Using XML with Caché

Version 5.1
15 June 2006
# Table of Contents

## 1 Introduction

1.1 Caché XML Features ................................................................. 1
1.2 Caché XML Architecture ......................................................... 2

## 2 XML Overview

2.1 XML Basics ............................................................................. 5
2.2 Validation and DTDs ................................................................. 6
2.3 XML Schemas ........................................................................... 7

## 3 The Caché Object-XML Projection

3.1 Changing the Default XML Projection ....................................... 10
  3.1.1 XMLNAME: Element Names .............................................. 11
  3.1.2 XMLPROJECTION: How Properties are Projected .............. 11
  3.1.3 XMLITEMNAME: Naming Items within a Collection ............ 12
  3.1.4 XMLKEYNAME: Naming Keys for Array Properties ............ 12
  3.1.5 XMLIO: Controlling Import and Export ............................ 12
  3.1.6 XMLTYPE: Controlling XML Schema Name ..................... 13
  3.1.7 XSDTYPE: Controlling XML Schema Type ...................... 13
3.2 Sample XML Projections ......................................................... 13
  3.2.1 Default Projection .......................................................... 13
  3.2.2 Changing Element Names ............................................... 14
  3.2.3 Changing Elements to Attributes ...................................... 14
  3.2.4 Disabling XML Projection of a Property .......................... 14
  3.2.5 Projecting Mixed Content for a Property ......................... 15
  3.2.6 Projecting a Property as Content .................................... 16
3.3 XML Projection of Collections ............................................... 16
  3.3.1 List Collections ........................................................... 16
  3.3.2 Array Collections ........................................................ 18

## 4 Serving XML from Caché

4.1 A Simple Caché XML Server ................................................... 19
4.2 Serving Objects as XML .......................................................... 21
  4.2.1 The %XML.Adaptor Class ............................................... 21
  4.2.2 Modifying the XML Server Class ..................................... 22

## 5 Importing XML Content into Caché

................................................................. 25
List of Figures

Object-XML Projection ........................................................................................................ 9
Caché SAX Interface .......................................................................................................... 37
List of Tables

XML Parameters ............................................................................................................... 10
XMLPROJECTION Values ............................................................................................. 11
XMLIO Values ............................................................................................................... 12
Node Types .................................................................................................................. 33
Node Values ............................................................................................................... 34
Element Information .................................................................................................. 34
1

Introduction

This document describes how Caché works with XML.

Caché brings the power of objects to XML processing: you are not constrained to using XML as simple text files (or relational BLOBs) nor are you forced to wedge complex XML documents into relational rows and columns. Instead, you can use objects as a direct representation of XML documents and vice versa. As Caché includes a native object database, you can use such objects directly with a database; you do not need additional complex middle-ware or conversion technologies.

You can use XML with Caché in a variety of ways, including:

- As a standard format in messaging applications. This includes industry-standard protocols as well as more home-grown solutions.
- As a standard format for data exchange between applications and users.
- As a standard representation for external data storage. This may include traditional database records or it may include more complex content such as documentation.

In addition, Caché uses its built-in XML capabilities to support SOAP and Web Services. For more information refer to Using SOAP and Web Services with Caché.

1.1 Caché XML Features

Caché offers the following XML-related features:
• Objects can automatically “project” themselves as XML documents. This includes projecting object properties as XML elements and attributes as well as supporting nested elements (represented as object collections, embedded objects, or object relationships). The projected XML can be served as standard files or as online content (such as in response to HTTP requests). The object projection can also be used to automatically generate an XML DTD or Schema (the standard XML methods for defining the contents of an XML document).

• XML documents can be automatically transformed into an equivalent object representation. You can use the resulting objects to perform business logic or for storage in the Caché object database. Incoming XML can be read from files, streams, or from incoming HTTP requests. Caché can validate any incoming XML using industry-standard XML DTD or Schema validation.

• For special applications, you can customize Caché's XML support in a number of ways. This includes an easy way to create custom XML server code as well as the ability to create custom XML import mechanisms (using Caché's SAX interface).

• Caché includes, within Studio, an XML Schema Wizard that automatically generates class definitions from XML schemas. For more information, refer to the section on XML Schemas.

1.2 Caché XML Architecture

Caché includes the following components for working with XML:

• The Caché SAX parser—a built-in SAX XML validating parser using the standard Xerces library. SAX is a parsing engine that provides complete XML validation and document parsing. Caché SAX communicates with a Caché process using a high-performance, in-process call-in mechanism. Using Caché SAX you can process XML documents using either Caché's built-in XML support (q.v.) or by providing your own custom SAX interface classes within Caché.

• The Caché XML library—a set of Caché classes for working with XML. These classes include %XML.Adaptor, which gives any class the ability to automatically project (serialize) its object instances to XML and vice versa; %XML.Reader which provides the ability to read XML documents and automatically convert them into object instances; and %XML.TextReader which provides a way to read generic XML documents without defining an object mapping.
These components are described in more detail later in this document.
This chapter contains a very basic overview of XML. It includes the following topics:

- **XML Basics**
- **Validation and DTDs**
- **XML Schemas**

To learn more about XML, you can go to almost any technical bookstore. You can also visit the W3C website: [http://www.w3.org](http://www.w3.org)

## 2.1 XML Basics

XML, the eXtensible Markup Language, provides a standardized, textual representation of data. Such a representation is referred to as a “markup language” because it starts with plain text content and adds a series of special markup characters that provide meaning to the content.

XML uses tags, like those used in HTML, to mark data in a text document. For instance, in HTML, a commonly used tag is the “<b>” tag, which marks text as boldface:

```html
<b>This text is in bold.</b>
```

What makes XML special is that it allows you to define your own document types each with their own set of tags. For example, you could create a document containing a tag called “MySpecialText”:

```xml
<MySpecialText>
Wow!
</MySpecialText>
```
In this example, MySpecialText is known as an element, so that the content of this MySpecialText element is “Wow!” Note that it is exactly clear where MySpecialText begins and ends, because there are a pair of tags that define the beginning and the end of the element (“<MySpecialText>” is its opening tag and “</MySpecialText>” is its closing tag); XML requires that all elements have opening and closing tags.

In practice, elements are likely to refer to different parts of data records, such as:

```xml
<Student level="undergraduate">
  <Name>Barnes, Gerry</Name>
  <DOB>1981-04-23</DOB>
</Student>
```

In this example, the Student element has a “level” attribute, where one possible value is “undergraduate”. Note that attribute values must be enclosed within quotation marks.

There are no particular rules about the use of elements and attributes in XML. Generally, elements are properties of data (such as Name and DOB above), while attributes define the structure of data being described (such as unique IDs). An example of this is:

```xml
<Person ID="230902">Smith, Leslie</Person>
```

Attributes often specify differences between different kinds of records, so that, in the Student example, an undergraduate might have an element specifying the student’s Minor, while a graduate student might have a ThesisAdvisor element.

## 2.2 Validation and DTDs

The primary function of XML is to allow you to define documents using a simple, well-defined structure. You can define such a document structure using a Document Type Definition (DTD).

A DTD (which is a series of text directives contained within an XML document or in an external file) provides the definition of the elements and attributes that can be used within a document. A DTD specifies all the valid elements for a document. In the Student example, suppose that the DTD specifies all different groups of people in a college’s database. In this case, there might be elements for Student, Professor, Instructor, Researcher, StaffMember, Alumnus, and so on; each element would have its own structure, which the DTD would specify.

In general, all XML documents (with or without a DTD) have to be well-formed. That is, they have to follow certain rules, including:
2.3 XML Schemas

A relatively new feature of the XML standard is the XML schema. An XML schema provides an alternative to the DTD for specifying meta-information for a set of XML documents. As with a DTD, you can use a schema to validate the contents of specific XML documents.

XML schemas offer several advantages over DTDs for certain applications including:

• An XML schema is itself a valid XML document making it easier to develop tools that operate on schemas.
• An XML schema can specify a richer set of features and includes type information for values.
Here is a sample XML schema defining a very simple Person record:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="MyApp.Person">
    <xsd:sequence>
      <xsd:element name="Age" type="xsd:integer"/>
      <xsd:element name="DOB" type="xsd:string"/>
      <xsd:element name="Home" type="s:MyApp.Address"/>
      <xsd:element name="Name" type="xsd:string"/>
      <xsd:element name="Office" type="s:MyApp.Address"/>
      <xsd:element name="SSN" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

For more information on XML schemas, refer to the W3C link [http://www.w3.org/TR/xmlschema-0/](http://www.w3.org/TR/xmlschema-0/).
The Caché Object-XML Projection

Caché projects objects as XML documents using an “XML projection”. The XML projection for a given class determines how object properties correspond to XML elements (or attributes) and vice versa.

A class that supports an XML projection is said to be “XML-enabled”. Any class that extends (inherits from) the %XML.Adaptor class (part of the Caché library) is automatically XML-enabled.

By default, the XML projection works as follows:

- Object instances correspond to top-level XML elements. By default the class name is used as an element name. A class’ package name is not used as part of its element name.
- Object properties correspond to XML elements or attributes within an enclosing top-level XML element. By default the property name is used as an element or attribute name. Also by default, only public properties are projected.
• Literal properties are projected using their “XSD” (SOAP-encoded) value (that is the LogicalToXSD method of their data type class is used, if present).

• Properties are projected to XML in the order of their sequence number; that is, the order in which they appear within Caché Studio.

• Collections, relationships, and embedded objects are projected as nested elements.

• Character streams are projected in the same manner as strings.

• Binary streams are projected using base64 encoding.

• A XML-enabled class includes a method, XMLDTD, that will provide a DTD for its XML projection.

• A XML-enabled class includes a method, XMLSchema, that will provide an XML Schema for its XML projection.

• As XML is a data representation, methods are not projected to XML.

3.1 Changing the Default XML Projection

You can control how a class is projected to XML by specifying values for certain class and property parameters.

XML Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLIO</td>
<td>Controls how a property is imported and exported.</td>
</tr>
<tr>
<td>XMLITEMNAME</td>
<td>Controls the name used for individual items within a collection when projected to XML.</td>
</tr>
<tr>
<td>XMLKEYNAME</td>
<td>Controls the name of the attribute used to hold the key values for array elements when projected to XML.</td>
</tr>
<tr>
<td>XMLNAME</td>
<td>Controls the name (element or attribute) used for a property or class when projected to XML.</td>
</tr>
<tr>
<td>XMLPROJECTION</td>
<td>Controls how a property is projected to XML.</td>
</tr>
<tr>
<td>XMLTYPE</td>
<td>Controls how a class name is represented in an XML schema.</td>
</tr>
<tr>
<td>XSDTYPE</td>
<td>Provides a way to override the XSD type used for a property when creating an XML Schema for a class.</td>
</tr>
</tbody>
</table>
These parameters are described in the following sections.

### 3.1.1 XMLNAME: Element Names

Using the *XMLNAME* parameter, you can change the name of the XML element corresponding to an object entity. You can do this at both the class and property level:

- Specify the XML element name that corresponds to a class using the *XMLNAME* class parameter:
  
  Parameter XMLNAME = "company";

- Specify the XML element or attribute name corresponding to a property *XMLNAME* property parameter:
  
  Property Name As %String(XMLNAME = "CompanyName");

### 3.1.2 XMLPROJECTION: How Properties are Projected

You can specify whether a property is represented as XML elements (the default) or attributes by specifying a value for its *XMLPROJECTION* property parameter.

The value of the *XMLPROJECTION* parameter can be one of the following:

<table>
<thead>
<tr>
<th>XMLPROJECTION</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>The property is not projected to XML.</td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td>The property is projected as an XML attribute.</td>
</tr>
<tr>
<td>ELEMENT</td>
<td>The property is projected as an XML element (the default for non-collection properties).</td>
</tr>
<tr>
<td>COLLECTION</td>
<td>The property is projected as a collection of items (the default for collection properties). See Collections below.</td>
</tr>
<tr>
<td>CONTENT</td>
<td>The property is projected as the primary content for this class (i.e., the contents of the property are written without any enclosing element). See below.</td>
</tr>
</tbody>
</table>

If *XMLPROJECTION* is not specified, then a default projection (i.e., one that is relevant to the current property) is used.

Refer to the following sections for examples of the *XMLPROJECTION* parameter.
Note that you can project a property not projected by default (such as a private property) by explicitly setting the value of the property’s XMLPROJECTION parameter:

Property InternalID As %Integer(XMLPROJECTION = "ATTRIBUTE") [Private];

### 3.1.3 XMLITEMNAME: Naming Items within a Collection

The XMLITEMNAME parameter lets you control the XML name used for items within a collection.

For more information, refer to the section on Collections below.

### 3.1.4 XMLKEYNAME: Naming Keys for Array Properties

The XMLKEYNAME parameter lets you control the XML name used for the key values for items within an array.

For more information, refer to the section on Collections below.

### 3.1.5 XMLIO: Controlling Import and Export

The XMLIO parameter controls how the XMLImport and XMLExport methods handle a property.

The XMLIO parameter can take one of the following values:

<table>
<thead>
<tr>
<th>XMLIO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>This property is used by XMLImport but is ignored by XMLExport.</td>
</tr>
<tr>
<td>OUT</td>
<td>This property is used by XMLExport but is ignored by XMLImport. If an XML element corresponding to this property is found in an XML document, then XMLImport will return an error.</td>
</tr>
<tr>
<td>INOUT</td>
<td>This property is used by both XMLImport and XMLExport.</td>
</tr>
<tr>
<td>CALC</td>
<td>This property is used by XMLExport but is ignored by XMLImport. If an XML element corresponding to this property is found in an XML document, then XMLImport will ignore it. Typically, this is used for calculated properties (whose value is based on the value of other properties) so that you can export a document with all values and ignore the calculated values on import.</td>
</tr>
</tbody>
</table>
If you do not specify a value for the XMLIO parameter, then the default value (INOUT) is used.

### 3.1.6 XMLTYPE: Controlling XML Schema Name

The XMLTYPE parameter lets you control the default type name (the classname with package) that is used in XML schemas output by the XMLSchema method.

The XMLTYPE parameter is distinguished from XMLNAME which controls the default top level tag name. You can use XMLTYPE to distinguish between two classes in different packages with the same name for SOAP usage.

### 3.1.7 XSDTYPE: Controlling XML Schema Type

Using the XSDTYPE parameter, you can override the default XSD (XML Schema Type) used for the XML schema generated for a class.

### 3.2 Sample XML Projections

The following examples demonstrate the default XML representation of an XML-enabled class and the effects of modifying the various XML parameter values.

#### 3.2.1 Default Projection

Here is the definition of a simple XML-enabled class:

```clj
Class MyApp.Company Extends (%Persistent,%XML.Adaptor) [ClassType=persistent] { 
    Property Name As %String; 
    Property CompanyID As %String; 
}
```

Here is a sample XML representation of an instance of this class:

```xml
<Company>
  <Name>XYZ Corp.</Name>
  <CompanyId>Q4673</CompanyId>
</Company>
```
3.2.2 Changing Element Names

Here is the same example with the XML element names used for the class and the CompanyId property modified using the XMLNAME parameter:

```plaintext
Class MyApp.Company Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Parameter XMLNAME = "CompanyData";

    Property Name As %String;
    Property CompanyID As %String(XMLNAME = "Id");
}
```

Here is a sample XML representation of an instance of this class:

```xml
<CompanyData>
    <Name>XYZ Corp.</Name>
    <Id>Q4673</Id>
</CompanyData>
```

3.2.3 Changing Elements to Attributes

Here is the same example with its CompanyId property projected as an XML attribute:

```plaintext
Class MyApp.Company Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Parameter XMLNAME = "CompanyData";

    Property Name As %String;
    Property CompanyID As %String(XMLNAME = "Id", XMLPROJECTION="ATTRIBUTE");
}
```

Here is a sample XML representation of an instance of this class:

```xml
<CompanyData Id="Q4673">
    <Name>XYZ Corp.</Name>
</CompanyData>
```

3.2.4 Disabling XML Projection of a Property

To prevent a property from being projected to XML, you can set its the XMLPROJECTION parameter to “NONE”.

Here is the same example with its Name property excluded from its XML projection:

```plaintext
Class MyApp.Company Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Parameter XMLNAME = "CompanyData";

    Property Name As %String(XMLPROJECTION="NONE");
    Property CompanyID As %String(XMLNAME = "Id", XMLPROJECTION="ATTRIBUTE");
}
```
Here is a sample XML representation of an instance of this class:

```xml
<CompanyData Id="Q4673">
</CompanyData>
```

### 3.2.5 Projecting Mixed Content for a Property

By default, a literal property (string or character stream) is assumed to not contain any XML elements. When a literal property value is projected to XML any “<” or “>” characters are escaped as “&lt;” and “&gt;”, respectively.

If you want the contents of a literal property to contain XML elements (for example, a memo field containing embedded markup commands that are not part of your data model) then you have to indicate that this property contains *mixed* content. The easiest way to do this is to use the `%XML.String` data type class included in the Caché class library.

Here is the same example with an additional `Memo` property of type `%XML.String` that contains mixed content:

```plaintext
Class MyApp.Company Extends (%Persistent,%XML.Adaptor) [ClassType=persistent] 
{
  Parameter XMLNAME = "CompanyData";

  Property Name As %String;
  Property CompanyID As %String(XMLNAME = "Id", XMLPROJECTION="ATTRIBUTE");
  Property Memo As %XML.String(MAXLEN=2000);
}
```

Here is a sample XML representation of an instance of this class:

```xml
<CompanyData Id="Q4673">
  <Name>XYZ Corp.</Name>
  <Memo>This is a <emphasis>very</emphasis> important customer.</Memo>
</CompanyData>
```

The `%XML.String` class is simply a subclass of the `%String` data type class that defines a special `CONTENT` parameter to “MIXED”.

Note that your application is responsible for ensuring that a mixed content property contains valid markup; the `%XML.String` class does not provide this validation.

If you wish to project a character stream containing mixed content, you need to set the `CONTENT` parameter for the stream property to “MIXED”:

```plaintext
Property MyStream
  As %CharacterStream(CONTENT = "MIXED");
```
3.2.6 Projecting a Property as Content

Sometimes you may wish to project a class to XML with no sub-elements; only a main element, possibly some attributes, and some text content. For example:

```xml
<Part Code="T22">Left-handed Screw Driver</Part>
```

To do this, create a class (typically as an embedded or serial object class) that contains a property whose `XMLPROJECTION` parameter is “CONTENT”.

For example, here is a class that corresponds to the above XML representation:

```caché
Class MyApp.Part Extends (%SerialObject,%XML.Adaptor) [ClassType=serial]
{
   Property Content As %String(MAXLEN=100,XMLPROJECTION="CONTENT");
   Property Code As %String(XMLPROJECTION="ATTRIBUTE");
}
```

There are some restrictions when using an `XMLPROJECTION` of “CONTENT” within a class:

1. There can only be one property projected as content. It must be a single-valued (non-collection), literal property.
2. All additional properties must be projected as attributes; The class cannot contain any properties that are projected as elements.

3.3 XML Projection of Collections

There are several ways in which the elements of a collection (list, array, or relationship) can be projected to XML. The easiest way to illustrate this is with some examples (for both lists and arrays).

3.3.1 List Collections

Suppose we define a simple class, MyApp.Test, that contains a single list property, Data:

```caché
Class MyApp.Test Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
   Property Data As List Of %String;
}
```

Here is a sample of the default XML representation of an instance of this class:
In this case, because Data is a collection, its XMLPROJECTION parameter is automatically set to “COLLECTION”. This places an outer <Data> tag (this name is determined by the XMLNAME parameter of the collection property) around a set of one or more <DataItem> tags enclosing the individual items within the collection.

Each item within a collection is enclosed within a tag. By default, the name of this tag is the string “Item” appended to the XMLNAME parameter for the collection. You can override this default using the XMLITEMNAME parameter of the collection property:

```
Class MyApp.Test Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Property Data As List of %String(XMLITEMNAME="Value");
}
```

Here is the corresponding XML representation of an instance of this class:

```
<Test>
  <Data>
    <DataItem>Value 1</DataItem>
    <DataItem>Value 2</DataItem>
    <DataItem>Value 3</DataItem>
  </Data>
</Test>
```

It is possible to project the contents of a collection with no enclosing tag (sometimes referred to as an “unwrapped array”). To do this, set the XMLPROJECTION parameter to “ELEMENT”. This will cause each item within the collection to be projected as an individual element. In this case, the XMLNAME parameter (and not the XMLITEMNAME parameter) is used to determine the name of the elements. For example:

```
Class MyApp.Test Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Property Data As List of %String(XMLPROJECTION = "ELEMENT");
}
```

Here is the corresponding XML representation of an instance of this class:

```
<Test>
  <Data>
    <DataItem>Value 1</DataItem>
    <DataItem>Value 2</DataItem>
    <DataItem>Value 3</DataItem>
  </Data>
</Test>
```
Setting the `XMLPROJECTION` parameter of a collection property to “ATTRIBUTE” has no meaning and will generate a compile-time error.

### 3.3.2 Array Collections

Array collections are treated in the exact manner as list collections with one important difference: each item within an array has a corresponding *key* associated with it. When an array is projected to XML, its keys are projected as attributes within the individual collection elements.

Suppose we define a simple class, `MyApp.Test`, that contains a single array property, `Data`:

```core
Class MyApp.Test Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Property Data As Array Of %String;
}
```

Here is a sample of the default XML representation of an instance of this class:

```xml
<Test>
    <Data>
        <DataItem DataKey="Key1">Value 1</DataItem>
        <DataItem DataKey="Key2">Value 2</DataItem>
        <DataItem DataKey="Key3">Value 3</DataItem>
    </Data>
</Test>
```

By default, the name of the attribute containing the key is created by appending “Key” to the element name used for the collection property. You can change this name using the `XMLKEYNAME` parameter. For example:

```core
Class MyApp.Test Extends (%Persistent,%XML.Adaptor) [ClassType=persistent]
{
    Property Data As Array Of %String(XMLKEYNAME="MyKey");
}
```

Here is a sample of the default XML representation of an instance of this class:

```xml
<Test>
    <Data>
        <DataItem MyKey="Key1">Value 1</DataItem>
        <DataItem MyKey="Key2">Value 2</DataItem>
        <DataItem MyKey="Key3">Value 3</DataItem>
    </Data>
</Test>
```
Serving XML from Caché

Caché can serve XML data in a variety of ways.

In this chapter we will demonstrate:

• Building a simple HTTP-based XML server using CSP technology.
• Modifying this server to use the Caché XML projection to server XML documents based on objects stored within a database.

4.1 A Simple Caché XML Server

In this section we will demonstrate how to create a simple HTTP-based XML server using Caché. This server will use CSP (Caché Server Page) technology to serve static XML content in response to incoming HTTP requests. In the subsequent sections we will enhance this server so that it serves XML documents based on persistent database objects.

The XML server is simply a subclass of the %CSP.Page class that has been extended to serve XML instead of HTML. The definition of such a server class looks like this:
/// A simple XML server
Class XMLSamples.Server Extends %CSP.Page
{
    Parameter CONTENTTYPE = "text/xml";
    ClassMethod OnPage() As %Status
    {
        Write "<?xml version="1.0" ?>",
        Write "<Company>",
        Write "<Name>XYZ Corp.</Name>",
        Write "<Mission>To serve and project.</Mission>",
        Write "</Company>",
        Quit $$$OK
    }
}

This class extends the %CSP.Page class, overrides the CONTENTTYPE parameter to "text/xml" and overrides the OnPage method to write out XML content (in this case, static content).

You can create a similar class within Caché Studio as follows:

1. Start the Caché Studio and create a new Project within the local server's “Samples” namespace.

2. Create a new class definition: invoke the New Class Wizard using the New command within the File menu and specify “New Class Definition”.

3. On the first page of the wizard, enter “XMLSamples” as a package name and “Server” as a class name.

4. On the second page of the wizard, specify that the new class definition will be a “CSP page”.

5. On the third page of the wizard, specify that this CSP page will serve “XML”.

6. Press the Finish button.

7. At this point, the Studio Class Editor will display a new class definition similar to:

    Class XMLSamples.Server Extends %CSP.Page
    {
        Parameter CONTENTTYPE = "text/xml";
        ClassMethod OnPage() As %Status
        {
            Write "<?xml version="1.0" ?>",
            Quit $$$OK
        }
    }

8. Modify the implementation of the OnPage method so that it serves some simple XML content using the Write command:
Class XMLSamples.Server Extends %CSP.Page
{
Parameter CONTENTTYPE = "text/xml";

ClassMethod OnPage() As %Status
{
    Write "<?xml version="1.0" ?>",!
    Write "<Company>",!
    Write "<Name>XYZ Corp.</Name>",!
    Write "<Mission>To serve and project.</Mission>",!
    Write "</Company>",!
    Quit $$$OK
}
}

9. Save and Compile the class using the Compile command from the Build menu.

10. View the results of this class by requesting the following URL from a browser:
    http://127.0.0.1/csp/samples/XMLSamples.Server.cls

    Depending on your browser, this may be rendered as an XML tree or you may only see text. In either case, your browser's View Source command should display the actual XML sent.

At this point you have a fully functional, albeit simple, XML server. To make this server more useful you could add additional logic to the server's OnPage method.

In the next section, we will demonstrate how to modify this server so that it serves object data.

4.2 Serving Objects as XML

Any XML-enabled Caché class can automatically project its instances as XML using the XMLExport method provided by the %XML.Adaptor class.

In this section we will create an XML-enabled class and then invoke its XMLExport method within the simple XML server class.

4.2.1 The %XML.Adaptor Class

Here is an XML-enabled class definition for a Company class that corresponds to the company information served in the previous section:
Serving XML from Caché

/// A company object.
Class XMLSamples.Company Extends (%Persistent,%XML.Adaptor,%Populate)
[ ClassType = persistent ]
{
    Property Name As %String;
    Property City As %String(POPSPEC = "City()");
    Property YearFounded As %Integer(POPSPEC = "Integer(1975,2000)");
    Property Mission As %String(MAXLEN = 500, POPSPEC = "Mission()");
}

The %XML.Adaptor class provides methods that manage exporting an object as XML, import XML into an object, and generating a DTD for the object. The %Populate class provides automatic data population (so that we will have some data to serve).

You can write the contents of an object instance to the current Caché device by invoking the XMLExport method provided by the %XML.Adaptor superclass. To try this, do the following:

1. Create a new persistent class definition using Caché Studio similar to the XMLSamples.Company class listed above. Make sure that this class extends both the %Populate and %XML.Adaptor classes. You can do this within the New Class Wizard by selecting the “XML-Enabled” and “Automatic Data Population” options.

2. Copy the property definitions from the listing above into the new class definition. Save and compile the new class definition.

3. Start the Caché Terminal, and using the command line: switch the “Samples” namespace and invoke the Populate method in order to generate sample data for the new class:

```
ZN "SAMPLES"
Do ##class(XMLSamples.Company).Populate(10)
```

At this point, you have an XML-enabled persistent class with 10 instances saved within the database. From the command line you can now open an object instance and invoke its XMLExport method:

```
Set company = ##class(XMLSamples.Company).%OpenId(1)
Do company.XMLExport()
```

The XMLExport method will write the contents of the Company object with Object ID of “1” to the console as XML.

### 4.2.2 Modifying the XML Server Class

We will now combine the previous two samples to demonstrate how to modify the XML server class so that it serves XML generated from one or more persistent objects.

To do this, modify the OnPage method from the XMLSamples.Server class so that it opens an object instance and invokes its XMLExport method:
Class XMLSamples.XMLServer Extends %CSP.Page
{
    Parameter CONTENTTYPE = "text/xml";
    ClassMethod OnPage() As %Status
    {
        Write "<?xml version="1.0" ?>",!
        Set company = ##class(XMLSamples.Company).%OpenId(1)
        Set status = company.XMLExport()
        If ($system.Status.IsError(status)) {
            Do $system.OBJ.DisplayError(status)
            Quit status
        }
        Quit $system.Status.OK()
    }
}

View the results by requesting the following URL from a browser:

http://127.0.0.1/csp/samples/XMLSamples.Server.cls

As in the previous example, this will serve an XML document to the browser.

You can control how XML is exported for a class by providing values for the various XML parameters as described in the XML Projection chapter.
5

Importing XML Content into Caché

The Caché XML projection lets you read an XML document and create one or more instances of XML-enabled Caché objects that correspond to elements of that document. This is done using an instance of the %XML.Reader class.

The %XML.Reader class works in conjunction with the methods provided by the %XML.Adaptor class to do the following:

- It parses and validates the incoming XML document using the Caché SAX interface. The validation can include either DTD or XML Schema validation.
- It determines if any XML-enabled objects are correlated with the elements contained within the XML document and creates in-memory instances of these objects as it reads the document.

Note that the objects instances created by %XML.Reader are not stored within the database; they are in-memory objects. If you want to store objects within the database, you have to call the %Save method (for persistent objects) or copy the relevant property values to a persistent object and save it. The application must also decide when new data should be inserted into a database and when existing data should be updated; %XML.Reader has no way of making this distinction.

5.1 An XML Import Example

Suppose there is an XML file called test.xml that contains a “Person” element:
We start by defining an XML-enabled class, MyApp.MyPerson, that is an object representation of a person:

```plaintext
Class MyApp.MyPerson Extends (%Persistent,%XML.Adaptor) [ClassType=persistent] 
{
  Parameter XMLNAME = "Person";

  Property Name As %String;
}
```

To import this file into an instance of a MyApp.MyPerson class, create a class method called `Import`:

```plaintext
Class MyApp.MyPerson Extends (%Persistent,%XML.Adaptor) [ClassType=persistent] 
{
  // ...
  ClassMethod Import()
  {
    // Create an instance of %XML.Reader
    Set reader = ##class(%XML.Reader).%New()

    // Begin processing of the file
    Do reader.OpenFile("test.xml")

    // Associate a class name with the XML element name
    Do reader.Correlate("Person","MyApp.MyPerson")

    // Read objects from xml file
    While (reader.Next(.object,.sc)) {
      Write object.Name,!
    }

    // If error found during processing, show it
    If $system.Status.IsError(sc) do $system.OBJ.DisplayError(sc)
  }
}
```

This method performs several tasks:

- It parses `test.xml` using the Caché SAX interface. This includes validating the document against its DTD, if specified.
- It associates (using the `Correlate` method) the class MyApp.MyPerson (part of the MyApp package) with the XML element “Person”; each child element in Person becomes a property of MyApp.MyPerson.
- It reads each Person element from `test.xml` until there are no more Person elements, or an error is reported via the reference variable, `sc`. In either instance, `Next` returns false.
- Finally, if the loop terminates because of an error, it is displayed on the current output device.
As mentioned above, this example does not store objects to the database. As MyPerson is a persistent object, you could do this by adding the line:

```plaintext
Set savestatus = object.%Save()
If $system.Status.IsError(savestatus) {
  do $system.OBJ.DisplayError(savestatus)
}
```

within the While loop.

The functionality for extracting XML data from a document is provided by the %XML.Reader class. Its typical use is as in the example above: opening a file, correlating XML element(s) with Caché object(s), and reading element(s) from XML into Caché.

You can also read data from a stream or a string using the OpenStream and OpenString methods of the %XML.Reader class, respectively.
The %XML.TextReader Class

The %XML.TextReader class offers a simple, easy-to-use way to process XML documents. Specifically, %XML.TextReader provides a way to iterate over an arbitrary, well-formed XML document and view its contents (elements, attributes, etc.) as object properties. %XML.TextReader provides additional methods for navigating XML documents such as finding the next occurrence of a particular element within a document.

The %XML.TextReader class provides complete document validation, based on either a DTD or an XML Schema.

The %XML.TextReader provides an alternative method for working with XML within Caché. Depending on your application or requirements, you may use it instead of (or along with) Caché’s XML object serialization mechanism (based on the %XML.Adaptor class) or its SAX-based content handler (using the %SOAP.Parser class).

6.1 Overview

The %XML.TextReader class processes an XML document as follows:

1. It reads an XML document from one of a variety of sources: (a stream, a file, or a simple string). The document must be a well-formed XML document; that is, it must obey the basic rules of XML syntax.

2. It validates the XML document using either a DTD (if specified within the document) or an XML Schema (if specified in the call to load the document). It does this using the same machinery as the Caché SAX processor.
3. It creates a document object model within Caché temporary storage that represents the contents of the XML document.

4. It creates and returns an instance of %XML.TextReader object. This object is used for navigating across the XML document.

5. When the %XML.TextReader object is destroyed, it automatically cleans up any temporary storage used for the document object model.

Here is a simple Caché ObjectScript example that creates an %XML.TextReader object for an XML file and then uses it to list every element within the document:

### Using %XML.TextReader

```objectscript
ClassMethod ShowElements(filename As %String) {
    // Create an instance of %XML.TextReader (returned by reference)
    Set sc = ##class(%XML.TextReader).ParseFile(filename,.reader)
    If ($$$ISOK(sc)) {
        // Read all elements within the document
        While (reader.Read()) {
            If (reader.NodeType = "element") {
                Write reader.Name,!
            }
        }
    }
}
```

This example does the following:

1. It calls the `ParseFile` class method. This reads the source file, `filename`; creates an %XML.TextReader object; and returns it in the variable, `reader`, by reference.

2. If `ParseFile` is successful, it then invokes the `Read` method to find each successive node within the document.

3. It checks the type of each node using the `NodeType` property. If the node is an element it prints out its name to the current device.

You can use the other methods and properties of the %XML.TextReader class to perform more sophisticated operations.

### 6.2 Document Parsing

The first step in processing an XML document with the %XML.TextReader class is to select a document source and parse its contents.
6.2.1 Selecting the Document Source

The %XML.TextReader class lets you choose the kind of source that will provide the XML document to process. It does this by providing a number of class methods that take an input source (or input source name) and return an instance of %XML.TextReader object (by reference) if they successfully parse the XML document.

These methods include:

<table>
<thead>
<tr>
<th>Method</th>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParseStream</td>
<td>A stream object.</td>
<td>Reads an XML document from a stream object. The stream object must be derived from the %AbstractStream class.</td>
</tr>
<tr>
<td>ParseString</td>
<td>A string.</td>
<td>Reads an XML document contained within a string. The string must contain valid XML document.</td>
</tr>
</tbody>
</table>

6.2.2 Validation

Using the Flags argument of the appropriate ParseXXX method, you can control whether the XML parser validates documents.

By default, the %XML.TextReader class will perform DTD validation if the document contains a DTD declaration. It will perform XML Schema validation if you provide a schema specification via the SchemaSpec argument.

A schema specification is a string contain a comma-separated list of namespace/URL pairs:

"namespace URL,namespace URL"

Where namespace is the XML namespace used for the schema and URL is a URL identifying the location of the schema document. There is a single space character separating the namespace and URL values.

6.2.3 Entity Resolution

If you are not interested in (or have no idea about) custom resolution of external entities within XML documents, then skip this section.
Your XML document may contain references to external DTD or entity definitions that have to be resolved—that is, the source documents for these definitions have to be found by the parser. In this case, you can control how these external entities are found by providing an instance of an entity resolver class using the Resolver argument of the appropriate ParseXXX method.

An entity resolver class is a derived from the %XML.SAX.EntityResolver class and implements the resolveEntity method.

Whenever the XML processor finds a reference to an external entity (such as a DTD), it invokes the resolveEntity method of the provided entity resolver object and passes it the Public and System entity identifier strings found in the source document.

For example, suppose your XML document contains the following:

```xml
<!DOCTYPE chapter PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
  "c:\cachesys\csp\docbook\doctypes\docbook\docbookx.dtd">
```

In this case the resolveEntity method would be invoked with publicId set to “-//OASIS//DTD DocBook XML V4.1.2//EN” and systemId set to “c:\cachesys\csp\docbook\doctypes\docbook\docbookx.dtd”.

The resolveEntity method determines the correct source for the external entity, creates an instance of a stream object (derived from %AbstractStream), and returns it. The XML parser reads the entity definition from this stream and closes the stream.

For an example of this refer to the %XML.Catalog and %XML.CatalogResolver classes included in the Caché library. The %XML.Catalog class defines a simple database that associates public and system identifiers with URLs. The %XML.CatalogResolver class is an entity resolver class that uses this database to find the URL for a given identifier. The %XML.Catalog class can load its database from an SGML-style Catalog file; this is a file containing a mapping of identifiers to URLs in a standard format.

### 6.3 Document Model

The %XML.TextReader class parses an XML document and creates a document model structure containing all the components of the document. These components are referred to as nodes.

As you iterate over the contents of an XML document, moving from node to node, the various properties of the %XML.TextReader object take on values which correspond to the current node.
6.3.1 Node Type

Each node has a type, available via the `NodeType` property, which is one of the following:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“attribute”</td>
<td>An XML attribute.</td>
</tr>
<tr>
<td>“cdata”</td>
<td>A CDATA section.</td>
</tr>
<tr>
<td>“chars”</td>
<td>A set of characters (such as content of an element).</td>
</tr>
<tr>
<td>“comment”</td>
<td>An XML comment.</td>
</tr>
<tr>
<td>“element”</td>
<td>The start of an XML element.</td>
</tr>
<tr>
<td>“endelement”</td>
<td>The end of an XML element.</td>
</tr>
<tr>
<td>“entity”</td>
<td>An XML entity.</td>
</tr>
<tr>
<td>“processinginstruction”</td>
<td>An XML processing instruction.</td>
</tr>
<tr>
<td>“ignorablewhitespace”</td>
<td>The white space between markup in a mixed content model.</td>
</tr>
</tbody>
</table>

6.3.2 Node Name

Element and attribute nodes have names. These are available via the `Name` and `LocalName` properties.

Name is a fully qualified name (including a namespace, if present). LocalName is the name without any namespace prefix. For example, the element `<xyz:Info>` would have Name of “xyz:Info” and LocalName of “Info”.

For other node types other than element and attribute, the value of these properties is an empty string.

6.3.3 Node Value

A node may have a value, which is available via the `Value` property. You can test if a node has a value using the `HasValue` property.

The contents of the `Value` property varies according to the node type:
6.3.4 Element Information

Element nodes have the following additional information available:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeCount</td>
<td>The number of attributes in the current node.</td>
</tr>
<tr>
<td>IsEmptyElement</td>
<td>Is the current element empty?</td>
</tr>
<tr>
<td>HasAttributes</td>
<td>Does the current element contain attributes?</td>
</tr>
</tbody>
</table>

6.4 Reading Content

The %XML.TextReader class lets you navigate over the nodes within its document model. As you move from node to node, the various properties of the %XML.TextReader object take on values which correspond to the current node.

You can only move forward within an XML document; you cannot move back to a previously read node.
When you first create an %XML.TextReader object it does not have a current node; you must advance to a node (using one the methods described below) before the properties of the %XML.TextReader object contain values.

6.4.1 Reading the Next Node

To move to the next node within a document, use the **Read** method:

```csharp
While (reader.Read()) {
}
```

The **Read** method returns a true value until there are no more nodes to read. The loop in the above example will execute until it reaches the end of the document.

6.4.2 Finding a Specific Element

You can move to the next occurrence of a specific element within a document using the **ReadStartElement** method.

6.4.3 Reading Attribute Values

To find attribute names and values for an element you must do the following:

1. Move to an element node (using the **Read** method, for example).
2. Move to a specific attribute by index (ordinal position of the attribute within the element) using the **MoveToAttributeIndex** method or by name using the **MoveToAttributeName** method.
3. Use the properties of the %XML.TextReader object to get information on the attribute.
4. When you are finished with the attributes for the current element, you can move to the next element in the document by invoking one of the navigation methods such as **Read**. Alternatively, you can invoke the **MoveToElement** method to return to the element that contains the current attribute.

For example, the following code lists all the attributes for the current node by index number:

```csharp
If (reader.NodeType = "element") {
    // list attributes for this node
    For a = 1:1:reader.AttributeCount {
        Do reader.MoveToAttributeIndex(a)
        Write reader.LocalName," = ",reader.Value,!
    }
}
```
The following code finds the value of the color attribute for the current node:

```csharp
If (reader.NodeType == "element") {
    // find color attribute for this node
    If (reader.MoveToAttributeName("color")) {
        Write "color = ", reader.Value, !
    }
}
```
To create a custom set of XML-object import mappings, use the Caché SAX interface. SAX (Simple API for XML) is an event-driven XML parser that allows you to read an XML file and invoke methods upon encountering elements in the file. Caché comes with a SAX content handler, the %XML.SAX.ContentHandler class. To create a custom XML import and handling mechanism, create a subclass of %XML.SAX.ContentHandler; then, in the new class, override each of the default content handler's methods to perform whatever actions are required.

This operation is illustrated in the following diagram:

The process for creating and using a custom import mechanism is to:

1. Create a class that extends %XML.SAX.ContentHandler.
2. Include in that class the methods that you wish to override.

3. Include in that class a class method that reads an external file using the **ParseFile** method of the %XML.SAX.Parser class.

4. Invoke that class method, passing it the file to parse.

When parsing the file, the overridden methods perform their defined actions and content is properly handled.

### 7.1 About the SAX Content Handler

The %XML.SAX.ContentHandler class parses XML files and generates events when it reaches particular points in the XML file. At each of these points, you can override the class' method (which, by default, are empty); this allows you to create custom content handling for your application. The methods are:

- **characters** — triggered by character data.
- **endDocument** — triggered by the end of the document.
- **endElement** — triggered by the end of an element.
- **endPrefixMapping** — triggered by the end of a namespace prefix mapping.
- **ignorableWhitespace** — triggered by ignorable whitespace in element content.
- **skippedEntity** — triggered by a skipped entity.
- **startDocument** — triggered by the beginning of the document.
- **startElement** — triggered by the start of an element.
- **startPrefixMapping** — triggered by the start of an namespace prefix mapping.

There are also methods for handling errors and control processing:

- **error** — triggered by a recoverable parser error.
- **fatalError** — triggered by a fatal XML parsing error.
- **processingInstruction** — triggered by an XML processing instruction.
- **warning** — triggered by notification of a parser warning.
To better understand the mechanics of setting up and using a content handler, consider the following example: suppose you want to get a list of all the elements that appear in a file. To do this, you need simply to note every start element (since each must have a matching end element). The process is then to:

1. Create a class, here called MyApp.Handler, which extends %XML.SAX.ContentHandler:

   ```
   Class MyApp.Handler Extends %XML.SAX.ContentHandler
   {
   
   }
   ```

2. Override the ContentHandler class' `startElement` method with the following content:

   ```
   Class MyApp.MyHandler extends %XML.SAX.ContentHandler
   {
   // ...

   Method startElement(uri As %Library.String, localname As %Library.String,
                     qname As %Library.String, attrs As %Library.List)
   {
     // we've found an element
     Write !,"Element: ",localname
   }
   }
   ```

3. Add a class method to the `Handler` class that reads and parses an external file:

   ```
   Class MyApp.MyHandler extends %XML.SAX.ContentHandler
   {
   // ...
   ClassMethod ReadFile(file as %String)
   {
     // create an instance of MyHandler class
     Set handler = ##Class(MyHandler).%New()

     // now parse the file using the handler
     Set sc = ##Class(%XML.SAX.Parser).ParseFile(file,handler)
   }
   }
   ```

Note that this is a class method because it is invoked in an application to perform its processing. This method does the following:

a. It creates an instance of a content handler object:

   ```
   Set handler = ##Class(MyHandler).%New()
   ```

b. It invokes the `ParseFile` method of the `%XML.SAX.Parser` class. This validates and parses the document (specified by `file`) and invokes the various event handling methods of the content handler object:
Using the SAX Interface

Set sc = ##Class(%XML.SAX.Parser).ParseFile(file,handler)

Each time an event occurs while the parser parses the document (such as a start or end element), the parser invokes the appropriate methods in the content handler object. In the case of this example, the only overridden method is startElement, which then performs its defined behavior of writing out element names; for other events, such as reaching end elements, nothing happens (the default behavior).

c. When the ParseFile method reaches the end of the file it returns. The handler object goes out of scope and is automatically removed from memory.

4. At the appropriate point in the application, invoke the ReadFile method, passing it the file to parse:

Do ##class(Samples.MyHandler).ReadFile(filename)

where filename is the pathname of the file being read.

For instance, if the content of the file is:

<class>
  <instructor>Jennings,Dave</instructor>
  <student>Caulfield,Phoebe</student>
  <student>Gittes,Jake</student>
  ...
  <student>Spackler,Carl</student>
</class>

then the output of this example will be:

Element: instructor
Element: student
Element: student

...