# InterSystems® Ensemble

# Developing Ensemble Productions

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 to plan, develop, and test enterprise integration solutions using Ensemble. InterSystems Ensemble is a superset of the features and options of InterSystems Caché. This book refers you to books in both documentation sets.

This book contains the following sections:

- Introduction
- Programming in Ensemble
- Defining Ensemble Messages
- Defining Business Services
- Defining Business Processes
- Defining Business Operations
- Defining Alert Processors
- Defining Data Transformations
- Defining Business Metrics
- Defining an Enterprise Message Bank
- Using the Ensemble Record Mapper
- Using the Complex Record Mapper
- Handling Batches of Records Efficiently
- Less Common Tasks
- Testing and Debugging
- Deploying a Production
- Life Cycle of a Production and Its Parts

For a detailed outline, see the table of contents.

The following books provide related information:

- *Ensemble Best Practices* describes best practices for organizing and developing Ensemble productions.
- *Configuring Ensemble Productions* explains how to perform the configuration tasks related to creating an Ensemble production.
- *Developing BPL Processes* describes how to write Ensemble business processes using the Business Process Language (BPL).
- *Developing DTL Transformations* describes how to write Ensemble data transformations using the Data Transformation Language (DTL).
- *Developing Business Rules* explains how to define business rules that direct business process logic.
- *Developing Workflows* explains how to incorporate human interaction into automated business processes.

For general information, see *Using InterSystems Documentation*.

# 1

# Introduction

This chapter introduces the process of developing productions. It contains the following topics:

- Environmental Considerations
- A Look at a Production Definition
- Development Tools and Tasks
- Available Specialized Classes
- Using Studio in Ensemble

For an introduction to Ensemble concepts and options, see *Introducing Ensemble*. For information on how Ensemble processing works, see the first chapter in *Monitoring Ensemble*.

## 1.1 Environmental Considerations

You can use Ensemble only within an Ensemble-enabled namespace that has a specific web application. When you create classes, you should avoid using reserved package names. The following subsections give the details.

### 1.1.1 Ensemble-enabled Namespaces

An *Ensemble-enabled namespace* is a namespace that has global mappings, routine mappings, and package mappings that make the Ensemble system classes and data available to it. For general information on mappings, see "Configuring Namespaces" in the chapter "Configuring Caché" in the *Caché System Administration Guide*. (You can use the information in that section to see the actual mappings in any Ensemble-enabled namespace; the details may vary from release to release, but no work is necessary on your part.)

Of the system-provided namespaces, only ENSEMBLE and ENSDEMO are Ensemble-enabled.

Once you have successfully installed Ensemble, any new namespace that you create is by default Ensemble-enabled. If you clear the **Make this an Ensemble namespace** check box when creating a namespace, Ensemble creates the namespace with Ensemble disabled.

Important:    Except for the ENSEMBLE and USER namespaces, all system-provided namespaces are overwritten upon reinstallation or upgrade. For this reason, InterSystems recommends that customers always work in the ENSEMBLE namespace, the USER namespace (not suitable for Ensemble use), or in a new namespace that you create. For information on creating a new namespace, see "Configuring Data" in the chapter "Configuring Caché" in the *Caché System Administration Guide*.

## 1.1.2 Web Application Requirement

Also, you can use Ensemble in a namespace only if that namespace has an associated web application that is named
`/csp/`*namespace*, where namespace is the namespace name. (This is the default web application name for a namespace.)
For information on defining web applications, see "Applications" in the *Caché Security Administration Guide*.

## 1.1.3 Reserved Package Names

In any Ensemble-enabled namespace, avoid using the following package names: Demo, Ens, EnsLib, EnsPortal, or CSPX.
These packages are completely replaced during the upgrade process. If you define classes in these packages, you would
need to export the classes before upgrading and then import them after upgrading.

Also, InterSystems recommends that you avoid using any package names that *start* with Ens (case-sensitive). There are
two reasons for this recommendation:

* When you compile classes in packages with names that start with Ens, the compiler writes the generated routines into
  the ENSLIB system database. (The compiler does this because all routines with names that start with Ens are mapped
  to that database.) This means that when you upgrade the instance, thus replacing the ENSLIB database, the upgrade
  removes the generated routines, leaving only the class definitions. At this point, in order to use the classes, it is necessary
  to recompile them.

  In contrast, when you upgrade the instance, it is not necessary to recompile classes in packages with names that do not
  start with Ens.

* If you define classes in packages with names that start with Ens, they are available in all Ensemble-enabled namespaces,
  which may or may not be desirable. One consequence is that it is not possible to have two classes with the same name
  and different contents in different Ensemble-enabled namespaces, if the package name starts with Ens.

# 1.2 A Look at a Production Definition

Although you create and configure productions in the Management Portal, it is instructive to get started by looking at the
definition of a production in Studio. This following shows a simple example of a production that is defined in the ENSDEMO
namespace:

```
Class Demo.FloodMonitor.Production Extends Ens.Production
{

XData ProductionDefinition
{
<Production Name="Demo.FloodMonitor.Production">
  <ActorPoolSize>1</ActorPoolSize>
  <Item Name="Demo.FloodMonitor.BusinessService"
        ClassName="Demo.FloodMonitor.BusinessService"
        PoolSize="1" Enabled="true" Foreground="false" InactivityTimeout="0">
  </Item>
  <Item Name="Demo.FloodMonitor.CustomBusinessProcess"
        ClassName="Demo.FloodMonitor.CustomBusinessProcess"
        PoolSize="1" Enabled="true" Foreground="false" InactivityTimeout="0">
  </Item>
  <Item Name="Demo.FloodMonitor.GeneratedBusinessProcess"
        ClassName="Demo.FloodMonitor.GeneratedBusinessProcess"
        PoolSize="1" Enabled="true" Foreground="false" InactivityTimeout="0">
  </Item>
  <Item Name="Demo.FloodMonitor.BusinessOperation"
        ClassName="Demo.FloodMonitor.BusinessOperation"
        PoolSize="1" Enabled="true" Foreground="false" InactivityTimeout="0">
  </Item>
</Production>
}
}
```

Note the following points:

- The production is a class, specifically is a subclass of Ens.Production.

- The XData ProductionDefinition block holds the configuration information for the production.

- Each <Item> is a business host; these are also called *configuration items*.

- Each business host refers to a class. ClassName specifies the class on which this host is based. This means that when the production creates an instance of this business host, it must create an instance of the specified class.

- The Name of an business host is an arbitrary string. Sometimes, it can be convenient to use the class name for this purpose, as in this example. This convention does not work when you create a large number of business hosts that use the same class.

  It is important to establish naming conventions at an early point during development. See *Ensemble Best Practices*. An absence of naming conventions will lead to confusion.

- The other values in the XData block are all settings. At the top, <ActorPoolSize> is a setting for the production. Within the business host definitions, PoolSize, Enabled, Foreground, and InactivityTimeout are settings for those business hosts.

# 1.3 Development Tools and Tasks

Ensemble productions consist primarily of class definitions and some supporting entities. The process of creating a production can require a small amount of programming or possibly a large amount, depending on your needs. As noted earlier, Ensemble provides graphical tools that enable nontechnical users to create business logic visually. These tools generate class definitions as needed.

While you develop a production, you use both the Management Portal and Studio as follows.

## 1.3.1 Portal Tasks

In the Management Portal, you use wizards to define and compile the following:

- The production class. See *Configuring Ensemble Productions*.

  When you apply changes to the production, Ensemble automatically saves them and compiles the production class.

- BPL business process classes. See *Developing BPL Processes*.

- DTL transformation classes. See *Developing DTL Transformations*.

- Business rule classes. See *Developing Business Rules*.

You also use the Management Portal for the following additional tasks:

- Defining reusable items for use in settings; these include Ensemble credentials, business partners, and so on. See *Configuring Ensemble Productions*.

- Starting and stopping the production. See *Managing Ensemble*.

- Examining the message flow, as part of your testing and debugging process. See *Monitoring Ensemble*.

- Test specific business hosts. See the chapter "Testing and Debugging."

### 1.3.2 Studio Tasks

In Studio, you define and compile the following classes:

- Message classes. See "Defining Ensemble Messages."

- Business service classes. See "Defining Business Services." Note that Ensemble provides business service classes that directly use specific adapters. You might be able to use one of them rather than create your own.

- Custom business process classes. See "Defining Business Processes."

- Business operation classes. See "Defining Business Operations." Note that Ensemble provides business operation classes that directly use specific adapters. You might be able to use one of them rather than create your own.

- Custom data transformation classes. See "Defining Data Transformations."

- Custom adapter classes. See "Less Common Tasks."

Also see other topics in the chapter "Less Common Tasks."

# 1.4 Available Specialized Classes

Ensemble provides many specialized adapter and business host classes that can reduce your development and testing time. For a summary of the most common options, see "Overview of Ensemble Options" in *Introducing Ensemble*. Rather than listing the specific classes, that chapter describes scenarios for which Ensemble provides tools. It provides references to the applicable books.

# 1.5 Using Studio in Ensemble

This section briefly discusses additional Studio features and notes that apply when you are creating an Ensemble production. It contains the following sections:

- Ensemble Features in Studio

- Exporting and Importing Productions

- Notes on Editing the Production Class

## 1.5.1 Ensemble Features in Studio

The Ensemble features of Studio become active whenever Studio is connected to an Ensemble-enabled namespace. These features are as follows:

- An additional tab in the Workspace window

- Additional items in the Other folder in the Workspace window

- An option to generate a project that contains all the code for a production

- Additional wizards for the **File > New** menu, not discussed here

### 1.5.1.1 Production Tab

For an Ensemble-enabled namespace, the Workspace window includes the **Production** tab. This tab lists classes related to Ensemble productions. This has the following folders:

| Folder | All classes in this folder are subclasses of... |
|---|---|
| Adapters | Ens.Adapter, including inbound and outbound adapters |
| Business Operations | Ens.BusinessOperation. |
| Business Processes | Ens.BusinessProcess. |
| Business Services | Ens.BusinessService. |
| Data Transformations | Ens.DataTransform. |
| Messages | Ens.Request or Ens.Response. |
| Productions | Ens.Production. |

### 1.5.1.2 Additional Items in Other

The **Other** folder in the Workspace window can contain additional items that may be unfamiliar to you. These include the following:

- Schema definitions for use with EDI (electronic document interchange) documents. The names of these definitions end in .HL7, .ASTM, .X12, and so on. For information, see *Ensemble Virtual Documents*.

- Dashboard definitions. Their names end in .dfi. For information, see "Creating Dashboards" in *Configuring Ensemble Productions*.

    Note that dashboards are not restricted to Ensemble-enabled namespaces. This item is listed here because .dfi items might be unfamiliar to you.

### 1.5.1.3 Option to Create a Project from a Production

Ensemble makes it easy to create a Studio project from a production, so that you can easily export a production from one system and import it into another. Use the following procedure:

1.  In the **Workspace** window, right-click a production.

2.  Click **Create Project from Production**.
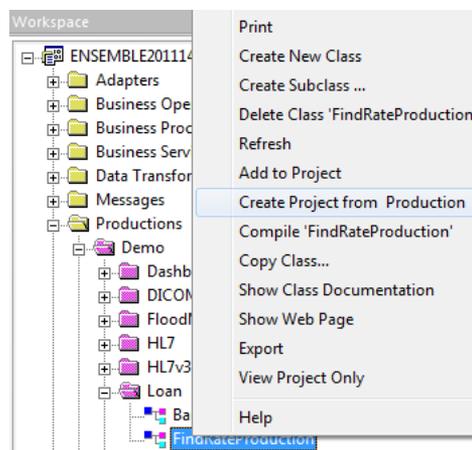
3.  Enter a project name and click **OK**. A project file is saved in the current namespace. The project includes all the detectable items for this production. This includes:

    •   The production class

    •   Each business host class

    •   Each adapter class

    •   Routing rules

    •   HL7 schemas

    •   Subclasses of routing engines

    •   Subtransforms

## 1.5.2 Exporting and Importing Productions

You can either use the management portal or Studio to export and import a production. The management portal makes it easier to include all required components in the export. For information on using the management portal to export and import a production, see Deploying a Production.

To export a production and then export it into another namespace or instance, do the following in Studio.

1.  Create a project from the production. See the previous topic.

2.  Export the project as an XML file.

3.  Import the XML file into another namespace or instance.

For information on importing and exporting projects, see *Using Studio*.

If the production requires data that is stored directly in globals, see "Managing Globals" in *Using Caché Globals*; this provides information on exporting and importing globals.

You can also export productions from the Management Portal. See *Configuring Ensemble Productions*.

## 1.5.3 Notes on Editing the Production Class

It is generally best to edit the production class within the Management Portal. By doing so, you can avoid introducing syntax errors and you take advantage of the type validation that the portal provides.

There are a couple of reasons, however, in which you might want to edit the class in Studio:

•   You might want to change the order of <Item> elements in the XData block.

•   You might want to add other class members, particularly methods, properties, and parameters.

The order of <Item> elements in the XData block controls one of the sorting options on the **Production Configuration** page. On this page, when you sort by number, the items are displayed in the order they are listed in the XData block, which is (initially) the order in which they were added to the production.

# 2
# Programming in Ensemble

This chapter discusses common programming tasks and topics in Ensemble. It includes the following sections:

## 2.1 Introduction

When you create business host classes or adapter classes, you typically implement callback methods, define additional methods as needed, and add or remove settings.

Within a business host or adapter, your methods typically do some or all of the following tasks:

- Get or set values of properties of the business host or adapter.

- Define callback methods. A *callback method* is an inherited method that does nothing by default; your task would be to override and implement this method.

- Execute inherited helper methods of the business host or adapter. *Helper method* is an informal term for a method meant for use by other methods.

  In business operations and business processes, your methods typically invoke inherited methods to send messages to other business hosts within the production.

- Generate log, alert, or trace notifications in response to various conditions that may arise. Ensemble uses the following terminology:

    – *Log entries* are intended to note items such as external, physical problems (for example, a bad network connection).

    Ensemble automatically writes these to the Event Log.

    – *Alerts* are intended to alert users, via an *alert processor* that you define and add to the production.

    Ensemble also automatically writes these to the Event Log.

    – *Trace items* are meant for debugging and diagnostic purposes. You can use these, for example, to locate program errors before deploying the production. Ensemble can write these to the Event Log, to the Terminal, or both.

Not all types of error or activity should necessarily generate these notifications. It is up to the developer to choose which occurrences to record, and how to record them. Note that the Event Log should not register program errors; these should be resolved before the production is released.

# 2.2 Key Principles

It is important to understand the programming practices that are best suited within Ensemble productions. Ensemble business hosts execute in separate processes, which means that you should make sure that:

- If a business host allocates any system resources (such as taking out locks or opening devices), the same business host should release them.

- If a business host starts a transaction, the same business host should complete it or roll it back.

- Any information that is to be shared between business hosts should be carried within a message sent between them (rather than via public variables).

Similar considerations apply to business rules and data transformations.

Also, you must often handle error codes received from Ensemble methods. The Ensemble development framework is designed to allow your custom code to be simple and linear as possible with regard to error codes. For example:

- The **OnProcessInput()** method of business services and the **OnMessage()** and other user-written MessageMap methods of business operations are wrapped by the Ensemble framework, so that you do not need to include any additional error trapping in your code.

- The adapter methods available for use by custom business operation code are guaranteed never to trap out, as are the framework methods available to custom code, such as **SendAlert()** and **SendRequestSync()**. Furthermore, the adapter methods automatically set suitable values for Retry and Suspend when error conditions do occur.

Given these precautions built into the Ensemble framework, InterSystems recommends that for normal circumstances your custom code should simply check the error code of each call, and if it is an error value, quit with that value. The following is an example of this coding style. Note the absence of traps or `Do {} While 0` blocks:

```
Class Test.FTP.FileSaveOperation Extends Ens.BusinessOperation
{
 Parameter ADAPTER = "EnsLib.File.OutboundAdapter";

 Method OnMessage(pRequest As Test.FTP.TransferRequest,
                  Output pResponse As Ens.Response) As %Status
 {
 Set pResponse=$$$NULLOREF
 Set tFilename=..Adapter.CreateTimestamp(pRequest.Filename,"%f_%Q")
 ; file with timestamp should not already exist
 $$$ASSERT('..Adapter.Exists(tFilename))
 Set tSC=..Adapter.PutStream(tFilename,pRequest.StreamIn) Quit:$$$ISERR(tSC) tSC
 Quit $$$OK
 }
}
```

More complicated scenarios are sometimes useful, such as for instance executing a number of SQL statements in a business operation using the SQL adapter and then calling a rollback before returning if any of them fail. However, the coding style in the previous example is the best practice in simple circumstances.

Later chapters in this book provide additional principles for specific kinds of business hosts.

# 2.3 Passing Values by Reference or as Output

If you are not familiar with passing values by reference or output, this section is intended to orient you to this practice.

Many Ensemble methods return at least two values: a status (an instance of %Status) *and* a response message or other returned value. Typically the response message is returned by reference or as output. If a value is returned by reference or as output, that means:

- When you define the method, the method must set the corresponding variable.

- When you invoke the method, you must include a period before the corresponding argument.

The following examples demonstrate these points.

## 2.3.1 Typical Callback Method

The following shows the signature of a typical callback method:

```
method OnRequest(request As %Library.Persistent, Output response As %Library.Persistent) as %Status
```

The keyword Output indicates that the second argument is meant to be returned as output. In your implementation of this method, you would need to do the following tasks, in order to satisfy the method signature:

1. Set a variable named response equal to an appropriate value. This variable must have a value when the method completes execution.

2. End with the Quit command, followed by the name of a variable that refers to an instance of %Status.

For example:

```
Method OnRequest(request As %Library.Persistent, Output response As %Library.Persistent) as %Status
{
    //other stuff
    set response=myObject
    set pSC=..MyMethod() ; returns a status code
    quit pSC
}
```

This example discusses a value returned as output, but the details are the same for a value passed by reference.

### 2.3.2 Typical Helper Method

The following shows the signature of a typical inherited helper method:

```
method SendRequestSync(pTargetDispatchName As %String,
                       pRequest As %Library.Persistent,
                       ByRef pResponse As %Library.Persistent,
                       pTimeout As %Numeric = -1,
                       pDescription As %String = "") as %Status
```

The keyword `ByRef` indicates that the third argument is meant to be returned by reference. To invoke this method, you would use the following:

```
 set sc=##class(pkg.class).SendRequestSync(target,request,.response,timeout,description).
```

Notice the period before the third argument.

This example discusses a value passed by reference, but the details are the same for a value returned as output.

# 2.4 Adding and Removing Settings

To provide new settings for a production, a business host, or an adapter, modify its class definition as follows:

1.  Add a property for each configuration setting you wish to define.

2.  Add a class parameter called *SETTINGS* to the class.

3.  Set the value of *SETTINGS* to be a comma-delimited list of the names of the properties you have just defined. For example:

    ```
    Property foo As %String;

    Property bar As %String;

    Parameter SETTINGS = "foo,bar";
    ```

    See the following section for additional details for *SETTINGS*.

The `foo` and `bar` settings now automatically appear in the configuration display on the **Production Configuration** page whenever an item of that class is selected for configuration.

To remove an inherited configuration setting, list the property in the *SETTINGS* class parameter, preceded by a hyphen (-). For example:

```
Parameter SETTINGS = "-foo";
```

# 2.5 Specifying Categories and Controls for Settings

By default, a setting is displayed in the **Additional Settings** category on the **Details** tab when you configure a production. By default, the setting provides one of the following input mechanisms, depending on the property on which the setting is based:

*   In most cases, the setting is displayed with a plain input field.

*   If the property specifies the *VALUELIST* parameter, the setting is displayed with a drop-down list.

*   If the property is of type %Boolean, the setting is displayed as a check box.

In all cases, the InitialExpression of the property (if specified) controls the initial state of the input field.

You can override both the location and control type. To do so, include the setting name in *SETTINGS* as follows:

```
Parameter SETTINGS = "somesetting:category:control";
```

Or:

```
Parameter SETTINGS = "somesetting:category";
```

```
Parameter SETTINGS = "somesetting::control";
```

Where *category* and *control* indicate the category and control to use, respectively. The following subsections provide details.

You can include multiple settings, as follows:

```
Parameter SETTINGS = "setting1:category:control1,setting2:control2:editor,setting3:category:control3";
```

For example (with disallowed line breaks included):

```
Parameter SETTINGS = "HTTPServer:Basic,HTTPPort:Basic,SSLConfig:Connection:sslConfigSelector,
ProxyServer:Connection,ProxyPort:Connection,ProxyHTTPS:Connection,
URL:Basic,Credentials:Basic:credentialsSelector,UseCookies,ResponseTimeout:Connection"
```

## 2.5.1 Category for a Setting

*category* is one of the following case-sensitive, literal values:

- `Info` — Places the setting into the **Information Settings** category.

- `Basic` — Places the setting into the **Basic Settings** category.

- `Connection` — Places the setting into the **Connection Settings** category.

- `Additional` — Places the setting into the **Additional Settings** category (the default).

- `Alerting` — Places the setting into the **Alerting Control** category.

- `Dev` — Places the setting into the **Development and Debugging** category.

Or use your own category name.

## 2.5.2 Control for a Setting

*control* specifies the name of a specific Zen control to use when viewing and modifying the setting in the **Production Configuration** page. Use `selector` or use the name of a class in the package EnsPortal.Component. See "Examples for Settings Controls."

To supply extra values to your component in order to display a suitable set of options, append a set of name-value pairs as follows:

```
myControl?Prop1=Value1&Prop2=Value2&Prop3=Value3
```

Where:

- *Prop1*, *Prop2*, *Prop3*, and so on are names of properties of the control. For example: `multiSelect`

- *Value1*, *Value2*, *Value3*, and so on are the corresponding values.

  Use the syntax *@propertyname* to refer to a property of the Javascript zenPage object. The supported variables are these:

  – *currHostId* refers to the ID of the current Ens.Config.Item.

---

– *productionId* refers to the ID of the Ens.Config.Production in use.

Use \@ to indicate the character @ as is.

To pass values to the context property of a control, enclose the value in curly braces. See the subsection.

### 2.5.2.1 Passing Values to the context Property of a Control

Ensemble uses the general-purpose selector component, which you can also use; to do so, specify *control* as selector and then append property/value pairs.

The selector component allows almost any list of data or options to be displayed to the user and which allows the user to type data if needed. To configure the component in this way, use the following syntax to set the context property of the control:

```
context={ContextClass/ContextMethod?Arg1=Value1&Arg2=Value2&Arg3=Value3}
```

Where:

- *ContextClass* is the name of the context class. This must be a subclass of %ZEN.Portal.ContextSearch or Ens.ContextSearch. See "Examples for Settings Controls."

- *ContextMethod* is a method in that class. This method provides the data from which the user chooses values.

- *Arg1*, *Arg2*, *Arg3*, and so on are the names of multidimensional arguments of that method.

- *Value1*, *Value2*, *Value3*, and so on are values for those arguments. The rules for values are the same as given previously (for example, you can use *@propertyname*).

**Note:** The selector component also has a property named multiSelect. By default, the user can select only one item. To enable the user to select multiple items, include a property/value pair as follows: multiSelect=1.

The argument names and values must be URL encoded. For example, replace % with %25.

### 2.5.2.2 Examples for Settings Controls

The following list gives examples for *control*, organized by the kind of data that they enable the user to select.

These examples use Ensemble classes. Many of these examples use the Ens.ContextSearch class, which provides a large number of useful methods. If this list does not cover your scenario, see the class documentation for Ens.ContextSearch to determine whether any of the existing methods supply the data you need. If that class does not cover your scenario, you can create your own subclass of Ens.ContextSearch.

**business hosts in the same production**

```
selector?multiSelect=1&context={Ens.ContextSearch/ProductionItems?targets=1&productionName=@productionId}
```

Or, if the user should choose only one business host:

```
selector?context={Ens.ContextSearch/ProductionItems?targets=1&productionName=@productionId}
```

**business partners**

```
partnerSelector
```

**business rules**

```
ruleSelector
```

**character sets**

```
selector?context={Ens.ContextSearch/CharacterSets}
```

**credential sets**

```
credentialsSelector
```

**framing**

```
selector?context={Ens.ContextSearch/getDisplayList?host=@currHostId&prop=Framing}
```

**local interfaces**

```
selector?context={Ens.ContextSearch/TCPLocalInterfaces}
```

**schema categories appropriate for a specific business host**

```
selector?context={Ens.ContextSearch/SchemaCategories?host=classname}
```

Where *classname* is the class name of the business host. For example:

```
selector?context={Ens.ContextSearch/SearchTableClasses?host=EnsLib.MsgRouter.RoutingEngineST}
```

**search table classes appropriate for a specific business host**

```
selector?context={Ens.ContextSearch/SearchTableClasses?host=classname}
```

Where *classname* is the class name of the business host. For example:

```
selector?context={Ens.ContextSearch/SearchTableClasses?host=EnsLib.EDI.ASTM.Service.Standard}
```

**SSL configurations**

```
sslConfigSelector
```

# 2.6 Specifying Default Values for Settings

As you define business host classes (and possibly adapter classes), you should consider how to control the default values for any settings of those items. Ensemble can take the default value for a setting from one of three sources:

- The production definition.

- The values defined for the Ensemble instance, but stored outside the production. For information, see "Defining Production Defaults" in *Configuring Ensemble Productions*.

- The default value for the property as defined in the host class. In this case, the default value is determined by the InitialExpression property keyword.

Some settings are dependent on the environment, such as TCP/IP addresses or file paths; typically you configure these settings to have their source outside the production, while others, such as ReplyCodeActions are design decisions and most likely you develop your application to retrieve these from the production definition.

You can develop your production to have configuration settings come from different sources. The primary purpose is to make it easier to move productions from one Ensemble instance to another, such as from *test* to *live*.

Once you define a production, you can change the source of both production and business host settings on the **Production Configuration** page of the Management Portal. See *Configuring Ensemble Productions* for details.

The use of default settings allows you to define production and business host settings outside the production definition where you can preserve them during a production upgrade. To facilitate updating productions or moving productions from one system to another, you can omit settings and take their values from a structure that is installed on the system. When a setting is missing, Ensemble retrieves the default setting from outside the production definition if one exists.

See the descriptions for the following methods in the Ens.Director class entry of the *Class Reference* for programming details:

- **GetProductionSettingValue()**

- **GetProductionSettings()**

# 2.7 Accessing Properties and Methods from a Business Host

When you define a method in a business host class, you might need to access properties or methods of that class or of the associated adapter. This section briefly describes how to do these things.

Within an instance method in a business host, you can use the following syntaxes:

- `..bushostproperty`

  Accesses a setting or any other property of the business host. (Remember that all settings are properties of their respective classes.)

- `..bushostmethod()`

  Accesses an instance method of the business host.

- `..Adapter.adapterproperty`

  Accesses a setting or any other property of the adapter. (Note that every business host has the property Adapter. Use that property to access the adapter and then use dot syntax to access properties of the adapter.)

- `..Adapter.adaptermethod()`

  Accesses an instance method of the adapter.

# 2.8 Accessing Production Settings

You might need to access a setting of the production. To do so, use the macro `$$$ConfigProdSetting`. For example, `$$$ConfigProdSetting("mySetting")` retrieves the value of the production setting called `mySetting`. InterSystems suggests you wrap this macro in a $GET call for safety; for example:

```
set myvalue=$GET($$$ConfigProdSetting("mySetting"))
```

For details about $GET, see the *Caché ObjectScript Reference*.

Also see "Using Ens.Director to Access Settings" in the chapter "Advanced Topics."

# 2.9 Choosing How to Send Messages

In business operations and business processes, your methods typically invoke inherited methods to send messages to other business hosts within the production. This section discusses the options.

## 2.9.1 Synchronous and Asynchronous Sending

When you define business service, business process, and business operation classes, you specify how to send a request message from that business host. There are two primary options:

- *Synchronously* — The caller stops all processing to wait for the response.

- *Asynchronously* — The caller does not wait; immediately after sending the request the caller resumes other processing. When sending a request asynchronously, the caller specifies one of two options regarding the response to this request:

    - Ask to receive the response when it arrives.

    - Ignore the possibility of a response.

The choice of how to send a message is *not* recorded in the message itself and is not part of the definition of the message. Instead, this is determined by the business host class that sends the message.

## 2.9.2 Deferred Sending

In addition to the straightforward alternatives of synchronous (wait) and asynchronous (do not wait), it is possible to send messages outside Ensemble using a mechanism called *deferred response*.

Suppose a business process wishes to invoke an action outside Ensemble. It sends a request to a business operation, which performs the invocation and returns the response. The business process is the intended recipient of any response; the business operation is simply the means by which the request goes out and the response comes in. The business operation will relay a response back if the business process made the request synchronously, or if it made the request asynchronously with asynchronous response requested. The following diagram summarizes this mechanism.



Now suppose the business operation that receives a request from a business process has been written to use the deferred response feature. The original sender is unaware of the fact that the response is going to be deferred by the business operation. Deferring the response is a design decision made by the developer of the business operation. If the business operation does in fact defer the response, when the original sender receives the response at the end of the deferral period, it is unaware that the response was ever deferred.

A business operation defers a response by calling its **DeferResponse**() method to generate a token that represents the original sender and the original request. The business operation must also find a way to communicate this token to the external entity, which is then responsible for including this token in any later responses to Ensemble. For example, if the external destination is email, a business operation can include the token string in the subject line of the outgoing email. The entity receiving this email can extract this token from the request subject line and use it in the response subject line. In the following diagram, the item "t" represents this token.

Between the time when the business operation defers the request, and when the response is finally received by the original sender, the request message has a status of Deferred. After the original sender receives the corresponding response, the request message status changes from Deferred to Completed.

An incoming event in response to the request can be picked up and returned to the original sender by any business host in the production. Exactly where the event arrives in Ensemble depends on the design of the production; typically, it is the task of a business service to receive incoming events from outside Ensemble. The business host that receives the incoming event must also receive the deferred response token with the event. The business host then calls its **SendDeferredResponse**() method to create the appropriate response message from the incoming event data and direct this response to the original sender. The original sender receives the response without any knowledge of how it was returned. The following figure shows a request and its deferred response.



# 2.10 Generating Event Log Entries

The Event Log is a table that records events that have occurred in the production running in a given namespace. The Management Portal provides a page that displays this log, which is intended primarily for system administrators, but which is also useful during development. (For details on this page, see *Monitoring Ensemble Productions*.)

The primary purpose of the Event Log is to provide diagnostic information that would be useful to a system administrator in case of a problem while the production is running.

Ensemble automatically generates Event Log entries, and you can add your own entries. Any given event is one of the following types: Assert, Info, Warning, Error, and Status. (The Event Log can also include alert messages and trace items, discussed in the next sections.)

To generate Event Log entries:

1. Identify the events to log.

   Not all types of error or activity should necessarily generate Event Log entries. You must choose the occurrences to note, the type to use, and the information to record. For example, Event Log entries should appear in case of an external, physical problem, such as a bad network connection.

   The Event Log should not register program errors; these should be resolved before the production is released.

2. Modify the applicable parts of the production (typically business host classes) to generate Event Log entries in ObjectScript or in Basic, as described in the following subsections.

**Important:** If you need to notify users actively about certain conditions or events, use alerts, which are discussed in the next section and in "Defining Alert Processors," later in this book.

## 2.10.1 Generating Event Log Entries in ObjectScript

Within business host classes or other code used by a production, you can generate Event Log entries in ObjectScript. To do so, use any of the following macros. These macros are defined in the Ensemble.inc include file, which is automatically included in Ensemble system classes:

| Macro | Details |
|---|---|
| $$$LOGINFO(*message*) | Writes an entry of type Info. Here and later in this table, *message* is a string literal or an ObjectScript expression that evaluates to a string. |
| $$$LOGERROR(*message*) | Writes an entry of type Error. |
| $$$LOGWARNING(*message*) | Writes an entry of type Warning. |
| $$$LOGSTATUS(*status_code*) | Writes an entry of type Error or Info, depending on the value of the given *status_code*, which must be an instance of %Status. |
| $$$ASSERT(*condition*) | Writes an entry of type Assert, if the argument is false. *condition* is an ObjectScript expression that evaluates to true or false. |
| $$$LOGASSERT(*condition*) | Writes an entry of type Assert, for any value of the argument. *condition* is an ObjectScript expression that evaluates to true or false. |

The following shows an example with an expression that combines static text with the values of class properties:

```
$$$LOGERROR("Awaiting connect on port "_..Port_" with timeout "_..CallInterval)
```

The following example uses an ObjectScript function:

```
$$$LOGINFO("Got data chunk, size="_$length(data)_"/"_tChunkSize)
```

## 2.10.2 Generating Event Log Entries in Basic

To generate Event Log entries in Caché Basic, use the following syntax:

```
"Ens.Util.Log".LogMethodName(Me.%ClassName(1),CurrentMethod,arg)
```

Where *LogMethodName* is one of the method names in the following table, *arg* is the argument for that method, and *CurrentMethod* is the name of the current method.

| Method | Details |
|---|---|
| LogInfo | Writes an entry of type Info. The argument is a string literal or an ObjectScript expression that evaluates to a string. |
| LogError | Writes an entry of type Error. The argument is a string literal or an ObjectScript expression that evaluates to a string. |
| LogWarning | Writes an entry of type Warning. The argument is a string literal or an ObjectScript expression that evaluates to a string. |
| LogStatus | Writes an entry of type Error or Info, depending on the value of argument. |
| LogAssert | Writes an entry of type Assert, if the argument is false. The argument is an ObjectScript expression that evaluates to true or false. |

For example:

```
"Ens.Util.Log".LogInfo(Me.%ClassName(1),MyMethod,"log this")
```

# 2.11 Generating Alerts

An *alert* sends notifications to applicable users while an Ensemble production is running, in the event that an alert event occurs. The intention is to alert a system administrator or service technician to the presence of a problem. Alerts may be delivered via email, text pager, or another mechanism. All alerts *also* write messages to the Ensemble Event Log, with the type Alert.

The Ensemble alert mechanism works as follows:

- When you create business host classes for the production, include code that:

  1. Detects undesirable conditions or other circumstances that a user must address.

  2. Generates alerts on those occasions.

- You define and configure an *alert processor*, which is a business host, named Ens.Alert. The alert processor can optionally manage the alert to track the process of resolving the event. For details on defining an alert processor, see Defining an Alert Processor. Any production can include no more than one alert processor.

In a business host class (other than a BPL process class), do the following to generate an alert:

1. Create an instance of Ens.AlertRequest.

2. Set the AlertText property of this instance. Specify it as a string that provides enough information so that the technician has a good idea of how to address the problem.

3. Invoke the **SendAlert()** method of the business host class. This method runs asynchronously and thus does not delay the normal activities of the business host.

For example, in ObjectScript:

```
set alert=##Class(Ens.AlertRequest).%New()
set msg="Suspended HL7 Message "_pRequest.%Id()
  _" because ACK.MSA.Acknowledgment Code = '"_tAcknowledgmentCode_"'"
set alert.AlertText=msg
do ..SendAlert(alert)
```

**Note:** For information on generating alerts in BPL, see *Developing BPL Processes*.

# 2.12 Adding Trace Elements

*Tracing* is a tool for use primarily during development. You add trace elements so that you can see the behavior of various elements in a production, for the purpose of debugging or diagnosis. To add trace elements to a production, you identify the areas in your code (typically business host classes) where you would like to see runtime information. In those areas, you add lines of code that (potentially) write trace messages. Note that these are messages only in a general sense; trace messages are simply strings and are unrelated to Ens.Message and its subclasses.

In most cases, you can define two kinds of trace elements: user elements and system elements. In most cases, it is more appropriate to define user trace elements.

**Note:**     For information on writing trace elements in BPL, DTL, or business rules, see *Developing BPL Processes*, *Developing DTL Transformations*, and *Developing Business Rules*.

Also, for information on enabling tracing, see the chapter "Enabling Tracing" in *Monitoring Ensemble*.

## 2.12.1 Writing Trace Messages in ObjectScript

To write trace messages in ObjectScript, use the following lines of code:

- To write a user trace message:

  ```
  $$$TRACE(trace_message)
  ```

  Where *trace_message* is a string containing useful information about the context in which you add this line of code.

- To write a system trace message (less common):

  ```
  $$$sysTRACE(trace_message)
  ```

  You might see $$$sysTRACE in Ensemble system code, but the appropriate choice for your own business host classes is generally $$$TRACE.

For example:

```
$$$TRACE("received application for "_request.CustomerName)
```

## 2.12.2 Writing Trace Messages in Basic

To write trace messages in Caché Basic, use the following lines of code:

- To write a user trace message:

  ```
  "Ens.Util.Trace".WriteTrace("user",Me.%ClassName(1),CurrentMethod,arg)
  ```

  Where *CurrentMethod* is the name of the current method and *arg* is the string that you want written to the console.

- To write a system trace message (less common):

  ```
  "Ens.Util.Trace".WriteTrace("system",Me.%ClassName(1),CurrentMethod,arg)
  ```

## 2.12.3 Writing Trace Messages in BPL or DTL

To write user trace messages in a BPL business process or in a DTL data transformation, use the <trace> element. See the *Ensemble Business Process Language Reference* or the *Ensemble Data Transformation Language Reference*.

# 3
# Defining Ensemble Messages

This chapter describes how to define the classes that define Ensemble message bodies. It contains the following sections:

- Introduction
- Creating a Simple Message Body Class
- Creating a Complex Message Body Class

## 3.1 Introduction

A message body can be *any persistent object*.

In practice, you often create a subclass of Ens.Request or Ens.Response and add properties. This creates the *standard message body*. If you use these classes, you have easy access to the various built-in features for viewing the contents of messages from the Management Portal. These features help developers and administrators detect errors in a running production, especially if the production uses message content to determine where the message should be sent.

Some electronic documents — Electronic Data Interchange (EDI) formats such as HL7 or X12 — contain data that is arbitrarily long and complex. In this case, it is better to use an alternative class, a class that represents an Ensemble *virtual document*. In this case, the message body does not have a set of properties to contain the message contents. For details, see *Ensemble Virtual Documents*.

Most of the examples in this book assume a standard message body, with a relatively small number of message properties.

## 3.2 Creating a Simple Message Body Class

To create a message class (to be used as the message body), create a class that:

- Extends either Ens.Request or Ens.Response.
- Contains properties as needed to represent elements of data to be carried in the message.

The following shows a simple example:

```
Class Demo.Loan.Msg.CreditRatingResponse Extends Ens.Response
{

Property TaxID As %String;

Property CreditRating As %Integer;

}
```

The class can also contain methods. For example:

```
Class Demo.Loan.Msg.Application Extends Ens.Request{

Property Amount As %Integer;
Property Name As %String;
Property TaxID As %String;
Property Nationality As %String;
Property BusinessOperationType As %String;
Property Destination As %String;

Method RecordNumber() As %String
{
  If ..%Id()="" Do ..%Save()
  Quit ..%Id()
}

Method GetRecordNumberText(pFormatAsHTML As %Boolean = 0) As %String
{
  Set tCRLF=$s(pFormatAsHTML:"<br>",1:$c(13,10))
  Set tText=""
  Set tText=tText_"Your loan application has been received,"_tCRLF
  Set tText=tText_"and is being processed."_tCRLF
  Set tText=tText_"Your record number is "_..RecordNumber()_"."_tCRLF
  Set tText=tText_"You'll receive a reply from FindRate"_tCRLF
  Set tText=tText_"within 2 business days."_tCRLF
  Set tText=tText_"Thank you for applying with FindRate."_tCRLF
  Quit tText
}

}
```

If you create a message class that does not extend Ens.Request or Ens.Response:

• Base the class on a persistent class.

• Optionally include %XML.Adaptor and either Ens.Util.RequestBodyMethods or Ens.Util.ResponseBodyMethods as superclasses. The %XML.Adaptor class provides support for displaying the message in XML form in the Management Portal. The Ens.Util.RequestBodyMethods and Ens.Util.ResponseBodyMethods classes provide additional support for displaying and working with the message in the Management Portal.

# 3.3 Creating a Complex Message Body Class

In the previous example, the message body class contained only simple properties. In some cases, you may need to define properties that use other classes. If so, you should carefully consider what to do when you purge message bodies (as described in *Managing Ensemble*).

When you purge message bodies, Ensemble deletes only the specific message body object. For example, consider the following message class:

```
Class MyApp.Messages.Person Extends Ens.Response
{
Property Name As %String;

Property MRN As %String;

Property BirthDate As %Date;

Property Address As MyApp.Messages.Address;

}
```

The `Address` class is as follows:

```
Class MyApp.Messages.Address Extends %Persistent
{
Property StreetAddress As %String;

Property City As %String;

Property State As %String;

Property ZIP As %String;

}
```

In this case, if you purge message bodies, Ensemble deletes instances of `MyApp.Messages.Person`, but does not delete instances of `MyApp.Messages.Address`.

If your message body class uses other classes as properties and if your application requires that any referenced objects should also be purged, use one of the following approaches:

- Make sure that the referenced classes are serial. For example, redefine the `Address` class as follows:

  ```
  Class MyApp.Messages.Address Extends %SerialObject
  {
  ...
  }
  ```

  In this case, the data for the `Address` class is stored as part of the `Person` class (and is thus automatically purged at the same time).

- Define the property as a suitable relationship. See "Persistent Behavior of Relationships" in the chapter "Relationships" in *Using Caché Objects*.

- Add a delete trigger or **%OnDelete()** method to the message class so that this class deletes the appropriate records in the referenced classes.

- Optionally include %XML.Adaptor as a superclass so that the properties defined in the referenced class can be displayed in the Management Portal.

# 4

# Defining Business Services

This chapter describes how to define business service classes. It contains the following sections:

- Introduction
- Key Principles
- Defining a Business Service Class
- Implementing the OnProcessInput() Method
- Sending Request Messages
- Processing Only One Event Per Call Interval

**Tip:** Ensemble provides specialized business service classes that use specific inbound adapters, and one of those might be suitable for your needs. If so, no programming would be needed. For a partial list, see "Connectivity Options" in *Introducing Ensemble*.

## 4.1 Introduction

A business service is responsible for accepting requests from external applications into Ensemble. The following figure shows how it works:
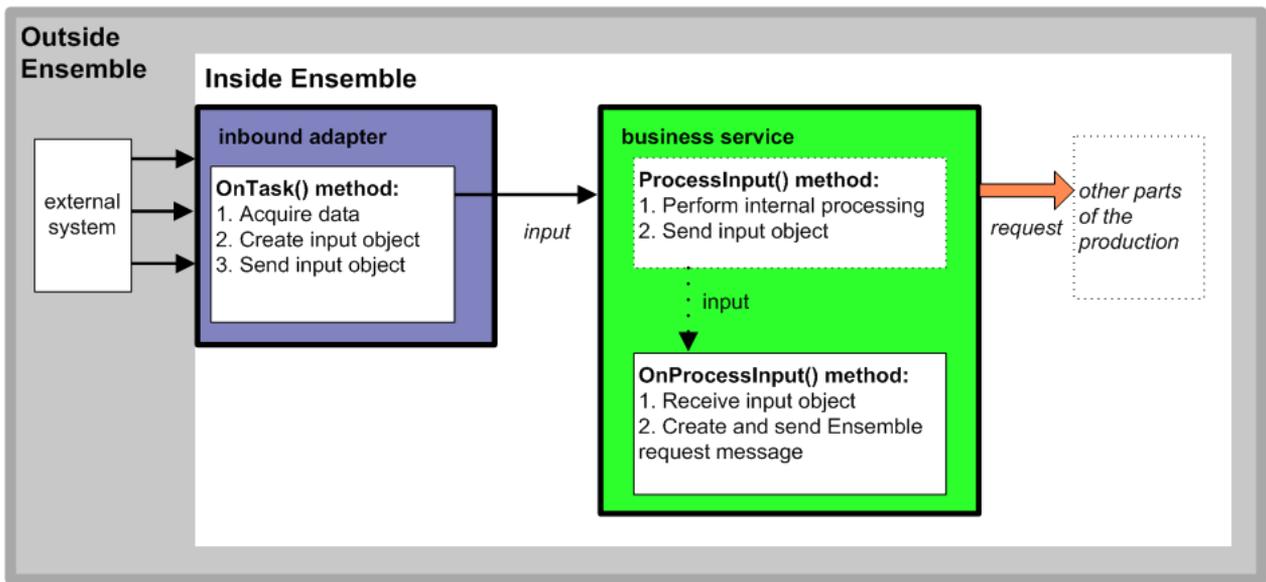
Note that this figure shows only the input flow of data, not the optional response.

A business service is responsible for the following activities:

- Waiting for a specific external event (such as notification from an application, receipt of a TCP message, etc.).

- Reading, parsing, and validating the data accompanying such an event,

- Returning, if required, an acknowledgment to the external application indicating that the event was received.

- Creating an instance of an Ensemble request message and forwarding it on to the appropriate business process or business operation for processing.

The purpose of a business service is usually to receive data input. In most cases, a business service has an inbound adapter associated with it. However, in some cases an adapter is not required, either because an application is capable of sending Ensemble request messages into the service or because the business service has been written to handle external calls of a particular kind, for example from a composite application. A business service of this type is called an *adapterless* business service.

When a business service has an inbound adapter, it is in the data "pulling" (as opposed to "pushing") mode. In this mode, the business service polls the adapter at regular intervals to see if it has data. Meanwhile, if the adapter encounters input data at any time, it calls the business service to process the input.

When a business service does not have an adapter, it does not "pull" data. Instead, client applications call the business service and tell it to process input (this is a data "pushing" mode).

# 4.2 Key Principles

First, be sure to read the chapter "Programming in Ensemble."

Within a business service, you can access properties and methods of the associated adapter, which is available as the Adapter property of the business service. This means that you can alter the default behavior of the adapter; it may or may not be appropriate to do so. It is useful to remember the principle of encapsulation. The idea of encapsulation is that the adapter class should be responsible for the technology-specific logic, while the business service class should be responsible for the production-specific logic.

If you find that it is necessary to greatly or frequently alter the behavior of an adapter class from within a business service class, it might be more appropriate to create a customized subclass of the adapter class. See "Less Common Tasks."

This principle also applies to business operations.

# 4.3 Defining a Business Service Class

To create a business service class, define a class as follows:

- Your class must extend Ens.BusinessService (or a subclass).

- In your class, the *ADAPTER* parameter must equal the name of the adapter class for this business service to use.

    **Tip:** If you simply want a business service to wake up and run periodically without concern for events outside Ensemble, use the adapter class Ens.InboundAdapter.

- Your class must implement the OnProcessInput() method, as described in "Implementing the OnProcessInput() Method."

- Your class can add or remove settings. See "Adding and Removing Settings," earlier in this book.

- Your class can implement any or all of the startup and teardown methods. See "Overriding Start and Stop Behavior," later in this book.

- Your class can contain methods to accomplish work internal to itself.

**Note:** Studio provides a wizard that you can use to create a class to get started. To access this wizard, click **File —> New** and then click the **Production** tab. Then click **Business Service** and click **OK**.

For examples of business service classes, see the adapter guides.

# 4.4 Implementing the OnProcessInput() Method

Within your business service class, your **OnProcessInput()** method can have the following generic signature:

```
Method OnProcessInput(pInput As %RegisteredObject, pOutput As %RegisteredObject) As %Status
```

Here *pInput* is the input object that the adapter will send to this business service, and *pOutput* is the output object.

First look at the adapter class that you have selected. InterSystems recommends that you edit the **OnProcessInput()** method signature to use the specific input argument needed with the adapter.

The **OnProcessInput()** method should do some or all of the following:

1. Optionally set properties of the business service class (at any appropriate time). The business service property of greatest interest is %WaitForNextCallInterval. Its value controls how frequently Ensemble invokes the **OnTask()** method of the adapter.

    For other properties, see the *Class Reference* for Ens.BusinessService.

2. Validate, if necessary, the input object.

3. Examine the input object and decide how to use it.

4. Create an instance of a request message class, which will be the message that your business service sends.

For information on creating messages, see the chapter "Creating Ensemble Messages."

5. For the request message, set its properties as appropriate, using values in the input object.

6. Determine where you want to send the request message. When you send the message, you will need to use the configuration name of a business host within the production.

7. Send the request message to a destination within the production (a business process or business operation). See the next section.

8. Make sure that you set the output argument (pOutput). Typically you set this equal to the response message that you have received. This step is required.

9. Return an appropriate status. This step is required.

# 4.5 Sending Request Messages

In your business service class, your implementation of **OnProcessInput()** should send a request message to some destination within the production. To so, call one of the following instance methods of the business service class, as appropriate for your needs:

- **SendRequestSync()** sends a message synchronously (waits for a response). For details, see "Using the SendRequestSync() Method."

- **SendRequestAsync()** sends a message asynchronously (does not wait for a response). For details, see "Using the SendRequestAsync() Method."

- **SendDeferredResponse()** sends a response that was previously deferred. This method is less commonly used. For details, see "Using the SendDeferredResponse() Method."

Each of these methods returns a status, an instance of %Status.

These methods are also defined — with the identical method signatures — in Ens.BusinessProcess and Ens.BusinessOperation, although their internals are different in those classes. This means that you can invoke these instance methods from within your business process and business operation classes.

## 4.5.1 Using the SendRequestSync() Method

To send a synchronous request, use the **SendRequestSync()** method as follows:

```
Set tSC = ..SendRequestSync(pTargetDispatchName, pRequest, .pResponse, pTimeout)
```

Where:

- *pTargetDispatchName* — The configuration name of the business process or business operation to which the request is sent.

- *pRequest* — A request message. See "Defining Ensemble Messages."

- *pResponse* — (By reference) A response message. This object receives the data returned by the response.

- *pTimeout* — (Optional) The number of seconds to wait for a response. The default is –1 (wait forever).

This method returns a status, an instance of %Status.

If no response is expected, you can use **SendRequestAsync()** instead of **SendRequestSync()**.

## 4.5.2 Using the SendRequestAsync() Method

To send an asynchronous request, use the **SendRequestAsync()** method as follows:

```
Set tSC = ..SendRequestAsync(pTargetDispatchName, pRequest)
```

Where:

- *pTargetDispatchName* — The configuration name of the business process or business operation to which the request is sent.

- *pRequest* — A request message. See "Defining Ensemble Messages."

This method returns a status, an instance of %Status.

## 4.5.3 Using the SendDeferredResponse() Method

All business hosts support the **SendDeferredResponse()** method. This method permits a business host to participate in the Ensemble deferred response mechanism. The business host identifies a previously deferred request, creates the actual response message, and sends this response to the business host that originated the request. See "Using Deferred Sending" in the chapter "Programming in Ensemble."

This topic describes the role of a business service in this mechanism. Suppose an incoming event arrives in Ensemble along with a deferred response token, and suppose the arrival point for this event is a business service. This business service then calls **SendDeferredResponse()** to create a response and direct it to the caller that originated the request. The **SendDeferredResponse()** call looks like this:

```
Set sc = ..SendDeferredResponse(token, pResponseBody)
```

Where:

- *token* — A string that identifies the deferred response so that the caller can match it to the original request. The business service obtains the token string through some mechanism unique to the production.

  For example, if the external destination is email, when sending a request for which it is willing to receive a deferred response, a business operation can include the token string in the subject line of the outgoing email. The entity receiving this email can extract this token from the request subject line and use it in the response subject line. This preserves the token so that the business service receiving the response email can use it in a subsequent call to **SendDeferredResponse()**.

- *pResponseBody* — A response message. This object receives the data returned by the response. See "Defining Ensemble Messages."

This method returns a status, an instance of %Status.

# 4.6 Processing Only One Event Per Call Interval

If you want the business service to process only one event per call interval, set the %WaitForNextCallInterval property to 1 (true) in your implementation of **OnProcessInput()**:

```
set ..%WaitForNextCallInterval=1
```

This restricts the business service to processing only one input event per CallInterval, even when multiple input events exist.

This information applies to business services that use an adapter that have a property named CallInterval and that use that property as a polling interval.

# 5

# Defining Business Processes

Business processes are responsible for the higher level processing within a production. This chapter introduces them and discusses how to design and develop business process classes. It contains the following sections:

- Introduction
- Comparison of Business Logic Tools
- Key Principles
- Defining BPL Business Processes
- Defining Custom Business Processes

## 5.1 Introduction

By convention, business processes contain most of the logic of the production. They can contain their own logic and they can call business rules and data transformations, each of which also contains specialized logic. The following figure illustrates this:
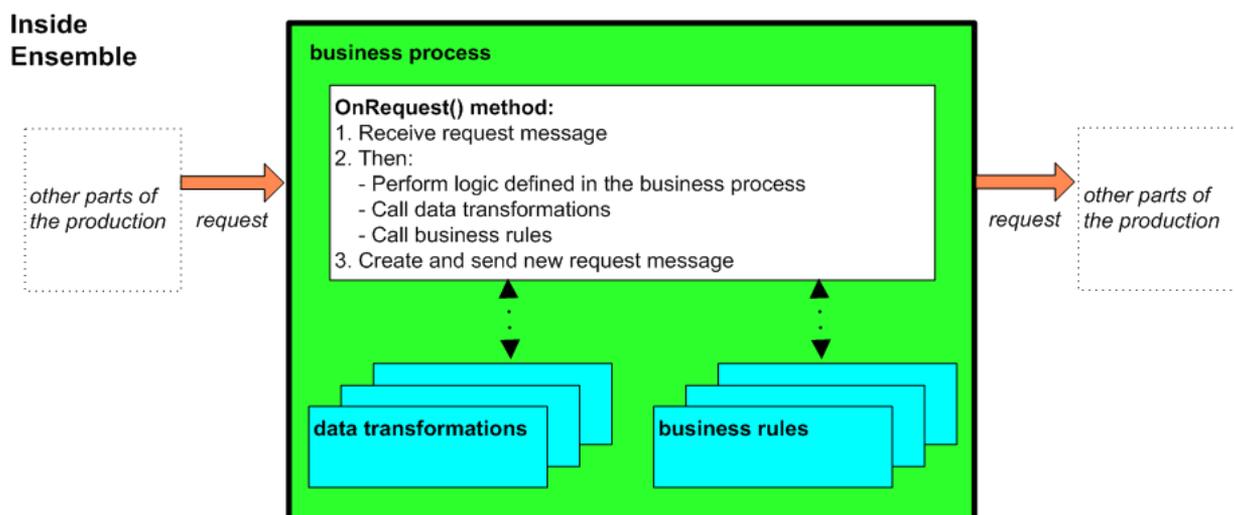


Note that this figure shows only the request messages.

There are many possible uses for business processes. In some cases, a business process coordinates a series of actions in one or more external applications. It contains the logic to determine the processing and it calls business operations or other business processes as needed. Business processes can also include human interaction; for details, see *Developing Workflows*.

Ensemble provides the following general types of business process:

- *BPL processes*, which are based on the class Ens.BusinessProcessBPL.

  Only BPL processes support the business process execution context and a graphical display of logic.

- *Routing processes*, which are based on the class EnsLib.MsgRouter.RoutingEngine or EnsLib.MsgRouter.VDocRoutingEngine.

  Ensemble provides a set of classes to route specific kinds of messages. To use these subclasses, no coding is generally necessary. For a list of books that discuss these classes, see "Types of Business Processes" in *Introducing Ensemble*.

- *Custom business processes*, which are based on the class Ens.BusinessProcess.

A production can include any mix of these business processes.

Note that Ens.BusinessProcessBPL, EnsLib.MsgRouter.RoutingEngine, and EnsLib.MsgRouter.VDocRoutingEngine are all based on Ens.BusinessProcess.

# 5.2 Comparison of Business Logic Tools

You will probably develop business processes in conjunction with the data transformations and business rules that they use. Data transformations and business rules are intended to contain specific kinds of logic:

- Data transformations alter the message
- Business rules return values or specify where to send messages (or potentially both)

There is, however, overlap among the options available in business processes, data transformations, and business rules. To assist you in determining how to create these items, the following table compares them. It discusses BPL (the most common business process), DTL (the most common data transformations), and business rules.

| Option | Supported in BPL? | Supported in DTL? | Supported in business rules? |
|---|---|---|---|
| Retrieving information about the business process | *Yes* (business execution context variables) | No | No |
| Assigning a value | *Yes* (<assign>) | *Yes* (**assign** action) | *Yes* (**assign** action) |
| Calling a data transformation | *Yes* (<transform>) | *Yes* (**subtransform** action) | *Yes* (**send** action) |
| Calling a business rule | *Yes* (<call>) | No | *Yes* (**delegate** action) |
| Calling custom code | *Yes* (<code>) | *Yes* (**code** action) | No |
| Invoking SQL | *Yes* (<sql>) | *Yes* (**sql** action) | No |
| Conditional logic | *Yes* (<if>, <switch>, <branch>) | *Yes* (**if** action) | No |
| Looping | *Yes* (<foreach>, <while>, <until>) | *Yes* (**for each** action ) | No |

| Option | Supported in BPL? | Supported in DTL? | Supported in business rules? |
|---|---|---|---|
| Sending an alert | *Yes* (<alert>) | No | No |
| Including trace elements | *Yes* (<trace>) | *Yes* (**trace** action) | *Yes* (**trace** action) |
| Sending a request message to a business operation or process | *Yes* (<call>) | No | *Yes* (**send** action) |
| Waiting for a response from asynchronous requests | *Yes* (<sync>) | No | No |
| Deleting the message | No | No | *Yes* (**delete** action) |
| Performing error handling | *Yes* (<throw>, <catch>, and others) | No | No |
| Delaying execution for a specified duration or until a future time | *Yes* (<delay>) | No | No |
| Sending a primary response before execution is complete | *Yes* (<reply>) | No | No |
| Using XPATH and XSLT | *Yes* (<xpath>, <xslt>) | No | No |
| Storing a message temporarily to acknowledge a milestone | *Yes* (<milestone>) | No | No |

For details on DTL transformations and business rules, see *Developing DTL Transformations* and *Developing Business Rules*.

# 5.3 Key Principles

First, be sure to read the chapter "Programming in Ensemble."

When you develop business processes, consider the following key principles:

- Sometimes it is desirable to make the response object be a modified version of the incoming request object, and it may be useful to make modifications in stages. However, do not modify the incoming request object. Instead copy it to a context variable (or, for a custom business process, copy data to local variable). Then modify the copy.

- Be careful when sending messages synchronously (which you can do only within a custom business process or within <code> in BPL).

  When a business process A calls business process B synchronously, process A does not continue until the response is received. If process A requires completion of calls to other processes (B) in order to complete itself, and if those processes share the pool of actor jobs, the actor pool can become deadlocked if there is no free actor job to process the called business process (B).

  This happens because the calling business process cannot complete and free the actor job until the called business process returns, but the called business process cannot execute because there is no free actor job to execute it.

Also note that Ensemble cannot shut down during a true synchronous call.

It is better to use **SendRequestAsync()** and handle response messages in the **OnResponse()** method. If you need to call synchronously, you can avoid this problem by configuring the called business process (B) to use its own job pool.

- If a single-job business process makes a request and waits for the response, the process loses FIFO capability.

  Because synchronous calls from BPL are implemented asynchronously by the compiler, the business process will go to disk after issuing the call. The actor can then continue to dequeue fresh business process or responses to other business processes.

  FIFO processing is only guaranteed for a single-job business process if it makes no call requests that need a response. If a business process must make a call and still maintain FIFO, it must use a <code> activity that invokes **SendRequestSync()**. However, in that case the preceding bullet item applies.

- BPL provides a useful advantage: It handles every aspect of synchronous and asynchronous messaging in a streamlined and elegant way that prevents common problems with contention and deadlocks. Even when a call is marked synchronous, if any wait time is required, the Ensemble framework quietly frees the job that was executing the call on behalf of the BPL business process so that it can do other work while BPL waits for the synchronous response. Later on, the framework silently coordinates receipt of the synchronous response and wakes up the BPL business process to resume its work.

  If you use custom code, it is very easy to (accidentally) design a production prone to deadlock. All you need to do is to create a sequence of business processes that send synchronous requests and use a limited actor pool. If the production then receives the same number of messages as actors, and a synchronous send occurs simultaneously for all the messages, the production ends up in a deadlock where all the actor processes are in the state of waiting for another process to release messages from queues. The most dangerous aspect of this issue is that testing does not necessarily produce the conditions that cause the deadlock. It is quite possible that the deadlock will first be encountered after deployment. At this point it will not be clear why Ensemble has "seized up," and the costs of this unforeseen problem may be high.

# 5.4 Defining BPL Business Processes

A BPL business process is a class based on Ens.BusinessProcessBPL. In this case, you can create and edit the process visually within either the Management Portal or Studio. For information, see *Developing BPL Processes*.

# 5.5 Defining Custom Business Processes

To create a custom business process class, define a class as follows:

- Your class must extend Ens.BusinessProcess (or a subclass).

- Your class must implement the OnRequest() and OnResponse() methods, as described in next two sections.

- Your class can add or remove settings. See "Adding and Removing Settings," earlier in this book.

- Your class can implement any or all of the startup and teardown methods. See "Overriding Start and Stop Behavior," later in this book.

- Your class can contain methods to accomplish work internal to itself.

# 5.5.1 Implementing the OnRequest() Method

A custom business process class must implement the OnRequest() method. An Ensemble production calls this method whenever an initial request for a specific business process arrives on the appropriate queue and is assigned a job in which to execute.

This method has the following signature:

```
method OnRequest(request As %Library.Persistent, Output response As %Library.Persistent) as %Status
```

Where:

- *request* — The incoming request object.

- *response* — The response returned by this business process.

## 5.5.1.1 Example

The following is an example of an **OnRequest()** method, from the Demo.Loan.BankUS sample class in the ENSDEMO namespace:

```
Method OnRequest(request As Demo.Loan.Msg.Application, Output response As Demo.Loan.Msg.Approval)
        As %Status
{
  Set $ZT="Trap",tSC=$$$OK
  Do {
    $$$TRACE("received application for "_request.Name)
    #;
    If $zcrc(request.Name,2)#5=0 {
      Set response = ##class(Demo.Loan.Msg.Approval).%New()
      Set response.BankName = "BankUS"
      Set response.IsApproved = 0
      $$$TRACE("application is denied because of bank holiday")
      Quit
    }
    #;
    Set tRequest = ##class(Demo.Loan.Msg.PrimeRateRequest).%New()
    Set tSC =..SendRequestAsync("Demo.Loan.WebOperations",tRequest,1,"PrimeRate")
    #;
    Set tRequest = ##class(Demo.Loan.Msg.CreditRatingRequest).%New()
    Set tRequest.SSN = request.SSN
    Set tSC =..SendRequestAsync("Demo.Loan.WebOperations",tRequest,1,"CreditRating")
    #;
    Set tSC = ..SetTimer("PT15S")
    #;
    Quit
  } While (0)
Exit
  Quit tSC
Trap
  Set $ZT="",tSC=$$$EnsSystemError Goto Exit
}
```

# 5.5.2 Implementing the OnResponse() Method

A custom business process class must implement the OnResponse() method. A production calls this method whenever a response for a specific business process arrives on the appropriate queue and is assigned a job in which to execute. Typically this is a response to an asynchronous request made by the business process.

This method has the following signature:

```
method OnResponse(request As %Library.Persistent,
                  ByRef response As %Library.Persistent,
                  callrequest As %Library.Persistent,
                  callresponse As %Library.Persistent,
                  pCompletionKey As %String) as %Status
```

This method takes the following arguments:

- *request* — The initial request object sent to this business process.

- *response* — The response object that will eventually be returned by this business process.

- *callrequest* — The request object associated with the incoming response.

- *callresponse* — The incoming response object.

- *pCompletionKey* — The completion key value associated with the incoming response. This is set by the call to the **SendRequestAsync()** method that made the request.

### 5.5.2.1 Example

The following is an example of an **OnResponse()** method, from the Demo.Loan.BankUS sample class in the ENSDEMO namespace:

```
/// Handle a 'Response'
Method OnResponse(request As Ens.Request,
                  ByRef response As Ens.Response,
                  callrequest As Ens.Request,
                  callresponse As Ens.Response,
                  pCompletionKey As %String) As %Status
{
    Set $ZT="Trap",tSC=$$$OK
    Do {
        If pCompletionKey="PrimeRate" {
            Set ..PrimeRate = callresponse.PrimeRate
        } Elseif pCompletionKey="CreditRating" {
            Set ..CreditRating = callresponse.CreditRating
        }
        Quit
    } While (0)
Exit
    Quit tSC
Trap
    Set $ZT="",tSC=$$$EnsSystemError Goto Exit
}
```

## 5.5.3 Methods to Use in OnRequest() and OnResponse()

When you implement **OnRequest()** and **OnResponse()**, you can use the following methods of the Ens.BusinessProcess class:

- **SendDeferredResponse()**

- **SendRequestSync()**

    But see "Key Principles," earlier in this chapter.

- **SendRequestAsync()**

- **SetTimer()**

- **IsComponent()**

For details on these methods, see the *Class Reference* for Ens.BusinessProcess.

# 6

# Defining Business Operations

This chapter describes how to define business operation classes. It contains the following sections:

- Introduction
- Key Principles
- Defining a Business Operation Class
- Defining a Message Map
- Defining Message Handler Methods
- Business Operation Properties
- Sending Requests to Targets within the Production
- Suspending Messages

**Tip:**     Ensemble provides specialized business operation classes that use specific outbound adapters, and one of those might be suitable for your needs. If so, no programming would be needed. For a partial list, see "Connectivity Options" in *Introducing Ensemble*.

## 6.1 Introduction

A business operation is responsible for sending requests from Ensemble to an external application or system. The following figure shows how it works:
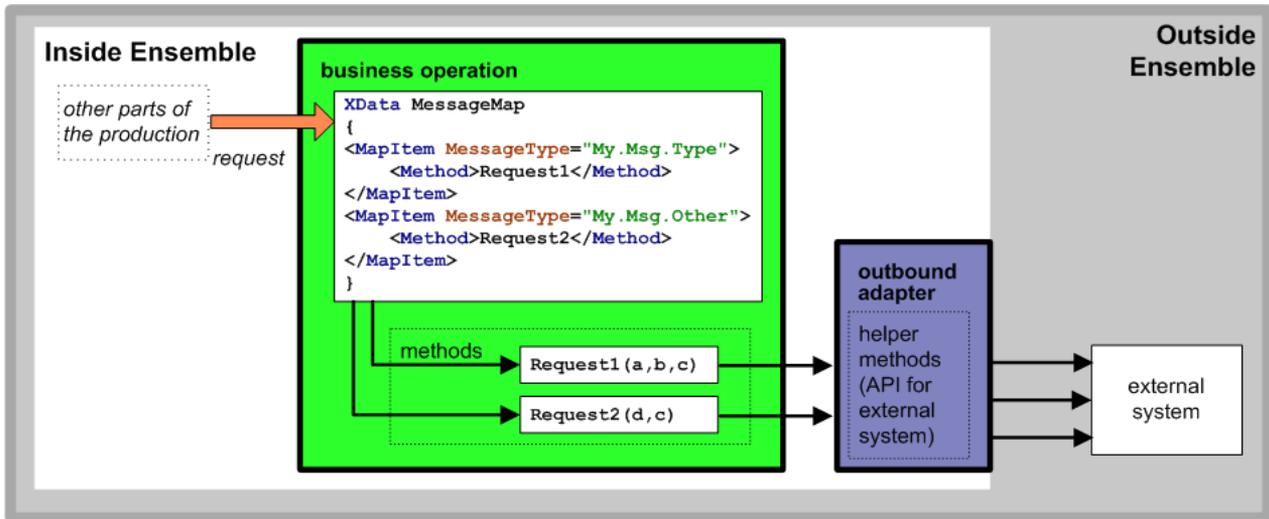
Note that this figure shows only the input flow of data, not the optional response.

A business operation is responsible for the following activities:

- Waiting for requests from business services or business processes.

- Dispatching, via a message map, the request to a specific method within the business operation. Each method within a business operation class represents a specific action within an external application.

- Transforming the request object into a form usable by the associated outbound adapter and asking the outbound adapter to send a request to the external application.

- Returning, if requested, a response object to the caller.

Each business operation contains a message map that specifies which external operation to perform, depending on the type of request message that it received. The message map contains one or more entries, each of which corresponds to one invocation of the associated outbound adapter.

# 6.2 Key Principles

First, be sure to read the chapter "Programming in Ensemble."

By convention, a business operation is an extremely specific operation that contains very little logic and does what is requested of it without calling further operations or branching in any way. When the design of the production demands logic, this is contained in a business process.

Many productions provide a large set of extremely simple business operations. In these cases, business processes contain the logic that determines *when* each operation should be called.

Also see "Key Principles" in the chapter "Defining Business Services."

# 6.3 Defining a Business Operation Class

To create a business operation class, define a class as follows:

- Your business operation class must extend Ens.BusinessOperation (or a subclass).

- In your class, the *ADAPTER* parameter should usually equal the name of the adapter class for this business service to use.

  Or you can define a business operation with no associated outbound adapter class. In this case, the business operation itself must contain the logic needed to communicate with an external application.

- In your class, the *INVOCATION* parameter must specify the invocation style you want to use, which must be one of the following.

  - **Queue** means the message is created within one background job and placed on a queue, at which time the original job is released. Later, when the message is processed, a different background job will be allocated for the task. This is the most common setting.

  - **InProc** means the message will be formulated, sent, and delivered in the same job in which it was created. The job will not be available again in the sender's pool until the message is delivered to the target. This is only suitable for special cases.

- Your class should define a *message map* that includes at least one entry. A message map is an XData block entry that has the following structure:

```
XData MessageMap
{
<MapItems>
  <MapItem MessageType="messageclass">
    <Method>methodname</Method>
  </MapItem>
  ...
</MapItems>
}
```

  See "Defining a Message Map."

- Your class must define all the methods named in the message map. These methods are known as *message handlers*. Each message handler should have the following signature:

```
Method Sample(pReq As RequestClass, Output pResp As ResponseClass) As %Status
```

  Here *Sample* is the name of the method, *RequestClass* is the name of an Ensemble request message class, and *ResponseClass* is the name of an Ensemble response message class. In general, these methods will refer to properties and methods of the Adapter property of your business operation. For details, see "Defining Message Handler Methods."

- Your class can add or remove settings. See "Adding and Removing Settings," earlier in this book.

- Your class can implement any or all of the startup and teardown methods. See "Overriding Start and Stop Behavior," later in this book.

  Ensemble is an integration platform potentially communicating with many other heterogeneous devices; therefore, it does not make property values dependent on server platform, time zone, time formatting, or other localization issues that may apply. Rather, InterSystems recommends you handle such cases in your production implementation. If your production requires different initial settings for property values, set the value in the **OnInit()** method of the business operation. See "Overriding Start and Stop Behavior," later in this book.

- Your class can contain methods to accomplish work internal to itself.

The following example shows the general structure that you need:

```
Class MyProduction.NewOperation Extends Ens.BusinessOperation
{
Parameter ADAPTER = "MyProduction.MyOutboundAdapter";

Parameter INVOCATION = "Queue";

Method SampleCall(pRequest As Ens.Request, Output pResponse As Ens.Response) As %Status
{
  Quit $$$ERROR($$$NotImplemented)
}

XData MessageMap
{
<MapItems>
  <MapItem MessageType="Ens.Request">
    <Method>SampleCall</Method>
  </MapItem>
</MapItems>
}
}
```

**Note:** Studio provides a wizard that you can use to create a business operation stub similar to the preceding. To access this wizard, click **File —> New** and then click the **Production** tab. Then click **Business Operation** and click **OK**.

For examples of business operation classes, see the adapter guides.

# 6.4 Defining a Message Map

A message map is an XML document, contained within an XData MessageMap block in the business operation host class. For example:

```
Class MyProduction.Operation Extends Ens.BusinessOperation
{

XData MessageMap
{
<MapItems>
  <MapItem MessageType="MyProduction.MyRequest">
    <Method>MethodA</Method>
  </MapItem>
  <MapItem MessageType="Ens.StringRequest">
    <Method>MethodB</Method>
  </MapItem>
</MapItems>
}

}
```

The operation of the message map is straightforward. When the business operation receives an incoming request, it searches, starting at the top of the message map, through each *MapItem* until it finds the first one whose *MessageType* attribute matches the type of the incoming message. It then invokes the operation method associated with this *MapItem*.

Some things to keep in mind about message maps:

- The message map is searched from top to bottom; once a match is found, no more searching is performed.

- If the incoming request object is a *subclass* of a given *MessageType* then it is considered a match. If you want to filter out subclasses, be sure to place them above any super classes within the message map.

- If the incoming request does not match any of the *MapItem* entries, then an error is written to the Ensemble Event Log and the request is ignored.

# 6.5 Defining Message Handler Methods

When you create a business operation class, typically the biggest task is writing message handlers for use with this adapter, that is, methods that receive Ensemble messages and then invoke methods of the adapter in order to communicate with targets outside the production.

Each message handler method should have the following signature:

```
Method Sample(pReq As RequestClass, Output pResp As ResponseClass) As %Status
```

Here *Sample* is the name of the method, *RequestClass* is the name of an Ensemble request message class, and *ResponseClass* is the name of an Ensemble response message class.

In general, the method should do some or all of the following:

1.  Optionally set properties of the business operation class (at any appropriate time). See "Business Operation Properties."

2.  Examine the input object.

3.  Create an instance of the response class.

4.  Call the applicable method or methods of the adapter. These methods are available via the Adapter property of your business operation. For example:

    ```
    Set tSc=..Adapter.SendMail(email,.pf)
    ```

    This method is discussed after these steps.

    Or, to send messages to a target within the production, see "Sending Messages to a Target within the Production."

5.  Examine the response.

6.  Use information in the response to create an Ensemble response message (an instance of Ens.Response or a subclass), which the method returns as output.

    For information on defining message classes, see the chapter "Defining Ensemble Messages."

7.  Make sure that you set the output argument (`pOutput`). Typically you set this equal to the response message. This step is required.

8.  Return an appropriate status. This step is required.

# 6.6 Business Operation Properties

Within an operation method, the following properties of the business operation class are available:

| Property | Description |
| --- | --- |
| %ConfigName | The configuration name for this business operation. |
| %SessionId | The session ID of the current message being processed. |
| Adapter | The associated outbound adapter for this business operation. |
| DeferResponse | To defer the response from this business operation for later delivery, set the DeferResponse property to the integer value 1 (true) and obtain a deferred response delivery token before exiting the business operation. |

| Property | Description |
|---|---|
| FailureTimeout | The length of time (in seconds) during which to continue retry attempts. After this number of seconds has elapsed, give up and return an error code. See Retry and RetryInterval. |
| Retry | Set this property to the integer value 1 (true) if you want to retry the current message. Typically, the retry feature is used when the external application is not responding and you wish to retry without generating an error. See RetryInterval and FailureTimeout. |
| RetryInterval | How frequently (in seconds) to retry access to the output system if this message is marked for retry. See Retry and FailureTimeout. |
| SuspendMessage | Set this property to the integer value 1 (true) if you want the business operation to mark its current in-progress message as having Suspended status. See the section "Suspending Messages." |

# 6.7 Sending Requests to Targets within the Production

Although a business operation is primarily responsible for delivering a request to the specific external application, it can also send messages to other business operations or to business processes, as needed. To send messages to a target within the production, call **SendRequestSync()**, **SendRequestAsync()**, or **SendDeferredResponse()**.

For information on these methods, see "Sending Request Messages" in the chapter "Defining Business Services."

Ens.BusinessOperation defines an additional method that you can use: DeferResponse().

## 6.7.1 The DeferResponse() Method

This method returns a %Status value indicating success or failure. It provides one by-reference argument, *token*, which returns the deferred response delivery token required for a later call to **SendDeferredResponse()**. For example:

```
Set sc=..DeferResponse(.token)
// Send the token out somewhere...
Quit $$$O
```

For an overview of deferred sending, see "Using Deferred Sending" in the chapter "Programming in Ensemble."

# 6.8 Suspending Messages

If you want the business operation to mark its current in-progress message as having Suspended status, set the business operation property SuspendMessage to the integer value 1 (true). Typically a business operation will do this for messages that have been rejected by the external system for some reason.

Ensemble places a Suspended message on a special queue so that a system administrator can diagnose the problem, fix the problem, and then resend the message. The system administrator can perform a simple resend (to the original target) or can send it to a new destination. For information, see *Monitoring Ensemble*.

The following sample method is from a business operation that sends a document to an external system. The method sets the SuspendMessage property to 1 if an error returns from the call to **Validate()** the document that is about to be sent:

```
Method validateAndIndex(pDoc As MyX12.Document) As %Status
{
  If ""=..Validation||'$method($this,"OnValidate",pDoc,..Validation,.tSC) {
    Set tSC=##class(MyX12.Validator).Validate(pDoc,..Validation)
```

```
    }
    Set:'$D(tSC) tSC=$$$OK
    If $$$ISERR(tSC) {
      Set ..SuspendMessage=1
      Do ..SendAlert(##Class(Ens.AlertRequest).%New($LB(
          ..%ConfigName,"Suspended document "_pDoc.%Id()_
          " because it failed validation using spec '"
          _..Validation_"' with error "_
          $$$StatusDisplayString(tSC))))
      Quit tSC
    }
    If ""'=..SearchTableClass {
      TRY {
        Set tSCStore=$classmethod(..SearchTableClass,"IndexDoc",pDoc)
        If $$$ISERR(tSCStore)
          $$$LOGWARNING("Failed to create SearchTable entries")
      }
      CATCH errobj {
        $$$LOGWARNING("Failed to invoke SearchTable class")
      }
    }
    Quit $$$OK
}
```

# 7

# Defining an Alert Processor

System alerts and user-generated alerts provide a way to inform users of problems in the Ensemble production. An alert processor is a business host that notifies applicable users via email, text pager, or other mechanism, about a problem that must be corrected. In many cases, you can define an alert processor without creating custom code. See Monitoring Alerts for information on adding an alert processsor to a production. This chapter describes how to create an alert processor with custom code. It includes the following topics:

- Background Information

- Coding a Simple Email Alert Processor

- Coding a Simple Outbound Adapter Alert Processor

- Coding a Routing Alert Processor

- Adding Custom Code to Alert Management

## 7.1 Background Information

Your business hosts can send alerts; see "Generating Alerts" in the chapter "Programming in Ensemble." Ensemble also automatically sends alerts on specific occasions, depending on the values of settings in the production. If the production includes a business host named **Ens.Alert**, Ensemble automatically sends a specialized request message (Ens.AlertRequest) to that business host. This business host is the *alert processor* for the production; any production can contain no more than one of these.

The alert processor can then use the information in this message to determine who must be contacted. There are a couple of general scenarios:

- If you can handle all alerts via the same output mechanism, the alert processor can be a business operation that uses the applicable adapter. See "Using a Simple Email Alert Processor" and "Using a Simple Outbound Adapter Alert Processor."

- If you need to route the alert, but do not need to track alert resolution, see "Using a Routing Alert Processor."

- If you want to track the alert resolution process, see "Using Alert Management to Track Alert Resolution."

In all cases, Ensemble also writes the information to the Ensemble Event Log, with the type Alert.

**Note:** **Ens.Alert** is the required name of the *business host* that serves as the alert processor. Do not confuse this with a class name. The alert processor can use any class name.

# 7.2 Using a Simple Email Alert Processor

If it is appropriate to send all alerts via email, use the class EnsLib.EMail.AlertOperation for the Ens.Alert component. This specialized business operation does the following:

- The *ADAPTER* parameter is specified as EnsLib.EMail.OutboundAdapter.

  This adapter provides settings to specify email recipients, as well as information needed to use an SMTP email server.

  It also provides helper methods to send email via the configured server.

- The **%OnProcessInput()** method is implemented. This method expects Ens.AlertRequest as input.

  This method does the following:

  1. Reads the alert text from the Ens.AlertRequest.

  2. Creates an email message (an instance of %Net.MailMessage) and writes the alert text into that.

  3. Sends the email message to all configured recipients.

You might be able to use this class without modification. Or you could create and use a subclass of it.

# 7.3 Using a Simple Outbound Adapter Alert Processor

If you can handle all the alerts via the same output mechanism, but you cannot use EnsLib.EMail.AlertOperation, then create a business operation named Ens.Alert as follows:

- Specify the *ADAPTER* parameter as the name of a suitable adapter class.

- Implement the **%OnProcessInput()** method. The signature of this method should be as follows:

  ```
  Method OnMessage(pRequest As Ens.AlertRequest, Output pResponse As Ens.Response) As %Status
  ```

  In your implementation, call methods of the adapter as needed.

  See "Defining Business Operations" and see the adapter books.

You might want to define the class so that details such as email addresses and telephone numbers are configurable. See "Adding and Removing Settings," earlier in this chapter.

# 7.4 Using a Routing Alert Processor

If you need to contact users via multiple output mechanisms, the alert processor should be a business process that determines how to route the Ens.AlertRequest messages. In this case, your production must contain an additional business operation for each output mechanism, and the alert processor forwards messages to those business operations.

## 7.4.1 Defining the Alert Processor as a Routing Process

To define the alert processor as a routing process, create a business process class that can receive Ens.AlertRequest messages.

The business process would examine the messages and forward them to different business operations, depending on the alert contents and any logic that you include.

Your logic may need to consider the following factors:

- Different requirements for various users

- Different requirements depending on the time of day

- The organization's problem-solving policies and procedures

You could use the class EnsLib.MsgRouter.RoutingEngine as the Ens.Alert routing process. This class provides the setting **Business Rule Name**. If you specify this setting as the name of a routing rule set, this business host uses the logic in that rule set to forward all the messages that it receives.

## 7.4.2 Defining the Business Operations

You can define each of the needed business operations as described in "Using a Simple Email Alert Processor" or "Using a Simple Outbound Adapter Alert Processor."

# 7.5 Adding Custom Code to Alert Management

Alert management allows you to assign alerts to users, track the status of alerts, and manage the progress of resolving alerts. For an overview of alert management, see Configuring Alert Management, which describes how to configure alert management components and define rules and data transformations for alert management. This section describes how to add custom code to the alert management components.

The alert management framework has the following architecture:

- Single persistent object for a managed alert throughout its life cycle.

- Alert Manager, Notification Manager, and Alert Monitor have the same overall internal structure. When one of these components is invoked, it performs its function in three phases:

  1. First, the component executes the **OnProcess** method if it is implemented by a subclass. By implementing this method, you can include custom code in the component. If the **OnProcess** method sets a flag indicating that processing is complete, the component exits.

  2. Next, the component evaluates its rule or for the Notification Manager its data transformation, which sets parameters that control the component's actions.

  3. Finally, the component performs its action based on either the parameters set by the rule or the defaults set by the component's configuration.

- The Alert Notification operation is a simpler component that formats and forwards messages to its destinations.

## 7.5.1 Alert Manager

The Alert Manager has the class Ens.Alerting.AlertManager and must be named Ens.Alert. The Alert Manager receives alerts from all production components. The Alert Manager can promote an alert to a Managed Alert based on the conditions specified in a rule. The Alert Manager sends Managed Alerts to the Notification Manager.

The Alert Manager executes in three phases:

1.  If the component's class overrides the **OnCreateManagedAlert()** method, execute the override. You can provide custom code to process the alert request and create the managed alert in this method. If do not want the base Alert Manager code to evaluate the rule, create the managed alert, and send it to the Notification Manager, you should set the tProcessingComplete parameter to 1. In that case, the Alert Manager takes no further action.

2.  Evaluate the CreateManagedAlertRule rule. This rule has access to tAlertContext. If it returns a true value (1), the Alert Manager creates the managed alert. If it returns false, the Alert Manager does not create the managed alert and the alert is only written to the log. The alert context provides access to:

    *   Incoming alert

    *   Alert groups configured for the component that originated the alert

    *   Business partner configured for the component that originated the alert

    *   Alert owner

    The rule can suppress promoting the alert to a Managed Alert by returning 0 or can promote the alert to a Managed Alert by returning 1.

3.  If the rule sets tCreateAlert to 1, the Alert Manager creates a Managed Alert, or, if there is no CreateManagedAlertRule rule defined, the Alert Manager takes the default action and creates a Managed Alert. The Alert Manager creates the Managed Alert by calling the **OnCreateManagedAlert()** method, which can be overridden by a class extending Ens.Alerting.AlertManager. The default implementation of **OnCreateManagedAlert()** sets the production name in the Managed Alert and sets the current owner to be unassigned with the value as an empty string. If the Alert Manager creates a Managed Alert, it sends it to the Notification Manager.

## 7.5.2 Notification Manager

The Notification Manager has the class Ens.Alerting.NotificationManager and is responsible for determining what groups to notify and which notification operations to use.

The Notification Manager executes in three phases:

1.  If the component's class overrides the **OnProcessNotificationRequest()** method, execute the override. If the override sets the pProcessingComplete parameter to 1, the Notification Manager does not evaluate the tranformation or apply the default action.

2.  Executes the data transformation if one is configured. See Adding the Notification Manager and Defining Its Data Transformation for information about the data transformation.

3.  If the transformation sets the target.Notify property to 1 or if there is no data transformation, the Notification Manager sends the Alert Notification to the component listed in each target and passes the list of addresses to the target.

The Notification Manager does not receive or send the Managed Alert object, but instead uses the Notification Request object, which contains a reference to the persistent Managed Alert object.

## 7.5.3 Alert Monitor

The Alert Monitor queries for all open Managed Alerts where the current time exceeds the NextActionTime value. It makes the following SQL query:

```
"SELECT ID FROM Ens_Alerting.ManagedAlert WHERE IsOpen = 1 AND NextActionTime <= ?"
```

where the current time returned by $$$timeUTC is specified as the parameter.

The Alert Monitor handles each returned Managed Alert message separately. For each Managed Alert, it processes it in three phases:

1. If the component's class overrides the **OnProcessOverdueAlert()** method, execute the override. You can provide custom code to process the alert. If do not want the base Alert Monitor code to evaluate the rule, update the managed alert, and send it to the Notification Manager, you should set the tProcessingComplete parameter to 1. In that case, the Alert Monitor takes no further action.

2. Evaluate the OverdueAlertRule rule. This rule has access to tOverdueContext. The overdue context provides access to:

   - Incoming alert

   - Current time

   - NewNextActionTime

   - NewEscalationLevel

   The rule can suppress sending the reminder by returning 0, can set the next time that the managed alert will be found by the Alert Monitor by setting the NewNextActionTime or can escalate or de-escalate the Managed Alert by setting NewEscalationLevel.

   You can override the context of the alert rule and how the Alert Monitor processes the results:

   - You can add additional information to the alert rule context by overriding the **GetOnOverdueAlertContext()** method.

   - You can override how the Alert Monitor handles the rule results by overriding the **OnProcessOverdueRuleResult()** method. Otherwise, execute this method in the base class. The **OnProcessOverdueRuleResult()** method is responsible for escalating the managed alert. The override has access to the managed alert, tOverdueContext, tSendNotification, and tNotificationType. Note that you should either duplicate the functionality of the base class implementation or call it by calling ##super() .

3. If the rule returns 1, the Alert Monitor sends the managed alert to the Notification Manager.

## 7.5.4 Notification Operation

The Notification Operation sends notifications to groups of users. If you are sending notifications using more than one kind of mechanism, you can have a separate Notification Operation for each transmission method.

# 8

# Defining Data Transformations

This chapter discusses data transformations. It includes the following topics:

- Introduction
- Defining DTL Transformations
- Defining Custom Transformations

## 8.1 Introduction

A *data transformation* creates a new message that is a transformation of another message. When you *transform* a message, your data transformation sends a new message body that is a transformation of the original. Some of the transformations that occur during this process can include:

- Copying values from properties on the source to properties on the target.
- Performing calculations using the values of properties on the source.
- Copying the results of calculations to properties on the target.
- Assigning literal values to properties on the target.
- Ignoring any properties on the source that are not relevant to the target.

## 8.2 Defining DTL Transformations

A *DTL transformation* is a class based on Ens.DataTransformDTL. In this case, you can create and edit the transformation visually in the DTL editor, which you can access in the Management Portal or in Studio. The DTL editor is meant for use by nontechnical users. See *Developing DTL Transformations*.

## 8.3 Defining Custom Transformations

A *custom transformation* is a subclass of Ens.DataTransform that specifies:

- The name of the input (source) message class

- The name of the output (target) message class

- A series of operations that assign values to the properties of the output object

Each assignment operation consists of a call to the Ens.DataTransform class method **Transform()**. The argument is a simple expression that is evaluated to provide the value for one of the properties in the output class. The language of the expression can be Caché Basic or ObjectScript, and the expression can contain:

- Literal values

- Any property in the general-purpose execution context variable called *context*

- Properties on the source object

- Functions and operators from the expression language

- Calls to methods provided by Ensemble

- Calls to user-provided methods

# 9

# Defining Business Metrics

An Ensemble business metric measures or calculates one or more values, typically related to the performance of a production, for display in dashboards or in the Production Monitor. This chapter describes how to create and display business metrics. It contains the following sections:

- Introduction to Ensemble Business Metrics
- Defining a Single-instance Business Metric
- Defining a Multi-instance Business Metric
- Other Options in Business Metric Classes
- Adding Business Metrics to Dashboards
- Adding Business Metrics to the Production Monitor
- Setting and Getting Values Programmatically
- About the Business Metric Cache

**Note:** As an alternative, you can use third-party business activity monitoring products with Ensemble. These products can interoperate with Ensemble via any of its connectivity technologies, including web services, JDBC, ODBC, and more.

## 9.1 Introduction to Ensemble Business Metrics

A business metric is a specialized business service class that you include in a production. While the production is running, the business metric values are available for display; when it is not running, the values are null.

### 9.1.1 Business Metric Properties

The values in a business metric are called *properties*. There are two general kinds of business metric properties: simple properties and multidimensional properties with autohistory.

A *simple property* holds only one value at any time. The following example displays a business metric with two simple properties:

A *property with autohistory* holds multiple values, one value for each point in time, with the most recent value at the end. You can control the number of times when the value is recorded. The following example displays a business metric with two properties that have autohistory:



As a developer of business metric classes, you are free to provide values for metric properties in any manner you choose: such values can be based on data stored within Ensemble messages or business process instances, data maintained within Ensemble by business processes, or data obtained by sending requests to external applications or databases.

## 9.1.2 Single- and Multi-instance Business Metrics

Also, there are two general kinds of business metrics: single-instance and multi-instance business metrics.

A *single-instance business metric* holds a single set of metric values. The examples in the previous section showed single-instance business metrics.

A *multi-instance business metric* holds multiple sets of metric values, one set for each instance defined by the metric. Multi-instance business metrics are useful when you have a number of similar items whose metrics you want to compare. Each

item is distinct, but has properties in common with the others. For example, each department might have a count of unassigned workflow tasks and a count of assigned workflow tasks. A business metric can have one instance for each department. The following shows an example multi-instance business metric that has two simple properties:



A multi-instance business metric can also have multidimensional properties with autohistory. In practice, however, it is not possible to simultaneously display the instances and the history. If you define such a business metric, then when you add it to the dashboard, the dashboard displays the current values for all instances by default. You can include a filter so that the user can select a single instance; in that case, the dashboard can display the history for that instance.

For example:



Then, when a user selects an instance:

## 9.1.3 Business Metrics as Business Services

All business metric classes are derived from the class Ens.BusinessMetric, which is itself derived from the Ens.BusinessService class so that it has the full functionality of a business service. For example, you can add a business metric to a production definition, you can assign a logical name to a business metric, you can schedule it for periodic execution (to recalculate its metric properties at a given interval), and you can invoke business operations and business processes as part of the metric value calculation.

# 9.2 Defining a Single-instance Business Metric

To define a single-instance business metric, define a class that meets the following requirements:

- It must be a subclass of Ens.BusinessMetric.

   For this class, the *ADAPTER* parameter is Ens.InboundAdapter, which provides the **CallInterval** setting. This ensures that the business metric is invoked periodically.

- It must define one or more business metric properties. The details are different for simple properties and properties with autohistory.

- It can optionally specify values for property parameters, for example, to control the range of values.

- It must assign values to the business metric properties. To do so, it must implement the **OnCalculateMetrics()** method.

The following subsections provide the details.

Studio provides a wizard that you can use to create a stub class. To access this wizard, click **File** and then click **New**. Select the **Production** tab. Select the **Business Metric** icon and click **OK**.

Also see "Defining a Multi-instance Business Metric," later in this chapter.

## 9.2.1 Defining Simple Business Metric Properties

To define a simple business metric property, add a property to the business metric class as follows:

```
Property MetricProperty As Ens.DataType.Metric;
```

This property can hold either numeric or string values.

Where *MetricProperty* is the name of the business metric property. For example:

```
/// This metric tracks A/R totals
Property AccountsReceivable As Ens.DataType.Metric;
```

This property can hold either numeric or string values.

## 9.2.2 Defining Business Metric Properties with Autohistory

To define a business metric property with autohistory, add a property to the business metric class as follows:

```
Property MetricProperty As Ens.DataType.Metric (AUTOHISTORY=50) [MultiDimensional];
```

For the *AUTOHISTORY* parameter, you can use any positive integer. For example:

```
/// Recent Sales History
Property SalesHistory As Ens.DataType.Metric (AUTOHISTORY = 50) [ MultiDimensional ];
```

Generally, the purpose of this kind of property is to collect values at intervals, over time, so that the resulting series of numbers can be plotted on a chart. For this reason, the assigned values are typically numeric.

The rate of collection is controlled by the **Call Interval** setting of the configured business metric.

## 9.2.3 Specifying Other Parameters for Metric Properties

You can specify additional property parameters for the business metric properties. These parameters might include default values for the upper and lower limits that control the appearance of any meter that displays that metric. For example:

```
/// Total Sales for the current day.
Property TotalSales As Ens.DataType.Metric (RANGELOWER = 0, RANGEUPPER = 50, UNITS = "$US");
```

The following table lists the property parameters that you can use with Ens.DataType.Metric (in addition to *AUTOHISTORY*, which is discussed in the previous section). These parameters apply to metric properties that hold numeric values; none of them apply when the metric property holds a string value.

| Parameter | Description |
|---|---|
| *LINK* | Optional. A URL specifying a browser page related to this metric. If a dashboard user right-clicks on the associated meter and selects the **Drill Down** option, the browser will show this page. |
| *RANGELOWER* | Optional. A numeric value specifying the expected lower range for values of this metric. This value provides a default for the low range of any meters connected to this metric. |
| *RANGEUPPER* | Optional. A numeric value specifying the expected upper range for values of this metric. This value provides a default for the upper range of any meters connected to this metric. |
| *THRESHOLDLOWER* | Optional. A numeric value specifying the expected lower threshold value for this metric. This value provides a default for the lower threshold of any meters connected to this metric. |
| *THRESHOLDUPPER* | Optional. A numeric value specifying the expected upper threshold value for this metric. This value provides a default for the upper threshold of any meters connected to this metric. |

| Parameter | Description |
|---|---|
| *UNITS* | Optional. A user-defined string enclosed in double quotes that specifies the units for this metric. Examples include "$US" or "Liters". This string appears in the bottom half of the Ensemble dashboard display when the viewer clicks on the corresponding meter in the top half of the display. |

## 9.2.4 Assigning Values to Business Metric Properties

This section describes how to assign values to business metric properties for single-instance business metrics; the details for multi-instance business metrics are discussed later in this chapter.

To assign values to business metric properties, implement the **OnCalculateMetrics()** instance method of the business metric class. The purpose of **OnCalculateMetrics()** is to calculate, look up, or otherwise set a value for any metric properties in the class.

The following example defines two metric properties, Counter and SalesHistory. In this example, the **OnCalculateMetrics()** method simply increments the single-valued Counter property and updates the SalesHistory property to random value:

```
/// Example Business Metric class
Class MyProduction.MyMetric Extends Ens.BusinessMetric
{

/// Number of times these metrics have been calculated.
Property Counter As Ens.DataType.Metric
        (RANGELOWER = 0, RANGEUPPER = 100000, UNITS = "Events");

/// Total Sales for the current day.
Property SalesHistory As Ens.DataType.Metric
        (RANGELOWER = 0, RANGEUPPER = 50, AUTOHISTORY = 50, UNITS = "$US")
        [ MultiDimensional ];

/// Calculate and update the set of metrics for this class
Method OnCalculateMetrics() As %Status
{
  // set the values of our metrics
  Set ..Counter = ..Counter + 1
  Set ..SalesHistory = $GET(..SalesHistory) + $RANDOM(10) - 5

  Quit $$$OK
}

}
```

Notes:

- Notice that you specify the value for a property in the same way for simple properties and for properties with autohistory.

  (If you are familiar with multidimensional properties, note that you specify the value only for the unsubscribed top node of the property, as shown here. Because of the *AUTOHISTORY* parameter, Ensemble generates code that automatically maintains the lower nodes in the array, which is an integer-subscripted array. For example, `SalesHistory(1)` is the oldest value in the `SalesHistory` property.)

- In this method, you can optionally invoke business operations and business processes. You can also invoke any APIs needed to compute the values.

- Never allow a property with autohistory to contain a null value. The null values are not displayed, causing bar charts and line charts to contain the wrong number of bars or entries. This can result in a mismatch between the axis labels and the items that they label. To prevent such problems, replace any null values with zero.

- Before the **OnCalculateMetrics()** method is called, all metric properties are set to their last calculated value (if any). After the **OnCalculateMetrics()** is called, the values of all the metric properties are stored in the business metric cache for subsequent use by dashboards or other interested parties (if any).

# 9.3 Defining a Multi-instance Business Metric

To define a multi-instance business metric:

- Follow the instructions in the previous section, except that the details are slightly different for implementing **OnCalculateMetrics()**.

- Define the instance names. To do so, you can:

  - Define a static set of instance names. To do so, implement the **OnGetInstances()** method to assign a fixed list of names to an array. This approach is useful if the set of instances is static.

  - Define the instance names dynamically. To do so, add a **MetricInstances()** query to get a list of names from a column in an SQL database. This approach is useful if you expect the number or names of the items to change over time.

  - Combine these approaches. Use the **MetricInstances()** query to get an initial list and then use **OnGetInstances()** to add or replace names. (The business metric instance calls **MetricInstances()** first and then calls **OnGetInstances()**.)

- In your implementation of **OnCalculateMetrics()**, check the value of the %Instance property and assign values to the business metric properties as appropriate for that instance.

- Keep the following principles in mind:

  - The instance names are strings.

  - The instance names must be unique.

  - The instance names may be displayed to users on a dashboard, so use names that are concise, informative, and appropriate.

  - Try to keep the number of instances reasonable. Thousands of instances could be expensive to compute and difficult for users to understand.

The following subsections provide the details.

## 9.3.1 Defining a Static Set of Instance Names

To define a static set of instance names, override the **OnGetInstances()** method. This method is passed an array by reference. The **OnGetInstances()** method must fill this array with names using an ordinal number as a subscript, starting with 1. A simple example follows:

```
/// Return an array of metric instances
ClassMethod OnGetInstances(ByRef pInstSet As %String) As %Status
{
  Set pInstSet(1) = "Apple"
  Set pInstSet(2) = "Banana"
  Set pInstSet(3) = "Cherry"
  Quit $$$OK
}
```

## 9.3.2 Defining the Instance Names Dynamically

To define the set of instance names dynamically, add a query to the business metric class as follows:

- The query must be named **MetricInstances()**.

- It must take no arguments.

- The query must contain an SQL SELECT statement.

- The first column returned by the query returns the instance names.

For example:

```
/// Return current list of product names
Query MetricInstances() As %SQLQuery
{
  SELECT ProductName FROM MyApplication.Product
}
```

## 9.3.3 Assigning Values to Properties in a Multi-instance Metric

To assign values to properties in a multi-instance business metric, implement the **OnCalculateMetrics()** method. In your implementation, set the values as appropriate for each instance. To do so:

1. Check the value of the %Instance property. This property equals the name of one of the business metric instances.

   (Ensemble automatically processes one instance at a time, sequentially. For each instance, Ensemble executes the **OnCalculateMetrics()** instance method.)

2. Set the values of the business metric properties as needed for that instance.

The following shows an example:

```
Method OnCalculateMetrics() As %Status
{
  // get product name
  Set product = ..%Instance


  // find recent sales using SQL
  &SQL(SELECT SUM(Sales) INTO :sales
            FROM MyApplication.Product
            WHERE ProductName = :product)

  // update sales metric
  Set ..Sales = sales
  Quit $$$OK
}
```

This example uses the current instance name (%Instance) in an SQL query to retrieve the most recent sales data for that instance, then writes this data to the Sales property of the current instance of the class.

**Note:** You can also use the %Instance property elsewhere in the class when you want to substitute the current instance name.

# 9.4 Other Options in Business Metrics

This section describes other options within business metric classes.

## 9.4.1 Defining Actions for Use in Dashboards

A business metric class can define actions, which you can expose as user options in dashboards. An action can perform a combination of client-side activities (such as filtering and refreshing the dashboard) and server-side activities (such as invoking your own APIs). The action mechanism is quite general.

To define actions, implement the **%OnGetActionList()** and **%OnDashboardAction()** methods of the business metric class. For information on these methods, see the chapter "Defining Custom Actions" in the *DeepSee Implementation Guide*.

## 9.4.2 Implementing OnInit()

You can also override the **OnInit()** callback of the business metric class, to initialize any properties, for example. If you do so, you must ensure that it explicitly calls the **OnInit()** method provided by its superclass BusinessMetric, as shown below. If not, the corresponding dashboard element does not display properly:

```
Method OnInit() As %Status
{
    // . . .

    // invoke superclass implementation
    Quit ##super()
}
```

# 9.5 Adding Business Metrics to Dashboards

To add business metrics to dashboards, do the following:

1.  Add them to the appropriate production, in the same way that you would add any other business service.

2.  Configure the **Call Interval** setting as needed for each business metric.

3.  Create dashboards and add business metrics to them. For information, see *Configuring Ensemble*.

4.  Optionally extend the Production Monitor page to show information from your business metrics. See the next section.

# 9.6 Adding Business Metrics to the Production Monitor

In addition to displaying business metrics in dashboards, you can extend the Production Monitor page to show information from your business metric classes. To do so, set nodes in the ^Ens.Monitor.Settings global in your namespace, as follows:

| Node | Value |
|---|---|
| `^Ens.Monitor.Settings("MetricClasses",n,"Metric")` | Configuration name of the business metric, for the *n*th business metric. The Production Monitor page lists the business metrics in the order specified by *n*. |
| `^Ens.Monitor.Settings("MetricClasses",n,"Title")` | Display name for this business metric. The default is the configuration name of the business metric |
| `^Ens.Monitor.Settings("MetricClasses",n,"Instance")` | Instance name of this business metric. If the metric does not have instances, omit this. If the metric does have instances and you omit this, Ensemble uses the first instance, considering the order in which instances are defined. |

For example, do the following in the Terminal:

```
Set ^Ens.Monitor.Settings("MetricClasses",1,"Metric") = "MetricConfigName"
Set ^Ens.Monitor.Settings("MetricClasses",1,"Title") = "Title for Display"
Set ^Ens.Monitor.Settings("MetricClasses",1,"Instance") = "MetricInstanceName"
```

For each business metric that you add, the Production Monitor page indicates when it last updated the metric information, if there is any data for the given metric or instance, and whether or not the given metric is currently running.

For information on using the Production Monitor page, see "Monitoring a Production" in *Monitoring Ensemble*.

# 9.7 Setting and Getting Values Programmatically

In some cases, you might need programmatic access to metric properties. For example, you might want a business process to directly read or set a metric property. To do this, use the **GetMetric()** and **SetMetric()** class methods of Ens.BusinessMetric.

## 9.7.1 Using the GetMetric() Method

The **GetMetric()** class method reads the current value of a specified metric property from the business metric cache. Call this method as follows:

```
Set value = ##class(Ens.BusinessMetric).GetMetric(metric,property)
```

Where *metric* is the name of the business metric (the configuration name, not the class name) and *property* is the name of the metric property. If **GetMetric()** is unable to read the specified value, it returns an empty string.

To read values from multidimensional metric properties, there is a third, optional parameter that specifies which subnode of the property to read. For example:

```
Set value(1) = ##class(Ens.BusinessMetric).GetMetric(metric,property,1)
```

## 9.7.2 Using the SetMetric() Method

The **SetMetric()** class method sets the value of a specified metric property within the business metric cache. Call this method as follows:

```
Set tSC = ##class(Ens.BusinessMetric).SetMetric(metric,property,value)
```

Where *metric* is the name of the business metric (the configuration name, not the class name), *property* is the name of the metric property, and *value* is the value to which the metric property should be set.

**SetMetric()** returns a %Status code indicating success or failure. It is possible that **SetMetric()** is unable to acquire a lock for the business metric and may fail for that reason.

To set all the values of a multidimensional metric property, create the array of values and then pass the array by reference. For example:

```
For i=1:1:20 {
  Set data(i) = i*i
}
Set tSC = ##class(Ens.BusinessMetric).SetMetric("MyMetric","MyGraph",.data)
```

# 9.8 About the Business Metric Cache

So that Ensemble can retrieve metric values as efficiently as possible, it stores these values in a cache. This cache is the global ^Ens.Metrics, which has the following structure:

```
 ^Ens.Metrics(BusinessMetric,Instance,Property) = value
```

Where:

- *BusinessMetric* is the configured name of the business metric class.

- *Instance* is an instance number. Instances are numbered in the order in which they are defined.

- *Property* is the name of the business metric property.

# 10

# Defining an Enterprise Message Bank

This chapter describes how to define an optional, specialized kind of production called the Enterprise Message Bank. It includes the following sections:

- Overview

- Defining the Message Bank Server

- Adding a Message Bank Helper Class

- Notes about the Message Bank

After you have defined the Enterprise Message Bank, see "Configuring the Enterprise Message Bank"' in *Configuring Ensemble*.

## 10.1 Overview

The Enterprise Message Bank is an optional remote archiving facility where you can collect messages, Event Log items, and search table entries from *multiple* Ensemble client productions. It consists of the following components:

- The Message Bank server, which is a simple Ensemble production consisting exclusively of a Message Bank service that receives submissions from any number of client productions.

- A client operation (the Message Bank operation) that you add to an Ensemble production and configure with the address of a Message Bank server.

The following shows a conceptual example:
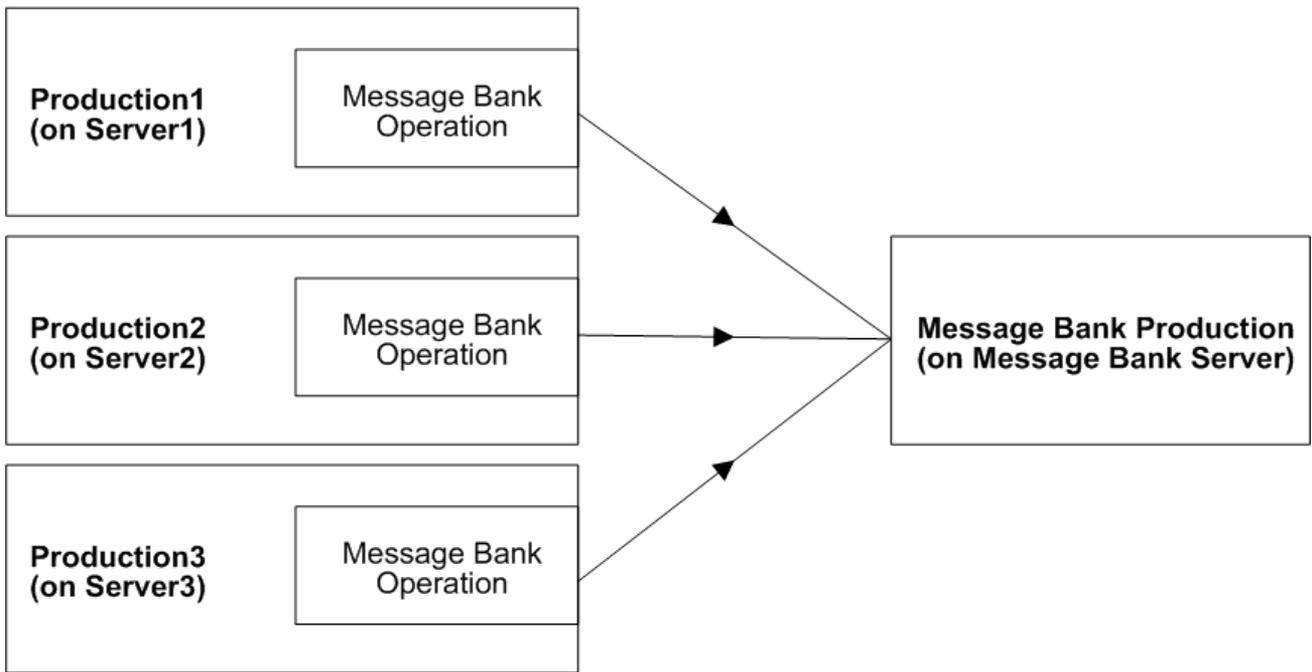
So that you can conveniently see the messages in the Message Bank, Ensemble provides the following additional options:

- For the Message Bank instance, the Management Portal automatically includes the Enterprise Monitor pages, where you can monitor the status of client productions, browse the Message Bank, and perform a search of messages from the monitored clients.

- For each client instance, you configure a link to the Enterprise Monitor in the Message Bank instance.

The following shows an example:
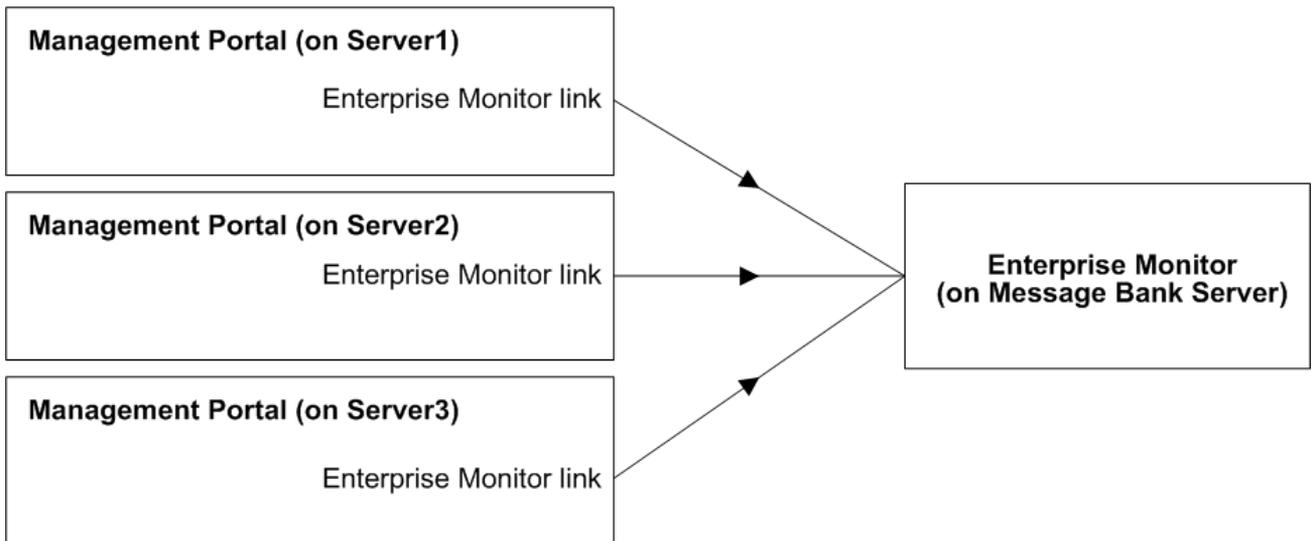


# 10.2 Defining the Message Bank Server

To define the Message Bank server, do the following on the server machine, in an Ensemble-enabled namespace:

1. Create a subclass of the Ens.Enterprise.MsgBank.Production abstract class.

2. Copy its `ProductionDefinition` XData block into your class.

3. Compile your new class in that namespace.

The following shows an example:

```
Class MyEnsemble.MyMessageBank Extends Ens.Enterprise.MsgBank.Production
{

XData ProductionDefinition
{
<Production Name="Ens.Enterprise.MsgBank.Production" TestingEnabled="false" LogGeneralTraceEvents="false">

  <Description>Production for receiving and collating message bank submissions from one or more client

   Ensemble namespaces and for maintaining a local repository of production status information about
each
   client namespace, for display on the Enterprise Monitor page. Open the Monitor page on the same
   machine that that is hosting this Production.</Description>
  <ActorPoolSize>0</ActorPoolSize>
  <Setting Target="Production" Name="ShutdownTimeout">120</Setting>
  <Setting Target="Production" Name="UpdateTimeout">10</Setting>
  <Item Name="MonitorService" Category="" ClassName="Ens.Enterprise.MonitorService" PoolSize="1"
        Enabled="true" Foreground="false" InactivityTimeout="0" Comment="Populates global
        ^CacheTemp.Ens.EntMonitorStatus by polling namespaces from Systems List every CallInterval
seconds"
        LogTraceEvents="false" Schedule="">
    <Setting Target="Host" Name="AlertGracePeriod">0</Setting>
    <Setting Target="Host" Name="AlertOnError">0</Setting>
    <Setting Target="Host" Name="ArchiveIO">0</Setting>
    <Setting Target="Adapter" Name="CallInterval">10</Setting>
  </Item>
  <Item Name="MsgBankService" Category="" ClassName="Ens.Enterprise.MsgBank.TCPService" PoolSize="100"

        Enabled="true" Foreground="false" InactivityTimeout="20" Comment="" LogTraceEvents="true"
Schedule="">
    <Setting Target="Host" Name="AlertGracePeriod">0</Setting>
    <Setting Target="Host" Name="AlertOnError">0</Setting>
    <Setting Target="Host" Name="ArchiveIO">0</Setting>
    <Setting Target="Adapter" Name="Endian">Big</Setting>
    <Setting Target="Adapter" Name="UseFileStream">0</Setting>
    <Setting Target="Adapter" Name="JobPerConnection">1</Setting>
    <Setting Target="Adapter" Name="AllowedIPAddresses"></Setting>
    <Setting Target="Adapter" Name="QSize">100</Setting>
    <Setting Target="Adapter" Name="CallInterval">5</Setting>
    <Setting Target="Adapter" Name="Port">9192</Setting>
    <Setting Target="Adapter" Name="StayConnected">-1</Setting>
    <Setting Target="Adapter" Name="ReadTimeout">10</Setting>
    <Setting Target="Adapter" Name="SSLConfig"></Setting>
  </Item>
</Production>
}

}
```

This production has the following services:

- Ens.Enterprise.MsgBank.TCPService — Accepts inbound data from the client productions. For details on configuring this service, see "Configuring the Message Bank Service on the Server" in *Configuring Ensemble Productions*.

- Ens.Enterprise.MonitorService — Collects status information from the client productions.

This production receives this data and maintains a local repository.

For testing purposes, you can put the Message Bank production on the same machine and instance as your regular productions, but it must be in a separate namespace from any production you plan to monitor.

# 10.3 Adding a Message Bank Helper Class

By default, the message bodies are not indexed and cannot be searched. You can add a helper class to implement the search capability in the Message Bank. To do so:

1. Create a subclass of Ens.Enterprise.MsgBank.BankHelperClass and implement its **OnBankMsg()** method. See the *InterSystems Class Reference* for details on the method.

   The **OnBankMsg()** method specifies any custom processing to perform when inbound messages include a message body.

   In your implementation of this method, you should decide whether to handle messages in process (and possibly reduce the input capacity), or whether the method should asynchronously forward the messages to a secondary process to balance the work more effectively.

2. In the Message Bank production, select the Ens.Enterprise.MsgBank.TCPService business service and specify the **Bank Helper Class** setting. For the value, use the name of your helper class.

# 10.4 Notes about the Message Bank

Note the following important characteristics of the Message Bank:

- The Message Bank has no synchronization dependencies for message body classes with the contributing productions; therefore, it receives a serialized form of each message. For virtual document message bodies (HL7, for example), it reparses the serialized document into an object, and receives and stores the search table entries from the contributing production as well. You can perform almost the same search for HL7 messages as on the original machine; however, the same is not true for other types of message bodies.

  To enable searching by HL7 custom schema properties in the Message Bank, make sure to place the custom schema definition in the Message Bank production namespace.

- For uniqueness, the Message Bank production prepends the numeric identifier of client productions to the message IDs.

- The Message Bank page also serves as a portal for viewing the state of multiple client productions and for invoking a resend service on them. For these extra features to work, the Message Bank needs to know the web addresses of the client productions. For details, see "Configuring the Enterprise Message Bank"' in *Configuring Ensemble*.

- The Message Bank mechanisms do not delete the messages from the originating production; you should handle that function with a separate purge process.

- It is possible to replay messages from the Message Bank back into the originating production or another client production. See "Using the Enterprise Message Bank"' in *Monitoring Ensemble*.

# 11

# Using the Ensemble Record Mapper

This chapter explains how to use the Record Mapper tool to work more easily with files containing delimited and fixed-width records in Ensemble productions. This chapter contains the following sections:

- Overview

- Creating and Editing a Record Map

- Using the CSV Record Wizard

- Record Map Class Structure

- Sample Record Mapper Production

The following two chapters described Complex Record Mapping, where an individual record consists of a group of heterogeneous subrecords, and batch processing, where multiple records are grouped to be processed in a batch:

- Using the Complex Record Mapper

- Handling Batches of Records Efficiently

## 11.1 Overview

The Ensemble Record Mapper tool provides a quick, efficient way to map data in text files to persistent Ensemble messages and back again. In particular, the Management Portal user interface allows you to visually create a representation of a text file and create a valid object representation of that data which maps to a single persistent Ensemble message object. The process of generating both the target object structure and an input/output parser is automated, leaving only a few options for the persistent structure of the object projection. Ensemble generates the objects in such a way as to be a single persistent tree to provide complete cascading delete operations.

The Management Portal also provides a **CSV Wizard** to help you convert CSV (Character Separated Value) files into a record map structure. This is particularly useful for files which contain column headers, as the wizard uses the header names to create the record map properties corresponding to the columns in the sample file.

The Record Mapper handles simple records which are either delimited or have fixed-width fields. A record map consists of a series of fields, which identify data in the record, and composites, which organize fields into a unit. In delimited records, the hierarchal level of composites specify the separator that is used between fields. Within delimited records, you can have simple fields that are repeating. You can not have repeating composites. You can optionally ignore any fields in the incoming text file so that they do not waste space in the stored records.

The Record Mapper is not capable of dealing with mixtures of delimited and fixed-width data, nor is it capable of dynamically adjusting its parser or object structure based on the content of the incoming record other than handling repeating simple fields. See Sample Record Mapper Production for details.

The Complex Record Mapper allows you to handle structured records containing different record types, including the ability to handle structures with repeating records and mixed delimited and fixed-field records.

An additional feature of the Record Mapper package allows you to batch heterogeneous records by implementing a class which inherits from EnsLib.RecordMap.Batch. This class handles parsing and writing out any headers and trailers associated with a specific batch. For simple headers and trailers, the Record Mapper user interface permits the creation of a batch of type EnsLib.RecordMap.SimpleBatch. You can extend either of these two batch implementations if you need to process more complex header and trailer data.

When a RecordMap batch operation is creating a batch from individual records, it stores the partially constructed batch in an intermediate file. You can specify the location of this file using the IntermediateFilePath property on the batch operation. On a mirrored system, you can store the intermediate file on a network drive that is accessible to both the main and failover system. Then if a failover occurs the failover system can continue to append records to the incomplete batch. Since the incomplete batch is stored in a file and not in a Caché database, it is not automatically copied to the mirror system.

# 11.2 Creating and Editing a Record Map

This section describes how to create and edit a record map. It contains the following sections:

- Introduction
- Getting Started
- Common Control Characters
- Editing the Record Map Properties
- Editing the Record Map Fields and Composites

## 11.2.1 Introduction

You can create a record map using the **Record Mapper** page of the Management Portal. If your file is a delimited file, you can also use the CSV Record Wizard to further automate the process. As you develop your record map, you can view how a sample file displays in the record map.

The **Record Mapper** page includes a visual representation of the record map structure, and a simple interface which allows you to enter and manipulate the more detailed settings available for the record map components. It also allows you to reposition sibling elements (elements at the same level). One of the more important features of the user interface is that if you have a sample input file, a sample parse of the file will be attempted when the current record map is saved. You can address small issues in your record map directly from the Management Portal.

You can also create a record map directly using XML and the model classes.

## 11.2.2 Getting Started

To start the Record Mapper, select **Ensemble** > **Build** > **Record Mapper**. From here, you have the following commands:

- **Open** — Displays the finder dialog box for you to choose an existing record map to open for editing.
- **New** — Initializes the page for you to enter a new record map structure.

- **Save** — Saves your record map structure as a class in the namespace in which you are working. Once saved, the object appears in the list of record maps.

- **Generate** — Generates the record map parser code and the related persistent message record class object.

  To generate an object manually use the **GenerateObject()** class method in EnsLib.RecordMap.Generator. It permits a number of options regarding the persistent structure of the generated objects, as noted in the comments for the method.

- **Delete** — Deletes the current record map. You can optionally delete the related persistent message record class and all stored instances of the classes.

- **CSV Wizard** — Opens the CSV Record Wizard to help automate the process of creating a record map from a sample file that contains comma separated values (CSV).

**Important:**     The **Save** operation only writes the current record map to disk. In contrast, the **Generate** operation generates the parser code and the persistent object structure for the underlying objects.

When you select the name of the record map on the left side of the page, you see the record settings on the right, where you can edit the properties of the record map itself. Before you can save a record map, you must add at least one field to your record map. The following sections describe these processes:

Once you have created a new Record Map or opened an existing one, the Record Mapper displays a summary of the fields defined in the Record Map on the left panel and, on the right panel, allows you to set the properties of the Record Map or of the selected field. If you have specified a sample data file, it is displayed above the left panel. For example, the following shows the Record Mapper with the Record Map properties in the right panel:

To export, import, or delete a Record Map, click **Ensemble**, **List**, and **Record Maps** to display the Record Map Lists page.

## 11.2.3 Common Control Characters

Within a record map, you can use literal control characters as well as printable characters in several places. For example, you can specify a tab character, which is a common control character as well as a comma, which is a printable character, as a separator. You can also specify control characters as a padding character or as one of the record terminator characters. To specify a control character in one of these contexts, you must specify the hexadecimal escape sequence for the character. If you select the space or tab character as the padding character, or CRLF (carriage return followed by a line feed), CR, or LF as the record terminator character in the Record Mapper, the Management Portal automatically generates the hexadecimal representation. If you are specifying another control character as the padding character or in the record terminator or any control character as a separator, you must enter the hexadecimal representation in the corresponding form field. The following table lists the hexadecimal escape sequence for commonly used control characters:

| Character | Hexadecimal representation |
|---|---|
| Tab | `\x09` |
| Line feed | `\x0A` |
| Carriage return | `\x0D` |
| Space | `\x20` |

For additional characters, see http://en.wikipedia.org/wiki/C0_and_C1_control_codes or other resources.

**Note:** If you specify a record terminator in the RecordMap, the incoming message must match the record terminator exactly. For example, if you specify CRLF (\x0D\x0A), then the incoming message record must match that sequence.

## 11.2.4 Editing the Record Map Properties

Whether you are entering properties for a new record map, starting from the wizard-generated map, or editing an existing map, the process is the same. For the record itself, enter or update values in the following fields:

**RecordMap Name**

Name of the record map. You should qualify the record map name with the package name. If you do not provide a package name and specify an unqualified record map name, the record map class is saved in the User package by default.

**Target Classname**

Name of the Ensemble class to represent the record. By default, the Record Mapper sets the target class name to a qualified name equal to the record map name followed by ".Record", but you can change the target class name. You should qualify the target class name with the package name. If you do not provide a package name and specify an unqualified target class name, the target class is saved in the User package by default.

**Batch Class**

Name of the batch class (if any) which should be associated with this record map.

**Type**

The type of record; options include the following:

- Delimited
- Fixed Width

**Character Encoding**

Character encoding for imported data records.

**Right justify**

Flag that specifies that padding characters should appear left of data in fields.

**Annotation**

Text that documents the purpose and use of the record map.

**Leading data**

Static characters which appear before any data of the actual record contents. If you are using the record map in a complex record map, you must identify the record with leading data.

**Padding Character**

Character used to pad the value. The padding character is removed by business services from the incoming message and used by business operations to pad the field value to fill fields in fixed-width record maps.

- None

- Space

- Tab

- Other

**Record Terminator**

Character or characters used to terminate the record.

- None

- CRLF

- CR

- LF

- Other

**Allow Early Terminator (fixed-width record maps only)**

Flag that specifies whether records can be terminated before the end. If allowed, record is treated as if it was padded with the padding character.

**Allow Complex Batching**

Flag that specifies whether record map can be used in a complex record map.

**Field separator (fixed-width record maps only)**

Optional single character used to separate fixed-width fields in records. If specified, input messages must contain this character between fields and business operations write this character between fields.

**Field separator(s) (delimited record maps only)**

A list of field separator characters. The first separator delimits the top-level fields in the record. The next separator delimits fields within a top-level composite field. Additional separators delimit fields within nested composite fields.

**Repeat separator (delimited record maps only)**

A single separator character that is used in all repeating fields.

**Quoting: None (delimited record maps only)**

Radio button that specifies there is no quote-style escaping.

**Quote Escaping (delimited record maps only)**

Radio button that enables quote-style escaping to allow a separator character to occur in a field value. Any input field can be quoted with the quote character. The field is considered all the characters between the start quote and end quote. Any separator character that appear within the quotes is treated as a literal character, not a separator. On output, any field that contains a separator in its value is quoted with the quote character.

**Quote All (delimited record maps only)**

Radio button that enables quote-style escaping to allow a separator character to occur in a field value. This has the same effect as Quote Escaping except that on output all fields are quoted whether they contain a separator character or not.

**Quote character (delimited record maps with quote escaping only)**

Character used to quote field contents. This field is displayed if you select the Quote Escaping or Quote All radio button. If you are using a control character as a quote, you must enter it in hexadecimal; see "Common Control Characters," earlier in this chapter.

## 11.2.5 Editing the Record Map Fields and Composites

The Record Mapper left panel displays a summary of the fields defined in the Record Map. If you select a field, the right panel accesses the field properties. For example:

Record maps consist of a sequence of fields and composites. Each composite consists of a series of fields and composites. The **Make Composite** and **Make Field** buttons switch between a composite and a data field. For composite fields, you only specify the name and the flag indicating the field is required. Click the green plus-sign icon on the record map to add a field or composite to the top level. Clicking on the plus-sign of a composite allows you to add a field or composite to it.

While you are adding fields to your record map, you can open a sample file to see how its data maps to the record you are creating.

For delimited record maps, fields within composite fields have different separators. For example, in a record, the top-level field are delimited by commas, but within a composite the fields are delimited by semicolons. For fixed-width record maps, composite fields help organize the data conceptually, but do not impact the processing of the input message.

When you create a composite field in the Record Mapper, composite fields set the default name as a qualified name that matches the composite structure. The qualified field names determine the structure of fields within the generated record class. If you modify the field names to have different qualified names, the level of composite fields in the record map is independent from the structure of the fields in the generated record class.

For each data field you enter the following properties:

**Name**

Name of the field.

**Datatype**

Data type of the field. Select from the following list or enter a custom datatype:

- %Boolean
- %Date
- %Decimal
- %Double
- %Integer
- %Numeric
- %String
- %Time
- %Timestamp

**Annotation**

Documents the purpose and use of the field in the record map.

**Width (fixed-width record maps only)**

Width of the field.

**Required**

Flag that specifies that the field is required.

**Repeating (delimited record maps only)**

Flag that specifies that the field may contain repeated values using the record map's repeat separator character.

**Ignore**

Flag that specifies the field is ignored on input and not included in the stored record. Using the **Ignore** property saves storage space for the stored records. On output, Ensemble outputs an empty value for ignored fields—for fixed-width records, it fills the field with spaces, and for delimited records, it writes two consecutive separators for the empty field.

**Trailing Data (fixed-width record maps only)**

Characters that must follow this field. Control characters must be entered in hexadecimal; see "Common Control Characters," earlier in this chapter.

**Datatype Parameters**

Parameters to apply to the data type separated by a semicolon.

**SQL Column Number**

The SQL column number of the field. This value must either omitted or be between 2 and 4096 (inclusive) as per the values for the SqlColumnNumber property keyword. The column number is of particular use when importing data from CSV files or similar data dumps, as the SQL representation can be replicated easily.

**Index**

>   Enumerated value that controls whether the property should be indexed; select one of the following:

*   (blank) — do not index

*   1

*   bitmap

*   idkey

*   unique

The left panel of the Record Mapper is a table with a summary of the field definitions. The columns specify:

*   Top-level field number.

*   Field name.

*   Summary of the properties of the field. The summary contains the following information, separated by ; (semicolon):

    *   ignored — present if the **Ignore** check box is selected.

    *   0..1 or 1..1 followed by the datatype and datatype parameter — for optional or required fields, respectively.

    *   Field width for fixed-width Record Maps.

    *   #nnn — for SQL Column Number, if specified.

    *   standard, bitmap, idkey, or unique — type of index, if specified.

    For example, an SSN field in a fixed-width Record Map could have a summary `0..1 %String(PATTERN=3N1"-"2N1"-"4N); 11; #5; standard`. This means it is an optional field, with a datatype and datatype parameters `%String(PATTERN=3N1"-"2N1"-"4N)`, has a field width of 11, has an SQL column number of 5, and has a standard index.

*   Icons that allow you to move the field up or down or to delete the field. For composite fields, the plus icon allows you to add a new subfield.

# 11.3 Using the CSV Record Wizard

Ensemble provides a wizard to help automate the process of creating a record map from a sample file that contains comma separated values (CSV). You initiate the CSV Record Wizard either by choosing it on the Ensemble **Build** submenu or by clicking **CSV Wizard** from the ribbon bar on the **Record Mapper** page. The wizard handles only files with a single level of separator and does not handle leading data.

From the wizard, you enter values for the following fields:

**Sample file**

>   Either enter the complete path with filename of your sample or click **Select file** to navigate and choose your sample file.

**RecordMap name**

>   Enter the name of the record map to generate from your sample file.

**Separator**

> Separator character used in the sample file. You must enter control characters in hexadecimal; see "Common Control Characters," earlier in this chapter.

**Record Terminator**

> Specify how the sample file terminates a record. Choose one of the following:
>
> - CRLF — each record ends with a carriage return, followed by a line feed.
>
> - CR — each record ends with only a carriage return.
>
> - LF — each record ends with only a line feed.
>
> - Other — each record ends with control characters. Enter control character values in hexadecimal; see "Common Control Characters," earlier in this chapter.

**Character Encoding**

> Select the type of character encoding used in the sample file.

**Sample has header row**

> Select this check box if the sample file you provide contains a header row.
>
> In this case, Ensemble removes any punctuation and white space from values in the header row, and then uses the resulting values as property names in the record map. (If you do not select this option, Ensemble specifies the property names as `Property1`, `Property2`, and so on.)

**Keep SQL Column order**

> Select this check box to keep the SQL column order in the generated object.

**Quote-style escaping in use**

> Select this check box and the quote character if the sample file uses quote-style escaping of the separator.

When you are finished filling out the wizard form, click **Create RecordMap** to generate a new record map from your sample file and return to the **Record Mapper** page. You can now refine your record map to add detail to the generated properties. See "Editing the Record Map Properties" for details.

# 11.4 Record Map Class Structure

There are two classes that describe a record map:

- RecordMap that describes the external structure of the record and implements the record parser and record writer.

- Generated record class that defines the structure of the object containing the data. This object allows you to reference the data in data transformations and in routing rule conditions.

A record map business service reads and parses the incoming data and creates a message, which is an instance of the generated record class. A business process can read, modify or generate an instance of the generated record class. Finally, a record map business operation uses the data in the instance to write the outgoing data using the RecordMap as a formatting template. Both the RecordMap class and the generated record class have hierarchical structures that describe the data, but the generated object structure does not have to be identical to the RecordMap structure.

When you create a new record map and then save it in the Management Portal, this action defines a class for that extends the RecordMap class. In order to define the generated record class, you must click **Generate** in the Management Portal, which calls the **GenerateObject()** method in the EnsLib.RecordMap.Generator class. If you are working in Studio, compiling the RecordMap class definition does not create the code for the generated record class. You must use the Management Portal or call the **Generator.GenerateObject()** method from the Terminal or from code.

The RecordMap consists of a sequence of fields and composites :

* A field defines a data field with the specified type. The field type can specify parameters such as, VALUELIST, MAXVAL, MAXLEN, and FORMAT. In fixed-width records, the Record Mapper uses the field width to set the default value for the MAXVAL or MAXLEN parameters.

* A composite consists of a sequence of fields and composites. Composites can be nested within a RecordMap.

By default, the Record Mapper in the Management Portal uses the composite level to set the qualified names of the fields. In delimited records, the nesting level of composites elements determines the separator used between fields as follows:

1. Fields in RecordMap that are not contained in a composite are delimited by the first separator.

2. Fields that occur in a composite that is in the RecordMap are delimited by the second separator.

3. Fields that occur in a composite that is itself within a composite are delimited by the third separator.

4. Each additional level of composite nesting increments the separator used to delimit the fields.

Composites in fixed-width records provide documentation about the structure of the data but do not impact how Ensemble treats the message.

Each RecordMap object has a corresponding record object structure. When you generate the RecordMap, the Record Mapper defines and compiles a record object that defines the object representation of the record map. By default, the Record Mapper in the Management Portal names the record "Record" qualified by the name of the RecordMap, but you can explicitly set of the name of the record object in the **Target Classname** field. By default, the Record Mapper names fields within composites by qualifying the name with the composites that contain it. If you use the default qualified names, the structure of the record object class properties will be consistent with the structure of the RecordMap fields and composites, but if you assign other names to the fields, the structure of the record object class properties will not match the structure of the RecordMap fields and composites.

The record object class extends the EnsLib.RecordMap.Base, %Persistent, %XML.Adaptor, and Ens.Request classes. If the *RECORDMAPGENERATED* parameter of the existing class is 0, then the target class is not modified by the record map framework — all changes are then the responsibility of the production developer. The properties in the generated record class are dependent on the names of the fields in the record map.

The properties of the record object class correspond to the fields of the record map and have the following names and types:

* Names of fields with simple unqualified names that appear anywhere in the RecordMap or in composites within it. These properties have a type determined by the type of the field.

* Top-level names of fields with qualified names that appear anywhere in the RecordMap or in composites within it. These properties have an object type with a class defined by the fields that share the same top-level qualified name. These classes extend the %SerialObject and %XML.Adaptor classes. These classes are defined within the scope of the generated record class name. These classes, in turn, have properties corresponding to the next level of name qualification.

Consider an example, where you are defining a delimited record map, where the data contains three levels of separators, such as where the top-level separator field delimits the information about a person, the next level delimits the information about identification number, name, and phone number; and the final level delimits the elements within the address and name. For example, the message could start with:

```
French Literature,TA,199-88-7777;Jones|Robert|Alfred;
```

To define a RecordMap to handle these separators, you would need a composite at the level of person and one at the level of name. Thus the default field name for the FamilyName field could be Person.Name.FamilyName. This default name creates a deep level of class names in the record object class, such as the class NewRecordMap.Record.Person.Name that contains properties such as NewRecordMap.Record.Person.Name.FamilyName. You can avoid this deep level by prefacing the field names with the $ (dollar sign) character. If you do this, the classes and properties are all defined directly in the record scope. Using the same example, the class NewRecordMap.Record.Name would contain properties such as NewRecordMap.Record.FamilyName.

**Note:** The names used to qualify the field names are used to define properties with an object type. Consequently, you cannot use a name both to qualify a field name and to be the last part of a field name, which would define a property with the same name with a data type.

# 11.5 Object Model for the RecordMap Structure

You can achieve the class structure behavior either by directly creating XML or by using the EnsLib.RecordMap.Model.* classes to create an object projection of the RecordMap. In general, the favored approach is to use Management Portal, but you may prefer to use the object model to create the RecordMap structure. The structure of these classes follows the RecordMap class structure; use the class reference for further information at this level.

# 11.6 Using a Record Map in a Production

When you choose to generate an object class on the Record Mapper page, you create a class that you can use in a business service of an Ensemble production.

## 11.6.1 Sample Record Mapper Production

The ENSDEMO provides a production named *Demo.RecordMap.Production*. This production demonstrates how to use Ensemble to process fixed files using the generated classes.

# 12

# Using the Complex Record Mapper

If the message format consists of multiple heterogeneous records, you can use the Complex Record Mapper and handle these complex records with built-in file or FTP services and operations. Typically, these complex records have a header record followed by a pattern of records and terminated by a trailer record. These records can either be fixed-fields records or delimited records and can be optional and repeat. Typically, the records have leading data that identifies the kind of record.

## 12.1 Overview

Complex record maps can describe structured records that can contain:

1. An optional header record.

2. Sequence of elements where each element can be a record defined by a RecordMap or a sequence. A sequence can contain a sequence of records and other sequences.

3. An optional trailer record.

The records within a sequence can either be delimited records or fixed-width records. Although it is possible to mix delimited and fixed-width records within a complex record map, typically all of the records are delimited or all are fixed-width.

The following delimited sample data can be described by a complex record map. The data consists of a header identifying a semester in a college and information about students and classes that each student takes.

```
SEM|194;2012;Fall;20
STU|12345;Adams;John;Michael;2;john.michael.adams@example.com;617-999-9999
CLS|18.034;1;Differential Equations;4
CLS|21W.759;1;Writing Science Fiction;4
STU|12346;Adams;Jane;Michelle;3;jane.michelle.adams@example.com;
CLS|21L.285;1;Modern Fiction;3
CLS|7.03;1;Genetics;4
STU|12347;Jones;Robert;Alfred;1;bobby.jones@example.com;
CLS|18.02;1;Calculus;4
```

The complex record map that describes this data consists of:

1. Header record identified by the leading data "SEM|".

2. A sequence of students, where each student consists of:

    a. Student record identified by the leading data "STU|".

    b. A repeating class record identified by the leading data "CLS|".

The sequence of students defines the repeating structure of the complex record but does not correspond to a record in the data.

A complex record map defines both a file structure and an object structure. The complex record map file service parses a file using the file structure defined by the complex record map and then stores the data in an object defined by the object structure. The complex record map file operation performs the reverse—it takes the data in the object and writes it out to a file using the file structure defined by the complex record map.

# 12.2 Creating and Editing a Complex Record Map

You can create a new complex record map or edit an existing one by using the Complex Record Mapper in the Management Portal or by importing an XML definition and editing one in Studio.

The complex map has a top-level sequence in which you can define the fields. If your complex record consists of a single sequence of records, where the sequence does not repeat, you can enter the record maps of the records directly in the complex record map. But, if the entire sequence can repeat between the header and trailer, then you should enter a repeating sequence record as the only element of the top-level sequence.

## 12.2.1 Getting Started

To access the Complex Record Mapper from the Management Portal, click **Ensemble**, **Build**, and **Complex Record Maps**. From here, you have the following commands:

- **Open** — Opens an existing complex record map.

- **New** — Creates a new complex record map.

- **Save** — Saves your complex record map structure as a class in the namespace in which you are working in the package specified in the **Complex RecordMap Name**.

- **Generate** — Generates the complex record map parser code and the related persistent message complex record class object.

  To generate an object manually use the **Generate()** class method in EnsLib.RecordMap.ComplexGenerator.

- **Delete** — Deletes the current complex record map. You can optionally delete the related persistent message complex record class and all stored instances of the classes.

**Important:** The **Save** operation only writes the current complex record map to disk. In contrast, the **Generate** operation generates the parser code and the persistent object structure for the underlying objects.

## 12.2.2 Editing the Complex Record Map Properties

Whether you are entering properties for a new complex record map, starting from the wizard-generated map, or editing an existing map, the process is the same. For the complex record itself, enter or update values in the following fields:

**Complex RecordMap Name**

> Name of the complex record map. You should qualify the complex record map name with the package name. If you do not provide a package name and specify an unqualified complex record map name, the complex record map class is saved in the User package by default.

**Target Classname**

Name of the Ensemble class to represent the complex record. By default, the Complex Record Mapper sets the target class name to a qualified name equal to the complex record map name followed by ".Batch", but you can change the target class name. You should qualify the target class name with the package name. If you do not provide a package name and specify an unqualified target class name, the target class is saved in the User package by default.

**Character Encoding**

Character encoding for imported data records. The specified for the complex record map should be the same as the encoding for all record maps included in the complex record map. If they are not the same, the character encoding for the complex record map overrides the character encoding for the record maps.

**Annotation**

Text that documents the purpose and use of the complex record map.

If you create a new complex record map, the Complex Record Mapper creates a definition that consists of the following elements.

- Complex map name and type—enter the name and class for the complex map.

- Header—if your complex record has a header, enter the name and the class of the record map that describes the header.

- Trailer—if your complex record has a trailer, enter the name and the class of the record map that describes the trailer.

## 12.2.3 Editing the Complex Record Map Records and Sequences

A complex record map consists of the following:

1. An optional header record.

2. Sequence of elements where each element can be:

    - Record defined by a RecordMap. It can have the following properties in the complex record map:

        – Required. A value 0 means the record is optional and a value 1 means it is required.

        – Repeating with a minimum and maximum number of occurrences.

    - A nested sequence of elements.

3. An optional trailer record.

Each record is defined by a record map. A sequence is defined in the complex record map definition. It describes the structure of the data in the message but does not itself correspond to any fields in the data.

The header and trailer records are each defined by a record map. Although it is optional to include a header or trailer record in the complex record map definition, if the definition contains a header record, then the data must contain a header record, and if the definition contains a trailer record, then the data must contain a trailer record. Header and trailer records cannot repeat.

Every sequence must contain at least one record or sequence.

When you are editing a record, you can click the **Make Sequence** button to replace the record with a sequence. When you are editing a sequence, you can click the **Make Record** button to replace the sequence with a record.

You can specify the following properties for a record:

- Record name.

- RecordMap that defines the record format. The RecordMap specifies the **Leading data** that identifies the record, whether the record has fixed columns or is delimited, the separators, and the record terminator. For details on defining a RecordMap, see Using the Ensemble Record Mapper.

- Whether the record is required.

- Whether the record can be repeated. If the record can be repeated, you can also specify:

  – Minimum number of repetitions

  – Maximum number of repetitions

- Annotation that documents the purpose and use of the record in the complex record map.

You can specify the following properties for a sequence:

- Sequence Name

- Whether the sequence is required.

- Whether the sequence can be repeated. If the sequence can be repeated, you can also specify:

  – Minimum number of repetitions

  – Maximum number of repetitions

- Annotation that documents the purpose and use of the sequence in the complex record map.

# 12.3 Complex Record Map Class Structure

There are two classes that describe a complex record map in a similar manner to the two classes that describe a record map. The two classes that describe a complex record map are:

- Complex record map that describes the external structure of the complex record and implements the complex record parser and writer.

- Generated complex record class that defines the structure of the object containing the data. This object allows you to reference the data in data transformations and in routing rule conditions.

A complex record map business service reads and parses the incoming data and creates a message, which is an instance of the generated record class. A business process can read, modify or generate an instance of the generated complex record class. Finally, a complex record map business operation uses the data in the instance to write the outgoing data using the complex record map as a formatting template. Both the complex record map class and the generated complex record class have hierarchical structures that describe the data. The complex record map class and the generated complex record class have parallel structures. This is different from the RecordMap class, where the generated record class can have a different hierarchal structure.

When you create a new complex record map and then save it in the Management Portal, this action defines a class for that extends the EnsLib.RecordMap.ComplexMap and Ens.Request classes. In order to define the generated record class, you must click **Generate** in the Management Portal, which calls the **Generate()** method in the EnsLib.ComplexGenerator class. If you are working in Studio, compiling the ComplexMap class definition does not create the code for the generated record class. You must use the Management Portal or call the **ComplexGenerator.Generate()** method from the Terminal or from code. The generated class extends the RecordMap.ComplexBatch and Ens.Request classes.

The ComplexMap class defines the complex record structure in an XData definition that defines the ComplexBatch with records specified by RecordReference elements and sequences defined by RecordSequence elements. If the

*RECORDMAPGENERATED* parameter of the existing class is 0, then the target class is not modified by the complex record map framework — all changes are then the responsibility of the production developer.

The ComplexBatch class has properties that correspond to the following top-level elements in the complex map definitions:

*   Header record, if specified. This property has its type set to the generated record class for the specified record map.

*   Record that has its type set to the generated record class for the specified record map or, if the record can be repeated, its type is set to an array of the generated record class.

*   Sequence that has its type set to the class defined for the sequence or, if the sequence can be repeated, its type is set to an array of this class.

*   Trailer record, if specified. This property has its type set to the generated record class for the specified record map.

A class is defined for each sequence. The sequence class extends the ComplexSequence and %XML.Adaptor classes. The sequence class is defined within the package and namespace defined for the ComplexBatch class. All sequence classes are defined in this level of the namespace even if they are contained within other sequences.

Each sequence has properties that correspond to the records and sequences that it contains.

# 12.4 Using a Complex Record Map in a Production

To create a production that uses complex records, you do the following:

1.  Create the individual record maps for each part of the complex record, including the header and trailer. See "Using the Record Mapper" for a description of how to create the individual record maps. Note that if you intend to use a sample file, you should create a sample file that contains only the part of the complex record that you are defining in the individual record map. The sample file should not contain a complete complex record.

2.  Use the Complex Record Mapper to define the structure of the complex record.

3.  Create a production and add one or more of the built-in complex record services and operations.

4.  If your production is simply passing a complex record from one application to another, you may be able to use a simple routing engine process. But, if your production is converting one complex record into a different complex record, you will create a Data Transformation in the routing engine. If both the input and output complex records have the same structure, you can create a simple data transformation that connects the source and target fields. For example, you could use a simple data transformation to convert a complex record containing delimited records to a complex record containing fixed column records. But if the input complex record does not have the same structure as the output complex record, you must add code either within the data transformation or in a Business Process Language (BPL) process.

# 13

# Handling Batches of Records Efficiently

The RecordMap feature imports a single record at a time, but if you are importing or exporting a large number of records, you can gain substantial efficiency improvement by using RecordMap Batch. The RecordMap Batch feature handles homogeneous records and processes all of the records in a batch at one time. The batch can optionally be preceded by a header record and followed by a trailer record.

To create a RecordMap batch, you implement a class which inherits from %Persistent and EnsLib.RecordMap.Batch. The Batch class contains methods that handle parsing and writing out any headers and trailers associated with a specific batch. You must provide code that parses and writes your headers. For simple headers and trailers, you can use the EnsLib.RecordMap.SimpleBatch class, which inherits from the Batch class and provides code for handling simple headers and trailers. You can extend either of these two batch implementations if you need to process more complex header and trailer data.

Batch processing follows the approach used for other Ensemble message formats like HL7 and X12. This is particularly relevant for the built-in business operations which handle RecordMap batch objects: these business operations accept either Batch objects or RecordMap objects which extend EnsLib.RecordMap.Base, or a request of type BatchRolloverRequest. When records in a particular batch are received, the Batch is opened and the batch header is written to a temporary file, followed by any objects within that batch received by the operation. If the request in synchronous, the classname, Id, and the count of previously written records for the batch will be returned in a EnsLib.RecordMap.BatchResponse. Receipt of the batch object (which may be the default batch) will trigger the batch trailer to be written to the temporary file, and this file will then be sent to the desired destination by the adapter for the business operation. If a Batch object is received by itself, then the entire Batch will be written out to a temporary file which will then be transferred to the desired location.

Batch operations also support a default batch option whereby records which do not already belong to a batch are added to a default batch. Output of this batch can be triggered by either sending the batch object to the operation, or sending a BatchRolloverRequest to the operation. The business operation can also be configured to use schedule- or count-based rollover for the default batch. These options are configured on the business operation, and can be used simultaneously.

The options for services primarily concern the way the Visual Trace displays messages within a batch.

The RecordMap Batch operation creates temporary files in the process of generating the final output file. You can control the location of these temporary files by specifying the **IntermediateFilePath** setting of the RecordMap Batch operation. If the namespace's database is being mirrored, it is important that all mirror members have access to the temporary file in order to successfully failover during a RecordMap Batch operation. See the *Caché High Availability Guide* for information on mirroring.

The ENSDEMO namespace contains the RecordMapBatch example that demonstrates how to use RecordMap Batch to import and export batches of records.

# 14

# Less Common Tasks

This chapter discusses the following less common development tasks:

- Defining Custom Utility Functions

- Rendering Connections When the Targets Are Dynamic

- Using Ens.Director to Start and Stop a Production

- Using Ens.Director to Access Settings

- Invoking a Business Service Directly

- Creating or Subclassing Inbound Adapters

- Creating or Subclassing Outbound Adapters

- Including Credentials in an Adapter Class

- Overriding Ensemble Credentials

- Overriding Start and Stop Behavior

- Programmatically Working with Lookup Tables

- Defining a Custom Archive Manager

## 14.1 Defining Custom Utility Functions

Ensemble provides a set of utility functions that can be invoked from business rules and from DTL; these are described in "Ensemble Utility Functions" in *Developing Business Rules*. You can add your own functions, and the business rules engine and Business Rule Editor accommodate your extensions automatically.

To add a new utility function:

1. Create a new class that is a subclass of Ens.Rule.FunctionSet. This class must not extend any other superclasses, only Ens.Rule.FunctionSet.

2. For each function you wish to define, add a class method to your new function set class. There is no support for polymorphism, so to be precise, you must mark these class methods as final. You can view this in the existing Ens.Util.FunctionSet methods (Ens.Util.FunctionSet is a superclass of Ens.Rule.FunctionSet).

3. Compile the new class. The new functions are now available for use in rule expressions. To invoke these functions, use the ClassMethod name from your subclass. Unlike functions defined in Ens.Rule.FunctionSet, user-defined

method names must be fully qualified with the class that they belong to. This happens automatically if you add them by selecting names from the wizards in the Management Portal.

As an example, the following function set class provides date and time functions for use in business rules. Its class methods **DayOfWeek()** and **TimeInSeconds()** invoke the ObjectScript functions $ZDATE and $PIECE to extract the desired date and time values from the ObjectScript special variable $HOROLOG:

```
/// Time functions to use in rule definitions.
Class Demo.HL7.MsgRouter.Functions Extends Ens.Rule.FunctionSet
{

/// Returns the ordinal position of the day in the week,
/// where 1 is Sunday, 2 is Monday, and so on.
ClassMethod DayOfWeek() As %Integer [ CodeMode = expression, Final ]
{
$zd($H,10)
}

/// Returns the time as a number of seconds since midnight.
ClassMethod TimeInSeconds() As %Integer [ CodeMode = expression, Final ]
{
$p($H,",",2)
}

}
```

The default language inside a method is ObjectScript, as in the previous example, but you can also use Caché Basic if you define the method using the Language keyword as described in:

- The "Language" entry in the "Class Keywords" section of the *Caché ObjectScript Reference*.

- The "Methods" chapter in *Using Caché Objects*.

For a full list of functions and special variables available in Caché Basic or ObjectScript, see:

- *Caché Basic Reference*

- *Caché ObjectScript Reference*

Once you have added a new function as described in this topic, the syntax for referring to it is slightly different than for built-in functions. Suppose you define this function in a class that inherits from Ens.Rule.FunctionSet.

```
ClassMethod normalizaSexo(value as %String) as %String
```

After you compile the class, when you use one of the Ensemble visual tools such as the Routing Rule Editor or Data Transformation Builder, you see your function name **normalizaSexo** included in the function selection box along with built-in functions like Strip, In, Contains, and so on.

Suppose you choose a built-in function from the function selection box and look at the generated code. You see that Ensemble has generated the function call using double-dot method call syntax (in DTL) or has simply referenced the function by name (in a business rule). (These syntax rules are explained in "Ensemble Utility Functions" in *Developing Business Rules*.)

The following example is an <assign> statement from DTL that references the built-in Strip function with double-dot syntax:

```
<assign property='target.cod'
        value='..Strip(source.{PatientIDExternalID.ID},"<>CW")'
        action='set'/>
```

However, if you create your own, user-defined functions, the syntax for DTL is different. It is not enough simply to identify the function; you must also identify the full class name for the class that contains the class method for your function. Suppose your function **normalizaSexo** was defined in a class named HP.Util.funciones. In that case, after you chose a function from the function selection box and looked at the generated code, you would see something like the following example:

```
<assign property='target.sexo'
        value='##class(HP.Util.funciones).normalizaSexo(source.{Sex})'
        action='set'/>
```

You need to be aware of this syntax variation if you wish to type statements like this directly into your DTL code, rather than using the Data Transformation Builder to generate the code.

# 14.2 Rendering Connections When the Targets Are Dynamic

The Management Portal automatically displays the connections to and from a given business host, when a user selects that business host. For example:
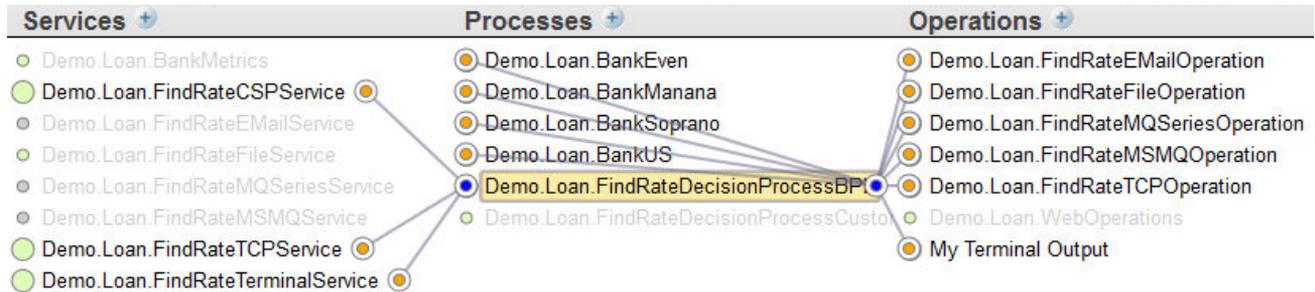


To do this, Ensemble reads the configuration settings for the business host and uses them.

If, however, the business service host its targets dynamically, at runtime, Ensemble cannot automatically display such connections. In this case, to display such connections, implement the **OnGetConnections()** callback method. Ensemble automatically calls this method (which does nothing by default) when it renders the configuration diagram.

**OnGetConnections()** has the following signature:

```
ClassMethod OnGetConnections(Output pArray As %String, item As Ens.Config.Item) [ CodeMode = generator
 ]
```

Where the arguments are as follows:

*   *pArray* — A string that this method creates. When the method returns, this string must contain a comma-separated list of the configured names of items to which this business service sends messages. Alternatively, if there are no targets at the present time, this string may be blank ("").

*   *item* — The Ens.Config.Item object that represents this business service.

For examples of overridden **OnGetConnections()** methods, use Studio to examine the built-in business services provided for use with electronic data interchange protocols such as HL7 and X12. These are described in detail in books such as the *Ensemble HL7 Version 2 Development Guide* and the *Ensemble X12 Development Guide*.

# 14.3 Using Ens.Director to Start and Stop a Production

During development, you typically use the Management Portal to start and stop a production. For live, deployed production, InterSystems recommends that you use the auto-start option as described in *Configuring Ensemble Productions*.

Another option is to start or stop a production programmatically. To do so, invoke the following methods in the Ens.Director class:

**StopProduction()**

Stop the currently running production in an Ensemble namespace:

```
Do ##class(Ens.Director).StopProduction()
```

**StartProduction()**

Start the specified production in the Ensemble namespace, as long as no other production is running:

```
Do ##class(Ens.Director).StartProduction("myProduction")
```

**RecoverProduction()**

Clean up a Troubled instance of a running production so that you can run a new instance in the Ensemble namespace:

```
Do ##class(Ens.Director).RecoverProduction()
```

It is not necessary to call **GetProductionStatus()** to see if the production terminated abnormally prior to calling **RecoverProduction()**. If the production is not Troubled, the method simply returns.

**GetProductionStatus()**

This method returns the production status via two output parameters, both of which are passed by reference. The first parameter returns the production name, but only when the status is Running, Suspended, or Troubled. The second parameter returns the production state, which is a numeric value equivalent to one of the following constants:

- $$$eProductionStateRunning

- $$$eProductionStateStopped

- $$$eProductionStateSuspended

- $$$eProductionStateTroubled

For example:

```
 Set tSC=##class(Ens.Director).GetProductionStatus(.tProductionName,.tState)
 Quit:$$$ISERR(tSC)
 If tState'=$$$eProductionStateRunning {
   $$$LOGINFO($$$Text("No Production is running.")) Quit
   }
```

You can use the production state macros such as $$$eProductionStateRunning in code outside of the Ensemble classes, for example in a general class or routine. To do this, you must add the following statement to the class:

```
#include Ensemble
```

It is not necessary to do this inside Ensemble classes, such as in business hosts.

Ens.Director provides many class methods, including many intended for use only by the Ensemble internal framework. InterSystems recommends that you use only the Ens.Director methods documented in this book, and only as documented.

**Note:** InterSystems recommends you *do not* use the **^%ZSTART** routine to control Ensemble production startup. The Ensemble startup mechanisms are much easier to use and are more closely tied to the production itself.

# 14.4 Using Ens.Director to Access Settings

The following Ens.Director class methods allow retrieval of production settings even when the production is not running:

### GetAdapterSettings()

Returns an array containing the values of all adapter settings for the identified configuration item: a business service or business operation. The array is subscripted by setting name. You can use the Caché $ORDER function to access the elements of the array. The first parameter for this method is a string that contains the production name and configuration item name separated by two vertical bars (||). The return value is a status value. If the status value is not $$$OK, the specified combination of production name (myProd) and configuration item name (myOp) could not be found.

```
Set tSC=##class(Ens.Director).GetAdapterSettings("myProd||myOp",.tSettings)
```

### GetAdapterSettingValue()

Returns the value of a named adapter setting for the identified configuration item: a business service or business operation. The first parameter is a string that contains the production name and configuration item name separated by two vertical bars (||). The second parameter is the name of a configuration setting. The third output parameter returns a status value from the call. For example:

```
Set val=##class(Ens.Director).GetAdapterSettingValue("myProd||myOp","QSize",.tSC)
```

If the returned status value is not $$$OK, the specified combination of production name (myProd) and configuration item name (myOp) could not be found, or a setting of the specified name (QSize) was not found in the configuration for that specified production and configuration item.

### GetCurrProductionSettings()

Returns an array containing the values of all production settings from the currently running production or the production most recently run. The array is subscripted by setting name. The return value for this method is a status value. If the status value is not $$$OK, no current production could be identified.

```
Set tSC=##class(Ens.Director).GetCurrProductionSettings(.tSettings)
```

### GetCurrProductionSettingValue()

Returns the string value of a named production setting from the currently running production or the production most recently run. The second output parameter returns a status value from the call. If this status value is not $$$OK, either a setting of the specified name was not found in the configuration for the current production, or no current production could be identified.

```
Set myValue=##class(Ens.Director).GetCurrProductionSettingValue("mySet",.tSC)
```

### GetHostSettings()

Returns an array containing the values of all settings for the identified configuration item: a business service, business process, or business operation. The array is subscripted by setting name. The first parameter for this method is a string that contains the production name and configuration item name separated by two vertical bars (||). The return value is a status value. If the status value is not $$$OK, the specified combination of production name (myProd) and configuration item name (myOp) could not be found.

```
Set tSC=##class(Ens.Director).GetHostSettings("myProd||myOp",.tSettings)
```

### GetHostSettingValue()

Returns the value of a named setting for the identified configuration item: a business service, business process, or business operation. The first parameter is a string that contains the production name and configuration item name separated by two vertical bars (||). The second parameter is the name of a configuration setting. The third output parameter returns a status value from the call. For example:

```
Set val=##class(Ens.Director).GetHostSettingValue("myProd||myOp","QSize",.tSC)
```

If the returned status value is not $$$OK, the specified combination of production name (myProd) and configuration item name (myOp) could not be found, or a setting of the specified name (QSize) was not found in the configuration for that specified production and configuration item.

### GetProductionSettings()

Returns an array containing the values of all production settings from the named production. The array is subscripted by setting name. The return value for this method is a status value. If the status value is not $$$OK, the specified production could not be found.

```
Set tSC=##class(Ens.Director).GetProductionSettings("myProd",.tSettings)
```

### GetProductionSettingValue()

Returns the value of a named production setting from the named production. The third output parameter returns a status value from the call. If this status value is not $$$OK, the specified production could not be found, or a setting of the specified name was not found in the configuration for the specified production.

```
Set val=##class(Ens.Director).GetProductionSettingValue("prod","set",.tSC)
```

Ens.Director provides many class methods, including many intended for use only by the Ensemble internal framework. InterSystems recommends that you use only the Ens.Director methods documented in this book, and only as documented.

# 14.5 Invoking a Business Service Directly

There are times when you want to invoke a business service directly, from a job that has been created by some other mechanism, such as a language binding, Caché Server Pages, SOAP, or a routine invoked from the operating system level. You can do so only if the value of the *ADAPTER* class parameter is null; this type of business service is called an *adapterless* business service.

For a business service to work, you must create an instance of the business service class. You *cannot* create this instance by calling the **%New()** method. Instead, you must use the method **CreateBusinessService()** of Ens.Director. For example:

```
Set tSC = ##class(Ens.Director).CreateBusinessService("MyService",.tService)
```

An Ensemble production does not allocate a job for this business service at production startup; it assumes a **Pool Size** setting of 0.

The **CreateBusinessService()** method does the following:

1.  It makes sure that a production is running and that the production defines the given business service.

2.  It makes sure that the given business service is currently enabled.

3.  It resolves the configuration name of the business service and instantiates the correct business service object using the correct configuration values (a production may define many business services using the same business service class but with different names and settings).

If the **CreateBusinessService()** method succeeds, it returns, by reference, an instance of the business service class. You can then invoke its **ProcessInput()** method directly. You must provide the **ProcessInput()** method with an instance of the input object it expects. For example:

```
If ($IsObject(tService)) {
  Set input = ##class(MyObject).%New()
  Set input.Value = 22
  Set tSC = tService.ProcessInput(input,.output)
}
```

If you call the **CreateBusinessService()** method and the business service calls an InProc business operation that requires access to credentials, see the compatibility note "New Global and Database Used to Store Credentials Passwords" in the *Ensemble Release Notes*.

Ens.Director provides many class methods, including many intended for use only by the Ensemble internal framework. InterSystems recommends that you use only the Ens.Director methods documented in this book, and only as documented.

# 14.6 Creating or Subclassing Inbound Adapters

This section describes how to create or subclass an inbound adapter.

## 14.6.1 Introduction to Inbound Adapters

An inbound adapter is responsible for receiving and validating requests from external systems.

Inbound adapter classes work in conjunction with a business service classes. In general, the inbound adapter contains general-purpose, reusable code while the business service contains production-specific code (such as special validation logic). Typically you implement inbound adapter classes using one of Ensemble's built-in adapter classes. The following figure shows how a production accepts incoming requests:



In general, when an external application makes a request for a certain action to be performed, the request comes into Ensemble via an inbound adapter, as shown in the previous figure. The requesting application is called a "client" application, because it has asked the production to do something. This application is a "client" of the production. The featured element at this step is the inbound adapter. This is a piece of code that "adapts" the client's native request format to a one that is understandable to the production. Each application that makes requests of a production must have its own inbound adapter. No change to the client application code is needed, because the adapter handles calls that are already native to the client application.

## 14.6.2 Defining an Inbound Adapter

To create an inbound adapter class, create a class as follows:

- Your class must extend Ens.InboundAdapter (or a subclass).

- Your class must implement the **OnTask()** method, as described in "Implementing the OnTask() Method."

- Your class can define settings. See "Adding and Removing Settings," earlier in this book.

- Your class can implement any or all of the startup and teardown methods. See "Overriding Start and Stop Behavior," in this chapter.

- Your class can include Ensemble credentials. See "Including Credentials in an Adapter Class," in this chapter.

- Your class can contain methods to accomplish work internal to itself.

The following shows an example:

```
Class MyProduction.InboundAdapter Extends Ens.InboundAdapter
{

Parameter SETTINGS = "IPAddress,TimeOut";

Property IPAddress As %String(MAXLEN=100);

Property TimeOut As %Integer(MINVAL=0, MAXVAL=10);

Property Counter As %Integer;

Method OnTask() As %Status
{
  #; First, receive a message (note, timeout is in ms)
  Set msg = ..ReceiveMessage(..CallInterval*1000,.tSC)

  If ($IsObject(msg)) {
    Set tSC=..BusinessHost.ProcessInput(msg)
  }

  Quit tSC
}

}
```

**Note:** Studio provides a wizard that you can use to create a class stub similar to the preceding. To access this wizard, click **File —> New** and then click the **Production** tab. Then click **Adapter** and click **OK**.

## 14.6.3 Implementing the OnTask() Method

The **OnTask()** method is where the actual work of the inbound adapter takes place. This method is intended to do the following things:

1. Check for an incoming event. The inbound adapter can do this in many different ways: For example, it could wait for an incoming I/O event (such as reading from a TCP socket), or it could periodically poll for the existence of external data (such as a file).

   Most prebuilt inbound adapters have a setting called CallInterval that controls the time interval between calls to **OnTask()**. You could use this approach as well.

2. Package the information from this event into an object of the type expected by the business service class.

3. Call the **ProcessInput()** method of the business service object.

4. If necessary, send an acknowledgment back to external system that the event was received.

5. If there is more input data, **OnTask()** can do either of the following:

   - Call **ProcessInput()** repeatedly until all the data is retrieved.

   - Call **ProcessInput()** only once per CallInterval, even if multiple input events exist.

When designing an inbound adapter, it is important to keep in mind that the **OnTask()** method must periodically return control to the business service; that is, the **OnTask()** method must not wait indefinitely for an incoming event. It should,

instead, wait for some period of time (say 10 seconds) and return control to the business service. The reason for this is that the business service object must periodically check for events from within Ensemble, such as notification that the production is being shut down.

Conversely, it is also important that the **OnTask()** method waits for events efficiently—sitting in a tight loop polling for events wastes CPU cycles and slows down an entire production. When an **OnTask()** method needs to wait, it should wait in such a way as to let its process go to sleep (such as waiting on an I/O event, or using the **Hang** command).

If your class is a subclass of an Ensemble adapter, then it probably implements the **OnTask()** method; therefore, your subclass might need to override a different method as specified by the inbound adapter class.

# 14.7 Creating or Subclassing Outbound Adapters

This section describes how to create or subclass an outbound adapter.

## 14.7.1 Introduction to Outbound Adapters

An outbound adapter is responsible for sending requests to external systems. The following figure shows how a production relays outgoing requests.



The outbound adapter is a piece of code that "adapts" the native programming interface of an external application or external database into a form that is understandable to an Ensemble production. Each external application or database that serves a production by means of a business operation must have its own outbound adapter. However, not every method in the external application or database needs to be mapped to the outbound adapter; only those operations that the production requires. As with inbound adapters, no change to the external application itself is needed to create an outbound adapter. And, the adapter itself is conceptually simple: It relays requests, responses, and data between the production and a specific application or database outside the production.

Outbound adapter classes work in conjunction with a business operation classes. In general, the outbound adapter contains general-purpose, reusable code while the business operation will contain production-specific code (such as special processing logic). Typically you will implement your outbound adapter classes using one of Ensemble's built-in adapter classes.

## 14.7.2 Defining an Outbound Adapter

To create an outbound adapter class, create a class as follows:

- Your class must extend Ens.OutboundAdapter (or a subclass).

- Your class must define one or more methods for the corresponding business operation to invoke. Every outbound adapter is free to define its own API (set of methods) for use by the associated business operation classes.

- Your class can define settings. See "Adding and Removing Settings," earlier in this book.

- Your class can implement any or all of the startup and teardown methods. See "Overriding Start and Stop Behavior," in this chapter.

- Your class can include Ensemble credentials. See "Including Credentials in an Adapter Class," in this chapter.

- Your class can contain methods to accomplish work internal to itself.

# 14.8 Including Credentials in an Adapter Class

To include Ensemble credentials in an adapter class, do the following in the class definition:

- Include a setting named Credentials.

- Define a method called **CredentialsSet()** that uses the value of the Credentials setting as a key to look up the username and password in the Credentials table. It then instantiates a credentials object that contains the username and password.

# 14.9 Overriding Ensemble Credentials

While the Ensemble credentials system centralizes management and keeps login data out of source code, sometimes you need to write code that gets credentials from another source. For example, your code might retrieve a username and password from a web form or cookie, and then use them with the HTTP outbound adapter to connect to some other site.

The way to handle this is in your business service or business operation code, do both of the following, before calling any adapter methods:

- Provide code that instantiates a credentials object and assigns username and password values to it

- Do not subsequently set the adapter Credentials property or call the adapter **CredentialsSet()** method, or the values may be reset.

For example:

```
If ..Adapter.Credentials="" {
   Set ..Adapter.%CredentialsObj=##class(Ens.Config.Credentials).%New()
}
Set ..Adapter.%CredentialsObj.Username = tUsername
Set ..Adapter.%CredentialsObj.Password = tPassword
```

Code such as this provides a credentials object that the EnsLib.HTTP.OutboundAdapter can use, but the values inside the object do not come from the Credentials table.

# 14.10 Overriding Start and Stop Behavior

Ensemble provides a set of callback methods that you can override in order to add custom processing at start and stop times during the life cycle of the production, its business hosts, or its adapters. By default, these methods do nothing.

## 14.10.1 Callbacks in the Production Class

If you have code that must execute before a production starts up, but that requires the Ensemble framework to be running before it can execute, you must override the **OnStart()** method in the production class. Place these code statements in **OnStart()** so that they execute in the proper sequence: that is, *after* Ensemble has started, but before the production begins accepting requests. The **OnStop()** method is also available to perform a set of tasks before the production finishes shutting down.

## 14.10.2 Callbacks in Business Host Classes

Each business host — business service, business process, or business operation — is a subclass of Ens.Host. In any of these classes you may override the **OnProductionStart()** method to provide code statements that you want Ensemble to execute on behalf of this host at production startup time. You can also implement the **OnProductionStop()** method.

For example, if your production requires different initial settings for property values, set the value in the **OnInit()** method of the business operation. For example, to change the initial setting of the LineTerminator property to depend on the operating system:

```
Method OnInit() As %Status
  {
      Set ..Adapter.LineTerminator="$Select($$$isUNIX:$C(10),1:$C(13,10))"
      Quit $$$OK
  }
```

## 14.10.3 Callbacks in Adapter Classes

An adapter class may override the **OnInit()** method. This method is called after the adapter object has been created and its configurable property values have been set. The **OnInit()** method provides a way for an adapter to perform any special setup actions.

For example, the following **OnInit()** method establishes a connection to a device when the adapter is started — assuming that this adapter also implements a **ConnectToDevice()** method:

```
Method OnInit() As %Status
{
  // Establish a connection to the input device
  Set tSC = ..ConnectToDevice()
  Quit tSC
}
```

An adapter class can also override the **OnTearDown()** method. This method is called during shutdown before the adapter object is destroyed. The **OnTearDown()** method provides a way for an adapter to perform any special cleanup actions.

For example, the following **OnTearDown()** method closes a connection to a device when the adapter is stopped, assuming that this adapter also implements a method named **CloseDevice()**:

```
Method OnTearDown() As %Status
{
  // close the input device
  Set tSC = ..CloseDevice()
  Quit tSC
}
```

# 14.11 Programmatically Working with Lookup Tables

Ensemble provides the utility function called Lookup() so that you can easily perform a table lookup from a business rule or DTL data transformation. This function works only after you have created at least one lookup table and have populated it with appropriate data.

For information on defining lookup tables, see "Defining Data Lookup Tables" in *Configuring Ensemble Productions*.

If you need more direct manipulation of lookup tables than the Management Portal provides, use the Ens.Util.LookupTable class. This class exposes lookup tables to access via objects or SQL. Additionally, it provides class methods to clear tables, export data as XML, and import data from XML.

Ens.Util.LookupTable provides the following string properties:

**TableName**

Name of the lookup table, up to 255 characters. You can view the lookup tables defined in a namespace by selecting **Ensemble**, **Configure**, and **Data Lookup Tables** in the Ensemble portal and then selecting **Open**.

**KeyName**

Key for the entry within the lookup table, up to 255 characters. This is the value from the **Key** field on the **Lookup Tables** page (select **Ensemble** > **Configure** > **Data Lookup Tables**).

**DataValue**

Value associated with this key in the lookup table, up to 32000 characters. This is the value from the **Value** field on the **Lookup Tables** page.

A sample SQL query might be:

```
SELECT KeyName,DataValue FROM Ens_Util.LookupTable WHERE TableName = 'myTab'
```

Ens.Util.LookupTable also provides the following class methods:

**%ClearTable()**

Deletes the contents of the specified lookup table.

```
do ##class(Ens.Util.LookupTable).%ClearTable("myTab")
```

**%Import()**

Imports lookup table data from the specified XML file. For the import to be successful, the file must use the same XML format as that provided by the **%Export()** method of this class.

```
do ##class(Ens.Util.LookupTable).%Import("myFile.xml")
```

**%Export()**

Exports lookup table data to the specified XML file. If the file exists, Ensemble overwrites it with new data. If the file does not already exist, Ensemble creates it. The following example exports only the contents of the specified lookup table, myTab:

```
do ##class(Ens.Util.LookupTable).%Export("myFile.xml","myTab")
```

The following example exports the contents of all lookup tables in the namespace:

```
do ##class(Ens.Util.LookupTable).%Export("myFile.xml")
```

The resulting XML file looks like the following example. Note that all entries, in all tables, appear as sibling <entry> elements inside a single <lookupTable> element.

```
<?xml version="1.0"?>
<lookupTable>
  <entry table="myOtherTab" key="myKeyA">aaaaaa</entry>
  <entry table="myOtherTab" key="myKeyB">bbbbbbbbb</entry>
  <entry table="myTab" key="myKey1">1111</entry>
  <entry table="myTab" key="myKey2">22222</entry>
  <entry table="myTab" key="myKey3">333333</entry>
</lookupTable>
```

For each <entry>, the *table* attribute identifies the table that contains the entry. The *key* attribute gives the name of the key. The text contents of the <entry> element provide the entry's value.

In addition to the XML format described above, you can use the SQL Import Wizard to import comma-separated value (CSV) files that list tables and keys.

# 14.12 Defining a Custom Archive Manager

For Ensemble, the Management Portal provides a tool called the Archive Manager; this is described in *Managing Ensemble*. You can define and use a custom Archive Manager. To do so, create a class as follows:

- It can use Ens.Archive.Manager as a superclass.

- It must define the DoArchive() method, which has the following signature:

  ```
  ClassMethod DoArchive() As %Status
  ```

An alternative option is to use the Enterprise Message Bank, which enables you to archive messages from *multiple* productions. For an overview, see "Defining the Enterprise Message Bank," earlier in this book.

# 15

# Testing and Debugging

This topic explains the facilities available for testing and debugging Ensemble productions. The information is also useful for troubleshooting and tuning production software that is already in use at the enterprise. Topics include:

- Correcting Production Problem States

- Testing from the Management Portal

- Debugging Production Code

- Enabling %ETN Logging

## 15.1 Correcting Production Problem States

If a production is Suspended or Troubled, read this section.

### 15.1.1 Recovering a Suspended Production

A production acquires the Suspended status when, at the end of the shutdown sequence, some queues still contain synchronous messages. If this is the case, you can start the Suspended production again to permit these messages to be processed.

However, if the problem that originally caused the Suspended status is not resolved, starting a Suspended production can just as easily add more synchronous messages to a queue, without discharging the previous messages and return the production to a Suspended status.

You may stop a Suspended production. InterSystems advises you do so only during initial development and not on a deployed, live installation.

If you find this is the case, you may sometimes need to take more drastic action to fully clean up after a production that goes into the Suspended state. Ensemble provides the **CleanProduction()** method for this case.

CAUTION:    Never use this procedure on a live, deployed Ensemble production. The **CleanProduction()** method removes all messages from queues and removes all current information about the production. Use this procedure only on a production that is still under development. If a live, deployed Ensemble production goes into a Suspended state, contact the InterSystems Worldwide Response Center (WRC) for assistance.

If you have exhausted all means to correct the problem and, therefore, are sure that a Suspended production *must* stop, discarding even synchronous messages that have not yet been delivered, then use the following procedure:

1. Start the Suspended production from the **Production Configuration** page by clicking **Start** in the ribbon bar; the production status changes to Running.

2. Now click **Stop** to verify the problem still exists. If the production stops successfully, there is no need to clean the production; if the production status returns to Suspended, continue with the next step.

3. Kill the production from a Terminal session as follows:

    a.  Change to the appropriate namespace:

    ```
    ZN "EnsSpace"
    ```

    Where EnsSpace is the name of the Ensemble namespace where the Suspended production runs.

    b.  Enter the following command:

    ```
    Do ##class(Ens.Director).CleanProduction()
    ```

4. Verify the production status from the **Production Configuration** page; the **Status** of the production should now be Stopped.

**Important:**    Only run the **CleanProduction()** method on a production in a Suspended state.

## 15.1.2 Recovering a Troubled Production

A production acquires a status of Troubled if Ensemble is stopped but the production did not shut down properly. This can happen if you restarted Ensemble or rebooted the machine without first stopping the production.

In this case the **Recover** command appears on the **Production Configuration** page. Click **Recover** to shut down and clean up the troubled instance of the production so that you can run a new instance when you are ready.

Or you may need to use the command line to recover the production. See "Using Ens.Director to Start and Stop a Production," earlier in this book.

# 15.2 Testing from the Management Portal

You can use the Management Portal to perform several tasks as you develop, test, and debug your productions:

*   Ability to view and modify system configuration.

*   Ability to start and stop a production.

*   Ability to view queues and their contents; messages and their details; adapters and actors and their status; business processes and their status; code and graphical representations of configured items.

*   Ability to view, sort, and selectively purge Event Log entries.

*   Ability to suspend (and later resend) messages whose connectivity is temporarily blocked.

*   Ability to filter and search the message warehouse for specific messages, by category or message content, using a graphical user interface or by entering SQL SELECT commands.

*   Ability to visually trace message activity using a graphical user interface.

*   Ability to create and view statistical reports.

Portal features that are most useful for developers are the Monitor Service, which constantly collects runtime data, the Testing Service, which you can use to issue simulated requests into a production that you are developing, and the Event

Log, which logs the status messages issued by business hosts. Use these features together to generate test data and study the results.

For information on using the portal, see *Managing Ensemble*.

The Management Portal has a Test menu where you can test both business hosts and data transformations. It contains the following items:

- Business Hosts — The **Testing Service** page (**Ensemble** > **Test** > **Business Hosts**) allows you to test business processes and business operations.

- Data Transformations — This option brings you to another page, where you can select a data transformation and click Test. For details, see the Testing Data Transformations section.

## 15.2.1 Using the Testing Service

The **Testing Service** page allows you to test a business process or business operation of a running Ensemble production in the active namespace. You can navigate to the Testing Service from the following locations in the Management Portal:

- From the **Ensemble** menu, click **Test**.

- From the configuration diagram on the **Production Configuration** page, click a business process or business operation in the left pane and click **Test** on the **Actions** tab.

Now, to use the Testing Service:

1. Make sure the appropriate production is running. See *Managing Ensemble*.

2. Make sure that testing is enabled for this production. From the **Production Configuration** page:

   a. Select the **Production Settings** link.

   b. On the **Settings** tab, open the **Development and Debugging** property list and check the **Testing Enabled** check box.

   c. Select **Apply**.

3. In the Management Portal, click **Ensemble** > **Test** > **Business Hosts** to display the **Testing Service** page.

   This page provides options that let you select either a **Business Process** or **Business Operation** as the target of your testing.

4. Select either **Business Process** or **Business Operation** as appropriate.

5. Select the testing target from the drop-down list.

6. Select the type of message to send. The page displays the following fields:

   - **Current Production** — The name of the currently running production (view-only).

   - **Target** — The business process or business operation that you selected in the previous Testing Service page (view-only).

   - **Request Type** — Select from a list of request messages. Only the request types that are valid for the **Target** are listed, including subclasses of supported types.

7. Enter values for the properties of the message that you selected.

   If the request message has no properties, none are displayed.

   If you are testing a virtual document message (HL7, X12), there is a free-form box where you can paste test message content. Below this box you can enter object properties for the message.

8. Click **Invoke** to submit the request with the values you entered and view the results.

If time elapses while the Testing Service attempts the request, a *Waiting* page displays the following view-only information:

- **Target** — The session ID associated with the request.

- **Request Type** — The request type of the selected target.

- **Session Id** — The session ID associated with the request.

- **Request Sent** — The date and time when the request was sent.

- **Response received** — The status *Waiting* and a graphical progress bar, indicating that work is being done.

Finally, the *Results* page displays any output values from the response generated by your request, including any errors with the full error message text.

You can perform one of the following commands when the test completes:

- Click **Done** to return to the Ensemble home page.

- Click **Trace** to navigate to the Visual Trace page to visually follow the path of the message through the production.

You can also use the classes and methods in the EnsLib.Testing package. See the EnsLib.Testing.Service entry in the *Class Reference* for details.

# 15.3 Debugging Production Code

The first step in debugging is to enable tracing as described in *Monitoring Ensemble Productions*. If this does not reveal the problem, you can step into the code using the debugger, as follows:

1. Using Studio, insert the *BREAK* command where you want to start debugging.

2. Enable the **Foreground** setting for the business host that uses the class you want to debug.

3. Start the production. The job that you marked in Step 2 runs in the foreground in the Terminal.

4. When that *BREAK* command is reached, the Terminal enters debug mode and you can step through the code.

For details, see "Command-Line Routine Debugging" in *Using Caché ObjectScript*.

You can also debug your code in Studio:

- With a business service, you still specify that it should be run in the foreground and use a BREAK to get the process ID.

- With a business process, you can use Studio to set a breakpoint and then attach to the process to debug using Studio. In contrast, if you use BPL, Studio displays in .INT code.

- With a business operation, you can use Studio to set a breakpoint and then attach to the operation process to debug.

For details, see "Using the Studio Debugger" in *Using Studio*.

# 15.4 Enabling %ETN Logging

The Event Log automatically includes partial information about system-level exceptions (including exceptions in your code). These Event Log entries end as follows by default:

```
-- logged as '-' number - @' quit arg1/arg2 }'
```

To get more complete information about such errors:

1. Set the `^Ens.Debug("LogETN")` global node to any value.

   This causes Ensemble to record additional details for system-level exceptions.

2. Rerun the code that you think caused the exception (for example, resend messages).

3. Recheck the Event Log, which now contains entries that end as follows:

   ```
   -- logged as '25 Sep 2012' number 15 @' quit arg1/arg2 }'
   ```

   This information refers to an entry in the Application Error Log — specifically it refers to error 15 in the Application Error Log for 25 September 2012.

4. Then to examine these exceptions, you can either:

   - Use the **View Application Error Log** page. To access this page, click **System Operation** > **System Logs** > **Application Error Log**.

   - Use the **^%ERN** routine. For details, see "Other Debugging Tools" in the chapter "Command-line Routine Debugging" in *Using Caché ObjectScript*.

# 16

# Deploying a Production

Typically, you develop a production on a development system and then, after completing and testing the production on a test deployment, you deploy it on a live production system. This chapter describes how to use the Management Portal to package a deployment from a development system and then to deploy it on another system. It also describes how you can develop and test changes to a production and then deploy those updates to a system running with live business data. This chapter includes the following sections:

- Overview of Deploying a Production
- Exporting a Production
- Deploying a Production on the Target System

## 16.1 Overview of Deploying a Production

You can deploy a production using either the Management Portal or Studio. The Management Portal automates some steps that you need to perform manually using Studio. If you have a live production that is being used and are developing updates to the production, you need to ensure that the live production is updated without interrupting your processing of business data. At its simplest level deploying a production is done by exporting the XML definition of the production from one system and importing and compiling the XML on a target system. The most important issues for a successful deployment from development to live systems are:

- Ensuring that the XML deployment file has all required components.
- Testing the deployment file on a test system before deploying it to the live system.
- Ensuring that the deployment file is loaded on the target system without disrupting the live production.

Typically, deploying a production to a live system is an iterative process, with the following steps:

1. Export the production from the development system.
2. Deploy the deployment file on a test system.
3. Ensure that the production has all required components and runs properly on the test system. If any failures are found fix them and repeat step 1.
4. After the production has been deployed to the test system without errors, deploy the deployment file to the live system. Monitor the live system to ensure that the production continues to run correctly.

You should ensure that the test system environment matches as closely as possible the environment of the live system. If you are updating an existing production, the production on the test system should match the production on the live system before the update is applied. If you are deploying a production on a new Ensemble installation, the test system should be a new Ensemble installation.

In order to update a component in a running production, you must do the following:

1. Load the updated XML on the system.

2. Compile the XML.

3. Update the running instances of the component to the new code by disabling and re-enabling the component.

The deployment process is slightly different depending on whether or not the target system is already running a version of the production. If the target system is running an older version of the production, then the deployment file should contain only the updated components and some configuration items, and, in most cases, it should not contain the definition of the production class. If the target system does not contain the production, the deployment file should contain all production components and settings.

If you use the **Deploy Production Changes** page (**Ensemble** > **Manage** > **Deploy Changes** > **Deploy**) to deploy updates to a running production, the portal automatically does the following:

1. Creates a rollback and log file.

2. Disables components that have configuration items in the deployment file.

3. Imports and compiles the XML. If there is a compilation error, the portal automatically rolls back the deployment.

4. Enables the disabled components

There are some conditions where you have to explicitly stop and restart a component or the entire production. If you are using Studio or importing the classes from the Management Portal **System Explorer**, then you have to perform these steps manually.

# 16.2 Exporting a Production

To export the XML for a production using the Management Portal, open the production, click **Production Settings** and the **Actions** tab and then click the **Export** button. Ensemble selects all business services, business processes, business operations, and some related classes, and then displays the following form to allow you to add export notes and additional components.

You can also export a business service, process, or operation by selecting the component in the production configuration and then clicking the **Export** button on the **Actions** tab. In both cases, you can add additional components to the package by clicking on one of the buttons and selecting a component. You can remove components from the package by clearing the check box.

You can use the export notes to describe what is in the deployment package. For example, you can describe whether a complete production is in the package or set of components that are an update to a production. The export notes are displayed when you are deploying the package to a target system using the Management Portal.

When you are exporting a deployment package, the first decision you should make is whether the target system has an older version of the production.

If you are deploying the production as a new installation, you should:

- Include the definition of the production class.

- Include the production settings.

- Include the definitions of all components used in the production.

- Exclude the production settings (ptd file) for each component. This would duplicate the definition in the production class.

If you are deploying the production to update a live version of the production, you should:

- Exclude the definition of the production class.

- Exclude the production settings unless there are changes and you want to override any local settings.

- Include the definition of all components that have been updated.

- Include the production settings (ptd) file for any component whose setting have been changed or that should be disabled before the XML is imported and compiled.

Although many components are included by default in the package, you have to add others manually by selecting one of the buttons in the **Add to package** section. For example, if any of the following are used in your production, you need to add them manually:

- Record maps—the defined and generated classes are included.

- Complex record maps—the defined and generated classes are included.

- Lookup tables

- User classes referenced in code

- System default settings or schedule specifications that are set as deployable

If you have a set of classes defined in a studio project file, you can include those classes in the deployment by clicking the **Studio Project Files** button and selecting the file.

The **Production Settings** button allows you to add the production ptd file. This XML defines the following:

- Production comments

- General pool size

- Whether testing is enabled and whether trace events should be logged.

You can deselect any component in the list by clearing its check box. You can select a component by checking its box. The **Select All** button checks all the boxes and the **Unselect All** button clears all check boxes.

Once you have selected the components for the deployment package, create it by clicking **Export**.

The deployment package contains the following information about how it was created:

- Name of the system running Ensemble

- Namespace containing the production

- Name of the source production

- User who exported the production

- UTC timestamp when the production was exported

You should keep a copy of the deployment file on your development system. You can use it to create a new deployment package with the latest changes to the components. Keeping a copy of the deployment file saves you from having to manually select the components to be included in the deployment file.

To create a new deployment package using an existing deployment package to select the components, do the following:

1. On the development system with the updated production, click **Production Settings** and the **Actions** tab and then the **Re-Export** button.

2. Select the file containing the older deployment package.

3. Ensemble selects the same components from the current production that were included in the older deployment package.

4. If there were any components missing from the older deployment package or if you have added new components to the production, add the missing components manually.

5. Click the **Export** button to save a new deployment package with the updated components.

**Note:** If a production uses XSD schemas for XML documents or uses an old format schema for X12 documents, the schemas are not included in the XML deployment file and have to be deployed through another mechanism. Ensemble can store X12 schemas in the current format, in an old format, or in both formats. When you create a deployment file, it can contain X12 schemas in the current format, but it does not contain any X12 schemas in the old format or any XSD schemas for XML documents. If your production uses an old format X12 schema or uses any XSD XML schema, you must deploy the schemas independently of deploying the production. For the schemas that are not included in the deployment file, they can be deployed to a target system by either of the following means:

- If the XML or X12 schema was originally imported from an XSD or SEF file and that file is still available, import the schema on the target system by importing that file. XSD files can be used to import XML schemas and SEF files can be used to import X12 schemas.

- Export the underlying Caché global that contains the schema and then import this on the target system. To export a global, select **System Explorer** > **Globals**, select the desired globals and then select **Export**. The X12 schemas are stored in the `EnsEDI.Description`, `EnsEDI.Schema`, `EnsEDI.X12.Description`, and `EnsEDI.X12.Schema` globals. The XML schemas are stored in the `EnsEDI.XML.Schema` global. See "Exporting Globals" in the *Using Caché Globals* guide for details on exporting globals.

# 16.3 Deploying a Production on a Target System

The Management Portal automates the process of deploying a production from a development system to a live system. This section describes what Ensemble does when you are loading a new version of a production on a live system.

Once you have the deployment package XML file, you can load it on a target system. In the Management Portal, select the correct namespace and click **Ensemble**, **Manage**, **Deployment Changes**, **Deploy**, and then click the **Open Deployment** button and select the file containing the XML deployment package. The form lists the new and changed items in the deployment package, displays the deployment notes that were specified when the package was created and allows you to specify the following deployment settings:

- Target production—specifies the production that the components will be added to. If the deployment package includes the production class from the source production, then the target production is set to the source production and cannot be changed. Otherwise, Ensemble sets the default production to the currently open production, but allows you to change it.

- Rollback file—specifies the file to contain the rollback information. The rollback file contains the current definitions of all components that are being replaced by the deployment.

- Deployment log file—contains a log of the changes caused by the deployment.

When you have read the deployment notes and made any changes to the deployment settings, complete the deployment by clicking the **Deploy** button. As part of deploying the package, Ensemble does the following to stop the production, load the new code, and then restart the production.

1.  Create and save the rollback package.

2.  Disable the components in the production that have a production settings (ptd) file in the deployment package.

3.  Import the XML file and compile the code. If there is an error compiling any component, the entire deployment is rolled back.

4.  Update the production settings.

5.  Write a log detailing the deployment.

6.  Enable the production components that were disabled if their current setting specify that they are enabled.

To undo the results of this deployment change, you can select the rollback file with an **Open Deployment** and then click the **Deploy** button.

If you are using studio, you use the **Tools Import Local** or **Import Remote** to import the XML file. But you must then manually compile the components and then disable and re-enable them in the production. Using the **Import** classes button on the Management Portal **System Explorer** does automatically compile the classes, but it does not create a rollback package and disable the components.

# A

# Life Cycle of a Production and Its Parts

This appendix describes the life cycle of a production and its component parts, for reference.

## A.1 Life Cycle of a Production

### A.1.1 Production Startup

When a production starts, the sequence of actions is as follows:

1. The production class is instantiated; its optional **OnStart()** method executes.

2. The production instantiates each business operation and executes its optional **OnProductionStart()** method.

3. The production instantiates each business process and executes its optional **OnProductionStart()** method.

4. The production clears the business metric cache of any metric values left over from a previous run.

5. The production instantiates each business service and executes its optional **OnProductionStart()** method.

6. The production processes any items already placed in queues. This includes asynchronous messages that were queued when the production stopped.

7. The production now accepts input from outside Ensemble.

### A.1.2 Production Shutdown

When a production stops, the sequence of actions is as follows:

1. The production takes each business service offline and executes its optional **OnProductionStop()** method. This action stops all requests from outside Ensemble.

2. All business hosts receive a signal to become Quiescent.

3. All queues go into a Quiescent state. This means that from this point forward, business hosts can only process queued messages with High priority (synchronous messages). Asynchronous messages remain on their respective queues.

4. The production finishes processing all synchronous messages to the best of its ability.

5. The production takes each business process offline and executes its optional **OnProductionStop()** method.

6. The production takes each business operation offline and executes its optional **OnProductionStop()** method.

7. The production goes offline. Ensemble executes the optional **OnStop()** method in the production class.

# A.2 Life Cycle of a Business Service and Adapter

## A.2.1 Production Startup

When you start a production (or change the configuration of a specific business service), Ensemble automatically performs the following tasks for each configured business service class (that is, for every business service listed in the production definition):

1. Ensemble invokes the business service's **OnProductionStart()** callback method, if defined.

   The **OnProductionStart()** method is a class method that is invoked once for each business service class listed in the production configuration. A business service class can use this callback to perform any class-wide initialization it may require.

2. Ensemble creates one or more background processes in which to execute the business service.

   The number of background processes is determined by the business service's PoolSize property within the production configuration. Each background process is referred to as an instance of the business service and contains an instance of a business service object.

   Ensemble only creates a background process for a business service if the following conditions are true:

   - The business service class must have an associated inbound adapter class as specified by its ADAPTER class parameter.

     A business service class with no associated inbound adapter is referred to as an "adapterless service." Instead of waiting for external events, such a service is invoked in-process (perhaps by a composite application).

   - The business service's Enabled property within the production configuration must be set to 1 (otherwise the business service is considered to be disabled and will not accept input).

   - The business service's PoolSize property within the production configuration must be set to a value greater than 0.

   If the business service's Foreground property within the production configuration is set to 1, then Ensemble will create a foreground process (that is, Ensemble will create a Terminal window) for the business service. This feature facilitates testing and debugging.

3. Ensemble initializes the system monitoring information used to monitor the status and operating history of the business service.

4. Within each background process, Ensemble does the following:

   a. Creates an instance of the business service class.

   b. Supplies the most recently configured values of any of the business service's settings

c. Invokes the business service's **OnInit()** callback method (if present). The **OnInit()** method is an instance method that provides a convenient place to execute any initialization logic for a business service.

d. Creates an instance of the associated adapter class (if one is defined) and supplies the most recently configured values of any of the adapter's settings.

## A.2.2 Runtime

While the production is running, the business service repeatedly calls the inbound adapter's **OnTask()** method. This OnTask loop is controlled by the business service's CallInterval setting and %WaitForNextCallInterval property as follows:

1. The business service calls the inbound adapter's **OnTask()** method.

2. **OnTask()** checks outside Ensemble for input events of interest to the business service:

   - If it finds input, **OnTask()** calls the **ProcessInput()** method of the associated business service object.

   - If it does not find input, **OnTask()** returns control to the business service, which waits for the next CallInterval to elapse before returning to step 1.

   - Multiple input events may exist. For example, if the business service uses File.InboundAdapter, there may be several files waiting in the designated directory.

     If there are multiple input events:

     – Typically, the **OnTask()** method calls **ProcessInput()** as many times as is necessary to process all the available input events until no more are found.

     – Alternatively, an inbound adapter can restrict **OnTask()** to call **ProcessInput()** only once per CallInterval, even if multiple input events exist. Rather than processing all the input events, **OnTask()** goes to sleep after processing the first event found.

3. **ProcessInput()** sets the business service %WaitForNextCallInterval property to 0 (false) and calls **OnProcessInput()** to handle the input event.

4. Upon completion, **ProcessInput()** returns control to **OnTask()**.

5. At this point, **OnTask()** may set %WaitForNextCallInterval to 1 (true). This restricts the business service to processing only one input event per CallInterval, even when multiple input events exist.

   Usually you want the business service to process all available input events without delay, so usually you do not want to do anything to change %WaitForNextCallInterval at this step. It should retain the 0 (false) value set by **ProcessInput()**.

   The adapter base class Ens.InboundAdapter has an **OnTask()** method that calls **ProcessInput()**, sets %WaitForNextCallInterval to 1, and returns.

   **Tip:** If you simply want a business service to wake up and run its **ProcessInput()** method once per CallInterval without concern for events outside Ensemble, use the adapter class Ens.InboundAdapter.

6. **OnTask()** returns.

7. The business service tests the value of its %WaitForNextCallInterval property:

   - If 1 (true), the business service waits for the CallInterval to elapse before returning to step 1.

   - If 0 (false), the business service immediately returns to step 1. The CallInterval does not come into play until **OnTask()** discovers there is no more input (see step 2).

## A.2.3 Production Shutdown

When a production stops, the following events related to business services occur:

1.  Ensemble disables each business service; no more incoming requests are accepted for this production.

2.  The **OnTearDown()** method in each inbound adapter is called.

3.  All inbound adapter and business service objects are destroyed and their background processes are killed.

4.  Each business service's **OnProductionStop()** class method is called, once for each configured item of that class in the production.

When a business service is disabled by a system administrator, or becomes inactive according to its configured schedule, the production continues to run but the associated inbound adapter is shut down, and its **OnTearDown()** method is executed.

# A.3 Life Cycle of a Business Process

Each time a production starts, Ensemble creates the public actor pool for the production. The value of the ActorPoolSize setting determines the number of jobs in the pool.

Within each job in the actor pool, there is an instance of an Ens.Actor object whose responsibility it is to manage the use of its job by business processes. This Ens.Actor instance is called an *actor*.

The business processes in a production can share the public message queue called Ens.Actor. This public queue is the target of all messages sent to any business process within a production that does not have its own, private queue. Actors listen on the Ens.Actor queue whenever they are free to host a business process. When a request arrives on the Ens.Actor queue, any actor that is free may assign its job to host the business process named in the request. Requests on the Ens.Actor queue are processed in the order in which they are received. Each successive request is claimed by the next available actor, on an ongoing basis.

For information on pools, see "Pool Size and Actor Pool Size" in *Configuring Ensemble*.

## A.3.1 Life Cycle in the Public Queue

The life cycle of a business process that uses the public queue is as follows:

1.  The initial request is addressed to the business process. The request message arrives on the Ens.Actor queue.

2.  As soon as an actor becomes available, it pulls the request message off the Ens.Actor queue.

3.  The actor creates a new instance of the appropriate business process class, supplies the most recent values of the settings of the business process, and invokes the **OnRequest()** method of that instance. The business process instance is now ongoing.

4.  While the business process instance is ongoing, whenever it is not active (for example, while it is waiting for input or feedback), it generally returns control to the actor that instantiated it. When this happens:

    a.  The actor suspends the business process instance.

    b.  The actor saves the current state of the instance to disk.

    c.  The actor returns its job to the actor pool.

5.  Whenever an ongoing business process has no assigned actor and a subsequent request or response arrives addressed to the business process (for example, when the anticipated input or feedback arrives) the following sequence occurs:

a. When an actor becomes available, it pulls the new request or response message from the Ens.Actor queue.

b. The actor restores the corresponding business process object from disk, complete with all of its state information.

c. The actor invokes the **OnRequest()** or **OnResponse()** instance method, as appropriate.

6. After the business process instance completes execution, its current actor invokes its **OnComplete()** method and marks the instance with the status of *IsComplete*. The actor also returns its job to the actor pool. No further events are sent to this business process instance.

## A.3.2 Life Cycle in a Private Queue

Alternatively, you can configure business processes to have private queues, bypassing the public Ens.Actor queue. The life cycle of a business process with a private queue runs exactly as described for a public queue, except that:

• The business process runs in jobs from the private pool only.

• The messages addressed to this business process arrive on its own, private queue, and do not arrive on the Ens.Actor queue.

# A.4 Life Cycle of a Business Operation and Adapter

Ensemble automatically manages the life cycle of each business operation.

## A.4.1 Production Startup

When you start a production (or change the configuration of a specific business operation), Ensemble automatically performs the following tasks for each configured business operation class (that is, for every business operation listed in the production definition):

1. It invokes the class' **OnProductionStart()** callback method, if defined.

   The **OnProductionStart()** method is a class method that is invoked once for each business operation class listed in the production configuration. A business operation class can use this callback to perform any class-wide initialization it may require.

2. It creates one or more background processes in which to execute the business operation.

   The number of background processes is determined by the business operation's PoolSize property within the production configuration. Each background process is referred to as an instance of the business operation and contains an instance of a business operation object.

   Ensemble will only create a background process for a business operation if the following conditions are true:

   • The business operation class must set its *INVOCATION* class parameter to "Queue".

   • The business operation's Enabled property within the production configuration must be set to 1 (otherwise the business operation is considered to be disabled). A disabled business operation still has an incoming message queue. Any requests posted to this queue will not be processed until the business operation is enabled.

   • The business operation's PoolSize property within the production configuration must be set to a value greater than 0.

If the business operation's Foreground property within the production configuration is set to 1, then Ensemble will create a foreground process (that is, it will create a Terminal window) for the business operation. This feature facilitates testing and debugging.

3.  It initializes the system monitoring information used to monitor the status and operating history of the business operation.

4.  Within each background process:

    a.  Ensemble creates an instance of the business operation class, supplies the most recently configured values of any of the business operation's settings, and invokes the business operation **OnInit()** callback method (if present). The **OnInit()** method is an instance method that provides a convenient place to execute any initialization logic for a business operation.

    b.  Ensemble creates an instance of the associated adapter class (if one is defined) and supplies the most recently configured values of any of the adapter's settings.

## A.4.2 Runtime

At runtime, a business operation does the following:

1.  It waits for requests to be sent (from business services, business processes, and other business operations) to its associated message queue.

2.  After the business operation retrieves a request from its message queue, it searches its message map for the operation method that corresponds to the request type. It then invokes that operation method.

3.  The operation method, using the data within the request object, makes a request to an external application. Typically it does this by calling one or methods of its associated outbound adapter object.

## A.4.3 Production Shutdown

When a production stops, the following events related to business operations occur:

1.  Ensemble waits for each business operation to reach a quiescent state (that is, Ensemble waits until each business operation has completed all of its synchronous requests).

2.  The **OnTearDown()** method in each outbound adapter is called.

3.  All outbound adapter and business operation objects are destroyed and their background processes are killed.

4.  Each business operation's **OnProductionStop()** class method is called, once for each configured item of that class in the production.

When a business operation is disabled by a system administrator, or becomes inactive according to its configured schedule, the production continues to run but the associated outbound adapter is shut down, and its **OnTearDown()** method is executed.