Ensemble Virtual Documents

Version 2017.2
2020-06-26

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Ensemble Virtual Documents
Ensemble Version 2017.2 2020-06-26

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

This book explains how the concept of virtual documents allows Ensemble to provide efficient support for Electronic Data Interchange (EDI) formats and for XML documents. This book provides the common information that applies to all Ensemble virtual document formats.

This book contains the following sections:

- Virtual Documents
- Schema Definitions
- Virtual Property Paths
- Using Virtual Documents in a Production
- Defining Search Tables
- Controlling Message Validation
- Creating Custom Schema Categories
- Portal Tools
- Syntax Guide for Virtual Property Paths
- Common Settings

For a detailed outline, see the table of contents.

The following books provide related information:

- *Developing Ensemble Productions* describes specific Ensemble development practices in detail.
- *Ensemble Best Practices* describes best practices for organizing and developing Ensemble productions.
- *Ensemble ASTM Development Guide* describes how to work with ASTM documents as Ensemble virtual documents.
- *Ensemble EDIFACT Development Guide* describes how to work with EDIFACT documents as Ensemble virtual documents.
- *Ensemble HL7 Version 2 Development Guide* describes how to work with HL7 version 2 messages as Ensemble virtual documents.
- *Ensemble X12 Development Guide* describes how to work with X12 documents as Ensemble virtual documents.
- *Ensemble XML Virtual Document Development Guide* describes how to work with XML documents as Ensemble virtual documents.

For general information, see *Using InterSystems Documentation*.

1

Virtual Documents

This chapter explains what virtual documents are, why there are useful, and how they are different from standard messages. It also briefly introduces tools that Ensemble provides so that virtual documents can be used in all the same ways as standard messages. This chapter contains the following sections:

1.1 Introduction

A *virtual document* is a kind of message that Ensemble parses only partially. To understand the purpose of virtual documents, it is useful to examine Ensemble messages a little more closely. Every Ensemble message consists of two parts:

- The *message header* contains the data needed to route the message within Ensemble. The message header is always the same type of object. This is a persistent object, meaning that it is stored within a table in the Ensemble database.
- The *message body* contains the message data. For standard messages, the message body is a persistent object. For virtual documents, the message body is implemented in a different way, as explained below.

An object represents each piece of data as a separate property. This is convenient in that any value in the object is easy to access. When writing code, you simply reference a class property by name to get its value.

For EDI (Electronic Data Interchange) formats, this approach becomes unwieldy and unnecessary. The approach is unwieldy because a large number of properties (possibly hundreds) would be required, and the process of creating an instance of the object can be slow. The standard approach is unnecessary because many applications use only a small number of the fields actually available in the document.

To address these issues, Ensemble provides an alternative type of message body called a *virtual document*. A virtual document allows you to send raw document content as a body of an Ensemble message, without creating objects to hold the contents of the document as a formal set of properties. The data in the virtual document is stored directly in an internal-use global, for greater processing speed.

The virtual document approach is also useful for XML documents (which can also be handled as standard messages).

1.2 Kinds of Virtual Documents

Ensemble can handle the following kinds of documents as virtual documents:

Kind of Document	See
ASTM documents	*Ensemble ASTM Development Guide*
EDIFACT documents	*Ensemble EDIFACT Development Guide*
HL7 version 2 messages	*Ensemble HL7 Version 2 Development Guide*
X12 documents	*Ensemble X12 Development Guide*
XML documents	*Ensemble XML Virtual Document Development Guide*

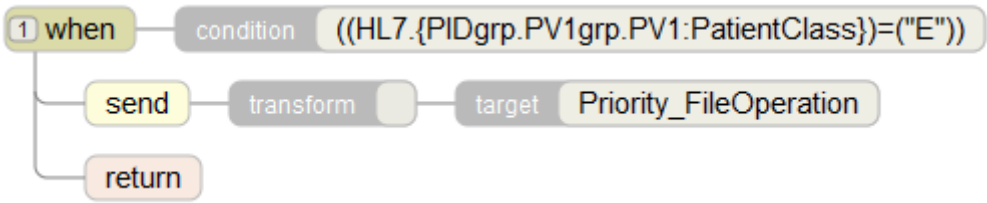
You can also handle XML documents as standard messages. To do so, you can generate classes from the corresponding XML schema. For information, see *Using Caché XML Tools*.

Other books in the Interoperability set do not describe virtual documents.

1.3 Access to Contents of Virtual Documents

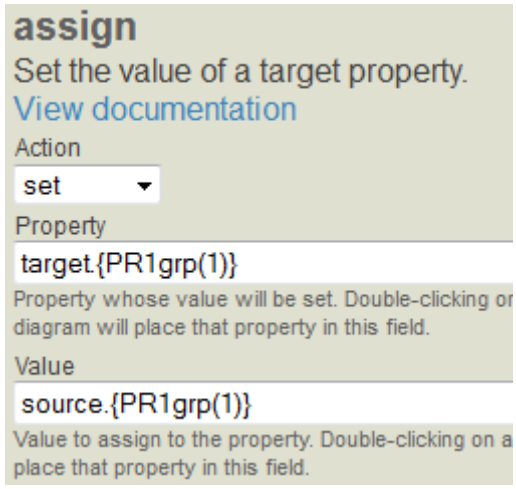
To work with data in a virtual document, you must be able to identify a specific data item within it. The data item is called a *virtual property*. An *virtual property path* is the syntax that Ensemble uses to specify the location of a virtual property. You can use virtual property paths in the following locations, among others:

- Business rules. The following shows an example:



  The curly braces enclose a virtual property path.

- Data transformations. The following shows an example:



For both **Property** and **Value**, the curly braces enclose a virtual property path.

As a consequence, your production can work with a virtual document in much the same way that it works with standard messages.

For details, see the chapter "Virtual Property Paths," later in this book.

1.4 Support for Filter and Search

For standard message bodies, each property of the message is directly searchable in the Management Portal. That is, the user can use the property for searching or filtering without having to know its path.

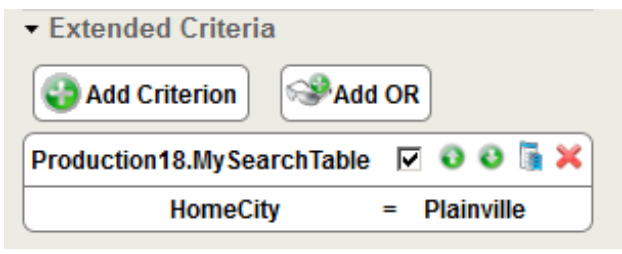
By default, with the exception of the message identifier, none of the data in an virtual document is directly searchable in the Management Portal. That is, a user must know the property path for a data item in order to use that item for searching or filtering.

To assist the user, you can use the following mechanism to make virtual properties directly searchable:

- You define a *search table class*; this class uses virtual property paths to define the searchable properties.
- You configure the applicable business hosts to use the search table class.

  As the business host receives messages, Ensemble indexes these properties as if they were properties in a standard message body.

Users can then use these properties directly without having to know the property paths that they use. For example:



For details on defining search tables, see "Defining Search Tables," later in this book.

2

Schema Definitions

This chapter introduces Ensemble schema definitions, which Ensemble uses to validate virtual documents (and to access data in them, as discussed in the chapter "Virtual Property Paths"). It contains the following sections:

- Introduction
- Schema Categories
- Document Structures
- Document Types (DocType)
- Tools

2.1 Introduction to Schema Definitions

Because a virtual document is not represented as an object (with a corresponding class definition), Ensemble requires additional tools to parse and validate the virtual document. These tools start with Ensemble schema definitions.

An Ensemble *schema definition* is a set of descriptions that represent a specific EDI standard or an XML schema. A schema definition is an Ensemble concept, not to be confused with other concepts such as a database schema or XML schema. It is, however, *based on* the corresponding EDI or XML schema.

Schema definitions are specific to a given Ensemble namespace and are stored in the Ensemble database for that namespace. Ensemble provides schema definitions for HL7 Version 2. In other cases, however, you must import the corresponding schema.

2.2 Schema Categories

Each schema definition provides Ensemble with a complete view of an EDI standard (or of an XML schema). There is one schema definition for HL7 Version 2, one for X12, one for ASTM, and so on. In practical terms, a schema definition may contain only a subset of the standard in question; this depends on how the corresponding EDI schema was imported into Ensemble.

In an Ensemble schema definition, a *schema category* is a grouping convention. Each schema definition contains one or more schema categories. For example, for HL7 Version 2, each schema category corresponds to a specific version of HL7 version 2 (2.1, 2.2, 2.3, and so on).

2.3 Document Structures

In an Ensemble schema definition, each schema category contains one or more *document structures*. Each document structure describes one type of document. For example, for HL7 Version 2, ADT_A01 is a document structure — if you are not familiar with HL7, just note that ADT_A01 is a kind of HL7 message.

Depending on the EDI standard, Ensemble uses different approaches to organize a schema definition into schema categories and document structures. For example, for HL7 Version 2, each schema category may contain many different document structures. In contrast, for X12, an Ensemble schema definition has a flatter organization, commonly with only one document structure per schema category:

2.4 Document Types (DocType)

Each Ensemble virtual document has a *document type*, often simply called *DocType* (which is the name of the property that stores this information). This corresponds to a specific part of a schema definition, the part that describes the expected structure of and values in this virtual document. The DocType enables Ensemble to validate and parse that virtual document.

The DocType for a virtual document is identified by a combination of the schema category and document structure. Specifically, the syntax for the DocType property is as follows:

```
category:structure
```

Where:

- *category* is the name of a schema category.
- *structure* is the name of a document structure within the referenced category. This piece is always required, even if the schema category has only one document structure.

For example, the following shows the DocType property of an ADT_A01 message for HL7 Version 2.5:

```
2.5:ADT_A01
```

2.5 Tools

The Management Portal provides the Schema Structures page, where you can import EDI or XML schemas (thus creating Ensemble schema definitions), export schemas, and browse schema definitions. See “Portal Tools,” later in this book.

3

Virtual Property Paths

This chapter introduces virtual property paths, which Ensemble uses to access data within virtual documents. It contains the following sections:

3.1 Introduction

To work with a virtual document, you must be able to identify a specific data item within it. The data item is called a *virtual property*. An *virtual property path* is the syntax that Ensemble uses to specify the location of a virtual property.

Except for XML virtual documents, you can use virtual property paths only if you have loaded the applicable EDI schema into Ensemble. Once the schema is loaded into Ensemble, Ensemble has all the information necessary to parse the corresponding documents as intended by the EDI schema.

3.2 Parsing EDI Documents: Segments and Fields

For ASTM, EDIFACT, HL7 version 2, and X12 documents, the raw data stream is divided into *segments*, which are further subdivided into *fields*. (The details are different for XML documents; see *Ensemble XML Virtual Document Development Guide*.)

The following example shows the raw contents of a partial HL7 message. The right-hand portion of the text is truncated for reasons of space:

```
MSH:;~\&:ST01C:A:HNS:A:20041209100007::ADT;A08:26070901:P:2.2:26070901::AL:::::::
PID:1::000616898;;;A;MR~00531098;;;;PI::LaRocca;Yan::19980202:F::4:924 Maple Blv
PD1::::36904;Malynko;Brendan;(SACKETT);;;PAC;DOC:Press;Chris:::::::::
NK1:1:Taylor;Chelsea:M;MOTHER:7702 Oak Street;;Reston;NV;93076;USA:854-495-4757:
PV1:1:O:"":3:::36904;Tsatsulin;Patrick;(SACKETT);;;PAC;DOC:36904;Zampitello;Emma
PV2:::::::::::::::::::::N:::::::::::::::::::::::::::N
AL1:"":"":"":"":"":""
DG1:1:FF:"":UIT::A:::::::::0:
DG1:2:I9:599.0:URIN TRACT INFECTION NOS::AM::::::::::
DG1:3:I9:599.0:URIN TRACT INFECTION NOS::F:::::::::1:
DG1:4:FF:"":UIT::W:::::::::1:
DG1:5:FF:::F:"":::"";"";"";"";"";""::0.00:::
```

A *terminator character* (often a carriage return) marks the end of the segment. In this case, each line is a segment.

Within a segment, *separator characters* mark the boundaries between fields. In this case, the colon is the separator character between fields and the semicolon is the separator between subfields.

A field can contain sub-fields and further subdivisions. These are delimited by other separator characters.

The details of the terminator character and separator characters depend upon the EDI format.

- For details on HL7 Version 2, see Separators. The default HL7.2 separators are:

  ```
  | ^ ~ \ &
  ```

  The fourth character (by default \) is used to escape characters rather than separate fields.

- For details on X12, see Separators. The default X12 separators are:

  ```
  * : \a ~ \r \n
  ```

- For details on EDIFACT see Separators. The default EDIFACT separators are:

  ```
  : + ? ' \r \n
  ```

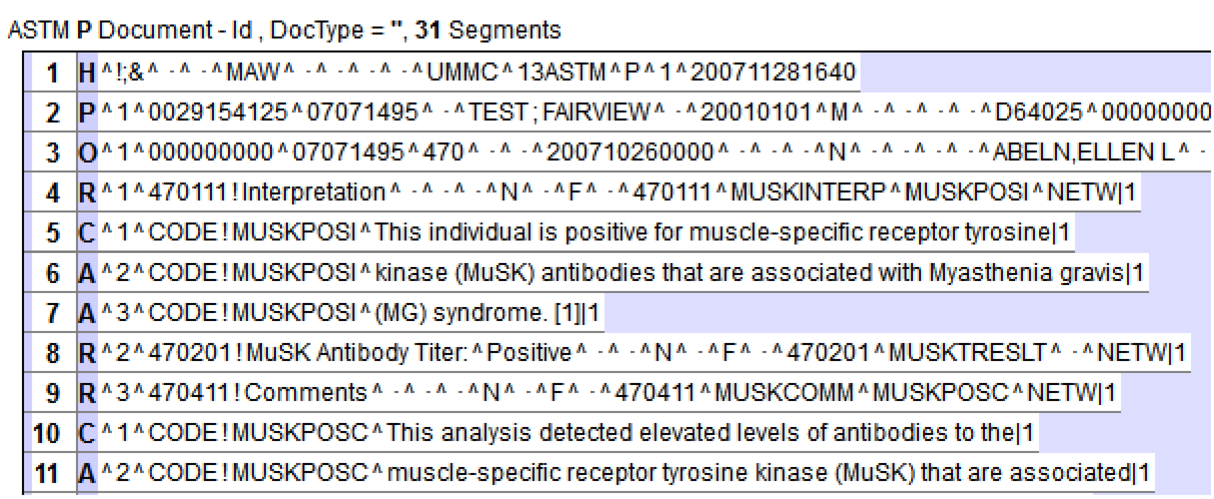
- For details on ASTM see Separators. The default ASTM separators are:

  ```
  | \ ^ &
  ```

  The fourth character (by default &) is used to escape characters rather than separate fields.

3.2.1 ASTM Example

The following figure shows how Ensemble parses an ASTM document. It shows the document as displayed in the Document Viewer page, described in “Portal Tools,” later in this book.



3.3 Segment Structures

Generally, an EDI standard defines a large number of possible segment structures to use as building blocks. Each document structure can contain only specific segments. Different document structures may also combine segments in different sequences or quantities. For example, for HL7 Version 2.3:

- An ADT_A01 message contains the following segments: MSN, EVN, PID, PD1, NK1, PV1, PV2, DB1, OBX, AL1, DG1, DRG, PR1, ROL, GT1, IN1, IN2, IN3, ACC, UB1, UB2
- In contrast, an ADT_A02 message contains only the following segments: MSN, EVN, PID, PD1, PV1, PV2, DB1, OBX

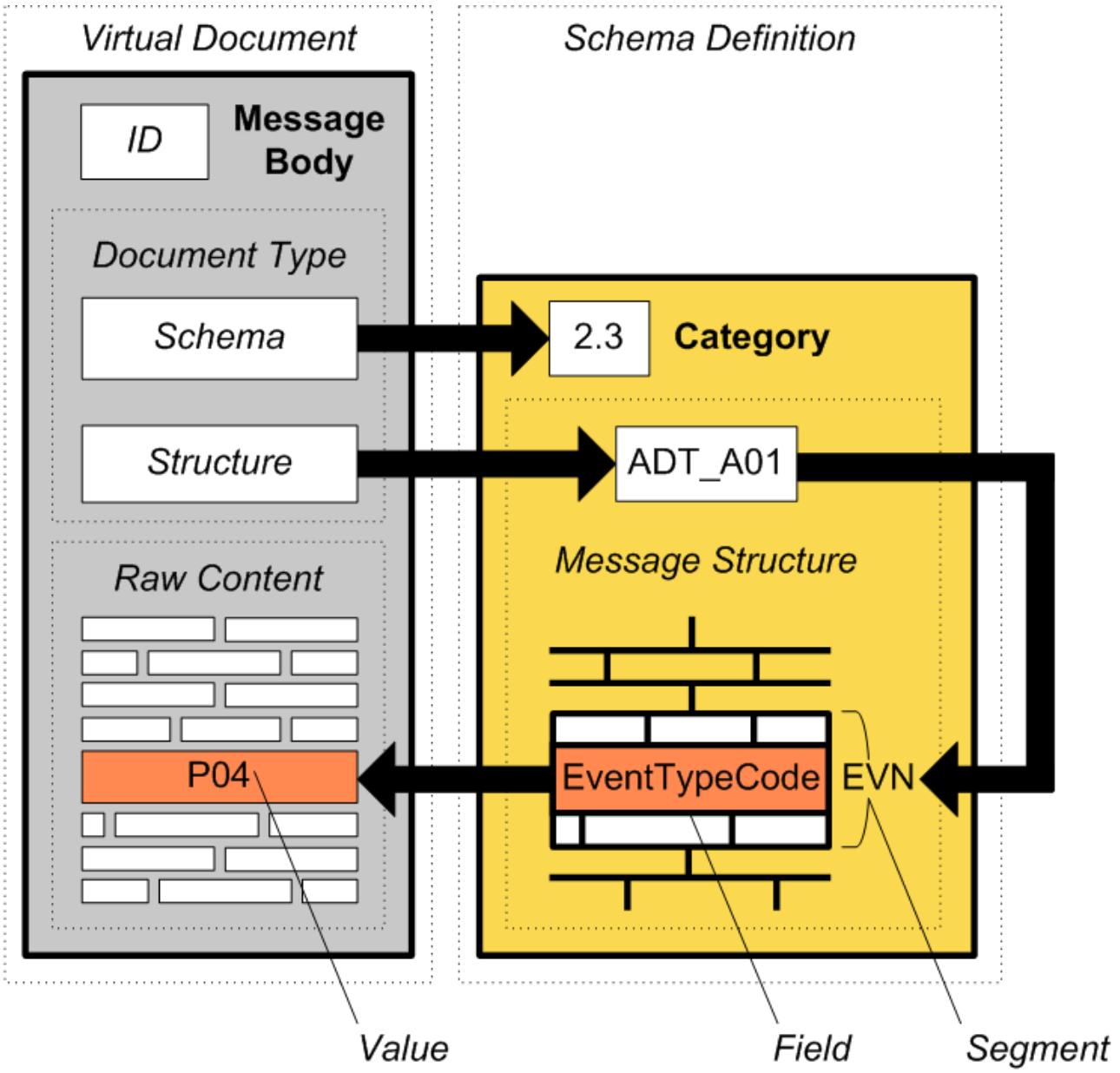
(The details are different for XML documents; see *Ensemble XML Virtual Document Development Guide*.)

3.4 Determining Virtual Property Paths

Conceptually, a *virtual property path* includes all the following units:

- *category*:*structure* — the DocType
- *segment*:*field* — the path to a data value within a virtual document of that DocType

The following figure illustrates this convention.



Generally the *segment* portion of the path identifies the target segment within a hierarchical document structure containing groups and repeating blocks of segments. For example, an NTE segment in a 2.3:ORM_O01 message might be identified as:

```
ORCgrp(1).OBRuniongrp.OBXgrp(3).NTE(1)
```

Similarly, the *field* portion of the path identifies a target field within a hierarchical structure of fields, subfields, and repeating groups within the target segment. As it happens, each *field* within NTE is simple, for example:

```
SourceofComment
```

So that the complete *segment*:*field* path looks like this:

```
ORCgrp(1).OBRuniongrp.OBXgrp(3).NTE(1):SourceofComment
```

Some fields can have complex hierarchical structures. Suppose we look at a PID segment in the same 2.3:ORM_O01 message structure. In the *segment* identified as:

```
PIDgrp.PID
```

There could be a *field* path as follows:

```
PatientIDInternalID(1).identifiertypecode
```

Unlike *segment* paths, the *field* portion of the path generally allows numbers instead of names for fields and subfields; for example, instead of the names previously shown, the following numbers may be used:

```
3(1).5
```

The Management Portal provides pages to help you determine the correct *segment*:*field* paths. (To access these pages, click **Ensemble**, and then click **Interoperate**.) The DTL editor also provides a view of the document structures used in a particular transformation.

(The details are different for XML documents; see *Ensemble XML Virtual Document Development Guide*.)

3.5 Virtual Document Classes

To work with virtual documents, there is no need to create message classes. Ensemble provides message classes, for example, one class to carry X12 documents, another for HL7 messages, another for XML documents, and so on. The business host classes automatically use the appropriate message class.

These messages are known collectively as *virtual documents*.

The virtual document classes provide properties to carry the information that Ensemble needs to process the messages. These properties include the following:

DocType property

Indicates the document type, a string in two parts as follows:

```
category:structure
```

Where:

- *category* is the name of a schema category.
- *structure* is the name of a document structure within the referenced *category*.

The following examples show DocType values for various HL7 messages:

- `2.3.1:ORM_O01`
- `2.4:ORD_O04`
- `myCustomCategory:MFN_M03`

This property is visible in the Management Portal. You can use it for filtering, for example.

RawContent property

Contains the first 32 KB of the raw message.

This property is also visible in the Management Portal and is useful in analyzing and reporting badly formed messages.

Note that this property is not indexed in any way. If you were to use this property in an SQL search query, the query would not execute very efficiently. It is not recommended to access this property programmatically (for example from a DTL). Instead, use virtual property paths to access the data that you need.

BuildMapStatus property

Contains a %Status value that indicates the success or failure of the most recent effort to map the raw content of the message to a particular DocType. You can test BuildMapStatus in code as follows:

- Use the macro `$$$ISOK(`*`myHL7Message`*`.BuildMapStatus)` in ObjectScript and the method `$SYSTEM.Status.IsOK(`*`myHL7Message`*`.BuildMapStatus)` in Basic. If the test returns a True value, BuildMapStatus contains a success value.
- Use the macro `$$$ISERR(`*`myHL7Message`*`.BuildMapStatus)` in ObjectScript and the method `$system.Status.IsError(`*`myHL7Message`*`.BuildMapStatus)` in Basic. If the test returns a True value, BuildMapStatus contains a failure value.

To view details of any BuildMapStatus failure codes, use the Schema Structures page as described in the chapter "Portal Tools."

The virtual document classes also provide the logic that Ensemble needs to interpret a virtual property path for a specific format and DocType. The classes provide the following instance methods which you can (depending on the context) use to get or set values within a virtual document:

GetValueAt() method

Returns the value of a virtual property in the message, given a virtual property path.

You can invoke this method (and the next one) from any place in the production where you have access to the message and you can execute custom code — for example, within a BPL <code> element.

SetValueAt() method

Sets a value in the message, given a virtual property path and a value. See the comments for **GetValueAt()**.

3.6 Testing Virtual Property 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:

1. Load the corresponding schema into Ensemble, if it is not yet loaded. To do so, see the chapter "Portal Tools."
2. In the Terminal or in test code:
   a. Create a string that contains the text of a suitable document.
   b. Use the **ImportFromString()** method of the applicable virtual document class to create an instance of a virtual document from this string.

      The following table lists the classes:

Document Type	Virtual Document Class
ASTM	EnsLib.EDI.ASTM.Document
EDIFACT	EnsLib.EDI.EDIFACT.Document
HL7 Version 2	EnsLib.HL7.Message
XML	EnsLib.EDI.XML.Document

c. Set the DocType property of this instance.

d. Use the **GetValueAt()** instance method of this instance.

The following method demonstrates step 2:

```
ClassMethod TestHL72Path()
{
 set string="MSH|^~\&|HIHLS6A-223103|AAH|Bayside Medical|AAH|20120421112842||ADT^A11|"
      _"20801130301008819401|P|2.5|||NE|NE"_$C(13,10)
  _"EVN|A11|20120118150140||VHF|ECM"_$C(13,10)
  _"PID|0001|126510^^^^MR|||Watson^Darby||19820611|M|||||||||5347607018|"
      _"  ||||||||||||||AAH"_$C(13,10)
  _"PV1|0001|CL|^^^VNIDT^^SNINFD||||130725LS^Smithers^Lucianne||||||||||CL|4137784E|"
      _"||||||||||||||||||S||VA|||20120118100000|20120118101000||||||V"
 set target=##class(EnsLib.HL7.Message).ImportFromString(string,.status)
 if 'status {do $system.Status.DisplayError(status)  quit}
 set target.DocType="2.5:ADT_A09"

 set pathvalue=target.GetValueAt("PID:PatientName.familyname",,.status)
 if 'status {do $system.Status.DisplayError(status)}
 write pathvalue
}
```

The following shows output from this method:

```
ENSDEMO>d ##class(EEDI.CheckPaths).TestHL72Path()
Watson
```

4

Using Virtual Documents in a Production

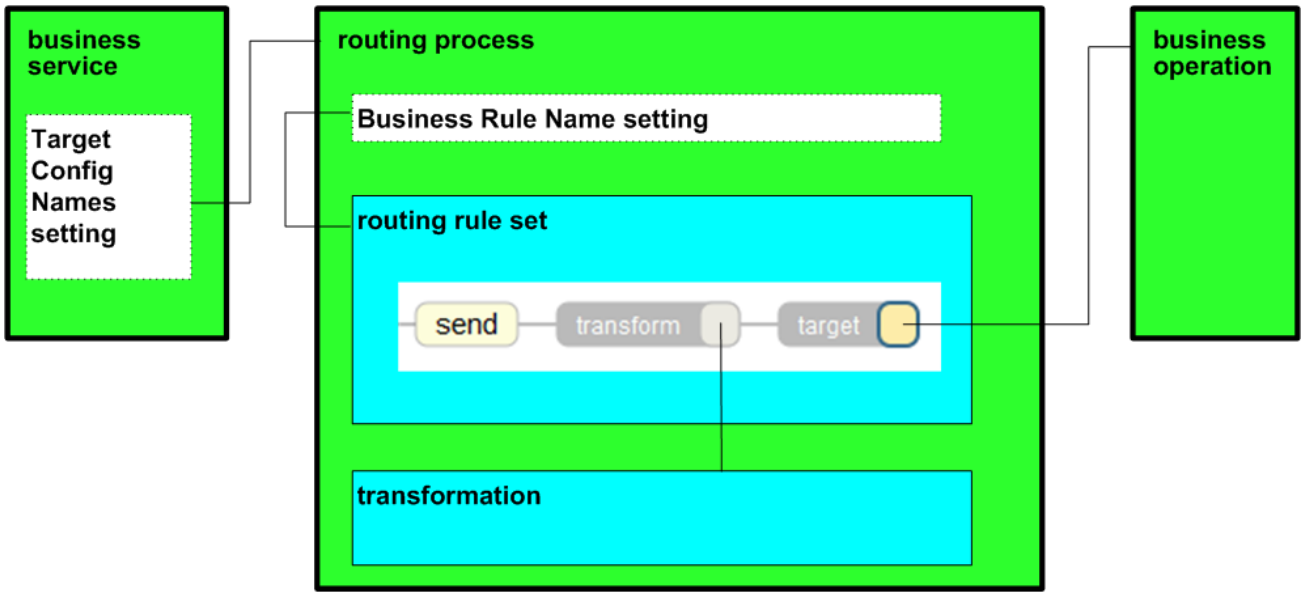
This chapter provides a brief and general overview of how to use virtual documents in a production. It contains the following sections:

4.1 Introduction

For each kind of virtual document, Ensemble provides a set of classes that you can use as business hosts, as follows:

Item	Classes
Business services	Ensemble provides one or more specialized business service classes, each with a different associated adapter. For example, for HL7 input, there are different classes for file input, FTP input, HTTP input, and so on. No coding is needed to use these classes.
Business processes	• EnsLib.HL7.MsgRouter.RoutingEngine, which is the standard Ensemble virtual document routing process class • Specialized subclasses of this routing process class No coding is needed to use these classes.
Business operations	Ensemble provides one or more specialized business operation classes, each with a different associated adapter. For example, for X12 documents, there are different classes for file output, TCP output, and FTP output. No coding is needed to use these classes.

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



Ensemble also includes schema definitions for HL7 Version 2, as well as one schema definition for ASTM. In other cases, you must load any applicable schemas in Ensemble, so that Ensemble can use them to validate and parse the virtual documents.

4.2 Business Services for Virtual Documents

For each virtual document format, Ensemble provides one or more specialized business service classes, each with a different associated inbound adapter. For example, for HL7 input, there are different classes for file input, FTP input, HTTP input, and so on. You use these classes to add business services to your production. With a few exceptions, these business hosts have the following configurable settings:

- Doc Schema Category — Specify a schema category that is relevant to the format and that is consistent with the expected messages for this business service. This setting provides some of the information that the business service requires in order to set the DocType of the message. Specifically, this setting determines the *category* part of the DocType.

  The business service also parses the document to determine the *structure* part of the DocType, and it then sets the value of that property.

- Search Table Class — Specify a suitable search table class that is consistent with the expected messages for this business service. The business service uses this to index each message that it processes.

- Target Config Names — Specify the business host or hosts to which this business service should send messages.

- Validation — Specify a string that indicates the kinds of validation for this service to perform.

  For details on the built-in validation process, see "Message Validation for Virtual Documents," later in this book.

  As part of the process of defining the production, you load any applicable EDI or XML schemas in Ensemble, so that Ensemble can use them to validate the virtual documents.

Some virtual document business services have additional configurable targets. For example, with FTP there is a persistent connection via which reply messages can be sent, so some business hosts have the setting **Reply Target Config Names**.

These business services have many other settings, to specify details that are specific to the EDI or XML format.

4.3 Business Processes for Virtual Documents

Ensemble provides specialized business processes for use with virtual documents. These processes are generally very similar to each other; each is designed as a routing process. A *routing process* routes and transforms messages by using these key items:

- Routing rules direct messages to their destinations based on message contents.
- Schema categories provide a means to validate and access message contents.
- Data transformations apply changes to prepare messages for their destinations.

These routing processes have the following configurable settings (among others):

- Business Rule Name — Name of the business rule set that this process should use.

  As part of the process of defining the production, you create this business rule set and any data transformations it requires.
- Validation — Specify a string that indicates the kinds of validation for this process to perform.

  For details on the built-in validation process, see "Message Validation for Virtual Documents," later in this book.
- Bad Message Handler — Name of the business host to which this process should send any bad messages, as determined by the validation process.

In some cases, the routing process has additional configurable targets, to handle reply messages, for example. It has additional settings, to specify details that are specific to the virtual document format.

For HL7 Version 2, Ensemble provides another specialized business process, a *sequence manager*. This business process ensures that related messages arrive at their targets with the proper sequence and timing.

4.4 Business Operations for Virtual Documents

For each virtual document format, Ensemble provides one or more specialized business operation classes, each with a different associated outbound adapter. You use these classes to add business operations to your production. With a few exceptions, these business hosts have the following configurable settings:

- Search Table Class — Specify a suitable search table class that is consistent with the expected messages for this business service. The business operation uses this to index each message that it processes.
- Validation — Specify a string that indicates the kinds of validation for this operation to perform.

  For details on the built-in validation process, see "Message Validation for Virtual Documents," later in this book.

These business operations have many other settings, to specify details that are specific to the virtual document format.

5

Defining Search Tables

This chapter describes generally how to define search tables for virtual documents. It includes the following topics:

- Defining a Search Table Class
- Example Search Table Class
- Defining a Custom Search Table Class
- Management of Search Tables
- Customizing Queries Used by the Management Portal

Be sure to perform these tasks in the same namespace that contains your production. When you create search tables, do not use reserved package names; see "Reserved Package Names" in *Developing Ensemble Productions*.

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

5.1 Defining a Search Table Class

To define a search table class, use the following general procedure:

- Create the class in the same namespace that contains your production. Also, do not use reserved package names; see "Reserved Package Names" in *Developing Ensemble Productions*.
- Create a subclass (or a copy, as you prefer) of the default search table class used for your type of virtual document:

Document Type	Default Search Table Class	Notes
HL7	EnsLib.HL7.SearchTable	Indexes a set of commonly needed properties; see "Properties That Are Indexed by Default" in the *Ensemble HL7 Version 2 Development Guide*.
X12	EnsLib.EDI.X12.SearchTable	Indexes the `Identifier` property, which corresponds to the X12 document ID.
ASTM	EnsLib.EDI.ASTM.SearchTable	Indexes the `Identifier` property, which corresponds to the ASTM document ID.
EDIFACT	EnsLib.EDI.EDIFACT.SearchTable	Indexes the `Identifier` property, which corresponds to the EDIFACT document ID.

Document Type	Default Search Table Class	Notes
XML	EnsLib.EDI.XML.SearchTable	Indexes the name of the root element of the XML document.

- In this subclass, include an XData block that defines the virtual properties as needed. The following subsection provides the details.

  This XData block does not need to include details about the virtual properties that are indexed by the superclass that you chose. For example, several of the search table classes index a property named `Identifier`. If you subclass any of those classes, your XData block does not need to include `Identifier`. You can improve search efficiency by identifying properties that are not selective. Properties are not selective when many messages have identical values for the property. When Ensemble is searching messages, it can be more efficient if it uses selective properties to limit the number of messages found before it tests the values of properties that are not selective. You can specify that a property is not selective by specifying `Unselective="true"` in the XData block.

- If this search table class is mapped to multiple namespaces, compile it in each of those namespaces, to ensure that the metadata local to each namespace is up to date.

  Important: Search table metadata is located in the default global database for each Ensemble namespace; therefore, changes to a search table class do not update metadata in all namespaces to which the class is mapped. For this reason, it is necessary to recompile the search table class in each of these namespaces.

When you compile this class, Ensemble generates code that dynamically fetches the local metadata for each search table property and then caches the metadata if the process is running as an Ensemble host. If the property metadata is not present, as in the case where a mapped search table class does not have local metadata for a new property, the class still indexes all other properties and returns an error to indicate the metadata was not present. Similarly, when the message bodies are deleted, Ensemble removes the corresponding entries from the search table; no work is required on your part.

To use the search table class, specify it as a configuration option (called **Search Table Class**) for the applicable business host. When that business host processes messages, it uses the configured search table class to index those messages.

5.1.1 XData Details for a Search Table Class

When you create a search table class, your goal is to provide one search table entry for each virtual property that you want to search and filter in the Message Browser, Rules Editor, and other parts of the Management Portal. To do this, you include an XData block like the following stub within your search table class:

```
XData SearchSpec [ XMLNamespace="http://www.intersystems.com/EnsSearchTable" ]
{
<Items>
   <Item DocType="doctype1" PropName="name1" PropType="type1" StoreNulls="boolean"
     Unselective="true">path1</Item>
   <Item DocType="doctype2" PropName="name2" PropType="type2" StoreNulls="boolean">path2</Item>
<  <Item DocType="doctype3" PropName="name3" PropType="type3" StoreNulls="boolean">path3</Item>
</Items>
}
```

Where:

- *path1*, *path2*, *path3*, and so on are virtual property paths.

  Each of these is a string expression. This string expression may include the following components in any combination:

  – Literal characters within double quotes.

  – Virtual property syntax within {} or []. This resolves to the value of the specified field in the X12 document. Square brackets differ from curly brackets in that square brackets enclose a *segment*:*field* combination that does

not require you to identify its containing document structure. For curly bracket syntax to resolve, the document structure must be known.

  - ObjectScript string operators, such as the underscore (_) for concatenation.
  - Functions, such as $PIECE or $EXTRACT.

- *doctype1*, *doctype2*, *doctype3*, and so on are DocType identifiers. Each of these is a schema category name and document structure name, separated by a colon, as follows:

  ```
  category:structure
  ```

  If the *category* is missing, any schema is matched; if the *structure* is missing; any structure is matched. A value of `""` (a blank string) matches any schema category and any document structure.

  This part of the <Item> specifies the DocType (or DocTypes) for which the given property path is to be indexed.

- *name1*, *name2*, *name3*, and so on are virtual property names of your choice.

  This is the name that Ensemble will display in the Management Portal. Choose any string that you expect to be meaningful, when viewed with others in the list.

  If you assign the same name to different <Item> elements, this has a convenient additive effect: When the user selects this name for a search, all of the entries with the same name are searched.

- *type1*, *type2*, *type3*, and so are optional type identifiers. Specify one of the following literal values:

  - `String:CaseSensitive`
  - `String:CaseInsensitive`
  - `Integer`
  - `Numeric`
  - `Boolean`
  - `DateTime:ODBC`
  - `DateTime:HL7` (supported only for virtual documents that carry HL7, ASTM, or EDIFACT messages)

  `String:CaseSensitive` is the default.

- *boolean* is an optional flag that controls what happens when a search encounters an empty field in the document. If this flag is true, Ensemble returns a valid pointer to an empty string. If this flag is false, Ensemble returns a Not Found status and a null pointer.

  Specify either 1 (which means true) or 0 (which means false). The default is 0.

- `Unselective="true"` specifies that a property value typically does not select a small number of messages. Ensemble uses this information to search more efficiently. The default value is `Unselective="false"`.

Important: When Ensemble indexes virtual documents (thus adding to the search tables), it replaces any vertical bar (|) with a plus sign (+). Take this into consideration when you use the search table to search for content. For example, to search for a message that contains `my|string`, use `my+string` as the search criterion.

5.2 Example Search Table Class

The following example shows a search table class. The `SearchSpec` XData block contains the <Item> elements that define the search table.

```
Class Demo.HL7.MsgRouter.SearchTable Extends EnsLib.HL7.SearchTable
{

XData SearchSpec [ XMLNamespace="http://www.intersystems.com/EnsSearchTable" ]
{
<Items>
   <!-- Items that do not depend on DocType, indexing any HL7 message -->
   <Item DocType="" PropName="SendingFacilApp" >{1:4}_"|"_{1:3}</Item>
   <Item DocType="" PropName="RecvingFacilApp" >{1:6}_"|"_{1:5}</Item>
   <Item DocType="" PropName="MSHDateTime" PropType="DateTime:HL7" >{1:7}</Item>

   <!-- Get fields from named segments found in any HL7 message -->
   <Item DocType="" PropName="PatientName" >[PID:5]</Item>
   <Item DocType="" PropName="InsuranceCo" >[IN1:4]</Item>

   <!-- Get patient name from any HL7 message declared type ADT_A05 -->
   <Item DocType=":ADT_A05" PropName="PatientName" >{3:5}</Item>

   <!-- Get specific field from specific segment when the        -->
   <!-- HL7 message is assigned a specific DocType. Only in this -->
   <!-- case can you use names for segments, instead of numbers. -->
   <Item DocType="Demo.HL7.MsgRouter.Schema:ORM_O01 " PropName="ServiceId" >
     {ORCgrp().OBRuniongrp.OBRunion.OBR:UniversalServiceID.text}
   </Item>
   <Item DocType="2.3.1:ORU_R01 " PropName="ServiceId" >
     {PIDgrpgrp().ORCgrp(1).OBR:UniversalServiceID.text}
   </Item>
</Items>
}
}
```

The XMLNamespace declaration (as shown in the preceding example) enables Studio to provide word completion as you type.

5.3 Defining Custom Search Table Classes

In some cases, the basic search table mechanism described in this chapter might not enable you to index messages as needed. In such cases, you can define and use custom search table classes.

The class can define two kinds of properties. Within this topic, these properties are called: *standard properties* (which are stored in the search table) and *virtual properties* (which are not stored in the search table but instead are retrieved at runtime). Either kind of property is either indexed or not. If you index a property, more disk space is consumed but queries for that property run more quickly. The Management Portal displays the indexed properties as a group above the non-indexed ones, so that users can select them appropriately.

To define a custom search table class, define a class as follows:

- Extend Ens.CustomSearchTable.

  This class defines one standard class property, DocId, which is indexed.

- Define additional class properties as needed, and add indices for these class properties. For example:

  ```
  Property Type As %String(COLLATION = "EXACT");

  Index Type On Type [ Type = bitmap ];
  ```

  Note that collection properties are not currently directly supported by the query generation mechanisms. For a collection property, use the **GetVirtualPropertyList()** method mechanism described below.

- Optionally implement the **GetPropertyList()** method as needed.

  ```
  classmethod GetPropertyList(Output pIndexedProperties As %List, Output pProperties As %List) as
  %Status
  ```

  Where:

  – *pIndexProperties* is a $LISTBUILD list of standard properties that should be indexed.

  – *pProperties* is a $LISTBUILD list of standard properties to define.

  The purpose of this step is to indicate which class properties are to be used as standard properties (see the definitions before this list), as well as which of those should be indexed.

  By default, this method is generated, and Ensemble uses all class properties of the search table class as standard properties and indexes them all, except for private, internal, transient, and multidimensional properties.

  For virtual properties, implement **GetVirtualPropertyList()** instead (or in addition).

- Optionally implement the **GetVirtualPropertyList()** method as needed.

  ```
  classmethod GetVirtualPropertyList(Output GetVirtualPropertyList As %List,
                                     Output pVirtualProperties  As %List)
                                     as %Status
  ```

  Where:

  – *GetVirtualPropertyList* is a $LISTBUILD list of virtual properties that should be indexed.

  – *pVirtualProperties* is a $LISTBUILD list of the virtual properties to define.

  The purpose of this step is to indicate which class properties are to be used as virtual properties, as well as which of those should be indexed.

  For standard properties, implement **GetPropertyList()** instead (or in addition).

- If you implement **GetVirtualPropertyList()**, also implement the **GetVirtualProperty()** method. This method must return the value of a virtual property, given a document ID and a virtual property name:

  ```
  classmethod GetVirtualProperty(pDocID As %String,
                                 pPropName As %String,
                                 Output pPropValue As %String,
                                 ByRef pUserArgs) as %Status
  ```

  Where:

  – *pDocID* is the ID of a document in the custom search table.

  – *pPropName* is the name of a virtual property.

  – *pPropValue* is the value of that property.

  – *pUserArgs* specifies any arguments.

- Implement the **OnIndexDoc()** method.

  ```
  ClassMethod OnIndexDoc(pDocObj As %Persistent, pSearchTable As Ens.CustomSearchTable) As %Status
  ```

  This method should specify how to populate a given row in the search table from properties in a supplied message.

For **OnProcessCondition()** and additional options, see the class reference for Ens.CustomSearchTable.

For an example, see Demo.CustomSearchTable.Sample.

5.4 Management of Search Tables

The class Ens.DocClassMap manages all the search tables (including custom search tables). It writes to and reads from a global (*^Ens.DocClassMap*). This global indicates, for each message class, which search tables contain data for it. Note that you should never edit this global directly.

The Ensemble classes use this class to remove search table entries when message bodies are deleted.

It should not normally be necessary to use this class directly. However, if the data in *^Ens.DocClassMap* is lost or damaged, use the **RebuildMap()** method of this class to recreate the global. For details, see the class reference for Ens.DocClassMap.

5.5 Customizing Queries Used by the Management Portal

When users search for messages in the **Message Viewer** and the **Message Bank Message Viewer** pages in the Management Portal, Ensemble generates and then uses queries. In advanced cases, you can customize how Ensemble generates these queries. To do so, use the following general procedure:

- Define a subclass of EnsPortal.MsgFilter.AbstractAssistant. For details, see the class reference for that class.
- Set the name of the class into *^EnsPortal.Settings("MessageViewer","AssistantClass")* for the Message Viewer or *^EnsPortal.Settings("MsgBankViewer","AssistantClass")* for the Message Bank Viewer.

6

Controlling Message Validation

For virtual documents other than ASTM documents, Ensemble can provide message validation, with an option to include a business host to handle bad messages. You can also override the validation logic, if wanted. This chapter describes the details. It includes the following topics:

- Introduction to Message Validation
- Basic Validation Options and Logic
- Overriding the Validation Logic
- Defining Bad Message Handlers

6.1 Introduction to Message Validation

For virtual documents other than ASTM documents, one or more of the specialized business host classes include the **Validation** setting, which you use to specify how that business host should validate messages that it receives, before attempting further work.

If the message does not fail validation, the business host sends the message to the specified normal target or targets.

If the document does fail validation, the details are different depending on the kind of business host:

- A business service or business operation does not send the message anywhere.
- A routing process includes the additional setting Bad Message Handler, which is meant to be the name of a business host. If the document fails validation, the routing process forwards the document to its bad message handler, as specified by this setting. If there is no bad message handler, the routing process does not route the document, but logs an error. The routing process may also include a setting to enable you to send an alert.

6.2 Basic Validation Options and Logic

This topic describes the allowed values for the Validation setting and describes how Ensemble validates a message. (Note that HL7 Version 2 has an additional flag related to Z-segments and has additional logic; see the *Ensemble HL7 Version 2 Development Guide*.)

Value	Meaning
d	Validation examines the DocType property of the document to see if it has a value.
m	Validation verifies that the document segment structure is well formed, and that it can be parsed using the schema identified in the DocType property of the document.
dm	For most virtual document formats, this is the default setting. Both d and m are active.
1	Same as dm, for backward compatibility with previous releases.
(a blank string)	The routing process skips validation and routes all documents as given in the associated routing rule set.

6.2.1 The d Validation Flag

If the `d` flag is present in the **Validation** string, Ensemble examines the **Schema Category** specified in the message (as set by the business service) and compares it with the message DocType. If the **Schema Category** in the message is blank, the message is automatically declared bad. However, if **Schema Category** is not blank, but does not match the message DocType, validation can continue, as long as there are more **Validation** flags defined, such as m.

6.2.2 The m Validation Flag

If the `m` flag is present in the **Validation** string, Ensemble searches for a way to either validate the message, or declare it bad The details depend upon the virtual document type.

6.3 Overriding the Validation Logic

The virtual document routing process (EnsLib.MsgRouter.VDocRoutingEngine), and its subclasses provide default validation logic. Most of the specialized virtual document business service and operation classes also provide validation logic. You can override the validation logic; to do so, create and use a subclass of the applicable class.

6.3.1 Overriding the Validation Logic in a Routing Process Class

If you create a subclass of an Ensemble class and then override the **OnValidate()** method, you can:

- Extend or replace the list of accepted values for the **Validation** setting.
- Determine how the routing process will validate documents, as controlled by your own **Validation** options.

When you override the **OnValidate()** method, you may also override the **Validation** property definition in the same subclass. Pay careful attention to the following details:

- The `InitialExpression` value specifies the default for the **Validation** configuration setting.
- The comments that precede the **Validation** property definition are used as a tooltip for the **Validation** setting. Use the `///` convention and leave no white space lines between the last comment line and the property definition. This allows Management Portal users to view your comments as a tooltip.

  As an example, the following excerpt shows some of the comments that appear with the **Validation** property definition in the EnsLib.HL7.MsgRouter.RoutingEngine class:

```
/// 'd' - require DocType
/// 'm' - don't tolerate BuildMap errors (includes 'z' by default;
///       specify '-z' to tolerate unrecognized trailing Z-segments)
/// 'z' - don't tolerate unrecognized trailing Z-segments
Property Validation As %String(MAXLEN = 20) [ InitialExpression = "dm-z", Transient ];
```

6.3.2 Overriding the Validation Logic in a Business Service or Operation Class

The virtual document business service and operation classes each provide a **Validation** property and **OnValidate()** method that you can override. By default, this property is *not* exposed as a setting for any of these business services or business operations, and by default, no **OnValidate()** activity ever occurs in these classes. You can change this if you want to validate documents at the incoming or outgoing sides of the interface, rather than at the routing engine as is the usual case. To accomplish this, follow these steps:

1. Use the instructions for overriding **Validation** and **OnValidate()** provided above.
2. If you want your users to be able to choose a type of validation, also add the **Validation** property as a setting. See "Adding and Removing Settings" in *Developing Ensemble Productions*.

6.4 Defining Bad Message Handlers

A routing process has the setting **Bad Message Handler**. The purpose of this setting is to indicate the business host to which the process should send messages that are found to be bad, according to the Bad Message Handler of the business process. To define a bad message handler, first decide how you want to handle the bad message. Typically you create a business operation. This business operation could do either or both the following, for example:

- (Via a file adapter) Write the contents of the message to a file.
- (Depending on a configuration setting) Trigger an alert whenever it encounters a bad message.

Note that the business process sends the bad message to this business host *instead of* its usual target for validated messages.

If a message is bad and if the **Bad Message Handler** setting is not specified, the routing process simply stops the validation sequence and does not send the message.

7

Creating Custom Schema Categories

It may be necessary to create custom schema categories in Ensemble. This chapter describes the details. It includes the following topics:

- When Custom Schema Categories Are Needed
- Ways to Create Custom Schema Categories
- Syntax for Schema Categories in Ensemble

Important: Never edit the built-in schema category definitions. When you need a custom schema (or, for HL7, when you need custom message types that contain Z-segments), create a custom schema category definition that uses a built-in schema category definition as its schema base.

Be sure to perform these tasks in the same namespace that contains your production.

7.1 When Custom Schema Categories Are Needed

The ASTM, EDIFACT, HL7, and X12 standards are extensible.

For example, a common practice in customizing ASTM is to add custom segments to an otherwise standard ASTM document structure.

Similarly, the common practice in customizing HL7 is to add custom segments to an otherwise standard HL7 message structure. By convention, the custom segments have names that begin with the letter Z (ZEK, ZPM, etc.), so custom segments are called *Z-segments*. A custom message structure often consists of a standard message with a few Z-segments added to it.

If you are working with documents that do not follow a standard schema that you have loaded, it is necessary to create the corresponding custom schema category if you want to do either of the following:

- Perform structural document validation
- Use segment and field path names in BPL, DTL, and routing rule syntax

7.2 Ways to Create Custom Schema Categories

There are four general ways to create custom schema categories:

- To create a custom HL7 schema, you can use the Custom Schema Editor as described in "Using the Custom Schema Editor" in the *HL7 Version 2 Development Guide*.
- Create a copy of the schema file that most closely meets your needs. Edit that file. Then import it, as described in the chapter "Portal Tools."
- In Studio, open the schema category definition that most closely meets your needs. Use **File > Save As** to create a new definition. Then edit the copy. The section "Syntax for Schema Categories in Ensemble" provides details for the syntax of the definition.
- Use a Studio wizard to create a new custom schema category definition, and then edit that. See "Creating Custom X12 Schemas" in the *Ensemble X12 Development Guide*. The same process can be used for ASTM, EDIFACT, and HL7.

The Studio Workspace window displays a list of the schema categories in the given namespace. See the **Other** folder. These definitions have file extensions such as .AST, .HL7, and .X12.

7.3 Syntax for Schema Categories in Ensemble

Ensemble uses the same XML-based syntax to represent schema category definitions for ASTM, EDIFACT, HL7, and X12. The following shows an example. Note that the actual text is longer and wider than the example, which contains ellipses (...) for omitted items and is truncated at right.

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

<Category name="2.5" std="1">

 <MessageGroup name='ACK' description='General acknowledgment message'/>
 <MessageGroup name='ADR' description='ADT response'/>
 <MessageGroup name='ADT' description='ADT message'/>
 <MessageGroup name='BAR' description='Add/change billing account'/>
                :
 <MessageEvent name='A03' description='ADT/ACK -  Discharge/end visit'/>
 <MessageEvent name='A04' description='ADT/ACK -  Register a patient'/>
 <MessageEvent name='A05' description='ADT/ACK -  Pre-admit a patient'/>
 <MessageEvent name='A06' description='ADT/ACK -  Change an outpatient to an inpatient'/>
                :
 <MessageType name='ACK' structure='ACK'/>
 <MessageType name='ADR_A19' structure='ADR_A19'/>
 <MessageType name='ADT_A01' structure='ADT_A01' returntype='ACK_A01'/>
 <MessageType name='ADT_A02' structure='ADT_A02' returntype='ACK_A02'/>
                :
 <MessageStructure name='ORR_O02' definition='MSH~MSA~[~{~ERR~}~]~[~{~NTE~}~]~[~[~PID~[~{~NTE~}~]~]~{~OR
 <MessageStructure name='ORS_O06' definition='MSH~MSA~[~{~ERR~}~]~[~{~SFT~}~]~[~{~NTE~}~]~[~[~PID~[~{~NT
 <MessageStructure name='ORU_R01' definition='MSH~[~{~SFT~}~]~{~[~PID~[~PD1~]~[~{~NTE~}~]~[~{~NK1~}~]~[~
 <MessageStructure name='ORU_R30' definition='MSH~[~{~SFT~}~]~PID~[~PD1~]~[~PV1~[~PV2~]~]~ORC~OBR~[~{~NT
 <MessageStructure name='OSQ_Q06' definition='MSH~[~{~SFT~}~]~QRD~[~QRF~]~[~DSC~]'/>
                :
 <SegmentStructure name='BLG' description='Billing'>
      <SegmentSubStructure piece='1' description='When to Charge' datastruct='CCD' length='40' required=
      <SegmentSubStructure piece='2' description='Charge Type' length='50' required='O' ifrepeating='0'
      <SegmentSubStructure piece='3' description='Account ID' datastruct='CX' length='100' required='O'
      <SegmentSubStructure piece='4' description='Charge Type Reason' datastruct='CWE' length='60' requi
 </SegmentStructure>
                :
 <DataStructure name='FC' description='Financial Class'>
      <DataSubStructure piece='1' description='Financial Class Code' codetable='64'/>
      <DataSubStructure piece='2' description='Effective Date' datastruct='TS'/>
 </DataStructure>
                :
 <CodeTable name='109' tabletype='2' description='Report priority'>
     <Enumerate position='1' value='R' description='Routine'/>
     <Enumerate position='2' value='S' description='Stat'/>
 </CodeTable>
                :

</Category>
```

Inside the <Category> element, the example begins with two sets of XML elements that you do not need to work with: <MessageGroup> and <MessageEvent>. These elements group HL7 messages into functional categories. The example ends with two other XML elements that you do not need to work with: <DataStructure> and <CodeTable>. These elements define the range of possible values for certain fields in certain messages.

When you create a custom schema category, you supplement an existing schema by defining custom segments (Z-segments) and then stating which message types and message structures may contain those segments. To accomplish this, you only need to work with the XML elements <MessageType>, <MessageStructure>, and <SegmentStructure>.

A custom schema category definition is simpler than a built-in definition and contains fewer statements. Everything in the base category is included in the custom schema category definition. There is no need to repeat the definitions of standard message types. You only need to define custom message types. The conventions for doing this are as follows:

What to Define	How to Define It
Custom schema category definition	<Category>
Custom segments (called Z-segments for HL7 messages)	<SegmentStructure>
Any message structures that include custom segments	<MessageStructure>
Any message types that include message structures with custom segments. A message type identifies: • The message structure to send • The message structure to expect in response	<MessageType>

When viewed in Studio, a custom schema category definition resembles the following figure. The actual text is wider than the example, which is truncated at right. You can view the complete example by opening the schema definition file Demo.HL7.MsgRouter.Schema.HL7 in the ENSDEMO namespace.

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

<Category name="Demo.HL7.MsgRouter.Schema" base="2.3.1">
                :
 <MessageType name='ADT_A13' structure='ADT_A01' returntype='base:ACK_A13'/>
 <MessageType name='ADT_A16' structure='ADT_A16' returntype='base:ACK_A16'/>
 <MessageType name='ADT_A25' structure='ADT_A25' returntype='base:ACK_A25'/>
 <MessageType name='ADT_A31' structure='ADT_A01' returntype='base:ACK_A31'/>
 <MessageType name='ADT_Z44' structure='base:ADT_A30' returntype='base:ACK_A47'
 <MessageType name='ADT_Z47' structure='base:ADT_A30' returntype='base:ACK_A47'
                :
 <MessageStructure name='ADT_A01' definition='base:MSH~base:EVN~base:PID~[~base:PD1~]~[~{~base:NK1~}~]~b
 <MessageStructure name='ADT_A16' definition='base:MSH~base:EVN~base:PID~[~base:PD1~]~[~{~base:NK1~}~]~b
 <MessageStructure name='ADT_A25' definition='base:MSH~base:EVN~base:PID~[~base:PD1~]~base:PV1~[~base:PV
 <MessageStructure name='MFN_M03' definition='base:MSH~base:MFI~{~base:MFE~[~ZSI~]~base:OM1~[~base:Hxx~]
 <MessageStructure name='ORM_O01' definition='base:MSH~[~{~base:NTE~}~]~[~base:PID~[~base:PD1~]~[~{~base
                :
 <SegmentStructure name='ZNB' description='Newborn Abstract'>
      <SegmentSubStructure piece='1' description='SetID' symbol='!' length='4' required='R' ifrepeating=
      <SegmentSubStructure piece='2' description='Birth Status Code' length='60' required='O' ifrepeatin
      <SegmentSubStructure piece='3' description='Birth Type Code' length='60' required='O' ifrepeating=
      <SegmentSubStructure piece='4' description='C-Section Indicator' length='2' required='O' ifrepeati
      <SegmentSubStructure piece='5' description='Gestation Period - Weeks' length='3' required='O' ifre
      <SegmentSubStructure piece='6' description='Encounter Number' symbol='!' length='12' required='R'
      <SegmentSubStructure piece='7' description='Newborn Code' symbol='!' length='60' required='R' ifre
      <SegmentSubStructure piece='8' description='Newborn Weight' length='10' required='O' ifrepeating='
      <SegmentSubStructure piece='9' description='Stillborn Indicator' length='8' required='O' ifrepeati
 </SegmentStructure>
                :
</Category>
```

7.3.1 <Category>

The <Category> element is the top-level container for the XML document that describes the custom schema category. The following is an example of <Category> syntax for a custom schema category definition:

```
<Category name="Demo.HL7.MsgRouter.Schema" base="2.3.1">
  <!-- All the statements in the custom schema category definition -->
</Category>
```

The following table describes the <Category> attributes.

Attribute	Description	Value
name	Name displayed in the Schema Structures page in the list of available schema categories.	String. For convenience, use the name of the .HL7 file or other schema file
std	When 1 (true), this <Category> block describes a standard HL7 schema category. The default is 0 (false).	For standard schema category definitions only. Do not use *std* in a custom schema.
base	Identifies the schema category that is the base for this custom schema category. Every definition in the schema base is automatically included in the custom category; statements in the custom schema category simply add to the base.	The *name* of a standard or custom schema category defined using a <Category> block in another .HL7 file or other schema file.

7.3.2 <SegmentStructure>

A <Category> element may contain one or more <SegmentStructure> elements. Each <SegmentStructure> element defines the structure of a custom segment (Z-segment). The following is an example of <SegmentStructure> element syntax:

```
<SegmentStructure name='ZNB' description='Newborn Abstract'>
  <SegmentSubStructure piece='1' description='SetID' symbol='!'
                       length='4' required='R' ifrepeating='0'/>
  <SegmentSubStructure piece='2' description='Birth Status Code'
                       length='60' required='O' ifrepeating='0'/>
  <SegmentSubStructure piece='3' description='Birth Type Code'
                       length='60' required='O' ifrepeating='0'/>
  <SegmentSubStructure piece='4' description='C-Section Indicator'
                       length='2' required='O' ifrepeating='0'/>
  <SegmentSubStructure piece='5' description='Gestation Period - Weeks'
                       length='3' required='O' ifrepeating='0'/>
  <SegmentSubStructure piece='6' description='Encounter Number' symbol='!'
                       length='12' required='R' ifrepeating='0'/>
  <SegmentSubStructure piece='7' description='Newborn Code' symbol='!'
                       length='60' required='R' ifrepeating='0'/>
  <SegmentSubStructure piece='8' description='Newborn Weight'
                       length='10' required='O' ifrepeating='0'/>
  <SegmentSubStructure piece='9' description='Stillborn Indicator'
                       length='8' required='O' ifrepeating='0'/>
</SegmentStructure>
```

The following table describes the <SegmentStructure> attributes.

Attribute	Description	Value
name	Name displayed in the Schema Structures page in the list of available segment structures.	3–character string. By convention, custom segment names begin with the letter Z.
description	Text description of the segment contents, displayed in the Schema Structures page and as a tooltip for the Document Viewer page.	String

7.3.3 <SegmentSubStructure>

A <SegmentStructure> element may contain one or more <SegmentSubStructure> elements. Each <SegmentSubStructure> element describes one field of the custom segment, in sequential order from top to bottom. The following is an example of <SegmentSubStructure> element syntax:

```
<SegmentSubStructure piece='6' description='Encounter Number' symbol='!'
                     length='12' required='R' ifrepeating='0'/>
```

The following table describes the <SegmentSubStructure> attributes.

Attribute	Description	Value
piece	Number displayed in the Schema Structures page when the user asks to view details of the segment that contains this field. This number can be used to identify the field in a virtual property path.	Integer. Each <SegmentSubStructure> within a <SegmentStructure> must use *piece* values in sequential order, beginning at 1 and incrementing by 1.
codetable	Code table that enumerates a list of valid values for this field. This attribute is typically not used in a custom schema.	The *name* of a code table defined using a <CodeTable> block.
datastruct	Data structure that specifies how to interpret the values in this field. This attribute is typically not used in a custom schema.	The *name* of a data structure defined using a <DataStructure> block.
description	Text description of the field contents, displayed in the Schema Structures page and as a tooltip for the Document Viewer page.	String
symbol	Symbol that indicates the requirements for presence, absence, or repetition of this field within the segment. This field is optional. It serves as an indicator on the Schema Structures page. It does not actually control the requirement or repetition of fields. See *required* and *ifrepeating*.	A single character: • ! means 1 only. The field *must* appear, but only once. • ? means 0 or 1. The field *may* appear, but at most once. • + means 1 or more. The field may repeat one or more times. • * means 0 or more. The field may repeat zero or more times. • & means the field may be present, and may repeat, but only under certain conditions.
length	Upper limit on the number of characters that can be present in this field.	Integer
required	Whether or not this field must be present in the segment.	A single character: • C means conditional • O means optional • R means required
ifrepeating	Whether or not this field may repeat within the segment.	Integer. 0 means no, 1 means yes.

7.3.4 <MessageStructure>

A <Category> element may contain one or more <MessageStructure> elements. Each <MessageStructure> element provides a specification for the number and arrangements of segments in a message structure. The following is an example of <MessageStructure> element syntax:

```
<MessageStructure
  name='MFN_M03'
  definition='base:MSH~base:MFI~{~base:MFE~[~ZSI~]~base:OM1~[~base:Hxx~]~}'
  description='HNB MFN message'
  />
```

The following table describes the <MessageStructure> attributes.

Attribute	Description	Value
name	Name displayed in the Schema Structures page in the list of available message structures.	3–character string, plus an underscore (_), plus a 3–character string.
definition	Specification for the number and arrangements of segments in the message structure. May include a mix of standard and custom message segments. See syntax rules below.	String that includes the 3–character *name* values for standard or custom message segments defined using <SegmentStructure>
description	Text description of the field contents, displayed in the Schema Structures page and as a tooltip for the Document Viewer page.	String

Syntax for the *definition* string works as follows:

- Keep the entire string all on one line
- List each segment sequentially from left to right
- When listing a segment, use its *name* value as defined by a <SegmentStructure>
- Separate a segment from the next segment by a ~ (tilde) character
- If a segment or block of segments repeats, enclose the repeating part in {~ and ~}
- If a segment or block of segments is optional, enclose the optional part in [~ and ~]

Within a *definition*, a *name* may be simple, in which case Ensemble assumes the value refers to a custom <MessageStructure> block within the same .HL7 file. Alternatively, a *name* may refer to a standard message structure from the schema base. This means it is defined in the .HL7 file identified by the *base* attribute in the containing <Category> element. To indicate this, the *name* must use the prefix:

```
base:
```

So that Ensemble can find the appropriate <SegmentStructure> in the other file. No other external .HL7 file can be referenced; only the schema base.

In the <MessageStructure> example at the beginning of this topic, only the ZSI segment is defined in the same .HL7 file. The other segments (MSH, MFI, MFE, OM1, and Hxx) are all defined in the schema base.

7.3.5 <MessageType>

A <Category> element may contain one or more <MessageType> elements. <MessageType> entries define any message types that include message structures that have custom segments. A <MessageType> element is a simple list of two items:

- A message structure to send
- A message structure to expect in response

The following is an example of <MessageType> element syntax:

```
<MessageType name='ADT_A31' structure='ADT_A01' returntype='base:ACK_A31'/>
```

The following table describes the <MessageType> attributes.

Attribute	Description	Value
name	Name displayed in the Schema Structures page in the list of available message types.	3–character string, plus an underscore (_), plus a 3–character string.
structure	The message structure to send.	The *name* of a standard or custom message structure defined using <MessageStructure>
returntype	The message structure to expect in response. This must be a valid ACK message structure. Make sure the *returntype* has a value from the schema base. For example: `returntype="base:ACK"`	The *name* of a standard or custom message structure defined using <MessageStructure>

structure values can be simple *name* values from the current .HL7 file. If a file contains syntax like the following, where the *structure* is MFN_M03:

```
<MessageType name='MFN_M03' structure='MFN_M03' returntype='base:MFK_M03'/>
```

Then the same file must also contain syntax like the following, to define MFN_M03:

```
<MessageStructure
  name='MFN_M03'
  definition='base:MSH~base:MFI~{~base:MFE~[~ZSI~]~base:OM1~[~base:Hxx~]~}'
  description='HNB MFN message'
  />
```

Alternatively, *structure* or *returntype* values can refer to a standard message structures from the schema base. To indicate this, the values must use the prefix:

`base:`

So that Ensemble can find the appropriate <MessageStructure> entries, in the .HL7 file identified by the *base* attribute in the containing <Category> element. No other external .HL7 file can be referenced; only the schema base.

The following example uses both styles of syntax. This example defines a custom message type MFN_M03 that sends a custom MFN_M03 and receives a standard MFK_M03 as a response.

```
<MessageType name='MFN_M03' structure='MFN_M03' returntype='base:MFK_M03'/>
```

8

Portal Tools

The Management Portal provides pages to help you view the Ensemble schema categories and their subdivisions, view documents, and perform related tasks. This chapter describes how to use these pages in general, and it contains the following topics:

- Accessing the Tools
- Using the Schema Structures Page
- Using the Document Viewer Page

8.1 Accessing the Tools

1. In the Management Portal, switch to the appropriate namespace.

   To do so, click **Switch** in the title bar, select the namespace, and click **OK**.
2. Click **Ensemble**.
3. Click **Interoperate**.
4. Click the menu option that corresponds to the EDI format in which you are interested.

8.2 Using the Schema Structures Page

Ensemble provides versions of the Schema Structures page for ASTM, HL7 Version 2, X12, and XML. Depending on the format, you can use this page to import, export, and view schemas.

To access this page:

1. Click **Ensemble**.
2. Click **Interoperate**.
3. Click the menu option that corresponds to the format in which you are interested.
4. Click the option ending with **Schema Structures** and then click **Go**.

   Ensemble displays a page that lists the Ensemble schemas of this type in this namespace.

In most cases, this page displays a list of schema categories on the left; these are all the schema categories in this namespace related to this format. For example, for HL7 Version 2, this list is as follows:

Category	Base	Standard
2.1		Yes
2.2		Yes
2.3		Yes
2.4		Yes
2.5		Yes
2.6		Yes
2.7		Yes
2.3.1		Yes
2.5.1		Yes

These columns are as follows:

- **Category** identifies the schema category.
- **Base** specifies the standard category on which a custom category is based.
- **Standard** indicates whether this schema category is standard (**Yes**) or custom (**No**).

This area enables you to specify which schema category you are interested in examining. When you click a schema category, Ensemble displays details for that category in the tabs on the right.

Depending on the EDI format, the tabs on the right are as follows:

- **DocType Structures** identifies the sequence and grouping of segments within a message structure.
- **Segment Structures** lists the fields in each segment.
- **Data Structures** lists the contents of composite data fields.
- **Code Tables** lists the values that can be used within an enumerated field.

On this page, you can do some or all the following, depending on the EDI format:

- Import a schema into this namespace, for this EDI standard. To do so, click **Import**, use **Browse** to choose a file, and click **OK**.
- Export a schema to a file. To do so, select the schema category, then click **Export**, enter a filename, select a file type, and click **OK**.
- Remove a custom schema from this namespace. To do so, select a custom schema and click **Delete**, and then click **OK**. You cannot delete any of the standard schemas installed with Ensemble.

  The schema is immediately removed.

  CAUTION: You cannot undo this operation.

- Create a new schema based on an existing schema. To do so, click **New** and then specify the base schema and the new custom schema name. You can then specify the details for the custom schema.

8.3 Using the Document Viewer Page

For information on the document viewer, see Using the X12 Document Viewer Page or Using the HL7 Message Viewer Page. The information presented in these guides is equally applicable to ASTM, XML, EDIFACT, or any other EDI format.

Syntax Guide for Virtual Property Paths

This reference describes the syntax details that support the virtual property model. Except where noted, this information applies equally to HL7 Version 2, X12, and so forth. It provides the following reference information:

- Virtual Property Shortcuts When DocType Is Unimportant
- Virtual Property Path Basics
- Curly Bracket { } Syntax
- Square Bracket [ ] Syntax
- Parenthesis () Syntax
- Angle Bracket <> Syntax

The following table summarizes where the virtual property path syntaxes can be used:

Syntax	BPL	DTL	Business Rules	Search Filters and Search Tables
GetValueAt()	supported	supported	*not supported*	*not supported*
{ } Syntax	supported (except for in <code> and <sql> elements)	supported (except for in <code> and <sql> elements)	supported	supported
[ ] Syntax	*not supported*	*not supported*	supported	supported
() Syntax	*not supported*	supported	supported	*not supported*
<> Syntax	*not supported*	*not supported*	supported	*not supported*

Virtual Property Shortcuts When DocType Is Unimportant

Describes virtual property shortcuts that you can use when DocType is unimportant.

Details

Often when you need to identify a virtual property path, the specific DocType is clear from the context, so you only need to identify the *segment*:*field* path. There are also cases when a specific DocType is not important and any DocType from the schema definition that matches your search criteria is of interest. Consequently, there are two important shortcuts for virtual property syntax used in BPL, DTL, and routing rules:

- Curly brackets

  {*segment*:*field*}

  Curly bracket syntax is available in BPL, DTL, business rules (including routing rules), search filters, or search tables. The *segment*:*field* combination inside the curly brackets may use segment and field names, or numeric positions. However, names work only when the DocType (*category*:*structure*) is clearly identified within the current context. For example, a DTL data transformation always identifies the DocType of its source and target in its <transform> element.

- Square brackets

  [*segment*:*field*]

  Square bracket syntax is available for business rules, search tables, and search filters only. The *segment* must be a name; *field* may be a name or number. Names can only be resolved when the DocType (*category*:*structure*) is known at runtime. Ensemble can resolve a numeric *field* without knowing the specific message structure or schema. If there is more than one result that matches the pattern in square brackets, this syntax returns a string that contains all matching values, each value enclosed in <> angle brackets.

The following shortcuts are available only when defining routing rules:

- Round brackets or parentheses

  (*multi-valued-property-path*)

- Angle brackets

  <*context*|*expression*>

Virtual Property Path Basics

Describes how to create a virtual property path.

Introduction

For most EDI formats (not XML virtual documents), a virtual property path has the following syntax:

```
segmentorsubsegmentID:fieldorsubfieldID
```

Where:

- *segmentorsubsegmentID* refers to a segment or a subsegment. This identifier follows the rules given in "Segment and Subsegment Identifiers."
- *fieldorsubfieldID* refers to a field or subfield within the parent segment or subsegment. This identifier follows the rules given in "Field and Subfield Identifiers."

A virtual property path is relative to a specific message structure, and might not be valid in any other message structure. The Management Portal provides pages to help you determine paths. See the chapter "Portal Tools."

Important: For XML virtual documents, the syntax is different. See *Ensemble XML Virtual Document Development Guide*.

Segment and Subsegment Identifiers

To refer to a segment, use either of the following:

- Symbolic name of the segment.

  If a message has multiple segments of the same name, append `(n)` to the end of the identifier, where *n* is the 1–based position of the desired segment.

- Number of the segment as contained in the message. Typically this number is not known, so this syntax is less useful. An exception (for HL7 Version 2) is the MSH segment, which is always first.

A segment can have subsegments (which can have further subsegments). To identify a subsegment, use the following syntax:

```
segmentID.subsegmentID
```

Where both *segmentID* and *subsegmentID* follow the preceding rules for identifying a segment. Note that you cannot mix symbolic names and numeric identifiers. That is, if you use a symbolic name for any part of this syntax, you must use symbolic names in all parts of this syntax. Similarly, if you use the numeric identifier in any part of this syntax, you must use the numeric identifiers in all parts of this syntax.

Field and Subfield Identifiers

To refer to a field, use either of the following:

- Symbolic name of the field.

  If the parent segment or subsegment has multiple fields of the same name, append `(n)` to the end of the identifier, where *n* is the 1–based position of the desired field.

- Number of the field within the parent segment or subsegment

A field can have subfields (which can have further subfields). To identify a subfield, use the following syntax:

```
fieldID.subfieldID
```

Where both *fieldID* and *subfieldID* follow the preceding rules for identifying a field. Note that you cannot mix symbolic names and numeric identifiers. That is, if you use a symbolic name for any part of this syntax, you must use symbolic names in all parts of this syntax. Similarly, if you use the numeric identifier in any part of this syntax, you must use the numeric identifiers in all parts of this syntax.

Examples

The following virtual property path accesses the `streetaddress` subfield of the `Address` field of the second `NK1` segment:

```
NK1(2):Address.streetaddress
```

The following virtual property path accesses the `streetaddress` subfield of the `ContactAddress` field of the `AUTgrp.CTD` subsegment of the first `PR1grp` segment:

```
PR1grp(1).AUTgrp.CTD:ContactAddress.streetaddress
```

Special Variations for Repeating Fields

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

Iterating Through the Repeating Fields

When you are using curly bracket `{}` notation in BPL or DTL, the shortcut `()` iterates through every instance of a repeating field. For example, consider the following single line of DTL:

```
<assign property='target.{PID:3().4}' value='"001"'/>
```

This line is equivalent to the following three lines of equally valid DTL:

```
<foreach key='i' property='target.{PID:3()}'>
   <assign property='target.{PID:3(i).4}' value='"001"'/>
</foreach>
```

The same `()` convention is also available in BPL.

Counting Fields

If the path refers to a repeating field, you can use `(*)` to return the number of fields. For example, ORCgroup is a repeating field in an HL7 message. The following expression returns the number of ORC groups in the first PID group:

```
HL7.{PIDgrpgrp(1).ORCgrp(*)}
```

Important: To count fields within DTL, use `("*")` instead of `(*)`.

This syntax is also available for collection properties in standard messages.

Accessing the Last Field in a Set of Repeating Fields

If the path refers to a repeating field, you can use `.(-)` to return the last field.

This syntax is also available for collection properties in standard messages.

Appending to a Set of Repeating Fields

If the path refers to a repeating field, you can use `.()` to append another field.

This syntax is also available for collection properties in standard messages.

Curly Bracket { } Syntax

Describes how to use curly bracket { } syntax to access virtual properties.

Where Applicable

You can use this syntax in business rules, search tables, search filters, BPL elements (other than <code> and <sql>), and DTL elements (other than <code> and <sql>).

For curly bracket syntax to resolve, the message structure must be known. If the message structure is unknown in the current context, use square bracket [ ] syntax instead, if possible.

Details

To use curly bracket { } syntax to access a virtual property, use the following syntax:

```
message.{myVirtualPropertyPath}
```

Where:

- *message* is a variable that refers to the current message. The name of this variable depends upon the context.
- *myVirtualPropertyPath* is a virtual property path as described earlier in this reference.

The preceding syntax is equivalent to the following method call:

```
message.GetValueAt("myVirtualPropertyPath")
```

Curly bracket {} syntax is simpler and so is commonly used where available.

Wholesale Copy

When you are using curly bracket {} notation in BPL or DTL, you can copy whole segments, groups of segments, or whole composite fields within a segment. To do so, omit the field part of the virtual property path (as well as the colon separator). Thus, the following DTL <assign> statements are all legal:

```
<assign property='target.{MSH}' value='source.{MSH}' />
<assign property='target.{DG1()}' value='source.{DG1()}' />
<assign property='target.{DG1(1):DiagnosingClinician}' value='"^Bones^Billy"'/>
```

Note: If the source and target types are different, such as copying from an EnsLib.HL7.Message to an EnsLib.EDI.XML.Document, you cannot use the wholesale copy to assign subproperties, even if the structures appear to be parallel. For such a copy, you must assign each leaf node of a structure independently and add a For Each action to process iterations.

The last line of the previous example uses the caret (^) as a component separator character. For details specific to each EDI format, see:

- "Separators" in the reference section of the *Ensemble HL7 Version 2 Development Guide*
- "Separators" in the reference section of the *Ensemble ASTM Development Guide*
- "Separators" in the reference section of the *Ensemble EDIFACT Development Guide*
- "Separators" in the reference section of the *Ensemble X12 Development Guide*

Examples

Here is a BPL example:

```
<if condition='source.{MSH:9.1}="BAR"'>
```

Here is a DTL example:

```
<transform targetClass='EnsLib.HL7.Message'
           targetDocType='2.3:ADT_A01'
           sourceClass='EnsLib.HL7.Message'
           sourceDocType='2.3:ADT_A01'
           create='copy'
           language='objectscript'>
  <assign property='target.{PR1grp(1).PR1:Anesthesiologist.nametype}'
          value='"M"'
          action='set'/>
</transform>
```

Here is an example from a business rule:

```
Document.Name
Document.Parent.DocType
Document.{PIDgrp.PV1grp.PV1:18}
Document.{PIDgrp.PID:PatientName.familylastname}
Document.{ORCgrp(1).OBRuniongrp.OBRunion.OBR:4.3}
```

Here is an excerpt from a search table:

```
<Item DocType="" PropName="MSHControlID"
                 PropType="String:CaseSensitive" >{1:10}</Item>
```

Square Bracket [ ] Syntax

Describes how to use square bracket [ ] syntax to access virtual properties.

Where Applicable

You can use this syntax in business rules, search tables, and search filters. This syntax is available for ASTM, EDIFACT, HL7 version 2, and X12 documents.

Details

To use square bracket [ ] syntax to access a virtual property, use the following syntax:

```
[myVirtualPropertyPath]
```

Where *myVirtualPropertyPath* is a virtual property path as described earlier in this reference, except that field identifiers must be in numeric format.

This syntax finds values in named segments regardless of message structure. If there is more than one instance of the *segment* type in the message, this syntax returns a string that contains all matching values, each value enclosed in <> angle brackets. For example, if the syntax returns multiple values *a*, *b*, and *c*, they appear in a single string like this:

<a><b><c>

When you use square brackets, Ensemble can resolve the numeric path without knowing the specific message structure or schema. This is different from curly brackets `{}` which require you to identify the message structure. For example, a DTL data transformation identifies the message structure of the source and the target messages with attributes of the <transform> element (*sourceDocType* and *targetDocType*) so that you can use curly bracket syntax.

Square bracket syntax supports the repeating field shortcut `()` only in the *field* portion of the property path (*segment*:*field*).

Example

The following excerpt from a search table class shows two valid ways to match all the FT1 segments found in a message that contains several FT1 segments:

```
<Items>
  <Item DocType="" PropName="TransactionAmt">[FT1:12.1]</Item>
  <Item DocType="" PropName="TxType">[FT1:6]</Item>
</Items>
```

Comparison to FindSegmentValues()

The syntax is similar to but does not equate directly with the default behavior of **FindSegmentValues()**. Instead, it modifies the separator and encloses the result in angle brackets, so the square bracket syntax equates to the following method call:

`"<"_msg.FindSegmentValues("`*segment*`:`*field*`",,"><")_">"`

Parenthesis () Syntax

Describes how to use parenthesis () syntax to access virtual properties.

Where Applicable

You can use this syntax in business rules and in DTL transformations.

Details

To use parenthesis syntax to access a virtual property, use the following syntax:

```
message.(multi-valued-property-path)
```

Where:

- *message* is a variable that refers to the current message. The name of this variable depends upon the context.
- *multi-valued-property-path* is a virtual property path that uses the repeating field shortcut () to iterate through every instance of a repeating field, as described earlier in this reference.

The preceding syntax is equivalent to the following method call:

```
message.GetValueAt("multi-valued-property-path")
```

If the syntax returns multiple values *a*, *b*, and *c*, they appear in a single string enclosed in <> angle brackets, like this:

<a><b><c>

Example

For example, in an HL7 routing rule, the syntax `HL7.(NK1():1)` finds the values of the first field in all of the multiple NK1 segments in the HL7 message object represented by the special variable `HL7`.

Angle Bracket <> Syntax

Describes how to use angle bracket <> syntax to access virtual properties.

Where Applicable

You can use this syntax in business rules.

Details

To use angle bracket syntax to access a virtual property, use the following syntax:

```
message<xpathexpression>
```

Where

- *message* is a variable that refers to the current message. The name of this variable depends upon the context.
- *xpathexpression* is an XPath expression.

The preceding syntax is equivalent to the following:

`GetXPathValues(message.stream,"`*context*`|`*expression*`")`

GetXPathValues() is a convenience method in the rules engine. It operates on a message that contains a stream property whose contents are an XML document. The method applies an XPath expression to the XML document within the stream property, and returns all matching values. If the *context*`|` part of the XPath argument is missing, Ensemble searches the entire XML document.

If the syntax returns multiple values *a*, *b*, and *c* they appear in a single string enclosed in <> angle brackets, like this:

<*a*><*b*><*c*>

Example

In an HL7 routing rule, the syntax `HL7.<fracture>` results in a match if the XML document in the message stream property contains the word `fracture`.

Common Settings

This section provides the following reference information:

- Settings for Business Services
- Settings for Routing Processes
- Settings for Business Operations

Settings for Business Services

Provides reference information for settings of business services that handle virtual documents.

Summary

Many business services that handle virtual documents have the following settings:

Group	Settings
Basic Settings	Target Config Names, Doc Schema Category
Additional Settings	Search Table Class, Validation

Doc Schema Category

Category to apply to incoming document type names to produce a complete DocType specification. Combines with the document type name to produce a DocType assignment. This setting may also contain multiple comma-separated type names followed by = and a DocTypeCategory or full DocType values to apply to documents declared as that type.

A trailing asterisk (*) at the end of a given partial type name matches any types beginning with the partial entry.

Note that a DocType assignment may be needed for Validation or Search Table Class indexing.

Search Table Class

Specifies the class to use to index virtual properties in the inbound documents. The default depends on the EDI format, but you can create and use your own search table class. See the chapter "Defining Search Tables."

In either case, be sure that the category given by Doc Schema Category includes the DocType values (if any) in the search table class.

Target Config Names

Configuration items to which to send the received documents.

Validation

Determines the validation that a business host performs when it receives an incoming virtual document.

If the document fails validation, the business service or business operation does not send it. (The details are different for business processes.)

The following table lists the possible values for **Validation**:

Value	Meaning
d	Validation examines the DocType property of the document to see if it has a value.
m	Validation verifies that the document segment structure is well formed, and that it can be parsed using the schema identified in the DocType property of the document.
dm	Both d and m are active.
1	Same as dm, for backward compatibility with previous releases.
(a blank string)	The business host skips validation and routes all documents.

For a longer discussion, see "Basic Validation Options and Logic."

Settings for Routing Processes

Provides reference information for settings of EnsLib.HL7.MsgRouter.RoutingEngine business process, which you use to route most kinds of virtual documents.

Summary

EnsLib.HL7.MsgRouter.RoutingEngine has the following settings:

Group	Settings
Basic Settings	Validation, Business Rule Name
Additional Settings	Act On Transform Error, Act On Validation Error, Alert On Bad Message, Bad Message Handler, Response From, Reply Target Config Names, Response Timeout, Force Sync Send
Development and Debugging	Rule Logging

The remaining settings are common to all business processes. See "Settings for All Business Processes" in *Configuring Ensemble Productions*.

Act On Transform Error

If True, causes errors returned by a transformation to stop rule evaluation and the error to be handled by Reply Code Actions setting.

Act On Validation Error

If True, causes errors returned by validation to be handled by Reply Code Actions setting.

Alert On Bad Message

If True, any document that fails validation automatically triggers an alert.

Bad Message Handler

If the document fails validation, and if the routing process has a configured **Bad Message Handler**, it sends the bad document to this business operation instead of its usual target for documents that pass validation.

See "Defining Bad Message Handlers," earlier in this book.

Business Rule Name

The full name of the routing rule set for this routing process.

Force Sync Send

If True, make synchronous calls for all "send" actions from this routing process. If False, allow these calls to be made asynchronously. This setting is intended to ensure FIFO ordering in the following case: This routing process and its target business operations all have **Pool Size** set to 1, and ancillary business operations might be called asynchronously from within a data transformation or business operation called from this routing process.

If **Force Sync Send** is True, this can cause deadlock if another business process is called by a target that is called synchronously from this routing process.

Note that if there are multiple "send" targets, **Force Sync Send** means these targets will be called one after another in serial fashion, with the next being called after the previous call completes. Also note that synchronous calls are not subject to the Response Timeout setting.

Reply Target Config Names

Specifies a comma-separated list of configuration items within the production to which the business service should relay any *reply* documents that it receives. Usually the list contains one item, but it can be longer. The list can include business processes or business operations, or a combination of both.

This setting takes effect only if the Response From setting has a value.

Response From

A comma-separated list of configured items within the production. This list identifies the targets from which a response may be forwarded back to the original caller, if the caller requested a response.

If a **Response From** string is specified, the response returned to the caller is the first response that arrives back from any target in the list. If there are no responses, an empty "OK" response header is returned.

The **Response From** string also allows special characters, as follows:

- The `*` character by itself matches any target in the production, so the first response from any target is returned. If there are no responses, an empty "OK" response header is returned.
- If the list of targets begins with a `+` character, the responses from all targets return together, as a list of document header IDs in the response header. If none of the targets responds, an empty OK response header is returned.
- If the list of targets begins with a `-` character, only error responses will be returned, as a list of document header IDs in the response header. If none of the targets responds with an error, an empty OK response header is returned.

If this setting value is unspecified, nothing is returned.

Response Timeout

Maximum length of time to wait for asynchronous responses before returning a "timed-out error" response header. A value of -1 means to wait forever. Note that a value of 0 is not useful, because every response would time out. This setting takes effect only if the **Response From** field has a value.

Rule Logging

If logging is enabled controls the level of logging in rules. You can specify the following flags:

- `e`—Log errors only. All errors will be logged irrespective of other flags, so setting the value to 'e' or leaving the value empty will only log errors.
- `r`—Log return values. This is the default value for the setting, and is also automatic whenever the 'd' or 'c' flags are specified.
- `d`—Log user-defined debug actions in the rule. For details on the debug action, see "Adding Actions" in *Developing Business Rules*. This will also include 'r'.
- `c`—Log details of the conditions that are evaluated in the rule. This will also include 'r'.
- `a`—Log all available information. This is equivalent to 'rcd'.

Validation

For allowed values and basic information, see the Validation setting for business services.

If the document fails validation, the routing process forwards the document to its bad message handler, as specified by the Bad Message Handler setting. If there is no bad message handler, the routing process does not route the document, but logs an error. Also see Alert On Bad Message.

Settings for Business Operations

Provides reference information for settings of business operations that handle virtual documents.

Summary

Many business operations that handle virtual documents have the following settings:

Group	Settings
Additional Settings	Search Table Class, Validation

Search Table Class

See the Search Table Class setting for business services.

Validation

See the Validation setting for business services.