Using HTTP Adapters in Productions

Version 2019.1
2019-05-20

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

Using HTTP Adapters in Productions
InterSystems IRIS Data Platform Version 2019.1 2019-05-20

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 describes how to add HTTP adapters to a production, so that the production can send and receive HTTP requests and responses.

This book contains the following sections:

- About the HTTP Adapters
- Using the HTTP Inbound Adapter
- Using the HTTP Outbound Adapter
- Reference for Settings

For a detailed outline, see the table of contents.

For more information, try the following sources:

- For information about the pass-through service and operation, EnsLib.HTTP.GenericService and EnsLib.HTTP.GenericOperation, see "Configuring ESB Services and Operations" in *Using a Production as an ESB*. These classes enable you to receive an HTTP, REST, or SOAP request and pass it through to an external service.
- *Best Practices for Creating Productions* describes best practices for organizing and developing productions.
- *Developing Productions* explains how to perform the development tasks related to creating a production.
- *Configuring Productions* describes how to configure the settings for productions, business hosts, and adapters. It provides details on settings not discussed in this book.

1

About the HTTP Adapters

The HTTP adapters (EnsLib.HTTP.InboundAdapter and EnsLib.HTTP.OutboundAdapter) enable your production to send and receive HTTP requests and responses. This chapter provides a brief introduction to these adapters.

Tip: InterSystems IRIS™ also provides specialized business service classes that use these adapters, and one of those might be suitable for your needs. If so, no programming would be needed. See "Connectivity Options" in *Introducing Interoperability Productions*.

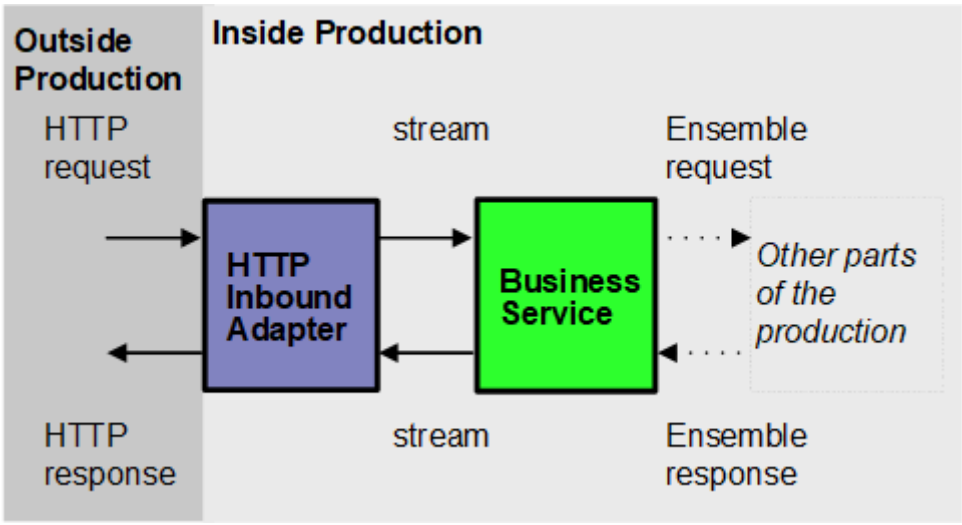
1.1 HTTP Inbound Adapter and Helper Classes

The EnsLib.HTTP.InboundAdapter is the HTTP listener for custom port listening, XML listening, and/or raw HTML handling. You use this adapter in cases when you prefer to listen on a private port rather than using a CSP page (which uses the standard web server to handle HTTP requests).

The class provides runtime settings that you use to specify items like the following:

- A local port, where the adapter will listen for input
- A list of IP addresses from which the adapter will accept input (if you want to restrict the possible sources)
- Settings that specify whether to use the character set given in the inbound request, and if not, what other character set to use

The inbound HTTP adapter listens on the specified port, reads the input, and sends the input as a stream (either binary or character, depending on the character set in use) to the associated business service. The business service, which you create and configure, uses this stream and communicates with the rest of the production.



When you work with the HTTP inbound adapter, there are two helper classes that you might use: %Library.GlobalCharacterStream and %Library.GlobalBinaryStream. The inbound adapter sends a stream to the associated business service. Specifically, this is an instance of %Library.GlobalCharacterStream or %Library.GlobalBinaryStream, depending on the character set being used. In general, these classes provide methods that you can use to read the contents of the stream, get the length of the stream, read a single line, rewind, append data, and so on. Both of these basic classes are documented in the documentation; for example, see the chapter "Working with Streams" in *Defining and Using Classes*.

1.2 HTTP Outbound Adapter and Helper Classes

EnsLib.HTTP.OutboundAdapter is the adapter for sending HTTP requests outside a production and receiving HTTP responses. This adapter provides settings to control items such as the following:

- Server and port to which the adapter will send HTTP requests
- URL path for the resource to request, at the given server and port
- An optional SSL configuration to use for the connection to the server
- Optional information to specify a proxy server through which the adapter can route requests

The adapter provides methods for sending HTTP POST, GET, and PUT actions:

- The main methods are PostFormData() and GetFormData(). Each accepts an output argument for the response object, a comma-separated list of form variable names, and a variable number of form variable arguments, one for each of the names in the comma-separated list. If you want to set multiple values for a form variable, you can use the same name multiple times in the list. Of course, you can also use these methods with no form variables to request flat scalar content such as a regular web page.
- For situations with a complicated set of form variables, use the methods PostFormDataArray() and GetFormDataArray(). These methods accept a multidimensional array instead of a variable argument list. This can help keep things organized, because you can provide multiple values for a given form variable as subnodes in the array, rather than as multiple entries in the list of names. You can also index the array by form variable name rather than by position.
- The low-level worker method SendFormDataArray() is available for situations when you need to use a PUT or some other unusual HTTP request, or where you need to customize aspects of your HTTP request other than form variables or cookies.

The adapter also provides properties and methods to manage cookies.

When you work with the HTTP outbound adapter, there are two helper classes that you can use:

- The HTTP outbound adapter uses %Net.HttpRequest to encapsulate the HTTP request that it sends.

  If you use PostFormData(), GetFormData(), PostFormDataArray(), or GetFormDataArray(), the adapter creates the request automatically and you do not have direct access to it.

  However, if you use SendFormDataArray(), you can create an instance of the %Net.HttpRequest class, set its properties, and use it to initialize the HTTP request that you send in that method. You use this technique when you need to set properties of the HTTP request (such as the proxy authentication) that cannot be set via the adapter.

- In all cases, the HTTP response is encapsulated in an instance of %Net.HttpResponse. The %Net.HttpResponse class provides methods to access the HTTP headers, as well as properties that contain the body of the response (a stream object), reason codes, the HTTP version of the server, and so on.

2

Using the HTTP Inbound Adapter

This chapter describes the default behavior of the HTTP inbound adapter (EnsLib.HTTP.InboundAdapter) and describes how to use this adapter in your productions. It discusses the following topics:

- Overall behavior of this adapter
- How to create a business service to use this adapter
- Details on how to implement OnProcessInput()
- How to implement the optional OnErrorStream() method
- How to use the HTTP request
- How to add and configure the business service

Tip: InterSystems also provides specialized business service classes that use this adapter, and one of those might be suitable for your needs. If so, no programming would be needed. See "Connectivity Options" in *Introducing InterSystems IRIS™ Interoperability*.

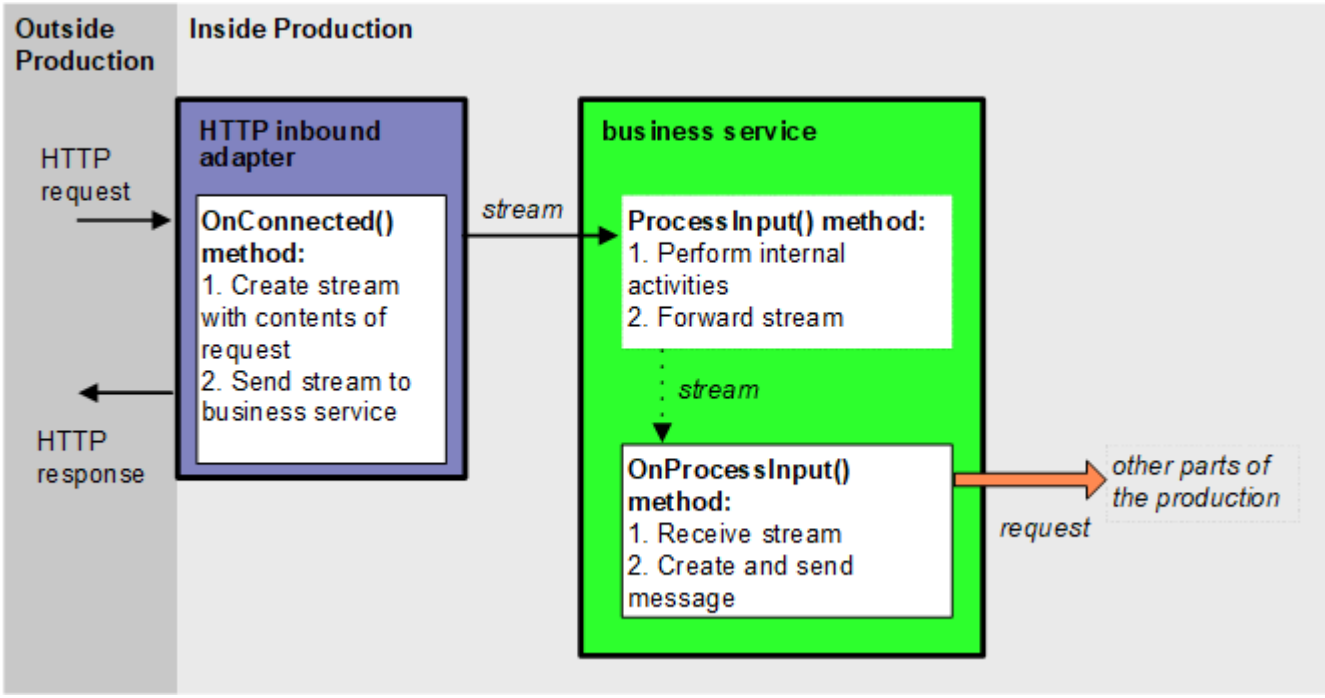
2.1 Overall Behavior

EnsLib.HTTP.InboundAdapter is an HTTP listener for custom port listening, XML listening, and/or raw HTML handling. You use this adapter in cases when you prefer to listen on a private port rather than using a CSP page (which uses the standard web server to handle HTTP requests).

First, the class provides runtime settings that you use to specify items like the following:

- A local port, where the adapter will listen for input
- A list of IP addresses from which the adapter will accept input (if you want to restrict the possible sources)
- Settings that specify whether to use the character set given in the inbound request, and if not, what other character set to use

The inbound HTTP adapter listens to a port on the local machine, reads the input, and sends the input as a stream to the associated business service. The business service, which you create and configure, uses this stream and communicates with the rest of the production. The following figure shows the overall flow:



In more detail:

1. The adapter receives an HTTP message and opens a TCP connection. (HTTP is a format of header and body data that is sent over a TCP connection.)
2. When the adapter connects, it executes its **OnConnected()** method, which first determines the character set to use. By default, it uses the character set specified in the inbound HTTP request. For details, however, see "Specifying the Character Set to Use."
3. The adapter chooses the appropriate translation table for the character set.
4. The adapter reads the body of the input, translates it, and places it into a new stream object.
   - If the adapter is using a non-binary character set, the stream is of type %GlobalCharacterStream.
   - If the adapter is using a binary character set, the stream is of type %GlobalBinaryStream.

   The adapter also extracts each HTTP header and adds that header to the Attributes property of the stream; this property is a multidimensional array, as discussed later.

   Also, if the URL includes form parameters, these are passed as follows:

   - If the HTTP request is a GET request, the adapter puts them into the Attributes array under the `"Params"` subscript.
   - If the HTTP request is a POST request, then the form variables are written into the request body.
5. The adapter then calls the internal ProcessInput() method of the business service class, passing the stream as an input argument.
6. The internal ProcessInput() method of the business service class executes. This method performs basic production tasks such as maintaining internal information as needed by all business services. You do not customize or override this method, which your business service class inherits.
7. The ProcessInput() method then calls your custom OnProcessInput() method, passing the stream object as input. The requirements for this method are described later in "Implementing the OnProcessInput() Method."
8. If ProcessInput() or OnProcessInput() returns an error, the production invokes the OnErrorStream() method of the business service.

The response message follows the same path, in reverse.

2.2 Creating a Business Service to Use the HTTP Inbound Adapter

To use this adapter in your production, create a new business service class as described here. Later, add it to your production and configure it. You must also create appropriate message classes, if none yet exist. See "Defining Messages" in *Developing Productions*.

The following list describes the basic requirements of the business service class:

- Your business service class should extend Ens.BusinessService.
- In your class, the *ADAPTER* parameter should equal EnsLib.HTTP.InboundAdapter.
- If you need to parse form variables from an HTTP POST request, implement the **OnInit()** callback method as follows:

  ```
  Method OnInit() As %Status
  {
      Set ..Adapter.ParseBodyFormVars=1
      Quit 1
  }
  ```

  This step is not necessary to parse the form variables from a GET request; these are automatically parsed into the Attributes property as discussed later in this chapter.
- Your class should implement the OnProcessInput() method, as described in "Implementing the OnProcessInput() Method."
- Your class can implement the OnErrorStream() method. See "Implementing the OnErrorStream() Method."
- For other options and general information, see "Defining a Business Service Class" in *Developing Productions*.

The following example shows the overall structure of your business service class. The details of the OnProcessInput() method depend on the character set that the adapter is using. If the adapter is using a non-binary character set, the general structure should be as follows:

```
Class EHTTP.NewService1 Extends Ens.BusinessService
{
Parameter ADAPTER = "EnsLib.HTTP.InboundAdapter";

Method OnProcessInput(pInput As %GlobalCharacterStream, pOutput As %RegisteredObject) As %Status
{
   set tsc=$$$OK
   //your code here
   Quit tsc
}
}
```

Or, if the adapter is using a binary character set, the **OnProcessInput()** method should be as follows instead:

```
Method OnProcessInput(pInput As %GlobalBinaryStream, pOutput As %RegisteredObject) As %Status
{
   set tsc=$$$OK
   //your code here
   Quit tsc
}
```

Note: Studio provides a wizard that you can use to create a business service stub similar to the preceding. To access this wizard, click **File —> New** and then click the **Production** tab. Then click **Business Service** and click **OK**. Note that the wizard provides a generic input argument. If you use the wizard, InterSystems recommends that you edit the method signature to use the specific input argument that you need; the input argument type should be %GlobalCharacterStream or %GlobalBinaryStream.

2.3 Implementing the OnProcessInput() Method

Within your custom business service class, the signature of your **OnProcessInput()** method depends on the character set that the adapter is using:

- If the adapter is using a non-binary character set, the signature should be as follows:

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

- If the adapter is using a binary character set, the signature should be as follows:

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

Here *pInput* is the input that the adapter will send to this business service. Also, *pOutput* is the generic output argument required in the method signature.

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

1. Examine the input object and extract the needed data from it. See "Using the HTTP Request," later in this chapter.
2. Create an instance of the request message, which will be the message that your business service sends.

   For information on creating message classes, see "Defining Messages" in *Developing Productions*.
3. For the request message, set its properties as appropriate, using values obtained from the input stream.

   If the attribute `Content-Type` is specified in the Attributes property, by default, that is used as the `Content-Type` of the request message. If `Content-Type` is not specified in the Attributes property, by default, the `Content-Type` of the request message is set to `"text/html"`. If these defaults are not appropriate, be sure to set this attribute. For example, the following code checks the value of the attribute `Content-Type` of the input stream and uses `"text/xml"` if the value is missing:

   ```
   Set outputContentType=$GET(pInput.Attributes("Content-Type"),"text/xml")
   ```
4. Call a suitable method of the business service to send the request to some destination within the production. Specifically, call **SendRequestSync()**, **SendRequestAsync()**, or (less common) **SendDeferredResponse()**. For details, see "Sending Request Messages" in *Developing Productions*

   Each of these methods returns a status (specifically, an instance of %Status).
5. Optionally check the status of the previous action and act upon it.
6. Optionally examine the response message that your business service has received and act upon it.
7. Make sure that you set the output argument (*pOutput*). This step is required.
8. Return an appropriate status. This step is required.

The following shows a simple example:

```
Method OnProcessInput(pInput As %GlobalCharacterStream, Output pOutput As %RegisteredObject) As %Status
{
    //get contents of inbound stream
```

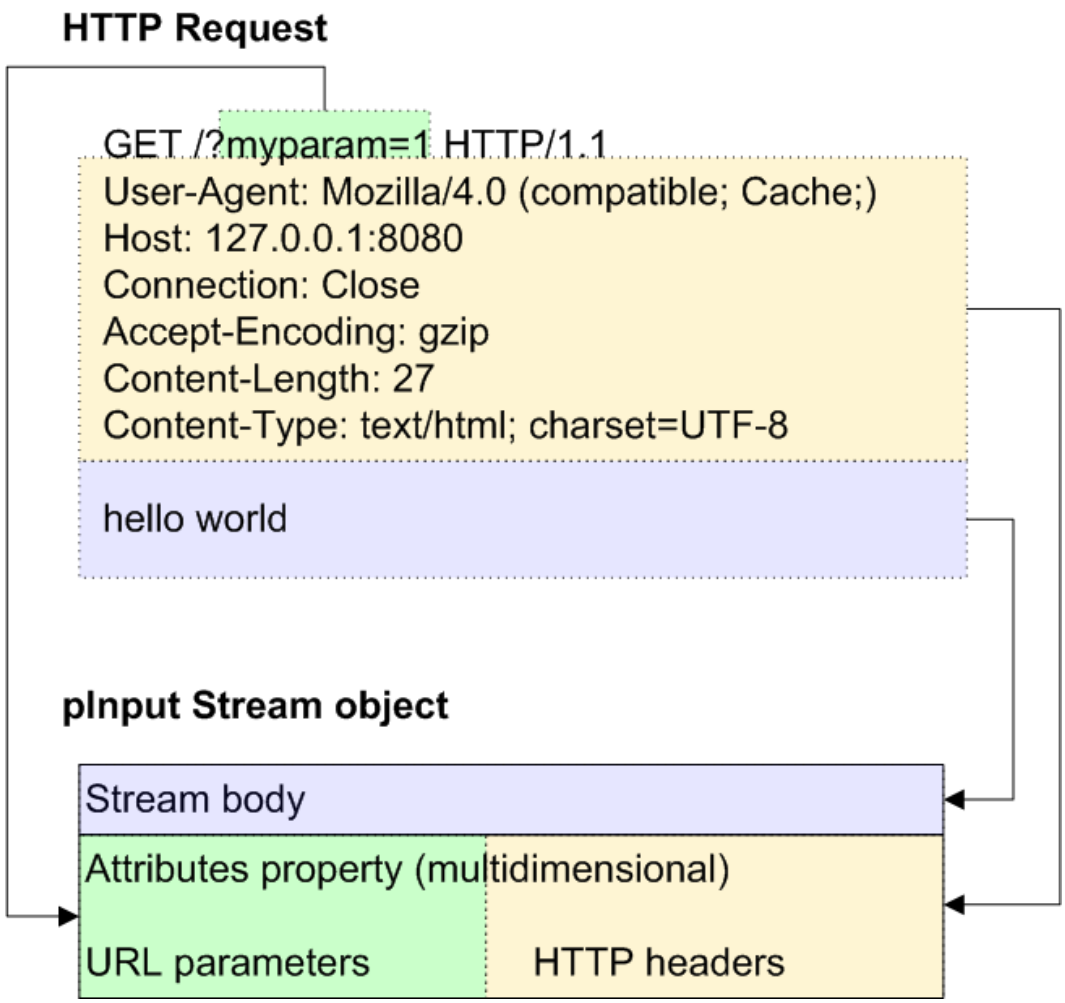
```
    //in this case, the stream contains a single value: a patient ID
    Set id=pInput.Read(,.tSC)

    //make sure Read went OK
    If $$$ISERR(tSC) do $System.Status.DisplayError(tSC)

    //create request object to send
   Set tRequest=##class(EHTTP.Request.Patient).%New()
    Set tRequest.patientID=id

   //send to lookup process
   Set tSC=..SendRequestSync("EHTTP.LookupProcess",tRequest,.tResponse)

    //define output for OnProcessInput
    Set pOutput=tResponse

    Quit tSC
}
```

2.4 Implementing the OnErrorStream() Method

If the **ProcessInput()** or **OnProcessInput()** method of your business service returns an error, the production invokes the **OnErrorStream()** method of the business service. You can implement this method to contain any desired error handling. The method should accept a status code as input and should return the desired output stream.

2.5 Using the HTTP Request

Within **OnProcessInput()**, the HTTP request is available as *pInput*, which is an instance of %GlobalCharacterStream or %GlobalBinaryStream, depending on your implementation. The following figure illustrates how this instance represents the HTTP request:



The body of the request is written to the *pInput* stream. For information on working with streams, see the documentation, for example, the chapter "Defining and Using Stream Properties" in *Defining and Using Classes.*

Additional data is available in the Attributes property of the *pInput* stream, as discussed in the following subsection. This includes the HTTP headers. If the request was a GET request, it also includes any form variables (URL parameters).

If the request was a POST request, then the form variables are available in the body.

2.5.1 About the Attributes Array

The Attributes property of the *pInput* stream is a multidimensional array that carries the following data:

Node	Contents
Attributes(*http_header_name*) Where *http_header_name* is the header name in lowercase, for example, `"content-length"`	Value of the given HTTP header
Attributes(`"Params"`,*form_variable_name*,*n*)	Value of the *n*th instance of the given URL form variable (if the HTTP request was a GET request).
Attributes(`"URL"`)	Complete URL of the HTTP request

So you can retrieve a header value as follows:

```
set contentlength = pInput.Attributes("content-length")
```

Or, to retrieve a URL form variable (for a GET request):

```
set pResponse.MessageRequestTimeStamp = pInput.Attributes("Params","REQUESTTIMESTAMP",1)
```

2.6 Adding and Configuring the Business Service

To add your business service to a production, use the Management Portal to do the following:

1. Add an instance of your custom business service class to the production.
2. Configure the adapter so that it can receive input. Specifically:
   - Specify the port on which the adapter will listen. To do so, specify the Port setting.
   - Optionally specify the IP addresses from which the adapter will accept input, if you want to limit the sources with which the adapter communicates.
   - Optionally specify the character set of the input data.
3. Enable the business service.
4. Run the production.

2.6.1 Specifying the Sources of HTTP Requests

You can configure the inbound HTTP adapter to recognize sources of HTTP requests in two different ways:

- You can permit HTTP requests from any server. This is the default.
- You can permit HTTP requests from a list of specific servers (optionally with specific ports).

  To do so, specify the Allowed IP Addresses setting. See “Reference for Settings.”

2.6.2 Specifying the Character Set to Use

When the EnsLib.HTTP.InboundAdapter receives input, it translates the characters in that input according to a translation table. To determine which translation table to use, the adapter first determines which character set the input uses.

In general, the HTTP `Content-Type` header of the input indicates which character set that request uses. By default, the adapter uses that character set.

However, you can control which character set the adapter uses, by using the following runtime settings:

- Force Charset
- Charset

See “Reference for Settings.”

For background information on character translation in InterSystems IRIS, see “Localization Support” in the *Orientation Guide for Server-Side Programming*.

3

Using the HTTP Outbound Adapter

This chapter describes the overall behavior of the HTTP outbound adapter (EnsLib.HTTP.OutboundAdapter) and describes how to use this adapter in your productions. It discusses the following topics:

- Overall behavior
- How to create a business operation to use this adapter
- Requirements of the message handler methods
- How to call HTTP commands from within those methods
- How to create custom HTTP requests
- How to use the HTTP response
- A couple of examples
- How to add and configure the business operation

Tip: InterSystems also provides specialized business service classes that use this adapter, and one of those might be suitable for your needs. If so, no programming would be needed. See "Connectivity Options" in *Introducing InterSystems IRIS™ Interoperability*.

3.1 Overall Behavior

Within a production, an outbound adapter is associated with a business operation that you create and configure. The business operation receives a message from within the production, looks up the message type, and executes the appropriate method. This method usually executes methods of the associated adapter.

EnsLib.HTTP.OutboundAdapter is the adapter for sending HTTP requests outside a production and receiving HTTP responses. This adapter provides settings to control items such as the following:

- Server and port to which the adapter will send HTTP requests
- URL path for the resource to request, at the given server and port
- An optional SSL configuration to use for the connection to the server
- Optional information to specify a proxy server through which the adapter can route requests

The adapter provides methods for sending HTTP POST, GET, and PUT actions:

- The main methods are PostFormData() and GetFormData(). Each accepts an output argument for the response object, a comma-separated list of form variable names, and a variable number of form variable arguments, one for each of the names in the comma-separated list. If you want to set multiple values for a form variable, you may use the same name multiple times in the list. Of course, you can also use these methods with no form variables to request flat scalar content such as a regular web page.
- For situations with a complicated set of form variables, use the methods PostFormDataArray() and GetFormDataArray(). These methods accept a multidimensional array instead of a variable argument list. This can help keep things organized, because multiple values for a given form variable can be given as subnodes in the array, rather than by using multiple entries in the list of names. You can also index the array by form variable name rather than by position.
- The low-level worker method SendFormDataArray() is available for situations when you need to use a PUT or some other unusual HTTP request, or where you need to customize aspects of your HTTP request other than form variables or cookies.

The HTTP requests are in the default character encoding of the local InterSystems IRIS server. To specify a different character encoding, create and send custom HTTP requests, as described in "Creating Custom HTTP Requests."

3.2 Creating a Business Operation to Use the Adapter

To create a business operation to use the EnsLib.HTTP.OutboundAdapter, you create a new business operation class. Later, add it to your production and configure it.

You must also create appropriate message classes, if none yet exist. See "Defining Messages" in *Developing Productions*.

The following list describes the basic requirements of the business operation class:

- Your business operation class should extend Ens.BusinessOperation.
- In your class, the *ADAPTER* parameter should equal EnsLib.HTTP.OutboundAdapter.
- In your class, the *INVOCATION* parameter should 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 is 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 released to 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>
}
```

- Your class should 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 a request message class, and *ResponseClass* is the name of a response message class. In general, the method code will refer to properties and methods of the Adapter property of your business operation.

For information on defining message classes, see "Defining Messages" in *Developing Productions*.

For information on defining the message handler methods, see "Creating Message Handler Methods," later in this chapter.

- For other options and general information, see "Defining a Business Operation Class" in *Developing Productions*.

The following example shows the general structure that you need:

```
Class EHTTP.NewOperation1 Extends Ens.BusinessOperation
{
Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";

Parameter INVOCATION = "Queue";

Method Sample(pReq As RequestClass, Output pResp As ResponseClass) As %Status
{
  Quit $$$ERROR($$$NotImplemented)
}

XData MessageMap
{
<MapItems>
  <MapItem MessageType="RequestClass">
    <Method>Sample</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**.

3.3 Creating Message Handler Methods

When you create a business operation class for use with EnsLib.HTTP.OutboundAdapter, typically your biggest task is writing message handlers for use with this adapter, that is, methods that receive production messages and then perform various HTTP operations.

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 a request message class, and *ResponseClass* is the name of a response message class.

In general, the method should do the following:

1. Examine the inbound request message.
2. Using the information from the inbound request, call a method of the Adapter property of your business operation. For example, call the **Get** method, which sends the HTTP GET command:

   ```
   set status=..Adapter.Get(.pHttpResponse,"Name",pRequest.Name)
   ```

   This example sends a GET request, passing in a parameter called `Name`, which has the value of `pRequest.Name`.

   The available methods are discussed in the next section. Each of them returns a status (specifically an instance of %Status).

These methods also return an HTTP response as output. In the preceding example, the response is placed into `pHttpResponse`. The response is an instance of %Net.HttpResponse; for information on using this object, see the chapter "Using the HTTP Response."

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

   For basic information on defining message classes, see "Defining Messages" in *Developing Productions*.
5. Make sure that you set the output argument (`pOutput`). Typically you set this equal to the response message. This step is required.
6. Return an appropriate status. This step is required.

The following shows an example:

```
Method PostMessage(pRequest As EHTTP.Request.OutboundPost,
Output pResponse As EHTTP.Response.OutboundPost) As %Status
{
  Set $ZT="Trap",tSC=$$$OK
  do {

    set input=pRequest.MessageStream
    set tResponse = ##class(%Net.HttpResponse).%New()
    set tsc = ..Adapter.Post(.tResponse,,input)

    set pResponse = ##class(EHTTP.Response.OutboundPost).%New()
    set len = tResponse.Data.SizeGet()
     While (tResponse.Data.AtEnd = 0) {
      do pResponse.MessageStream.Write(tResponse.Data.Read())
     }
    Quit
  } while (0)
Exit
  Quit tSC
Trap
  Set $ZT="",tSC=$$$EnsSystemError
  Goto Exit
}
```

3.4 Calling HTTP Commands

This section describes how to use the EnsLib.HTTP.OutboundAdapter to send HTTP commands (GET, POST, PUT, or DELETE). Within an HTTP command, you can send either form data or a request body. This section discusses the following topics:

- How to send form data
- How to send a request body
- Reference information for the adapter methods
- Basic information about the HTTP response

3.4.1 Sending Form Data

To send HTTP form data, use any of the following methods: **Post**, **PostFormDataArray()**, **Get**, **GetFormDataArray()**, or **SendFormDataArray()**. Each of these methods returns (as output) an HTTP response, an instance of %Net.HttpResponse. For information on using this object, see the chapter "Using the HTTP Response."

The details differ from method to method, but a partial example follows:

```
set tFormVar="USER,ROLE,PERIOD"

set tUserID=pRequest.UserID
set tRole=pRequest.UserRole
set tED=pRequest.EffectiveDate

set tSC=..Adapter.Get(.tResponse,tFormVar,tUserID,tRole,tED)
```

Note that this example assumes that the request message has the properties `UserID`, `UserRole`, and `EffectiveDate`.

3.4.2 Sending a Request Body

A request body can be either a stream or a string, depending on your needs. For information on creating as stream and writing to it, see the documentation; for example, the chapter "Defining and Using Stream Properties" in *Defining and Using Classes*.

Then use any of the following methods: **Post()**, **PostFormDataArray()**, **Get()**, or **GetFormDataArray()**. In this case, you will pass the request body as an argument, and you will leave the form data argument empty. Each of these methods returns (as output) an HTTP response, an instance of %Net.HttpResponse. For information on using this object, see the chapter "Using the HTTP Response."

The details differ from method to method, but a partial example follows:

```
set tsc = ..Adapter.Post(.tResponse,,pRequest.MessageStream)
```

Note that this example assumes that the request message has a property `MessageStream`.

3.4.2.1 Stream Attributes as HTTP Headers

If you create a stream and send that as the request body, you can force the HTTP outbound adapter to include some or all of the stream attributes as HTTP headers. To do so, set the SkipBodyAttrs property equal to a list of attributes that should *not* be used as HTTP headers. The default value is `"*"` which means that the stream attributes are ignored by default (and not used as headers). Note that this property is not available as a runtime setting.

For information on stream attributes, see the class reference for %Library.AbstractStream.

3.4.3 Reference Information for HTTP Methods

This section provides reference information on the methods you can use to invoke HTTP commands.

Post()

```
Method Post(Output pHttpResponse As %Net.HttpResponse,
            pFormVarNames As %String = "",
            pData...) As %Status
```

Sends the HTTP POST command to the configured destination (see "Specifying the Destination"), sending either form data or a request body.

The HTTP response is returned as output in the first argument. This is an instance of %Net.HttpResponse; for information on using this object, see the chapter "Using the HTTP Response."

With this method, do one of the following:

- To send form data to the named form variables, specify the *pFormVarNames* argument and *pData* arguments as needed. *pFormVarNames* is a comma-separated list of form variable names to use. For each name in the list, a *pData* argument should be supplied.

  To pass multiple values for a given form variable, include the variable name multiple times in the *pFormVarNames* list.

Any extra *pData* arguments you supply will be assigned to the last form variable in the list.

- To send a request body instead of form variables, leave *pFormVarNames* empty and pass the body text as the *pData* argument.

A data value passed for body text may be of either string or stream type. Data values passed for form variables must be of string type.

PostFormDataArray()

```
Method PostFormDataArray(Output pHttpResponse As %Net.HttpResponse,
                         pFormVarNames As %String = "",
                         ByRef pData) As %Status
```

Sends the HTTP POST command to the configured destination (see "Specifying the Destination"), sending form data to the named form variables.

pFormVarNames is a comma-separated list of form variable names to use. For each name in the list, a *pData* argument should be supplied.

The *pData* argument is an array. The top node of the array is not used. Each subnode is subscripted by the index of the corresponding form variable in the *pFormVarNames* list. The value at a given subscript should be specified as follows:

- For a form variable (*varname*) with a single value, the value at *pData("varname")* should be the form data value to send. There should be no subnodes.
- For a form variable (*varname*) with multiple values, the value *pData("varname")* should be the count of the values. Each of the values for this form variable should appear in a subnode, subscripted by its position in the node.

PostURL()

```
method PostURL(pURL As %String,
               Output pHttpResponse As %Net.HttpResponse,
               pFormVarNames As %String = "",
               pData...) as %Status
```

Sends the HTTP POST command to the specified URL (*pURL*), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

Get()

```
Method Get(Output pHttpResponse As %Net.HttpResponse,
           pFormVarNames As %String = "",
           pData...) As %Status
```

Sends the HTTP GET command to the configured destination (see "Specifying the Destination"), sending either form data or a request body. See the remarks for **Post()**.

GetFormDataArray()

```
Method GetFormDataArray(Output pHttpResponse As %Net.HttpResponse,
                        pFormVarNames As %String = "",
                        ByRef pData) As %Status
```

Sends the HTTP GET command to the configured destination (see "Specifying the Destination"), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

GetURL()

```
method GetURL(pURL As %String,
              Output pHttpResponse As %Net.HttpResponse,
              pFormVarNames As %String = "",
              pData...) as %Status
```

Sends the HTTP GET command to the specified URL (*pURL*), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

Put()

```
Method Get(Output pHttpResponse As %Net.HttpResponse,
           pFormVarNames As %String = "",
           pData...) As %Status
```

Sends the HTTP PUT command to the configured destination (see "Specifying the Destination"), sending either form data or a request body. See the remarks for **Post()**.

PutFormDataArray()

```
Method PutFormDataArray(Output pHttpResponse As %Net.HttpResponse,
                        pFormVarNames As %String = "",
                        ByRef pData) As %Status
```

Sends the HTTP PUT command to the configured destination (see "Specifying the Destination"), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

PutURL()

```
method PutURL(pURL As %String,
              Output pHttpResponse As %Net.HttpResponse,
              pFormVarNames As %String = "",
              pData...) as %Status
```

Sends the HTTP PUT command to the specified URL (*pURL*), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

Delete()

```
method Delete(Output pHttpResponse As %Net.HttpResponse,
              pFormVarNames As %String = "",
              pData...) as %Status
```

Sends the HTTP DELETE command to the configured destination (see "Specifying the Destination"), sending either form data or a request body. See the remarks for **Post()**.

DeleteFormDataArray()

```
method DeleteFormDataArray(Output pHttpResponse As %Net.HttpResponse,
                           pFormVarNames As %String = "",
                           ByRef pData) as %Status
```

Sends the HTTP DELETE command to the configured destination (see "Specifying the Destination"), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

DeleteURL()

```
method DeleteURL(pURL As %String,
              Output pHttpResponse As %Net.HttpResponse,
              pFormVarNames As %String = "",
              pData...) as %Status
```

Sends the HTTP DELETE command to the specified URL (*pURL*), sending form data to the named form variables. See the remarks for **PostFormDataArray()**.

SendFormDataArray()

```
Method SendFormDataArray(Output pHttpResponse As %Net.HttpResponse,
                         pOp As %String,
                         pHttpRequestIn As %Net.HttpRequest,
                         pFormVarNames As %String = "",
                         ByRef pData) As %Status
```

Sends the HTTP request to the configured destination (see "Specifying the Destination"), sending form data to the named form variables.

The *pOp* argument specifies the HTTP action to take. This should be one of the following: `"POST"` `"GET"` `"PUT"`

pFormVarNames is a comma-separated list of form variable names to use. For each name in the list, a *pData* argument should be supplied.

The *pData* argument is an array; see the remarks for the **GetFormDataArray()** method.

For special needs, create an instance of %Net.HTTPRequest or a subclass, specify its properties, and use this as the *pHttpRequestIn* argument. If you do this, the HTTP request is initialized with the properties of your instance. This method gives you more control than the other methods of the outbound adapter. For example, it enables you to add your own HTTP headers to the request.

3.4.4 Handling the HTTP Response

When you use any of the methods described in the previous subsection, you receive, as output, an HTTP response. This object is an instance of %Net.HttpResponse. The chapter "Using the HTTP Response" describes how to use this object.

3.5 Managing Cookies

A cookie is a text string in a response header that a server can ask a client to save and return as a header value in subsequent HTTP requests. Some servers use cookies to maintain open sessions,

The EnsLib.HTTP.OutboundAdapter provides the following properties and methods to manage cookies. Your custom methods can use these.

UseCookies property

Specifies whether to save cookies received in HTTP responses while this adapter is instantiated, and insert them in each subsequent HTTP request. If UseCookies is true, then any job associated with an instance of that adapter will maintain a collection of cookies and send the appropriate selection of them with each new request that it sends. If UseCookies is false (0), cookies will not be sent. This convention allows each of these jobs to maintain its own persistent session with any web server that requires it.

The default is false. This property is also available as a runtime setting.

%Cookies property

Contains an array of cookies. Indexed by Domain/Server; each element is $LB(name, domain, path, value, expires, secure). This property is InterSystems IRIS multidimensional array.

DeleteCookie() method

```
Method DeleteCookie(pName As %String,
                    pPath As %String,
                    pDomain As %String) As %Status
```

Deletes a particular cookie.

DeleteCookies() method

```
Method DeleteCookies(pDomain As %String = "",
                     pPath As %String = "") As %Status
```

Deletes all cookies from the specified domain and/or path.

Note: Remember that cookies are specific to an HTTP server. When you insert a cookie, you are using a connection to a specific server, and the cookie is not available on other servers.

3.6 Creating Custom HTTP Requests

If you use the more common methods of the HTTP outbound adapter (such as **Get**), the adapter automatically creates and sends an HTTP request, which can include either form data or a request body. In special cases, you may want to create a custom HTTP request so that you can specify details such as proxy authorization or a different character encoding.

To send the custom request, use the **SendFormDataArray()** method of the HTTP outbound adapter as described in "Calling HTTP Commands" in the previous chapter. Then see the chapter "Sending HTTP Requests and Reading HTTP Responses" of *Using Internet Utilities*.

3.7 Using the HTTP Response

After you use the outbound adapter to send an HTTP request, you receive a response object (%Net.HttpResponse). Specifically, the main methods of the HTTP.OutboundAdapter enable you to send an HTTP request. All these methods give you access to the HTTP response that is sent in return:

- If you use the **PostFormData()**, **PostFormDataArray()**, **GetFormData()** or **GetFormDataDataArray** method, the HTTP response is returned as output in the first argument of the method call. This argument is an instance of %Net.HttpResponse.
- If you use the **SendFormDataArray()** method, the HttpResponse property of the request is updated. This property is an instance of %Net.HttpResponse.

For details on these methods, see "Calling HTTP Commands" in the chapter "Using the HTTP Outbound Adapter."

For more information on using the HTTP response, see "Sending HTTP Requests and Reading HTTP Responses" in *Using Internet Utilities*.

3.8 Examples

This section provides a couple of examples.

3.8.1 Example with Post

The following example uses the HTTP outbound adapter and posts an XML message to the configured destination. First, the business operation class is as follows:

```
Class EHTTP.PostService Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.HTTP.OutboundAdapter";

Parameter INVOCATION = "Queue";

Method PostMessage(pRequest As EHTTP.Request.OutboundPost,
Output pResponse As EHTTP.Response.OutboundPost) As %Status
{
  Set $ZT="Trap",tSC=$$$OK
  do {

    set tResponse = ##class(%Net.HttpResponse).%New()
    set tsc = ..Adapter.Post(.tResponse,,pRequest.MessageStream)
    set stream = ""

    set pResponse = ##class(EHTTP.Response.OutboundPost).%New()
    set len = tResponse.Data.SizeGet()
     While (tResponse.Data.AtEnd = 0) {
      do pResponse.MessageStream.Write(tResponse.Data.Read())
     }
    Quit
  } while (0)
Exit
  Quit tSC
Trap
  Set $ZT="",tSC=$$$EnsSystemError
  Goto Exit
}

XData MessageMap
{
<MapItems>
  <MapItem MessageType="EHTTP.Request.OutboundPost">
    <Method>PostMessage</Method>
  </MapItem>
</MapItems>
}

}
```

The request message class is as follows:

```
Class EHTTP.Request.OutboundPost Extends Ens.Request
{

/// MessageStream contains the complete SOAP Message to post
Property MessageStream As %GlobalCharacterStream(CONTENT = "MIXED");

}
```

The response message class is as follows:

```
Class EHTTP.Response.OutboundPost Extends Ens.Response
{

/// MessageStream contains the Response to the SOAP Message post
Property MessageStream As %GlobalCharacterStream(CONTENT = "MIXED");

}
```

3.8.2 Example with Get

The following example uses the **Get()** method.

```
Method GetKPI(pRequest As EHTTP.Request.GetKPI,
ByRef pResponse As EHTTP.Response.GetKPI) As %Status
{
   set $ZT="ErrTrap",tSC=$$$OK
   do {
      set pResponse=##class(EHTTP.Response.GetKPI).%New()

      set tFormVar="KPINAME,PHYSICIAN,ROLE,PERIOD"

      set tRole=pRequest.UserRole
      set tUserID=pRequest.UserID
      set tED=pRequest.EffectiveDate
      set tYear=$p(tED,"-",1)
```

```
      set tMonth=$p(tED,"-",2)
      set tPeriod=tYear_tMonth

      if tRole="MFA" set tUserID="MEDICAL FACULTY ASSOCIATES"

      set tSC=..Adapter.Get
        (.tResponse,tFormVar,pRequest.KPI,tUserID,tRole,tPeriod)

      if '$IsObject(tResponse) Quit
      ;
      ;now parse the XML stream
      ;
      set reader=##class(%XML.Reader).%New()
      do tResponse.Data.Rewind()
      set ^drltrace("xml")=tResponse.Data.Read(32000)
      do tResponse.Data.Rewind()
      set tSC=reader.OpenStream(tResponse.Data)
      if $$$ISERR(tSC) {
         $$$LOGWARNING("Unable to open the XML Stream returned from IDX")
         $$$TRACE("XML request probably failed")
         do tResponse.Data.Rewind()
         set traceline=tResponse.Data.Read(1000)
         $$$TRACE(traceline)
         set tSC=$$$OK
         quit
      }
      ;
      ;Associate a class name with the XML element Name
      do reader.Correlate("KPIResult","EHTTP.IDX.KPIResult")

      if reader.Next(.tResults,.tSC)
      {

         set pResponse.KPIValues=tResults.KPIValues

      }
      if tSC=0 {
         $$$TRACE("Error reading values from IDX")
      }
   } while (0)
Exit
   Quit tSC
ErrTrap
   set $ZT="",tSC=$$$EnsSystemError
   goto Exit
}
```

This example retrieves an XML file from the target server, parses it, retrieving specific data.

3.9 Adding and Configuring the Business Operation

To add your business operation to a production, use the Management Portal to do the following:

1. Add an instance of your custom business operation class to the production.
2. Configure the adapter to communicate with a specific external data source. Specifically:
   - Specify the destination for HTTP requests
   - Optionally specify a proxy server
3. Enable the business operation.
4. Run the production.

3.9.1 Specifying the Destination Server and URL Path

Use the following runtime settings to specify the destination to which you will send HTTP requests:

- HTTP Server

- HTTP Port
- Credentials
- URL

3.9.1.1 Example

For example, suppose that the URL to which you are posting is `http://122.67.980.43/HTTPReceive.aspx`

In this case, you would use the following settings:

Setting	Value
HTTPServer	`122.67.980.43`
URL	`HTTPReceive.aspx`

3.9.2 Specifying a Proxy Server

Use the following runtime settings to route the HTTP request via a proxy server, if needed:

- Proxy Server
- Proxy Port
- Proxy HTTPS

Reference for Settings

This section provides the following reference information:

- Settings for the HTTP Inbound Adapter
- Settings for the HTTP Outbound Adapter

Also see "Settings in All Productions" in *Managing Productions*.

Settings for the HTTP Inbound Adapter

Provides reference information for settings of the HTTP inbound adapter, EnsLib.HTTP.InboundAdapter.

Summary

The inbound HTTP adapter has the following settings:

Group	Settings
Basic Settings	Call Interval, Port
Connection Settings	Job Per Connection, Allowed IP Addresses, OS Accept Connection Queue Size, Stay Connected, Read Timeout, SSL Configuration, Local Interface, Enable Standard Requests
Additional Settings	Charset, Force Charset, GenerateSuperSessionID

The remaining settings are common to all business services. For information, see "Settings for All Business Services" in *Configuring Productions*.

Allowed IP Addresses

Specifies a comma-separated list of remote IP addresses from which to accept connections. The adapter accepts IP addresses in dotted decimal form.

Note: IP address filtering is a means to control access on private networks, rather than for publicly accessible systems. InterSystems does not recommend relying on IP address filtering as a sole security mechanism, as it is possible for attackers to spoof IP addresses.

An optional :*port* designation is supported, so either of the following address formats is acceptable: 192.168.1.22 or 192.168.1.22:3298. If a port number is specified, connections from other ports will be refused.

If the string starts with an exclamation point (!) character, the inbound adapter initiates the connection rather than waiting for an incoming connection request. The inbound adapter initiates the connection to the specified address and then waits for a message. In this case, only one address may be given, and if a port is specified, it supersedes the value of the Port setting; otherwise, the Port setting is used.

Also see "Specifying the Sources of HTTP Requests," earlier in this book.

Call Interval

This adapter does not use polling. It listens for a connection, and once a connection is established, it listens for messages on that connection (and responds immediately, if possible). It does, however, periodically check to see whether there have been requests to shut down the adapter or make it quiescent. This setting specifies the interval, in seconds, for the EnsLib.HTTP.InboundAdapter to perform this check.

The default value is 5 seconds. The minimum is 0.1 seconds.

Charset

Specifies the character set of the incoming data. InterSystems IRIS™ automatically translates from this encoding. The setting value is not case-sensitive. Use `Binary` for binary files, or for any data in which newline and line feed characters are distinct or must remain unchanged, for example in HL7 Version 2 or EDI messages. Other settings may be useful when transferring text documents. Choices include:

- `Auto` — Use the encoding declared in the incoming HTTP header `Content-Type` field. This is the default.

- `AutoXML` — Use the encoding declared in the XML header of the incoming XML body content, if any.
- `Binary` — Read the raw bytes of the body without performing any character encoding transformation.
- `RawBytes` — Read the raw bytes of the body without performing any character encoding transformation.
- `Default` — Use the default character encoding of the local InterSystems IRIS server.
- `Latin1` — The ISO Latin1 8-bit encoding.
- `ISO-8859-1` — The ISO Latin1 8-bit encoding.
- `UTF-8` — The Unicode 8-bit encoding.
- `UCS2` — The Unicode 16-bit encoding.
- `UCS2-BE` — The Unicode 16-bit encoding in big-endian form.
- Any other alias from an international character encoding standard for which NLS (National Language Support) is installed in InterSystems IRIS.

For background information on character translation in InterSystems IRIS, see "Localization Support" in the *Orientation Guide for Server-Side Programming*.

Force Charset

If this setting is true, the adapter uses the Charset setting instead of any character set declared in the incoming HTTP header `Content-Type` field. The default is false.

Generate SuperSession ID

This property controls whether the message will have a SuperSessionID, which can be used to identify messages that cross from one namespace to another. If this property is set, the business service first checks the HTTP header of the inbound message for a SuperSession ID. If it has a SuperSessionID value, it uses it; otherwise, it generates a new SuperSession value. It sets the SuperSesssion value in the production message and can also return the value in any HTTP response it sends to the caller.

Job Per Connection

If this setting is true, the adapter spawns a new job to handle each incoming TCP connection and allows simultaneous handling of multiple connections. If it is false, the adapter does not spawn a new job for each connection. The default is true.

Local Interface

Specifies the network interface through which the connection should go. Select a value from the list or type a value. An empty value means use any interface.

Enable Standard Requests

Specifies that the service can receive data via the Web Gateway.

- In order for the CSP mechanism to work, HTTP Services must be configured either with their configuration name the same as the class name, or the invoking URL must include ?CfgItem= giving the config item name, or using a CSP application with a DispatchClass configured and the config item name as the next URL piece after the application name.
- To specify that the service only receives data from the CSP port and not from the special port, select **Enable Standard Requests** and set the pool size to 0.
- Note that if you use the CSP port, you cannot have a First-In-First-Out processing for messages.

- The default value of **Enable Standard Requests** for all HTTP services except HL7 is true. To maintain compatibility with previous versions, for HL7 HTTP services the default value of **Enable Standard Requests** is false.

OS Accept Connection Queue Size

Specifies the number of incoming connections should the operating system should hold open. Set to 0 if only one connection at a time is expected. Set to a large number if many clients will connecting rapidly.

Port

Identifies the TCP port on the local machine where the adapter is listening for HTTP requests. Avoid specifying a port number that is in the range used by the operating system for ephemeral outbound connections.

Read Timeout

Number of seconds to wait for each successive incoming read, following receipt of initial data from remote port.

Stay Connected

Specifies whether to keep the TCP connection open between requests.

- If this setting is zero, the adapter will disconnect immediately after each message is received.
- If this setting is positive, it specifies the idle time, in seconds. The adapter disconnects after this idle time.
- If this setting is -1, the adapter auto-connects on startup and then stays connected.

Settings for the HTTP Outbound Adapter

Provides reference information for settings of the HTTP outbound adapter, EnsLib.HTTP.OutboundAdapter.

Summary

The outbound HTTP adapter has the following settings:

Group	Settings
Basic Settings	HTTP Server, HTTP Port, URL, Credentials
Connection Settings	SSL Configuration, SSL Check Server Identity, Proxy Server, Proxy Port, Proxy HTTPS, Proxy Http Tunnel,Response Timeout, ConnectTimeout
Additional Settings	Use Cookies, SendSuperSession

The remaining settings are common to all business operations. For information, see "Settings for All Business Operations" in *Configuring Productions*.

Connect Timeout

Specifies the number of seconds to wait for the connection to the server to open. The default value is 5.

If the connection is not opened in this time period, the adapter retries repeatedly, up to the number of times given by Failure Timeout divided by Retry Interval.

Credentials

ID of the production credentials that can authorize a connection to the given destination URL. For information on creating production credentials, see *Configuring Productions*.

HTTP Port

TCP port on the server to send HTTP requests to (will use 80 by default, or 443 if SSLConfig is specified). If this is not standard (that is, not equal to 80), it is included in the `Host:` header of the HTTP request that you are sending. Also see "Specifying the Destination Server and URL Path," earlier in this book.

HTTP Server

IP address of the server to send HTTP requests to. This is used in the `Host:` header of the HTTP request that you are sending. Also see "Specifying the Destination Server and URL Path," earlier in this book.

Proxy HTTPS

Specifies whether the proxy (if any) uses HTTPS to communicate with the real HTTP/HTTPS server.

Proxy Port

Specifies the proxy server port on which to send HTTP requests, if using a proxy server (will use 8080 by default).

Proxy Server

Specifies the proxy server through which to send HTTP requests, if any.

Proxy HTTP Tunnel

Specifies whether the adapter uses the HTTP CONNECT command to establish a tunnel through the proxy to the target HTTP server. If true, the request uses the HTTP CONNECT command to establish a tunnel. The address of the proxy server is taken from the Proxy Server and Proxy Port properties. If Proxy Https is true, then once the tunnel is established, InterSystems IRIS™ negotiates the SSL connection. The default value is false.

Response Timeout

Specifies the timeout for getting a response from the server (the timeout for opening the connection to the server is set by ConnectTimeout). The default value is 30.

If no response is received, the adapter retries repeatedly, up to the number of times given by Failure Timeout divided by Retry Interval.

SSL Check Server Identity

Specifies that when making an SSL connection, the adapter should check that the server identity in the certificate matches the name of the system being connecting to. This defaults to specifying that the check should be made. Uncheck this for test and development systems where the name specified in the SSL certificate does not match the DNS name.

SSL Configuration

The name of an existing SSL/TLS configuration to use to authenticate this connection. Choose a client SSL/TLS configuration, because the adapter initiates the communication.

To create and manage SSL/TLS configurations, use the Management Portal. See the chapter "Using SSL/TLS with InterSystems IRIS" in the *Security Administration Guide*. The first field on the **Edit SSL/TLS Configuration** form is **Configuration Name**. Use this string as the value for the **SSL Configuration** setting.

SendSuperSession

The SendSuperSession is a Boolean setting that controls whether the outbound adapter creates a SuperSession header in the HTTP header and assigns an identifier to it. When finding a message, you can use the SuperSession value to match a message in one production with the related message in another production. Within a production, it is easy to track a message as it travels between business services, processes, and operations using the SessionId. But once a message leaves a business operation via an HTTP message and enters a different production, the production receiving the message assigns a new SessionId.

If **SendSuperSession** is selected, the HTTP outbound adapter does the following:

1. Check if the message has an empty value in Ens.MessageHeaderBase.SuperSession property. If it does have an empty value, the adapter generates a new value and stores it in the SuperSession property.
2. Stores the value of the SuperSession property in the private InterSystems.Ensemble.SuperSession HTTP header of the outgoing message.

When an HTTP incoming adapter receives a message, it checks for the SuperSession value in the incoming HTTP message header. If the value is present, it sets the Ens.MessageHeaderBase.SuperSession property. This property is preserved as the message passes from one production component to another.

Note: There are no tools to automate tracking messages between productions using SuperSession.

URL

URL path to request from the server (not including `http://` or the server address).

Also see "Specifying the Destination Server and URL Path," earlier in this book.

Use Cookies

Specifies whether to save cookies received in HTTP responses while this adapter is instantiated, and insert them in each subsequent HTTP request.