



Ensemble XML Virtual Document Development Guide

Version 2017.2
2020-06-26

Ensemble XML Virtual Document Development Guide

Ensemble Version 2017.2 2020-06-26

Copyright © 2020 InterSystems Corporation

All rights reserved.

InterSystems, InterSystems IRIS, InterSystems Caché, InterSystems Ensemble, and InterSystems HealthShare are registered trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)

Tel: +1-617-621-0700

Tel: +44 (0) 844 854 2917

Email: support@InterSystems.com

Table of Contents

About This Book	1
1 Introduction	3
2 Available Tools	5
2.1 Using the XML Schema Structures Page	5
2.2 Using the XML Document Viewer Page	5
2.3 Importing XML Schemas Programmatically	6
2.4 XML Classes	6
3 Configuration Steps	9
3.1 Loading XML Schemas into Ensemble	9
3.2 Adding a Business Service to Handle Inbound XML as Virtual Documents	9
3.3 Adding a Business Process to Handle XML Virtual Documents	10
3.4 Adding a Business Operation to Handle XML Virtual Documents	10
4 Overview of Property Paths in XML Virtual Documents	13
4.1 Orientation to Virtual Property Paths for XML Virtual Documents	13
4.1.1 Basic Syntax for Schema-dependent Paths	14
4.1.2 Basic Syntax for DOM-style Paths	14
4.2 Viewing Path Units for XML Virtual Documents	15
4.3 Redundant Inner Elements for Schema-dependent Paths	17
4.4 Repeating Fields	18
4.5 Duplicate Names	18
4.6 Choice Structures	19
4.7 Groups Included by Reference	20
5 Specifying Schema-dependent Paths for XML Virtual Documents	23
5.1 Getting or Setting the Contents of an XML Element	23
5.2 Getting or Setting the Value of an XML Attribute	24
5.3 Comments and Descriptions	24
5.4 Using Mixed Content When Setting Paths	25
5.5 Special Variations for Repeating Elements	25
5.5.1 Iterating Through the Repeating Elements	25
5.5.2 Counting Elements	26
5.6 Testing Schema-dependent Paths in the Terminal	26
6 Specifying DOM-style Paths for XML Virtual Documents	29
6.1 Getting or Setting Nodes (Basic Paths)	29
6.2 Using Mixed Content When Setting Paths	31
6.3 Using the Basic Path Modifiers	32
6.4 Using the Full() Function	33
6.5 Getting or Setting the Value of an XML Attribute	34
6.6 Using Path Modifiers to Insert or Append Nodes	34
6.7 Using the element() Function	37
6.8 Getting Positions of Elements	37
6.9 Getting Counts of Elements	38
6.10 Accessing Other Metadata	38
6.11 Summary of Path Modifiers	39
6.12 Variations for Documents That Use Namespaces	39

6.13 Testing DOM-style Paths in the Terminal	40
7 Defining Data Transformations for XML Virtual Documents	41
7.1 Creating a Data Transformation	41
7.2 Available Assignment Actions for XML Virtual Documents	42
7.3 Using Code	42
7.3.1 The pFormat Argument	43
7.4 Example 1: Copying Most of the Source Document	44
7.5 Example 2: Using Only a Few Parts of the Source Document	45
7.6 Example 3: Using Code and SetValueAt()	46
8 Defining Rule Sets for XML Virtual Documents	47
8.1 Creating a Rule Set	47
8.2 Example	48
9 Defining Search Tables for XML Virtual Documents	49
9.1 Introduction	49
9.2 Example	49
10 XML-Enabled Objects Compared to XML Virtual Documents	51
Reference for Settings	53
Settings for XML Business Services	54
Settings for XML Business Operations	55

About This Book

This book is one of a set that describes how to add virtual document interfaces and processing to Ensemble productions. It contains the following sections:

- [Introduction](#)
- [Available Tools](#)
- [Configuration Steps](#)
- [Overview of Property Paths in XML Virtual Documents](#)
- [Specifying Schema-dependent Paths](#)
- [Specifying DOM-style Paths](#)
- [Defining Data Transformations for XML Virtual Documents](#)
- [Defining Rule Sets for XML Virtual Documents](#)
- [Defining Search Tables for XML Virtual Documents](#)
- [XML-Enabled Objects Compared to XML Virtual Documents](#)
- [Reference for Settings](#)

For a detailed outline, see the [table of contents](#).

The following books provide related information:

- [Ensemble Best Practices](#) describes best practices for organizing and developing Ensemble productions.
- [Developing Ensemble Productions](#) explains how to perform the development tasks related to creating an Ensemble production.
- [Configuring Ensemble Productions](#) describes how to configure Ensemble productions, business hosts, and settings. It also provides reference information on settings not discussed in this book.
- [Ensemble Virtual Documents](#) describes the concept of Ensemble virtual documents and provides generic information on working with them.

For general information, see the *InterSystems Documentation Guide*.

1

Introduction

Ensemble provides support for XML documents as virtual documents. A *virtual document* is a kind of message that Ensemble parses only partially. This kind of message has the standard Ensemble message header and the standard message properties such as ID, Priority, and SessionId. The data in the message, however, is not available as message properties; instead it is stored directly in an internal-use global, for greater processing speed.

Ensemble provide tools so that you can access values in virtual documents, for use in data transformations, business rules, and searching and filtering messages. For background information, see [Ensemble Virtual Documents](#).

- Note:**
- When reading XML documents, Ensemble removes the XML declaration, all processing instructions, and all comments.
 - If the name of an element or attribute includes a period (.), Ensemble replaces that with a tilde (~).
For example, an XML element named `My.Element` appears as `My~Element` in Ensemble.

You can also work with XML documents as *standard* Ensemble messages. To do so, you can generate classes from the corresponding XML schema. For information, see [Using Caché XML Tools](#).

In some cases, it may be more efficient to use XML-enabled objects in productions instead of XML Virtual Documents. See “[XML-Enabled Objects Compared to XML Virtual Documents](#)” for more information.

2

Available Tools

This chapter provides an overview of the Ensemble tools that you can use to work with XML schemas and documents. It contains the following sections:

- [Using the XML Schema Structures Page](#)
- [Using the XML Document Viewer Page](#)
- [Importing XML Schemas Programmatically](#)
- [XML Classes](#)

2.1 Using the XML Schema Structures Page

The **XML Schemas** page enables you to import and view XML schema specifications.

To display this page, click **Ensemble > Interoperate > XML > XML Schema Structures**.

For general information on using this page, see “[Using the Schema Structures Page](#)” in *Ensemble Virtual Documents*.

Before importing a schema file, rename it so that its name is informative and unique within this Ensemble namespace. The filename is used as the schema category name in the Management Portal and elsewhere. If the filename ends with the file extension `.xsd`, the file extension is omitted from the schema category name. Otherwise the file extension is included in the name.

Note: You can use these schemas only to support processing of XML virtual documents as described in this book. Ensemble does not use them for any other purpose.

Important: After importing a schema file, do not remove the file from its current location in the file system. The XML parser uses the schema file rather than the schema stored in the Ensemble database.

2.2 Using the XML Document Viewer Page

The **XML Document** page enables you to display XML documents, parsing them in different ways, so that you can determine which DocType to use. You can also test transformations. The documents can be external files or documents from the Ensemble message archives.

To display this page, click **Ensemble > Interoperate > XML > XML Document Viewer**.

For general information on using this page, see “[Using the Document Viewer Page](#)” in *Ensemble Virtual Documents*.

2.3 Importing XML Schemas Programmatically

You can also load schemas programmatically by using the `EnsLib.EDI.XML.SchemaXSD` class directly. This class provides the **Import()** class method. The first argument to this method is the name of the file to import, including its full directory path. For example:

```
set status= ##class(EnsLib.EDI.XML.SchemaXSD).Import("c:\ENSEMBLE\myapp.xsd")
```

The `EnsLib.EDI.XML.SchemaXSD` class also provides the **ImportFiles()** method. For this method, you can specify the first argument in either of the following ways:

- As the name of a directory to import files from. Ensemble attempts to import all files in this directory, regardless of the file extensions. For example:

```
set status=##class(EnsLib.EDI.XML.SchemaXSD).ImportFiles("c:\ENSEMBLE\")
```

- As a list of filenames, separated by semicolons. You must include the full directory path for the first of these, and you can use wildcards in the filenames. For example:

```
set status=##class(EnsLib.EDI.XML.SchemaXSD).ImportFiles("c:\ENSEMBLE\*.xsd;*.XSD")
```

For more information, see the class reference for `EnsLib.EDI.XML.SchemaXSD`.

Important: After importing a schema file, do not remove the file from its current location in the file system. The XML parser uses the schema file rather than the schema stored in the Ensemble database.

2.4 XML Classes

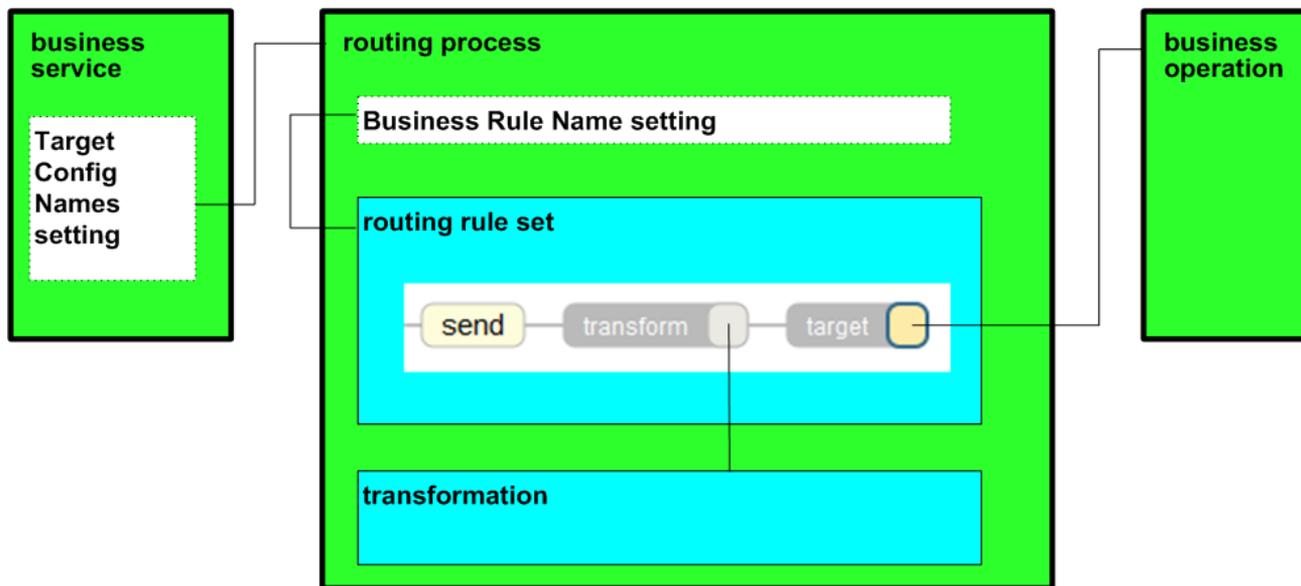
For reference, this section lists the classes that Ensemble provides to enable you to work with XML documents.

Item	Classes	Notes
XML business services	<ul style="list-style-type: none"> <code>EnsLib.EDI.XML.Service.FileService</code> <code>EnsLib.EDI.XML.Service.FTPService</code> 	Each of these business service classes uses a different adapter, as indicated by the class name.
XML routing process	<code>EnsLib.MsgRouter.VDocRoutingEngine</code>	This class is the standard virtual document routing process.
XML business operations	<ul style="list-style-type: none"> <code>EnsLib.EDI.XML.Operation.FileOperation</code> <code>EnsLib.EDI.XML.Operation.FTPOperation</code> 	Each of these business operation classes uses a different adapter, as indicated by the class name.
Messages	<code>EnsLib.EDI.XML.Document</code> (automatically used by the business host classes)	This is a specialized message class to carry XML documents as Ensemble virtual documents.

Item	Classes	Notes
Search table	EnsLib.EDI.XML.SearchTable	This is a specialized search table class for XML documents.

You can also create and use subclasses of these classes.

The business host classes include configurable targets. The following diagram shows some of them:



For information on other configurable targets, see “[Reference for Settings.](#)”

3

Configuration Steps

This chapter describes the configuration steps needed to use XML virtual documents in a production. It discusses the following topics:

- [How to load XML schemas into Ensemble](#)
- [How to configure a business service to handle inbound XML as virtual documents](#)
- [How to configure a business process to route XML virtual documents](#)
- [How to configure a business operation to handle XML virtual documents](#)

For information on settings not described here, see “[Reference for Settings](#).”

Later chapters describe how to create items to use in the production: [data transformations](#), [rule sets](#), and [search tables](#).

3.1 Loading XML Schemas into Ensemble

For XML virtual documents, it is useful, but not required, to load the corresponding XML schemas into Ensemble. If the schemas are available in Ensemble, then Ensemble can validate the documents, and you can use the schema-dependent virtual property paths (rather than only the DOM-style paths). Also, the DTL editor and the Business Rule Editor provide assistance with the document structure.

To load an XML schema into Ensemble, use the [XML Schema Structures](#) page, described in the chapter “[Available Tools](#).”

3.2 Adding a Business Service to Handle Inbound XML as Virtual Documents

To add a business service to handle inbound XML documents as Ensemble virtual documents, do the following:

1. To your production, add a business service that is based on the class `EnsLib.EDI.XML.Service.FileService` or `EnsLib.EDI.XML.Service.FTPService`.
2. Specify where this business service will find the inbound XML documents.

For example, for `EnsLib.EDI.XML.Service.FileService`, specify the **File Path** setting, which is the directory that the business service will check for new files.

3. Optionally specify other settings as needed. In particular, you might want to specify the following:
 - **Doc Schema Category**, which specifies the XML schema that applies to the inbound documents. Select a XML schema that you have previously loaded.

You must choose a schema if you want to validate the messages. The schema can also be used if you define search tables.
 - **Charset**, which specifies the character set of the inbound data. Ensemble automatically translates from this character encoding. For more options, see “[Charset](#)” in “[Reference for Settings](#)” in *Using File Adapters with Ensemble*.
 - **Search Table Class**. See the chapter “[Defining Search Tables for XML Virtual Documents](#).”

Make sure that this search table class is consistent with the kinds of messages received by this business host. For example, if the business host receives messages whose root element is <Transaction>, it would not be appropriate to use a search table class that used properties in an <Employee> element.
4. Specify where to send the XML documents. To do so, specify a comma-separated list of values for the **Target Config Names** setting. Each value should be the name of either a business process or a business operation.

3.3 Adding a Business Process to Handle XML Virtual Documents

To add a business process to handle XML virtual documents, do the following:

1. To your production, add a business process that is based on the class `EnsLib.MsgRouter.VDocRoutingEngine`.
2. For this business process, specify the **Business Rule Name** setting. Choose the appropriate business rule set that acts on XML virtual documents.

For information on defining these, see the chapter “[Defining Rule Sets for XML Virtual Documents](#).”
3. Optionally specify other settings as needed.
4. Configure the appropriate business host or hosts in the same production to send XML virtual documents to this business process:
 - For a business service, edit the **Target Config Names** setting to include the name of this business process.
 - For a business process, specify a **Business Rule Name** that routes messages to this business process.

3.4 Adding a Business Operation to Handle XML Virtual Documents

To add a business operation to send XML virtual documents to destinations outside of an Ensemble production, do the following:

1. To your production, add a business operation that is based on the class `EnsLib.EDI.XML.Operation.FileOperation` or `EnsLib.EDI.XML.Operation.FTPOperation`.
2. Specify settings of this business operation as needed.

For example, for `EnsLib.EDI.XML.Operation.FileOperation`, specify the **File Path** setting, which is the directory to which the business operation will write the files. The directory must exist and must be accessible.

3. Optionally specify the **Search Table Class** setting. See the chapter “[Defining Search Tables for XML Virtual Documents](#).”

Make sure that this search table class is consistent with the kinds of messages received by this business host. For example, if the business host receives messages whose root element is `<Transaction>`, it would not be appropriate to use a search table class that referred to an `<Employee>` element.

4. Configure the appropriate business host or hosts in the same production to send XML virtual documents to this business operation:
 - For a business service, edit the **Target Config Names** setting to include the name of this business operation.
 - For a business process, specify a **Business Rule Name** that routes messages to this business operation.

You might also want to add business operations to handle bad messages (for background, see “[Business Processes for Virtual Documents](#)” in *Ensemble Virtual Documents*).

4

Overview of Property Paths in XML Virtual Documents

This chapter provides an overview of property paths in XML virtual documents. It discusses the following:

- [Orientation to virtual property paths for XML documents](#)
- [How to view property path units for an XML schema](#)
- [How Ensemble handles redundant inner elements](#)
- [How Ensemble handles repeating elements](#)
- [How Ensemble handles duplicate elements](#)
- [How Ensemble handles choice structures](#)
- [How Ensemble handles groups included by reference](#)

The next two chapters describe in detail how to create property paths.

Note: The code examples in this chapter are fragments from data transformations, because data transformations generally use a richer set of property paths than do rule sets and search tables. Also, the emphasis is on DOM-style paths, because those are the paths that you must create manually. (In contrast, when you specify a schema to use, Ensemble displays the structure of the document and automatically generates schema-dependent paths when you drag and drop or when you use auto-completion.)

4.1 Orientation to Virtual Property Paths for XML Virtual Documents

This section briefly introduces virtual property paths for XML virtual documents.

As noted earlier, you can use schema-dependent paths only if you have loaded the corresponding XML schema. You can always use DOM-style paths, even when no schema is available.

4.1.1 Basic Syntax for Schema-dependent Paths

For XML virtual documents, a *schema-dependent path* consists of a set of *path units* separated by periods, as in the following example:

```
unit1.unit2.unit3
```

Where *unit1* is the name of a child XML element in the document, *unit2* is the name of a child element within *unit1*, and so on. The leaf unit is the name of either a child XML element or an XML attribute.

For example:

```
HomeAddress.City
```

For complete information, see the chapter “[Specifying Schema-dependent Paths.](#)”

4.1.2 Basic Syntax for DOM-style Paths

A *DOM-style path* always starts with a slash and has the basic structure shown in the following example:

```
/root_unit/unit1/unit2/unit3
```

Each path unit has the following form.

```
namespace_identifier:name
```

Where *namespace_identifier* represents the XML namespace; this is a token that Ensemble replaces with the actual namespace URI, as discussed in [a later subsection](#). This token is needed only if the element or attribute is in a namespace, as you will see later in this chapter.

name is the name of an XML element or attribute.

For example:

```
/$2:Patient/$2:HomeAddress/$2:City
```

For complete information, see “[Specifying DOM-style Paths.](#)”

4.1.2.1 XML Namespace Tokens

When you load a schema into Ensemble, Ensemble establishes a set of tokens for the namespaces used in that schema, for use in any DOM-style paths.

The token \$1 is used for first namespace that is declared in the schema; this usually corresponds to the XML schema namespace (<http://www.w3.org/2001/XMLSchema>). The token \$2 is used for the next namespace that is declared in the schema, \$3 is used for the third, and so on.

Ensemble assigns namespace tokens for all namespaces declared in the schema, whether or not those namespaces are actually used. Therefore, Ensemble might use \$3 or a higher value rather than \$2 for the items of interest to you, if additional namespaces are declared in the schema. It is practical to use the Management Portal to view the individual path units, as discussed in the [next section](#), to be sure that you are using the correct token for a specific path unit.

You can use namespace tokens if you have also loaded the corresponding schema (and have configured the applicable business host to use that schema). Otherwise, you must use the namespace prefixes exactly as given in the XML document.

4.2 Viewing Path Units for XML Virtual Documents

Until you are familiar with property paths for XML virtual documents, it is useful to use the Management Portal to view the individual path units. You can do this if you have loaded the corresponding schema.

To view the path units for the elements and attributes in a schema:

1. Load the schema as described in the [previous chapter](#).

For example, consider the following XML schema, shown here for reference, for the benefit of readers who are familiar with XML schemas:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified" targetNamespace="http://myapp.com"
        xmlns:myapp="http://myapp.com">
  <element name="Patient" type="myapp:Patient" />
  <complexType name="Patient">
    <sequence>
      <element minOccurs="0" name="Name" type="string" />
      <element minOccurs="0" name="FavoriteColors"
                type="myapp:ArrayOfFavoriteColorString" />
      <element minOccurs="0" name="Address" type="myapp:Address" />
      <element minOccurs="0" name="Doctor" type="myapp:Doctor" />
    </sequence>
    <attribute name="MRN" type="string" />
    <attribute name="DL" type="string" />
  </complexType>
  <complexType name="ArrayOfFavoriteColorString">
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="FavoriteColor"
                nillable="true" type="string" />
    </sequence>
  </complexType>
  <complexType name="Address">
    <sequence>
      <element minOccurs="0" name="Street" type="string" />
      <element minOccurs="0" name="City" type="string" />
      <element minOccurs="0" name="State" type="string" />
      <element minOccurs="0" name="ZIP" type="string" />
    </sequence>
  </complexType>
  <complexType name="Doctor">
    <sequence>
      <element minOccurs="0" name="Name" type="string" />
    </sequence>
  </complexType>
</schema>
```

The following shows an example XML document that obeys the schema shown in this section:

```
<?xml version="1.0" ?>
<Patient MRN='000111222' xmlns='http://myapp.com'>
  <Name>Georgina Hampton</Name>
  <FavoriteColors>
    <FavoriteColor>Red</FavoriteColor>
    <FavoriteColor>Green</FavoriteColor>
  </FavoriteColors>
  <Address>
    <Street>86 Bateson Way</Street>
    <City>Fall River</City>
  </Address>
  <Doctor>
    <Name>Dr. Randolph</Name>
  </Doctor>
</Patient>
```

2. Click **Ensemble > Interoperate > XML > XML Schema Structures**. This displays the **XML Schemas** page. The left column lists XML schemas loaded into this Ensemble namespace.
3. Click **Category** link in the row corresponding to the XML schema of interest.

If we do this for the XML schema shown previously, Ensemble then displays this:

XML DocType structures in Category **MyApp**

- Click the link for the document type of interest.

If we click **Patient**, Ensemble then displays this:

XML Document Structure / Document Type Definition

MyApp:Patient

Type: CT:Patient
xsd: element:complexType
Top Element: \$2:Patient

	Name	Type	Required	Element
1	Name		No	\$2:Name
2	FavoriteColors()	ArrayOfFavoriteColorString()	No	\$2:FavoriteColors/\$2:FavoriteColor[]
3	Address	Address	No	\$2:Address
4	Doctor	Doctor	No	\$2:Doctor/\$2:Name
5	MRN		No	@MRN
6	DL		No	@DL

On this page:

- Above the table, the value in large font displays the DocType value for this XML element. In this case, DocType is `MyApp:Patient`.
- The **Name** column shows path units in the format needed for schema-dependent paths.

In this case, this page tells us that we can use `Name`, `FavoriteColors`, `Address`, `Doctor`, `MRN`, and `DL` as path units in schema-dependent paths.

- The **Element** column shows path units in the format needed for DOM-style property paths.

In this case, this page tells us that we can use `$3:Name`, `$2:FavoriteColors/$2:FavoriteColor`, `$2:Address`, `$2:Doctor/$2:Name`, `@MRN`, and `@DL` as path units in DOM-style paths. Notice that `@MRN` and `@DL` do not have a namespace prefix; these attributes are not in any namespace.

- Click additional sub-items as wanted.

If we click **Address** in the **Name** column, Ensemble displays this:

XML Complex Type Structure Definition

MyApp:Address

Type: CT:Address
xsd: complexType

	Name	Type	Required	Element
1	Street		No	\$2:Street
2	City		No	\$2:City
3	State		No	\$2:State
4	ZIP		No	\$2:ZIP

This page displays any additional path units within *Address*.

In this case, this page tells us that we can use these additional path units in combination with the path unit that we used to get to this page, for example:

Schema-dependent path (partial)	...Address.Street
DOM-style path (partial)	/.../\$2:Address/\$2:Street

The following sections note specific variations due to schema variations.

4.3 Redundant Inner Elements for Schema-dependent Paths

For schema-dependent paths, Ensemble collapses redundant inner elements. This is best explained by example:

- The `<FavoriteColors>` element contains a sequence of multiple `<FavoriteColor>` elements. On the schema viewer page, `<FavoriteColors>` is shown simply as **FavoriteColors()** in the **Name** column (which shows the path unit for schema-dependent paths). This column is displayed in blue in the following figure.

XML Document Structure / Document Type Definition

MyApp:Patient

Type: CT:Patient**xsd:** element:complexType**Top Element:** \$2:Patient

	Name	Type	Required	Element
1	Name		No	\$2:Name
2	FavoriteColors()	ArrayOfFavoriteColorString()	No	\$2:FavoriteColors/\$2:FavoriteColor[]
3	Address	Address	No	\$2:Address
4	Doctor	Doctor	No	\$2:Doctor/\$2:Name
5	MRN		No	@MRN
6	DL		No	@DL

In contrast, the same element is shown as `$2:FavoriteColors/$2:FavoriteColorsItem` in the **Element** column on the right. This column shows the path unit for DOM-style paths.

For a sequence of multiple items of the same type, the schema-dependent path does not use the name of the inner element. (In contrast, the DOM-style path uses all the element names.) More generally, any redundant inner levels found in a schema are ignored in schema-dependent paths; the following item shows another example.

- The `<Doctor>` element includes a single `<Name>` element. On the schema viewer page, the `<Doctor>` item is shown as **Doctor** in the **Name** column, as shown in the previous figure.

Notice that the schema-dependent path to the data inside `<Doctor>` does not use the name of the inner element.

In contrast, the same item is shown as `$3:Doctor/$3:Name` in the **Element** column on the right. This column shows the path unit for DOM-style paths.

4.4 Repeating Fields

If a given element can occur multiple times, the **Name** column displays parentheses () at the end of the element name. For example, see the **FavoriteColors()** row in the preceding figure.

The **Type** and **Element** columns indicate the number of times the element can be repeated. In this case, the element can be repeated five times. If there is no number displayed in parentheses in the **Type** column, the element can be repeated any number of times.

4.5 Duplicate Names

If an XML schema has multiple elements at the same level that have the same name but different types, then Ensemble appends `_2`, `_3`, and so on, as needed to create unique names at that level. This procedure applies only to the schema-dependent paths. For example, consider a schema that defines the `<Person>` element to include two elements named

<Contact>. One is of type <Phone> and the other is of type <Assistant>. Ensemble displays the schema for the <Person> element as follows:

Type: DS:Person
xsd: complexType

	Name	Type	Required	Element
1	Contact	Phone	No	\$2:Contact
2	Contact_2	Assistant	No	\$2:Contact

Similarly, if the schema has multiple elements at the same level but in different namespaces, then Ensemble appends _2, _3, and so on, as needed to create unique names at that level. This procedure applies only to the schema-dependent paths.

4.6 Choice Structures

Some schemas include <choice> structures, like the following example:

```
<xsd:choice>
  <xsd:element name="OptionA" type="my:OptionType" />
  <xsd:element name="OptionB" type="my:OptionType" />
  <xsd:element name="OptionC" type="my:OptionType" />
</xsd:choice>
```

Ensemble represents this structure differently for the two kinds of paths. The following shows an example:

Type: CT:MainType
xsd: element:complexType
Top Element: \$2:Parent

	Name	Type	Required	Element
1	choice	#1	Yes	-
2	OtherElement	OtherType	Yes	OtherElement
3	OtherAttribute	date	No	@OtherAttribute

For schema-dependent paths, the **Name** displays a generic name for the <choice> structure, and the **Type** column displays a numeric placeholder. The **Element** column does not display anything.

If we click `choice`, Ensemble then displays the following:

Type: #choice
xsd: choice

	Name	Type	Required	Element
1	OptionA	OptionType	Yes	OptionA
2	OptionB	OptionType	Yes	OptionB
3	OptionC	OptionType	Yes	OptionC

In this case, these pages tell us that we can use the following paths to access OptionB:

Schema-dependent path (partial)	...Parent.choice.OptionB
DOM-style path (partial)	/.../Parent/OptionB

4.7 Groups Included by Reference

A schema can include a <group> that is included via the ref attribute. For example:

```
<s01:complexType name="Patient">
  <s01:sequence>
    <s01:element name="Name" type="s01:string" minOccurs="0"/>
    <s01:element name="Gender" type="s01:string" minOccurs="0"/>
    <s01:element name="BirthDate" type="s01:date" minOccurs="0"/>
    <s01:element name="HomeAddress" type="s02:Address" minOccurs="0"/>
    <s01:element name="FavoriteColors"
      type="s02:ArrayOfFavoriteColorsItemString" minOccurs="0"/>
    <s01:element name="Container" type="s02:ContainerType" minOccurs="0"/>
    <s01:element name="LatestImmunization" type="s02:Immunization" minOccurs="0"/>
    <s01:element ref="s02:Insurance" minOccurs="0"/>
    <s01:group ref="s02:BoilerPlate" minOccurs="1" maxOccurs="1"/>
  </s01:sequence>
  ...
<s01:group name="BoilerPlate">
  <s01:sequence>
    <s01:element name="One" type="s01:string"/>
    <s01:element name="Two" type="s01:string"/>
    <s01:element name="Three" type="s01:string"/>
  </s01:sequence>
</s01:group>
```

Ensemble represents this structure differently for the two kinds of paths. The following shows an example:

Type: CT:Patient
xsd: element:complexType
Top Element: \$2:Patient

	Name	Type	Required	Element
1	Name		No	\$2:Name
2	Gender		No	\$2:Gender
3	BirthDate	date	No	\$2:BirthDate
4	HomeAddress	Address	No	\$2:HomeAddress/\$2:Address
5	FavoriteColors()	ArrayOfFavoriteColorsItemString()	No	\$2:FavoriteColors/\$2:FavoriteColorsItem[]
6	Container	ContainerType	No	\$2:Container/\$2:Intermediate/\$2:Final
7	LatestImmunization	Immunization	No	\$2:LatestImmunization
8	Insurance	Insurance	No	\$2:Insurance
9	BoilerPlate	#9	Yes	BoilerPlate
10	MRN		No	@MRN

For schema-dependent paths, the **Name** displays the name of the group, and the **Type** column displays a numeric placeholder. The **Element** column also displays the name of the group.

If we click `BoilerPlate`, Ensemble then displays the following:

Type: CG:BoilerPlate
xsd: sequence:group

	Name	Type	Required	Element	Default	Description
1	One		Yes	\$2:One		
2	Two		Yes	\$2:Two		
3	Three		Yes	\$2:Three		

In this case, these pages tell us that we can use the following paths to access `Two`:

Schema-dependent path (partial)	<code>...Patient.BoilerPlate.Two</code>
DOM-style path (partial)	<code>/.../\$2:Patient/\$2:Two</code>

5

Specifying Schema-dependent Paths for XML Virtual Documents

This chapter describes how to specify schema-dependent paths for XML virtual documents. It discusses the following:

- [How to access contents of an element](#)
- [How to access the value of an attribute](#)
- [Comments and descriptions](#)
- [Mixed content](#)
- [Special variations for repeating elements](#)
- [How to test schema-dependent paths in the Terminal](#)

You can use these paths to access values and to set values.

The examples in this chapter use the schema shown in the [previous chapter](#).

5.1 Getting or Setting the Contents of an XML Element

To access the contents of an element, you can use one of the following schema-dependent paths. You also use these paths when you create more complex schema-dependent paths as discussed in later subsections.

Syntax	Refers to
<i>element_name</i>	Contents of the given element. <i>element_name</i> must be a child of the root element.
<i>parent.element_name</i>	Contents of the given element. <i>parent</i> is the full path to an element — that is, any syntax shown in this table. In this case, <i>element_name</i> is a child of the element referred to by <i>parent</i> .
<i>parent.element_name</i> (<i>n</i>)	Contents of the <i>n</i> th element with the name <i>element_name</i> within the given parent.
<i>parent.element_name</i> (-)	Contents of the last element with the name <i>element_name</i> within the given parent.

Consider the following XML document:

```
<?xml version="1.0" ?>
<Patient MRN='000111222' xmlns='http://myapp.com'>
  <Name>Georgina Hampton</Name>
  <FavoriteColors>
    <FavoriteColor>Red</FavoriteColor>
    <FavoriteColor>Green</FavoriteColor>
  </FavoriteColors>
  <Address>
    <Street>86 Bateson Way</Street>
    <City>Fall River</City>
  </Address>
  <Doctor>
    <Name>Dr. Randolph</Name>
  </Doctor>
</Patient>
```

The following table shows some example paths for this document:

Example Path	Current Path Value
Name	Georgina Hampton
FavoriteColors(1)	Red
FavoriteColors(2)	Green
FavoriteColors(-)	Green
Address	86 Bateson WayFall River
Address.Street	86 Bateson Way
Doctor	Dr. Randolph

5.2 Getting or Setting the Value of an XML Attribute

To access the value of an attribute, you can use one of the following schema-dependent paths. Here (and in the rest of this section), *element_reference* is a complete schema-dependent path as described in the previous table.

Syntax	Refers to
<i>element_reference.attribute_name</i>	Value of the <i>attribute_name</i> attribute of the element indicated by <i>element_reference</i> .

The following table shows an example path for the previous document:

Example Path	Current Path Value
MRN	000111222

5.3 Comments and Descriptions

Ensemble removes the comments when it reads XML files. Consequently, you should not use comments to document the schema. Instead of using comments, you can use the `description` or `altdesc` attributes available on most schema elements.

Although it is not useful in most cases, you can access a comment by using one of the following schema-dependent paths:

Syntax	Refers to
<i>element_reference</i> .#	Text of the first comment of the given element.
<i>element_reference</i> .#(<i>n</i>)	Text of the <i>n</i> th comment of the given element.
<i>element_reference</i> .#(-)	Text of the last comment.

Note: Ensemble removes all comments when it reads in XML files. The only comments that can be present are comments that have been added since the XML file was read. To add a comment, use `setValueAt()` with a path like one shown in the preceding table.

5.4 Using Mixed Content When Setting Paths

When you use `setValueAt()`, you can specify a value that consists of mixed content (that is, a value that consists of a mix of element and text nodes). For example:

```
set mixed="SOME TEXT<HOMETOWN>BELMONT</HOMETOWN>"
set status=target.SetValueAt(mixed,"Address")
```

The following table describes how Ensemble handles the value in different scenarios:

Path Refers to	How Ensemble Handles the Mixed Content
element or comment	Ensemble replaces the current contents of the element or comment with the given mixed content
attribute	Not supported

5.5 Special Variations for Repeating Elements

This section describes variations of virtual property paths that apply when you are referring to a repeating element.

5.5.1 Iterating Through the Repeating Elements

If the path refers to a repeating element, you can use the following syntax to iterate through every instance of that element.

Syntax	Refers to
<i>element_name</i> ()	Iterates through the elements of the given name, within the given context.

Suppose that we now use a data transformation that contains only the following code:

```
set status=target.SetValueAt("REPLACED COLOR","FavoriteColors()")
if 'status {do $system.Status.DisplayError(status) quit}
```

This line of code transforms the document shown previously in this chapter to the following:

```
<?xml version="1.0" ?>
<Patient MRN='000111222' xmlns='http://myapp.com'>
  <Name>Georgina Hampton</Name>
  <FavoriteColors>
    <FavoriteColor>REPLACED COLOR</FavoriteColor>
    <FavoriteColor>REPLACED COLOR</FavoriteColor>
  </FavoriteColors>
  <Address>
    <Street>86 Bateson Way</Street>
    <City>Fall River</City>
  </Address>
  <Doctor>
    <Name>Dr. Randolph</Name>
  </Doctor>
</Patient>
```

5.5.2 Counting Elements

If the path refers to a repeating element, you can use the following syntax to return the number of elements.

Syntax	Refers to
<i>element_name</i> (*)	Number of elements of the given name, within the given context. This syntax is valid only if the schema defines <i>element_name</i> as a repeating element.
<i>element_name</i> . *	Number of elements of the given name, within the given context. This syntax is valid for any <i>element_name</i> .

The following table shows example paths for the document shown previously in this chapter:

Example Path	Current Path Value
FavoriteColors. *	2
FavoriteColors(*)	2

5.6 Testing Schema-dependent Paths in the Terminal

It can be useful to test virtual document property paths in the Terminal before using them in business processes, data transformations, and so on, particularly when you are getting familiar with the syntax. To do so for schema-dependent XML paths, do the following:

1. Load the corresponding XML schema or schemas into Ensemble. To do so, use the [XML Schema Structures](#) page, described in the chapter “[Available Tools](#).”
2. Use the Management Portal to find the DocType value for the root element of the documents that you plan to test. For example:

XML Document Structure / Document Type Definition
MyApp:Patient

See “[Viewing Path Units for XML Virtual Documents](#),” earlier in this book.

3. In the Terminal or in test code:
 - a. Create a string that contains the text of a suitable XML document.

- b. Use the **ImportFromString()** method of `EnsLib.EDI.XML.Document` to create an instance of an XML virtual document from this string.
- c. Set the `DocType` property of this instance.
- d. Use the **GetValueAt()** and **SetValueAt()** methods of this instance.

The following method demonstrates step 3:

```
ClassMethod TestSchemaPath()
{
    set string="<Patient xmlns='http://myapp.com'>"
    _"<Name>Jack Brown</Name>"
    _"<Address><Street>233 Main St</Street></Address>"
    _"</Patient>"
    set target=##class(EnsLib.EDI.XML.Document).ImportFromString(string,.status)
    if 'status {do $system.Status.DisplayError(status) quit}

    //Use the DocType displayed in the Management Portal
    set target.DocType="MyApp:Patient"

    set pathvalue=target.GetValueAt("Address.Street",,.status)
    if 'status {do $system.Status.DisplayError(status) quit}
    write pathvalue
}
```

The following shows output from this method:

```
ENSDEMO>d ##class(Demo.CheckPaths).TestSchemaPath()
233 Main St
```

For additional options for **GetValueAt()**, see [“The pFormat Argument,”](#) later in this book.

6

Specifying DOM-style Paths for XML Virtual Documents

This chapter describes how to specify DOM-style paths for XML virtual documents. It discusses the following:

- [How to get or set a document node \(basic paths\)](#)
- [How to use mixed content when setting a value](#)
- [How to use basic path modifiers](#)
- [How to use the full\(\) function](#)
- [How to get or set the value of an attribute](#)
- [How to insert or append nodes](#)
- [How to use the element\(\) function](#)
- [How to get positions of elements](#)
- [How to count items](#)
- [How to get other metadata](#)
- [Summary of path modifiers](#)
- [How to modify a path to consider namespaces](#)
- [How to test DOM-style paths in the Terminal](#)

You can use these paths to access values and to set values (with noted exceptions).

Most of the following sections assume that the document does not use any XML namespaces. The [last section](#) gives information on adapting these paths for a document that does use XML namespaces.

The examples in this chapter use the schema shown in the chapter “[Overview of Property Paths in XML Virtual Documents](#).”

6.1 Getting or Setting Nodes (Basic Paths)

In an XML virtual document, there are five kinds of nodes: the root node, elements, text nodes, comments, and processing instructions. The root node and any element can have child nodes of any type. The other kinds of nodes cannot have child nodes. Attributes are not nodes.

The following table lists basic DOM-style paths to get or set many of the nodes of an XML virtual document. When there are multiple nodes of the same type or with the same name, and when you do not want the first one, see the [next section](#).

You also use these paths when you create more complex DOM-style paths as discussed in later subsections.

Syntax	Refers to
/	Contents of the root node. You can also use " ", if the context makes it clear that you are using a DOM-style path (that is, if no schema is loaded).
/root_element_name	Contents of the root element, whose name is <i>root_element_name</i> .
parent/element_name	Contents of the first element of the given name (<i>element_name</i>), within the given parent. Here <i>parent</i> is the full path to its parent element, including (as always) the initial slash.
element_reference/text()	First text node in the element indicated by <i>element_reference</i> .
element_reference/comment()	<p>First comment in the element indicated by <i>element_reference</i>.</p> <p>The value returned does not include the opening syntax (<!--) or the closing syntax (-->). Similarly, do not include the opening or closing syntax when setting the value.</p> <p>Ensemble removes all comments when it reads in XML files. The only comments that can be present are comments that you add. (To add them, use setValueAt() with a path like the one shown here.)</p>
element_reference/instruction()	<p>First processing instruction in the element indicated by <i>element_reference</i>.</p> <p>The value returned does not include the opening syntax (<?) or the closing syntax (?>). Similarly, do not include the opening or closing syntax when setting the value.</p> <p>Ensemble removes all processing instructions when it reads in XML files. The only instructions that can be present are instructions that you add. (To add them, use setValueAt() with a path like the one shown here.)</p>

Consider the following XML document:

```
<?xml version="1.0" ?>
<Patient xmlns='http://myapp.com'>Sample text node
  <!--Sample comment-->
  <!--Another comment-->
  <Name>Jane Doe</Name>
  <Address>
    <Street>100 Blank Way</Street>
  </Address>
</Patient>
```

The following table shows some example paths for this document:

Example Path	Current Path Value
/Patient/Name	Jane Doe

Example Path	Current Path Value
/Patient/Address	<Street>100 Blank Way</Street> In this case, the referenced element contains a child element (in contrast to the previous example). Note that Ensemble ignores whitespace when comparing DOM-style paths to values. That is, the value here matches the given path whether or not the document contains line breaks and indentation.
/Patient/Address/Street	100 Blank Way
/Patient/text()	Sample text node
/Patient/comment()	Sample comment

Suppose that we now use a data transformation that contains only the following code:

```
set status=target.SetValueAt("892 Broadway", "/Patient/Address/Street")
if 'status {do $system.Status.DisplayError(status) quit}
set status=target.SetValueAt("Dr. Badge", "/Patient/Doctor/Name")
if 'status {do $system.Status.DisplayError(status) quit}
```

Notice that one of these paths already exists and the other does not; both paths are valid. After we use this transformation, the new document would then look like the following:

```
<?xml version="1.0" ?>
<Patient xmlns='http://myapp.com'>Sample text node
  <!--Sample comment-->
  <!--Another comment-->
  <Name>Jane Doe</Name>
  <Address>892 Broadway</Address>
  <Doctor>
    <Name>Dr. Badge</Name>
  </Doctor>
</Patient>
```

6.2 Using Mixed Content When Setting Paths

When you use `setValueAt()` to set the value at a node, you can specify a value that consists of mixed content (that is, a value that consists of a mix of element and text nodes). For example:

```
set mixed="SOME TEXT<HOMETOWN>BELMONT</HOMETOWN>"
set status=target.SetValueAt(mixed, "/Patient/Address/Street")
```

The following table describes how Ensemble handles the value when you set the value of each different kind of node:

Node Type	How Ensemble Handles Mixed Content Provided for the Node Value
root	Not supported
element or comment	Ensemble replaces the current contents of the node with the given mixed content
text node or instruction	Ensemble escapes the XML special characters and then replaces the current contents of the given node

Note that attributes are not nodes.

6.3 Using the Basic Path Modifiers

You can add the following basic path modifiers to the end of basic paths (listed in the [previous section](#)), with noted exceptions. You can use the resulting paths in the same way that you use any of the basic paths.

[n]

Refers to an item by item position. Only instances of that item are counted; items of other types are ignored.

- When you get a value, this syntax returns the *n*th instance of the item to which the basic path refers (or an empty string otherwise).
- When you set a value, this syntax either overwrites or creates the *n*th instance of the item to which the basic path refers.

You can substitute a hyphen (-) to access the last instance. You can also omit the square brackets.

/[n]

Refers to a child element by child element position.

You can substitute a hyphen (-) to access the last child. You can also omit the square brackets.

Restrictions:

- You can use this only with a basic path that refers to an element; that is, you cannot use it with functions such as `comment()`.
- You can use this syntax only when getting a value, not when setting a value.

You can combine this path modifier with the other path modifiers, if you use the `/[n]` modifier as the *last* modifier.

[\$n]

Refers to an item by node position.

- When you get a value, this syntax returns the *n*th node, if that node is an instance of the item to which the basic path refers. Otherwise the path is invalid, and an error is returned.
- When you set a value, this syntax overwrites the *n*th node, if that node is an instance of the item to which the basic path refers. Otherwise the path is invalid, and an error is returned.

Different path modifiers, listed in a [later section](#), enable you to insert or append nodes. (Also see “[Summary of Path Modifiers](#)”.)

Consider the following XML document:

```
<?xml version="1.0" ?>
<Patient xmlns='http://myapp.com'>
  <!--Sample comment-->
  <!--Another comment-->
  Sample text node
  <Name>Fred Williams</Name>
  <FavoriteColors>
    <FavoriteColor>Red</FavoriteColor>
    <FavoriteColor>Green</FavoriteColor>
  </FavoriteColors>
  <Doctor>
    <Name>Dr. Arnold</Name>
  </Doctor>
</Patient>
```

The following table shows some example paths for this document:

Example Path	Current Path Value	Notes
/Patient/Name	Fred Williams	
/[1]/[1]	Fred Williams	This path accesses the first child element within the first element of the document (which is the only element in the document, according to the XML standard). The square brackets are optional here.
/Patient/FavoriteColors/[1]	Red	The square brackets are optional here.
/Patient/FavoriteColors/[2]	Green	
/[1]/[2]/[1]	Red	
/[1]/[2]/[2]	Green	
/Patient/Name[\$1]	An empty string	This path is invalid. The first node within <Patient> is not a <Name> element.
/Patient/Name[\$4]	Fred Williams	
/Patient/Doctor[\$6]	<Name xmlns='http://myapp.com'>Dr. Arnold</Name>	
/Patient/4	An empty string	This path is invalid. <Patient> does not have a fourth element.
/Patient/comment()[1]	Sample comment	For these paths, the square brackets are required, because without square brackets, these paths would be interpreted as element names.
/Patient/comment()[2]	Another comment	
/Patient/comment()[\$2]	Another comment	
/Patient/comment()[-]	Another comment	

6.4 Using the Full() Function

For a path that refers to an element (either a [basic path](#) or a path that uses [basic modifiers](#)), you can also obtain the opening and closing tags of the element. To do so, add `full()` to the end of the path.

You can use the `full()` function when you are setting a value. Within DTL, this is permitted only within a data transformation that uses the `append` action; see “[Assignment Actions for XML Virtual Documents](#),” later in this book.

Consider the following XML document:

```
<?xml version="1.0" ?>
<Patient xmlns='http://myapp.com'>
  <Name>Jack Brown</Name>
  <Address>
    <Street>233 Main St</Street>
  </Address>
</Patient>
```

The following table shows some example paths for this document:

Example Path	Current Path Value
/Patient/Name/full()	<Name xmlns='http://myapp.com'>Jack Brown</Name>
/Patient/Address/full()	<Address xmlns='http://myapp.com'><Street>233 Main St</Street></Address>
/Patient/Address/Street/full()	<Street xmlns='http://myapp.com'>233 Main St</Street>

For the root note, use of the **full()** function is implied. That is, the following two paths are equivalent:

```
/
/full()
```

Note: If you use **GetValueAt()**, you can also specify an additional format argument (*f*) that retrieves the full element. For details, see “[The pFormat Argument](#),” later in this book.

6.5 Getting or Setting the Value of an XML Attribute

To access the value of an attribute, you can use one of the following DOM-style paths. Here (and in the rest of this section), *element_reference* is a complete DOM-style path to an element.

Syntax	Refers to
<i>element_reference</i> /@ <i>attribute_name</i>	Value of the given attribute of the given element.
<i>element_reference</i> /@[<i>n</i>]	(For use only when retrieving values) Value of the <i>n</i> th attribute (in alphabetical order) of the given element.
<i>element_reference</i> /@[-]	Value of the last attribute of the given element.

You can omit the square brackets.

For example, consider the following XML document:

```
<?xml version="1.0" ?>
<Patient MRN='000111222' DL='123-45-6789' xmlns='http://myapp.com'>
  <Name>Liz Jones</Name>
</Patient>
```

The following table shows some example paths for this document:

Example Path	Current Path Value
/Patient/@MRN	000111222
/Patient/@[1]	000111222
/Patient/@2	123-45-6789

6.6 Using Path Modifiers to Insert or Append Nodes

To insert or append nodes, add the following path modifiers to the end of [basic paths](#). Use the path modifiers listed here only when you are setting a value.

Also see the [next section](#) for a couple of additional options.

[~n]

Inserts an instance of the item to which the basic path refers, right before the *n*th instance of that item, in the given context. Nothing is overwritten. See the following table for details.

Here and in the rest of this subsection, *n* is an integer.

Example Path	Behavior
/Patient/Episode[~5]	<p>Inserts a new <Episode> element within <Patient>, before the existing fifth <Episode> element.</p> <p>If <Patient> does not include five <Episode> elements, Ensemble performs <i>padding</i>; it creates empty <Episode> elements so that the inserted <Episode> is the fifth <Episode>. All the newly inserted elements are at the end of the <Patient> element.</p> <p>If the path refers to intermediate, nonexistent elements, Ensemble creates those.</p>
/Patient/element(Episode)[~5]	<p>Inserts an <Episode> element within <Patient>, before the existing fifth element.</p> <p>If <Patient> does not include five elements (of any type), this path is invalid. The element() function does not generate empty elements for padding.</p>
/Patient/[~5]	Not allowed, because there is no information about the kind of element to insert.
/Patient/element()[~5]	

For example, consider the following XML document:

```
<?xml version="1.0" ?>
<Patient xmlns='http://myapp.com'>
  <Name>Betty Hodgkins</Name>
  <FavoriteColors>
    <FavoriteColor>Purple</FavoriteColor>
  </FavoriteColors>
</Patient>
```

Also consider the following code from within a data transformation:

```
set status=target.SetValueAt("INSERTED COLOR","/Patient/FavoriteColors/FavoriteColor[~4]")
if 'status {do $system.Status.DisplayError(status) quit}
```

This line of code transforms the original document into the following:

```
<?xml version="1.0" ?>
<Patient>
  <Name>Betty Hodgkins</Name>
  <FavoriteColors>
    <FavoriteColor>Purple</FavoriteColor>
    <FavoriteColor/>
    <FavoriteColor/>
    <FavoriteColor>INSERTED COLOR</FavoriteColor>
  </FavoriteColors>
</Patient>
```

For another example, consider the following XML document:

```
<Patient xmlns='http://myapp.com'>
  <Name>Colin McMasters</Name>
  <Address>
    <Street>102 Windermere Lane</Street>
  </Address>
</Patient>
```

Also considering the following code from within a data transformation:

```
set status=target.SetValueAt("INSERTED ADDRESS","/Patient/Address/Street[~2]")
if 'status {do $system.Status.DisplayError(status) quit}
```

This line of code transforms the original document into the following:

```
<?xml version="1.0" ?>
<Patient>
  <Name>Colin McMasters</Name>
  <Address>
    <Street>102 Windermere Lane</Street>
    <Street>INSERTED ADDRESS</Street>
  </Address>
</Patient>
```

[~\$n]

Inserts an instance of the item to which the basic path refers, right before the *n*th node in the given parent. Nothing is overwritten. The path is invalid if the parent does not contain at least *n* nodes.

Example Path	Behavior
/Patient/Episode[~\$3]	Inserts a new <Episode> element within <Patient>, before the existing third node in that parent. The path is invalid if the parent does not have three nodes.
/Patient/element(Episode)[~\$3]	Not allowed. The element() function works only with element positions.
/Patient/[~3]	Not allowed, because there is no information about the kind of element to insert.
/Patient/element()[~3]	Not allowed for multiple reasons; see above items.

[~]

Appends an instance of the item to which the basic path refers, as the (new) last node of the given parent. Nothing is overwritten.

Example Path	Behavior
/Patient/Episode[~]	Appends a new <Episode> element within <Patient>, as the last node in that parent. If the path refers to intermediate, nonexistent elements, Ensemble creates those.
/Patient/element(Episode)[~]	Appends an <Episode> element within <Patient>, as the last node in that parent. If the path refers to intermediate, nonexistent elements, the path is invalid.
/Patient/[~]	Not allowed, because there is no information about the kind of element to append.
/Patient/element()[~]	

For example, the following shows part of a code element in a data transformation:

```

set status=target.SetValueAt("orange","/Patient/FavoriteColors/Color[~]")
if 'status {do $system.Status.DisplayError(status) quit}
set status=SetValueAt("pink","/Patient/FavoriteColors/Color[~]")
if 'status {do $system.Status.DisplayError(status) quit}

```

This adds two new <Color> children to the <FavoriteColors> element. If the <FavoriteColors> element does not exist, Ensemble creates it.

Also see “[Summary of Path Modifiers](#)”.

6.7 Using the element() Function

You can use the **element()** function when getting or setting values, as described in the following table:

Syntax	When Allowed	Behavior
<i>element_reference</i> /element()	When getting a value	Returns the first child element of the given element.
<i>element_reference</i> /element()[<i>n</i>]		Returns the <i>n</i> th child element of the given element.
<i>element_reference</i> /element()[-]		Returns the last child element of the given element.
<i>parent_element</i> /element(<i>element_name</i>)[~ <i>n</i>]	When setting a value	Inserts the specified element (given by the <i>element_name</i> argument) right before the <i>n</i> th child element of the given parent. This path is invalid if the given element does not have at least <i>n</i> child elements.
<i>parent_element</i> /element(<i>element_name</i>)[~]		Appends the specified element (given by the <i>element_name</i> argument) as the last node in the given parent.

6.8 Getting Positions of Elements

You can use the following syntaxes to get positions of elements.

Syntax	Returns
<i>element_reference</i> /position()	Element position of the given element within its parent.
<i>element_reference</i> /node-position()	Node position of the given element within its parent. For <i>node position</i> , Ensemble considers all kinds of nodes, not just elements.

6.9 Getting Counts of Elements

You can use the following syntaxes to get counts of elements.

Syntax	Returns
<ul style="list-style-type: none"> <code>element_reference/[*]</code> <code>element_reference/count()</code> 	Count of child elements within the given parent.
<ul style="list-style-type: none"> <code>parent/element_name[*]</code> <code>parent/element_name.count()</code> 	Count of elements <i>of the given name</i> , within the given parent. Notice that there is no slash after the name of the element (in contrast with the previous set of paths).
<ul style="list-style-type: none"> <code>element_reference/[\$*]</code> <code>element_reference/node-count()</code> 	Count of child nodes of the given element.
<ul style="list-style-type: none"> <code>element_reference/@[*]</code> <code>element_reference/@.count()</code> 	Count of attributes of the given element.

You can omit the square brackets in all cases except for `[*]`. Note that Ensemble also supports the **last()** function (equivalent to **count()**) and the **node-last()** function (equivalent to **node-count()**); you might prefer to use **last()** and **node-last()** if you are familiar with XPATH, which has a similar **last()** function.

6.10 Accessing Other Metadata

You can use the following functions to access other metadata of the XML virtual document. You can use these functions only at the *end* of a path.

Function	Returns
<code>/node-type()</code>	Type of the given node. This function returns one of the following values: <ul style="list-style-type: none"> <code>root</code> <code>element</code> <code>text</code> <code>comment</code> <code>instruction</code>
<code>/name()</code>	Full name of the given node. For example: <code>s01:Patient</code>
<code>/local-name()</code>	Local name of the given node. For example: <code>Patient</code>
<code>/prefix()</code>	Namespace prefix of the given node. For example: <code>s01</code>
<code>/namespace-uri()</code>	URI of the namespace to which the given node belongs. For example: <code>www.myapp.org</code>

Function	Returns
<code>/prefixes()</code>	All the namespace prefixes and their corresponding URIs, in the scope of the given element. This information is returned as a comma-separated list. Each list item consists of the namespace prefix, followed by an equal sign (=), followed by the URI. The default namespace URI is listed first without a prefix. For example: <code>=http://tempuri.org,s01=http://mysns.com</code>

6.11 Summary of Path Modifiers

The following table summarizes the path modifier for DOM-style paths:

Path modifier	Uses	Methods that can use paths that contain this modifier	Provides padding (as needed) when used with <code>SetValueAt()</code> ?
<code>[n]</code>	Getting or setting <i>n</i> th instance	<code>GetValueAt()</code> and <code>SetValueAt()</code>	Yes
<code>/[n]</code>	Getting <i>n</i> th child element	<code>GetValueAt()</code>	Not applicable
<code>[~n]</code>	Inserting <i>n</i> th instance	<code>SetValueAt()</code>	Yes
<code>[~]</code>	Appending instance	<code>SetValueAt()</code>	No
<code>[\$n]</code>	Getting or setting instance at <i>n</i> th node position	<code>GetValueAt()</code> and <code>SetValueAt()</code>	No
<code>[~\$n]</code>	Inserting instance at <i>n</i> th node position	<code>SetValueAt()</code>	No

6.12 Variations for Documents That Use Namespaces

If the document uses XML namespaces, for each element or attribute that is in a namespace, you must modify that section of the path to include a namespace prefix, followed by colon (:). A namespace prefix is one of the following:

- If you have loaded the corresponding XML schema, use a namespace token as described in “[XML Namespace Tokens](#),” earlier in this book. For example: use `$2:element_name` rather than `element_name`
- If you have not loaded the XML schema, use the namespace prefix exactly as it appears in the document. For example: `s01:Patient`
- Use the wildcard `*` to ignore the namespace. For example: `*:Patient`

Another option is to ignore all namespaces in the document. To do this, start the path with the wildcard `*:/` rather than `/`

For example: `*:/Patient/@MRN`

You cannot use any wildcards in a path when you are setting the value for that path.

Note: The output document of a DTL does not necessarily use the same namespace prefixes as the input document. The namespaces are the same, but the prefixes are generated. According to the XML standard, there is no significance to the choice of prefix.

6.13 Testing DOM-style Paths in the Terminal

It can be useful to test virtual document property paths in the Terminal before using them in business processes, data transformations, and so on, particularly when you are getting familiar with the syntax. To do so for DOM-style XML paths, do the following in the Terminal or in test code:

1. Create a string that contains the text of a suitable XML document.
2. Use the **ImportFromString()** method of `EnsLib.EDI.XML.Document` to create an instance of an XML virtual document from this string.
3. Use the **GetValueAt()** and **SetValueAt()** methods of this instance.

The following method demonstrates these steps:

```
ClassMethod TestDOMPath()  
{  
    set string="<Patient xmlns='http://myapp.com'>"  
    _"<Name>Jolene Bennett</Name>"  
    _"<Address><Street>899 Pandora Boulevard</Street></Address>"  
    _"</Patient>"  
    set target=##class(EnsLib.EDI.XML.Document).ImportFromString(string,.status)  
    if 'status {do $system.Status.DisplayError(status) quit}  
  
    set pathvalue=target.GetValueAt("/Patient/Name",,.status)  
    if 'status {do $system.Status.DisplayError(status) quit}  
    write pathvalue  
}
```

The following shows output from this method:

```
ENSDEMO>d ##class(Demo.CheckPaths).TestDOMPath()  
Jolene Bennett
```

For additional options for **GetValueAt()**, see “[The pFormat Argument](#),” later in this book.

7

Defining Data Transformations for XML Virtual Documents

This chapter discusses how to create data transformations (specifically DTL-based transformations) for XML virtual documents, for use in [rule sets](#). It discusses the following topics:

- [How to define a data transformation](#)
- [Details on the behavior of different actions](#)
- [Using code](#)
- [Example 1: copying most of the source document](#)
- [Example 2: using only a few parts of the source document](#)
- [Example 3: using code and SetValueAt\(\)](#)

7.1 Creating a Data Transformation

To create a data transformation for XML virtual documents:

1. Optionally load the applicable XML schema or schemas into Ensemble.
See “[Loading XML Schemas into Ensemble](#),” earlier in this book.
2. Use the DTL editor in the Management Portal or in Studio, as described in [Developing DTL Transformations](#).
3. Within the data transformation, use the following values:
 - For **Source Class** and **Target Class**, use **EnsLib.EDI.XML.Document**, the class with which Ensemble represents XML virtual documents.
 - For **Source Document Type**, optionally select the XML type expected in the message. Choose an XML type from one of the XML schemas you have loaded into Ensemble.
Leave this value blank if you do not have or do not want to use the schema.
 - For **Target Document Type**, optionally select a different XML type or remove the value.
Ensemble initializes **Target Document Type** with the value you select for **Source Document Type**, if any.

4. Create actions within the data transformation as usual, using the XML property paths described in the [previous chapter](#). There are two basic scenarios:
 - If you have loaded the schemas and have specified the source and target document types, the DTL editor displays each document structure as a tree. Then you can drag and drop to create the transformation. Ensemble creates actions that use schema-dependent paths. You can edit these to use DOM-style paths instead, if those are needed for some reason.
 - If you do not specify the document types, the document structures are not displayed as trees. In this case, it is necessary to add and edit the actions manually. You can use only DOM-style paths.

In either case, you can add code elements to support more complex processing.

After you save and compile the data transformation, it is available for use in a rule set; see the chapter “[Defining Rule Sets for XML Virtual Documents](#).”

7.2 Available Assignment Actions for XML Virtual Documents

For XML virtual documents, Ensemble supports the following assignment actions:

- **set** — Sets a value. If the type of the target element is "any", then the text can include XML markup. The XML markup must be well formed, but it will not be validated against any schema.
- **append** — Appends the new value to the target element, after any subnodes in that element.
- **clear** — Clears the text context of the target but retains the element and any children. Or, if the target is an attribute, the action clears its value but retains the attribute.
- **remove** — Removes the target element or attribute.

Note that **insert** is not supported.

7.3 Using Code

If you need to add code elements to support more complex processing, you directly invoke the **GetValueAt()** and **SetValueAt()** methods of the *source* and *target* variables. For `EnsLib.EDI.XML.Document`, these methods are as follows:

GetValueAt()

```
method GetValueAt(pPropertyPath As %String,  
                 pFormat As %String,  
                 Output pStatus As %Status) as %String
```

Where:

- *pPropertyPath* is an XML property path, as described [earlier in this book](#).
- *pFormat* is a set of flags that control the format of the returned string. See the [following subsection](#).
- *pStatus* is a status that indicates success or failure.

This method returns the current value at the given property path, or returns an empty string if the path is not valid.

SetValueAt()

```
method SetValueAt(pValue As %String,
                 pPropertyPath As %String,
                 pAction As %String = "set",
                 pKey As %String = "") as %Status
```

Where:

- *pValue* is a suitable value for the given XML property path.
- *pPropertyPath* is an XML property path, as described in the [previous chapter](#).
- *pAction* is either "set", "append", "clear", or "remove". For details, see the [previous section](#).
- *pKey* is not used for XML virtual documents.

This method evaluates the given property path, and (if the path is valid), uses *pValue* and *pAction* to modify the value at that path.

Important: It is useful to check the status values returned by these methods. The status contains specific information when you specify invalid paths or attempt actions that are not permitted. This information is particularly useful when you are debugging and can save you time.

7.3.1 The pFormat Argument

The *pFormat* argument for **GetValueAt()** is an optional string that controls the format of the returned string. This string can contain any suitable combination of the characters in the following table:

General Description	Character to Include in Format Setting	Specific Behavior
Line feeds and carriage returns	w	Adds a Windows-style carriage return and line feed combination after every text-free element.
	r	Uses the stored line feeds and carriage returns. This option takes precedence over the options w and n.
	n	Includes a new line (line feed) after every text-free element. In contrast to w, this option does not add a carriage return.
Indentation. Note that these options are used only if the output includes new lines.	i	Indents each new line with four spaces.
	Any integer from 1 to 9	Indents each new line with this number of spaces. This option takes precedence over the previous indentation option.
	t	Indents each new line with a tab. This option takes precedence over both of the previous indentation options.
	s	Uses the stored indentation whitespace. This option takes precedence over the previous indentation options.
Handling attributes	a	Alphabetizes the attributes in an element.
	q	Uses double quotes (rather than single quotes) to set off attribute values if possible.
Handling namespaces	p	Suppresses output of namespace prefixes.
	x	Suppresses output of namespace declarations.

General Description	Character to Include in Format Setting	Specific Behavior
Handling empty elements	e	Generates output for each empty element with an open tag and close tag pair. If this option is not set, empty elements are output as a single empty tag.
Other	c	Canonical output. This option takes precedence over the options <code>e i n t w</code> .
	f	Generates the full element (including both the starting and ending tags), not just the contents within the element.
	l	Includes information about the location of the schema file that was loaded into Ensemble. This option takes effect only if you use <code>f</code> .
	o	Includes any XML entities as is, rather than performing XML escaping for those entities.
	C(e)	Generates an XML header line that declares the given character encoding; <code>e</code> is the non-quoted name of a character encoding such as <code>UTF-8</code> . If <code>e</code> is empty, use the encoding defined by the adapter. If <code>e</code> begins with <code>!</code> then force the encoding of the output stream. Note that this will be applied automatically for file operations configured with a non-UTF-8 character set.

As noted above, the *pFormat* argument can equal a combination of these items. For example, if you use the value `C(UTF-8)q`, the outbound document is in the UTF-8 character set and attributes are set off with double quotes. For another example, if you use the value `C(UTF-16)a`, the outbound document is in the UTF-16 character set and attributes are alphabetized.

Note: This information also applies to the [Format](#) setting of an XML business operation.

7.4 Example 1: Copying Most of the Source Document

To easily define a data transformation that copies most of a source document, do the following in the Data Transformation Builder:

- On the **Transform** tab, select **copy** from the **Create** drop-down list.
Then, by default, the new document will be a copy of the original document.
- Define actions that partly or fully remove selected elements or attributes. To define such an action:
 1. In **Add Action**, click **clear** or **remove**.
 2. Double-click the target property that you want to clear or remove.
 3. Enter any value into **Value**; this field is required but is ignored in this case.

The following shows an example that uses schema-dependent paths:

```

Class Demo02.MyDTL Extends Ens.DataTransformDTL
{
XData DTL [ XMLNamespace = "http://www.intersystems.com/dtl" ]
{
<transform sourceClass='EnsLib.EDI.XML.Document' targetClass='EnsLib.EDI.XML.Document'
sourceDocType='Demo02:Patient' targetDocType='Demo02:Patient' create='copy' language='objectscript' >
<assign value="this value is ignored" property='target.{WorkAddress}' action='remove' />
<assign value="this value is ignored" property='target.{HomeAddress}' action='remove' />
</transform>
}
Parameter REPORTERRORS = 1;
}

```

This data transformation copies the source document to the target and then removes the <WorkAddress> and <HomeAddress> elements from the target.

The following shows an equivalent example that uses DOM-style property paths:

```

Class Demo02A.MyDTL Extends Ens.DataTransformDTL
{
XData DTL [ XMLNamespace = "http://www.intersystems.com/dtl" ]
{
<transform sourceClass='EnsLib.EDI.XML.Document' targetClass='EnsLib.EDI.XML.Document'
create='copy' language='objectscript' >
<assign value="this value is ignored" property='target.{/Patient/WorkAddress}' action='remove' />
<assign value="this value is ignored" property='target.{/Patient/HomeAddress}' action='remove' />
</transform>
}
Parameter REPORTERRORS = 1;
}

```

Notice that in this case, the data transformation does not specify the document types because they are unnecessary here.

7.5 Example 2: Using Only a Few Parts of the Source Document

To easily define a data transformation that uses only a few parts of a source document, do the following in the Data Transformation Builder:

- On the **Transform** tab, select **new** from the **Create** drop-down list.
Then, by default, the new document will be empty.
- Define actions that copy selected elements or attributes. To define such an action, drag and drop from the source document area to the target document area. Each action that you add this way is a **set** action.

The following shows an example that uses schema-dependent paths:

```

Class Demo05.MyDTL Extends Ens.DataTransformDTL
{
XData DTL [ XMLNamespace = "http://www.intersystems.com/dtl" ]
{
<transform sourceClass='EnsLib.EDI.XML.Document' targetClass='EnsLib.EDI.XML.Document'
sourceDocType='Demo05:Patient' targetDocType='Demo05:Patient' create='new' language='objectscript' >
<assign value='source.{MRN}' property='target.{MRN}' action='set' />
<assign value='source.{PrimaryCarePhysician}' property='target.{PrimaryCarePhysician}' action='set' />
</transform>
}
Parameter REPORTERRORS = 1;
}

```

This data transformation copies only the MRN and PrimaryCarePhysician properties from the source to the target.

The following shows an equivalent example that uses DOM-style property paths:

```
Class Demo05A.MyDTL Extends Ens.DataTransformDTL
{
  XData DTL [ XMLNamespace = "http://www.intersystems.com/dtl" ]
  {
    <transform sourceClass='EnsLib.EDI.XML.Document' targetClass='EnsLib.EDI.XML.Document'
    create='new' language='objectscript' >
    <assign value='source.{/Patient/MRN}' property='target.{/Patient/MRN}' action='set' />
    <assign value='source.{/Patient/PrimaryCarePhysician}'
    property='target.{/Patient/PrimaryCarePhysician}' action='set' />
    </transform>
  }

  Parameter REPORTERRORS = 1;
}
```

7.6 Example 3: Using Code and SetValueAt()

The following example uses the **code** action type and uses a DOM-style path. It adds an attribute and an XML comment to the root element:

```
Class Demo06.MyDTL Extends Ens.DataTransformDTL
{
  XData DTL [ XMLNamespace = "http://www.intersystems.com/dtl" ]
  {
    <transform sourceClass='EnsLib.EDI.XML.Document' targetClass='EnsLib.EDI.XML.Document'
    create='copy' language='objectscript' >
    <code>
    <![CDATA[
    //this part adds an attribute to the document
    set path="/1/@NewAttribute"
    set status=target.SetValueAt("New attribute value",path)
    if 'status {do ##class(MyApp.Utils).Trace("Demo06.MyDTL","Error setting path: ",path)}

    //this part adds a comment to the document
    set path="/1/comment()"
    set status=target.SetValueAt("This is an XML comment",path)
    if 'status {do ##class(MyApp.Utils).Trace("Demo06.MyDTL","Error setting path: ",path)}
    ]]>
    </code>
    </transform>
  }

  Parameter REPORTERRORS = 1;
}
```

If the **SetValueAt()** method returns an error, this transformation uses a utility method to record the details.

8

Defining Rule Sets for XML Virtual Documents

This chapter discusses how to create rule sets for XML virtual documents, for use in [business processes](#). It discusses the following topics:

- [How to create a rule set](#)
- [Example](#)

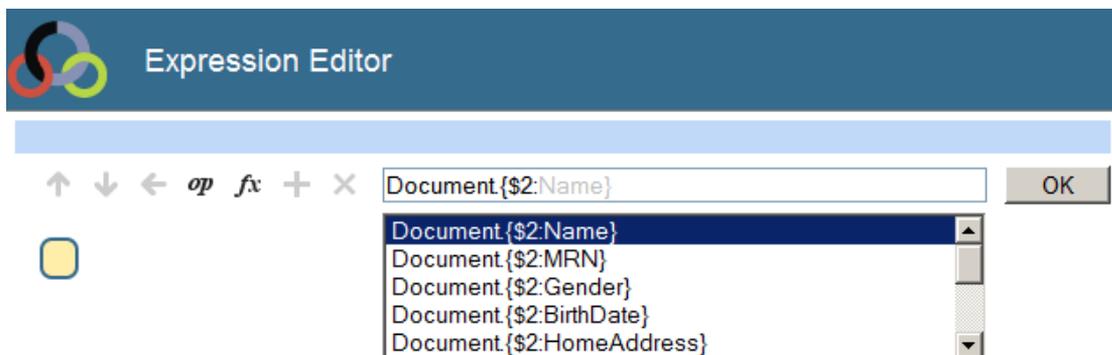
To configure a business process to use a rule set, specify its **Business Rule Name** setting; see “[Adding a Business Process to Handle XML Virtual Documents](#),” earlier in this book.

8.1 Creating a Rule Set

To create a rule set for XML virtual documents:

1. Optionally load the applicable XML schema or schemas into Ensemble.
See “[Loading XML Schemas into Ensemble](#)” earlier in this book.
2. Use the Rule Set editor in the Management Portal or in Studio, as described in [Developing Business Rules](#).
3. For the rule set basic definition, use **Virtual Document Message Routing Rule** for **Type**.
This choice sets **Context Class** to `EnsLib.MsgRouter.VDocRoutingEngine`. It also sets **Rule Assist Class** to `EnsLib.MsgRouter.VDocRuleAssist`.
4. For any rule constraint in the rule set, use the following values:
 - For **Message Class**, use **EnsLib.EDI.XML.Document**, the class with which Ensemble represents XML virtual documents.
 - For **Schema Category**, optionally select an XML schema that you have previously loaded into Ensemble.
Leave this value blank if you do not have or do not want to use the schema.
 - For **Document Name**, optionally select a document type defined in that schema.
Leave this value blank if you have not specified **Schema Category**.
5. Create rules as usual, using the XML property paths described [earlier in this book](#). There are two basic scenarios:

- If you have loaded the schema and have specified the target document type, the Expression Editor provides assistance when you start typing Document.



Notice that these property paths are schema-dependent paths, although you could edit them to be DOM-style paths instead, if those are needed for some reason.

- If you have not loaded the schema and specified the document type, you must type the path manually. You can use either schema-dependent paths or DOM-style paths.

After you save and compile the rule set, it is available for use in a business process.

8.2 Example

The following shows the class definition for a simple rule set. This rule set has one rule that uses a DOM-style path to check the <MRN> element of the <Patient> document. Depending on the returned value, the rule routes the message to either FileOut1 or FileOut2. Notice that in this case, the rule constraint does not refer to the XML schema or type.

```
Class Demo09.MyRules Extends Ens.Rule.Definition
{
Parameter RuleAssistClass = "EnsLib.MsgRouter.VDocRuleAssist";

XData RuleDefinition [ XMLNamespace = "http://www.intersystems.com/rule" ]
{
<ruleDefinition alias="" context="EnsLib.MsgRouter.VDocRoutingEngine">
<ruleSet name="" effectiveBegin="" effectiveEnd="">
<rule name="CheckMRN" disabled="false">
<constraint name="msgClass" value="EnsLib.EDI.XML.Document"></constraint>
<when condition="Document.{/$2:Patient/$2:MRN}=&quot;123456789&quot;">
<send transform="" target="FileOut1"></send>
<return></return>
</when>
<when condition="Document.{/$2:Patient/$2:MRN}!=&quot;123456789&quot;">
<send transform="" target="FileOut2"></send>
<return></return>
</when>
</rule>
</ruleSet>
</ruleDefinition>
}
}
```

9

Defining Search Tables for XML Virtual Documents

This chapter describes briefly how to define search tables for XML virtual documents. It discusses the following topics:

- [Introduction](#)
- [Example](#)

To configure a business service or business operation to use a search table class, specify the **Search Table Class** setting of that business host. See “[Configuration Steps](#),” earlier in this book.

9.1 Introduction

The XML search table class, `EnLib.EDI.XML.SearchTable` indexes only the name of the root element of the XML documents.

If you need more items to search, you can create a subclass. For details, see “[Defining a Search Table Class](#)” in *Ensemble Virtual Documents*.

Note: Ensemble does not retroactively index messages that were received before you added the search table class.

9.2 Example

The following shows an example:

```
XData SearchSpec [ XMLNamespace = "http://www.intersystems.com/EnsSearchTable" ]
{
<Items>
  <Item DocType="MyApp:Patient" PropName="Gender" >{*:/Patient/Gender}</Item>
  <Item DocType="MyApp:Patient" PropName="MRN" >{*:/Patient/@MRN}</Item>
</Items>
}
```


10

XML-Enabled Objects Compared to XML Virtual Documents

When developing a production with XML messages, you can use either of the following structures to contain the XML document:

- XML Virtual Documents
- XML-enabled objects

If you need to access a small number of the elements in the input XML document, you can use XML Virtual Documents, but if you need to access most of the elements in the input XML document, you should choose XML-enabled objects because they are more efficient, especially when an object with many elements is processed by a transformation. The following business services and operations are provided to handle XML-enabled objects:

- `EnsLib.XML.Object.Service.FileService`
- `EnsLib.XML.Object.Service.FTPService`
- `EnsLib.XML.Object.Operation.FileOperation`
- `EnsLib.XML.Object.Operation.FTPOperation`

The XML-enabled object business services read a file containing an XML document and convert it to one or more objects. You specify a property that defines the XML element to convert to objects. If the XML root document contains a single element, then the service converts it to one object, but if the XML root document contains a series of these elements, then the service converts them to separate objects.

To use the XML-enabled object services, you do the following:

1. Define a class that matches the structure of the input XML documents that you are processing. The class can either match the entire XML document or a repeating element within the root XML document. You can use the XML Schema Wizard to define this class. You can optionally define a `NAMESPACE` parameter for this class. This parameter specifies the XML namespace.
2. Specify the classname in the business service **Class Name** field.
3. Optionally, specify the element name in the **Element Name** field. If you specify this field, the service looks for one or more XML elements with this name within the root XML object. Each occurrence of this element is converted to an instance of the specified class. If you do not specify this field, the service matches the root document to the specified class.

4. Optionally, specify the **Format** parameter and optionally select **Ignore Null**. The **Format** parameter can have a value of "literal", "encoded", or "encoded12". These parameters specify the corresponding parameters for the %XML.Adaptor class.

The XML-enabled object business operations convert an object to an XML document and write the document to a file. In addition to specifying information about the XML class and element, you can specify properties that are used when the operation invokes the %XML.Writer class.

Specify the following properties in the operations:

- **Root Element Name** — If you specify this property, it is used as the root element name. If you omit this element, the operation uses the input element name.
- **Namespace** — Specifies the XML namespace except if the class defines a NAMESPACE property. In that case, the operation always uses the XML namespace defined in the class.
- **Expected Class Name** — Class name of the XML-enabled object. If the expected name does not match the actual name, the %XML.Writer adds an xsi:type attribute to the XML element.
- **Indentation Type** — Specifies the corresponding property for %XML.Writer. Indentation Type specifies if indentation of the XML output should take place and what type of indentation.
- **Indentation Depth** — Specifies the corresponding property for %XML.Writer. Indentation Depth specifies the number of indentation characters to be used for indentation. The default for "tab" is 1. The default for "space" is 4.
- **Charset** — Specifies the corresponding property for %XML.Writer. Charset is the charset to use for encoding the XML output. The default depends upon the output destination. "UTF-8" is the default for output to files and binary streams. On a Unicode system, "UTF-16" is the default for output to character streams and strings. On an 8-bit system, the default charset for the locale is the default charset for output to character streams and strings.
- **No XML Declaration** — Specifies the corresponding property for %XML.Writer. If No XML Declaration is 1 (true), the %XML.Writer does not write the XML declaration. The default is for the %XML.Writer to write the XML declaration unless Charset is not specified and the output is directed to a string or character stream in which case it does not write an XML declaration.
- **Runtime Ignore Null** — Specifies the corresponding property for %XML.Writer.
- **Element Qualified** — Specifies the corresponding property for %XML.Writer.
- **Attribute Qualified** — Specifies the corresponding property for %XML.Writer.
- **Default Namespace** — Specifies the corresponding property for %XML.Writer.
- **Suppress xmlns** — Specifies the corresponding property for %XML.Writer.
- **Format** — Specifies the corresponding property for %XML.Writer.
- **References inline** — Specifies the corresponding property for %XML.Writer.

Reference for Settings

This section provides the following reference information:

- [Settings for XML Business Services](#)
- [Settings for XML Business Operations](#)

For information on settings for the routing process (EnsLib.MsgRouter.VDocRoutingEngine), see “[Settings of a Virtual Document Routing Process](#)” in *Ensemble Virtual Documents*.

Settings for XML Business Services

Provides reference information for settings of XML virtual document business services.

Summary

XML virtual document business services provide the following settings:

Group	Settings	See
Basic Settings	Target Config Names , Doc Schema Category	“ Settings for Business Services ” in <i>Ensemble Virtual Documents</i>
Additional Settings	Search Table Class , Validation	<i>section in this topic</i>
	Reply Target Config Names	<i>section in this topic</i>

The remaining settings are either common to all business services or are determined by the file adapter. For information, see:

- “[Settings for All Business Services](#)” in *Configuring Ensemble Productions*
- “[Settings for the File Inbound Adapter](#)” in *Using File Adapters with Ensemble*

Reply Target Config Names

(File and FTP only) Comma-separated list of configuration items within the production to which the business service should relay any XML virtual documents *reply* messages. Usually the list contains one item, but it can be longer. The list can include both business processes and business operations.

Compare to [Target Config Names](#).

Validation

By default, validation of XML virtual documents is limited to testing whether the DocType is defined. To provide additional validation for XML virtual documents, you should subclass the `EnLib.MsgRouter.VDocRoutingEngine` class and override the **OnValidate** method, adding custom code to validate the XML document.

If you are validating the document, return a nonzero value, which suppresses any default validation. If the document passes validation, return 1 (\$\$OK) in `pStatus` to indicate success. If the document fails validation, return an error code in `pStatus`.

Settings for XML Business Operations

Provides reference information for settings of XML virtual document business operations.

Summary

XML virtual document business operations provide the following settings:

Group	Settings	
Basic Settings	Format	<i>section in this topic</i>
Additional Settings	Search Table Class	<i>“Settings for Business Operations” in Ensemble Virtual Documents</i>

The remaining settings are either common to all business operations or are determined by the file adapter. For information, see:

- “[Settings for All Business Operations](#)” in *Configuring Ensemble Productions*
- “[Settings for the File Outbound Adapter](#)” in *Using File Adapters with Ensemble*

Format

Specifies how to form the outbound document. You can leave this empty, in which case defaults are used. Or you can specify a string that contains a suitable combination of the characters listed in “[The pFormat Argument](#),” earlier in this book.

For example, if you use the value `C(UTF-8)q`, the outbound document is in the UTF-8 character set and attributes are set off with double quotes. For another example, if you use the value `C(UTF-16)a`, the outbound document is in the UTF-16 character set and attributes are alphabetized.

