InterSystems™
IRIS Data Platform

Adding SOAP Services and Web Clients to Productions

Version 2025.1
2025-06-03

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

Table of Contents

1

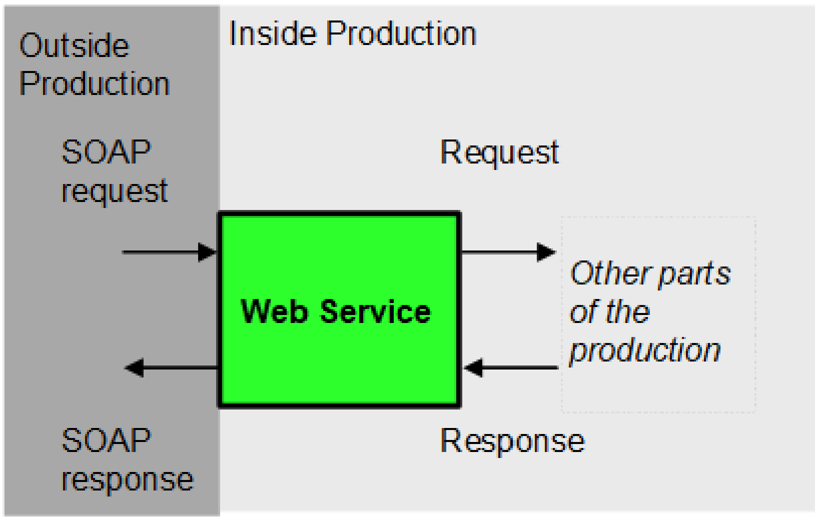
Introduction to Web Services in Productions

This page introduces how you can use web services within interoperability productions.

See Creating Web Services and Web Clients for information on the specific standards followed by InterSystems product support for SOAP and web services, including WSDL limitations.

1.1 InterSystems IRIS Support for Web Services

You can provide a SOAP-enabled front end for your production. To do so, you create an *production web service*, which is both a web service and a business service. Internally, your web methods generally receive SOAP request messages, use them to create and send request messages as needed within the production, receive the response messages, and use them to create SOAP response messages.



To enable you to create a production web service, InterSystems IRIS provides the base production web service class (EnsLib.SOAP.Service), as well as supporting classes in the %SOAP and %XML packages.

InterSystems IRIS provides powerful, built-in support for web services. The base production web service class does the following for you:

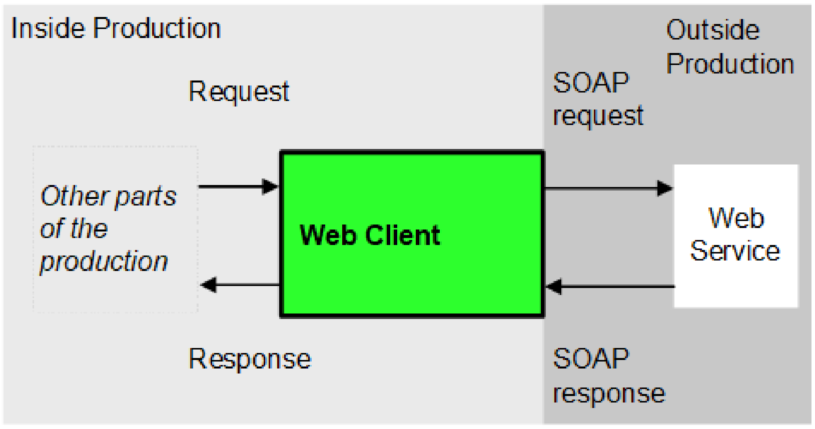
- Validates incoming SOAP messages.
- Unpacks SOAP messages, converts data to InterSystems IRIS representation, and invokes the corresponding method, which sends a request message to a destination inside the production.
- Runs the method.
- Receives a response message and then creates and returns a response message (a SOAP message) to the caller.

The SOAP specification does not include session support. However, it is often useful to maintain a session between a web client and the web service that it uses. You can do this with a production web service. If a web service uses sessions, it establishes a session ID and allows repeated calls on the service after one successfully authenticated call from a client.

The production web service class also provides the full functionality of any business service.

1.2 InterSystems IRIS Support for Web Clients

You can invoke an external web service from within a production. To do so, you create an *InterSystems IRIS web client*.



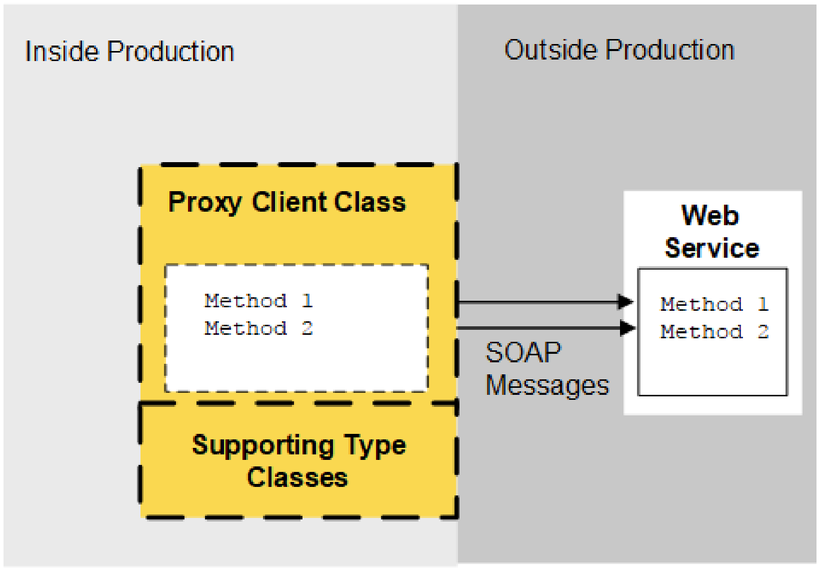
At a high level, your InterSystems IRIS web client receives InterSystems IRIS requests, converts them to SOAP requests and sends them to the appropriate web service. Similarly, it receives SOAP responses and converts them into InterSystems IRIS responses.

The InterSystems IRIS web client consists of the following parts, all of which you can generate programmatically:

- A *proxy client* class that defines a *proxy method* for each method defined by the web service. The purpose of the proxy client is to specify the location of the web service and to contain the proxy methods. Each proxy method uses the same signature used by the corresponding web service method and invokes that method when requested.
- A business operation that uses the InterSystems IRIS SOAP outbound adapter to invoke the proxy methods.
- Supporting classes as needed to define XML types and production messages.

1.2.1 The Proxy Client

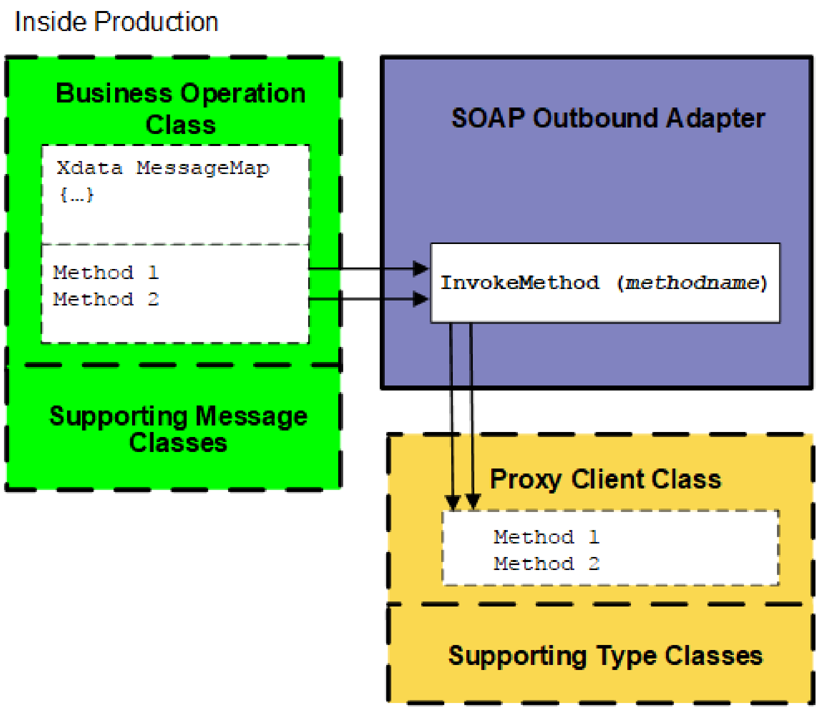
The generated classes include the proxy client class that defines a proxy method for each method of the web service. Each proxy method sends a SOAP request to the web service and receives the corresponding SOAP response.



As shown in the figure, the generated classes also include classes that define any XML types needed as input or output for the methods.

1.2.2 The Business Operation of an InterSystems IRIS Web Client

InterSystems IRIS can also generate a business operation class that invokes the proxy client, as well as classes that define message types as needed. The following figure shows how these classes work:



The classes and methods shown within dashed lines are all generated.

The business operation uses the SOAP outbound adapter, which provides useful runtime settings and the generic method **InvokeMethod()**. To invoke a proxy method in the proxy client class, the business operation class calls **InvokeMethod()**, passing to it the name of the method to run, as well as any arguments. In turn, **InvokeMethod()** calls the method of the proxy client class.

2

Creating a Web Service in a Production

This topic describes how to create an *production web service*, which is a web service in a production. When you do this, you are providing a SOAP-enabled interface to the production.

For settings not discussed here, see Settings in All Productions.

For an alternative approach, see Using the SOAP Inbound Adapter.

Tip: InterSystems IRIS® also provides specialized business service classes that use SOAP, and one of those might be suitable for your needs. If so, no programming would be needed. See Connectivity Options.

2.1 Overall Behavior

A production web service is based on EnsLib.SOAP.Service or a subclass. This class extends both Ens.BusinessService (so that it is a business service) and %SOAP.WebService (so that it can act as a web service as well). A production web service behaves as follows:

- Because it is a web service, it has a WSDL document (generated automatically) that describes the web methods available in it. The service can receive any SOAP message that conforms to the WSDL and sends SOAP responses in return.
- Because it is a business service, it is an integral part of the production to which you add it. Monitoring, error logging, runtime parameters, and all the rest of the production machinery are available as usual.

  Note: A production web service is not available unless the production is running (and the business service is enabled).

Communication with the outside world is done via SOAP request and response messages. InterSystems IRIS Interoperability request and response messages are used within the production.

2.2 Basic Requirements

To create a web service in a production, you 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 order to provide access to the web service, you also need to define a web application definition for the namespace in which the web service is running. The web application definition must be configured to use **CSP/ZEN** options instead of **REST** options. The option **Inbound Web Services** must be enabled.

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

- Your class should extend EnsLib.SOAP.Service. This class extends both Ens.BusinessService (so that it is a business service) and %SOAP.WebService (so that it can act as a web service as well).
- The class should define the *ADAPTER* parameter as null (`""`). For example:

  Class Member

  ```
  Parameter ADAPTER = "";
  ```

  Or, equivalently:

  Class Member

  ```
  Parameter ADAPTER;
  ```

- The class should specify values for other parameters:

Parameter	Description
SERVICENAME	Name of the web service. This name must start with a letter and must contain only alphanumeric characters. The default service name is `"MyProductionRequestWebService"`
NAMESPACE	URI that defines the target XML namespace for your web service, so that your service, and its contents, do not conflict with another service. This is initially set to `http://tempuri.org` which is a temporary URI used by SOAP developers during development.
TYPENAMESPACE	XML namespace for the schema in the types defined by the web service. If you do not specify this parameter, the schema is in the namespace given by *NAMESPACE* instead.
RESPONSENAMESPACE	URI that defines the XML namespace for the response messages. By default, this is equal to the namespace given by the *NAMESPACE* parameter.

- The class should define web methods, as described in Defining Web Methods.
- For other options and general information, see Defining a Business Service Class.

The following example shows in general what the class might look like:

Class Definition

```
Class Hospital.MyService Extends EnsLib.SOAP.Service
{

///For this business service, ADAPTER should be "" so that we use the normal SOAP processing
Parameter ADAPTER = "";

Parameter SERVICENAME = "MyService";

Parameter NAMESPACE = "http://www.myhospital.org";

Parameter USECLASSNAMESPACES = 1;

Method GetAuthorization(patientID As %Integer, RequestedOperation As %String,
LikelyOutcome As %String) As %Status [ WebMethod ]
{
```

```
    set request = ##class(Hospital.OperateRequest).%New()
    set request.PatientID = patientID
    set request.RequestedOperation = RequestedOperation
    set request.LikelyOutcome = LikelyOutcome
    set tSC=..SendRequestSync("Hospital.PermissionToOperateProcess",request,.response)
    // Create the SOAP response, set its properties, and return it.
}

}
```

2.3 Defining Web Methods for Use in InterSystems IRIS

This section describes the basic requirements for an InterSystems IRIS web method.

- Define the method within a subclass of EnsLib.SOAP.Service, as described in Basic Requirements.
- Mark the method with the WebMethod keyword.
- Ensure that all arguments and return values are XML-enabled:
  - If the method uses an object as an argument or a return value, you must ensure that the object is XML-enabled. That is, the class definitions for the types must extend %XML.Adaptor. The default settings for this class are normally suitable; if not, see Projecting Objects to XML.
  - If the method uses a data set as an argument or return value, you must ensure the data set is of type %XML.DataSet.
  - To use a collection (%ListOfObjects or %ArrayOfObjects) as an argument or a return value, you must ensure that the *ELEMENTTYPE* parameter of the collection is set and refers to an XML-enabled class.

Important: In most cases, web methods should be instance methods. Within a web method, it is often necessary to set properties of and invoke methods of the web service instance to fine-tune the behavior of the method. Because a class method cannot do these tasks, a class method is usually not suitable as a web method.

For additional notes, see Basic Requirements.

2.3.1 Basic Steps of an InterSystems IRIS Interoperability Web Method

Within a production web service, a web method should generally do the following:

1. Create a request message and set its properties with information from the inbound SOAP message.
2. Call a suitable method of the business service to send the request to a destination within the production. Specifically, call **SendRequestSync()**, **SendRequestAsync()**, or (less common) **SendDeferredResponse()**. For details, see Sending Request Messages.

   Each of these methods returns a status (specifically, an instance of %Status).
3. Check the status returned from the previous step and react appropriately.
4. Then:
   - In the case of success, look at the response message that is returned by reference and use it to create the return value of the web method. As noted previously, the return value must be XML-enabled so that it can be packaged as a SOAP response.
   - In the case of failure, call the **ReturnMethodStatusFault()** or **ReturnStatusFault()** method of the web service so that a SOAP fault can be returned and an Ens.Alert can be generated; see the next section for details.

2.3.2 Returning Faults to the Caller

By default, if an error occurs when a web method runs, the web service returns a SOAP message to the caller, but this message does not indicate where precisely the fault occurred. An example follows:

```
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Server Application Error</faultstring>
 <detail>
    <error xmlns='http://www.myapp.org' >
      <text>ERROR #5002: ObjectScript error: <INVALID OREF>
          zGetCustomerInfo+10^ESOAP.WebService.1</text>
    </error>
  </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

Your web methods should check for an error and use **ReturnMethodStatusFault()** or **ReturnStatusFault()**. In case of error, the message will be more informative, as follows:

```
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Method</faultcode>
  <faultstring>Server Application Error</faultstring>
  <faultactor>ESOAP.WebService</faultactor>
  <detail>
    <error xmlns='http://www.myapp.org' >
     <text>ERROR <Ens>ErrException:
           <DIVIDE>zGetCustomerRequest+8^ESOAP.MyOperation.1 -
           logged as '13 Jul 2007' number 4 @'    set x=100/0'</text>
   </error>
 </detail>
 </SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

The **ReturnMethodStatusFault()** and **ReturnStatusFault()** methods return a SOAP fault to the caller and then generate an exception which will create an alert (depending on settings). These methods have the following signatures:

```
ClassMethod ReturnStatusFault(pCode As %String,
                              pStatus As %Status)

ClassMethod ReturnMethodStatusFault(pStatus As %Status)
```

Here:

- *pCode* is a string that represents the error code to use in the <faultcode> element of the SOAP fault. The **ReturnMethodStatusFault()** method uses the generic error code `SOAP-ENV:Method`
- *pStatus* is the status to use in the returned SOAP fault. This is used to create the details of the SOAP fault.

Also notice that these methods set the <faultactor> element of the SOAP fault.

2.3.3 Example

The following shows a simple example:

Class Member

```
Method GetCustomerInfo(ID As %Numeric) As ESOAP.SOAPResponse [WebMethod]
{
    //create request message with given ID
    set request=##class(ESOAP.CustomerRequest).%New()
    set request.CustomerID=ID

    //send request message
    set sc= ..SendRequestSync("GetCustomerInfoBO",request,.response)
    if $$$ISERR(sc) do ..ReturnMethodStatusFault(sc)
    //use info from InterSystems IRIS response to create SOAP response
    set soapresponse=##class(ESOAP.SOAPResponse).%New()
    set soapresponse.CustomerID=response.CustomerID
    set soapresponse.Name=response.Name
    set soapresponse.Street=response.Street
    set soapresponse.City=response.City
    set soapresponse.State=response.State
    set soapresponse.Zip=response.Zip

    quit soapresponse
}
```

2.4 Viewing the WSDL

In InterSystems IRIS, a web service runs within an InterSystems IRIS web application. In turn, the web application is served by your choice of web server (the same web server that serves the Management Portal).

InterSystems IRIS automatically creates and publishes a WSDL document that describes your web service. Whenever you modify and recompile the web service, InterSystems IRIS automatically updates the WSDL correspondingly.

The URL has the following form, using the <baseURL> for your instance:

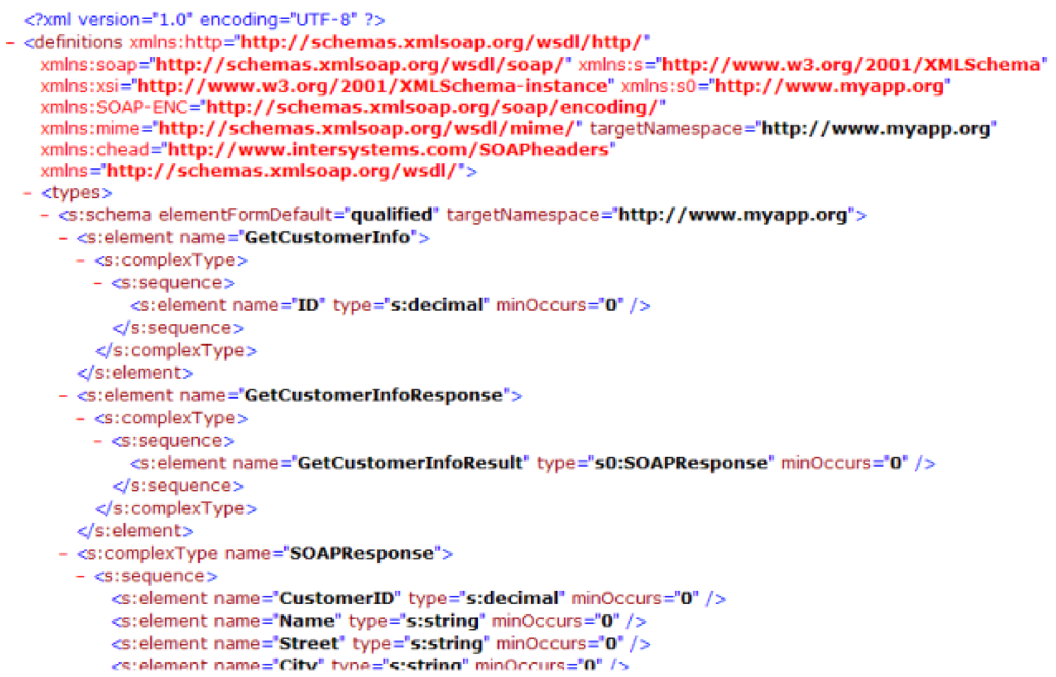
```
https:<baseURL>/csp/app/web_serv.cls?WSDL
```

Here */csp/app* is the name of the web application in which the web service resides, and *web_serv* is the class name of the web service. (Typically, */csp/app* is `/csp/`*namespace*, where *namespace* is the namespace that contains the web application and the production.)

For example:

```
https://devsys/csp/mysamples/MyApp.StockService.cls?WSDL
```

The browser displays the WSDL document as an XML document. The following shows an example:



```
  <?xml version="1.0" encoding="UTF-8" ?>
- <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:s0="http://www.myapp.org"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" targetNamespace="http://www.myapp.org"
    xmlns:chead="http://www.intersystems.com/SOAPheaders"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <s:schema elementFormDefault="qualified" targetNamespace="http://www.myapp.org">
      - <s:element name="GetCustomerInfo">
        - <s:complexType>
          - <s:sequence>
              <s:element name="ID" type="s:decimal" minOccurs="0" />
            </s:sequence>
          </s:complexType>
        </s:element>
      - <s:element name="GetCustomerInfoResponse">
        - <s:complexType>
          - <s:sequence>
              <s:element name="GetCustomerInfoResult" type="s0:SOAPResponse" minOccurs="0" />
            </s:sequence>
          </s:complexType>
        </s:element>
      - <s:complexType name="SOAPResponse">
        - <s:sequence>
            <s:element name="CustomerID" type="s:decimal" minOccurs="0" />
            <s:element name="Name" type="s:string" minOccurs="0" />
            <s:element name="Street" type="s:string" minOccurs="0" />
            <s:element name="City" type="s:string" minOccurs="0" />
```

2.5 Web Service Example

The following simple example shows a production web service that can be used to look up customer information, given a customer ID.

Class Definition

```
Class ESOAP.WebService Extends EnsLib.SOAP.Service
{

Parameter ADAPTER;

Parameter NAMESPACE = "http://www.myapp.org";

Parameter SERVICENAME = "CustomerLookupService";

Method GetCustomerInfo(ID As %Numeric) As ESOAP.SOAPResponse [WebMethod]
{
    //create request message with given ID
    set request=##class(ESOAP.CustomerRequest).%New()
    set request.CustomerID=ID

    //send request message
    set sc= ..SendRequestSync("GetCustomerInfoBO",request,.response)
    if $$$ISERR(sc) do ..ReturnMethodStatusFault(sc)

    //use info from InterSystems IRIS response to create SOAP response
    set soapresponse=##class(ESOAP.SOAPResponse).%New()
    set soapresponse.CustomerID=response.CustomerID
    set soapresponse.Name=response.Name
    set soapresponse.Street=response.Street
    set soapresponse.City=response.City
    set soapresponse.State=response.State
    set soapresponse.Zip=response.Zip

    quit soapresponse
}

}
```

The SOAP response class is as follows:

Class Definition

```
///
Class ESOAP.SOAPResponse Extends (%RegisteredObject, %XML.Adaptor)
{

Property CustomerID As %Numeric;
Property Name As %String;
Property Street As %String;
Property City As %String;
Property State As %String;
Property Zip As %Numeric;

}
```

Note the following points:

- The example web method (`GetCustomerInfo`) uses **SendRequestSync()** to communicate with a business operation elsewhere in the production. The method receives a response message and uses it to create a SOAP response message.
- The SOAP response class has the same properties as the corresponding production message response class. Unlike the production message response, however, the SOAP response class is XML-enabled and non-persistent.

2.6 Enabling SOAP Sessions

The SOAP specification does not include session support. However, it is often useful to maintain a session between a web client and the web service that it uses. You can do this with a production web service. If a web service uses sessions, it establishes a session ID and allows repeated calls on the service after one successfully authenticated call from a client.

Support for SOAP sessions is controlled by the *SOAPSESSION* class parameter. The default is 0, which means that the web service does not use sessions.

To enable SOAP sessions, create a subclass of EnsLib.SOAP.Service and set *SOAPSESSION* to 1 in the subclass. Base your production web service on this subclass.

For more information on SOAP sessions, see Creating Web Services and Web Clients.

2.7 Additional Options

Because your production web service extends %SOAP.WebService, you can use all the SOAP support provided by that class. This support includes options for the following customizations, among others:

- Customizing the SOAP headers
- Passing attachments in the SOAP messages
- Changing the binding style of the SOAP messages from document-style (the default) to rpc-style
- Changing the encoding style of the messages from literal (the default) to SOAP-encoded
- Customizing the XML types used in the SOAP messages
- Customizing the `SOAPAction` header used to invoke a web method
- Controlling whether elements are qualified (controlling the elementFormDefault attribute of the web service)
- Controlling the form of null arguments (to be an empty element rather than omitted)
- Writing the web method to have output parameters instead of return values

For these options and others, see Creating Web Services and Web Clients.

2.8 Adding and Configuring the Web Service

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

1. Add an instance of your custom class to the production.

   Important: Ensure that the configuration name is the same as the full class name, including package. This is a requirement for running a production web service.

2. Enable the business service.
3. Set the Pool Size setting to 0.

   For other settings, see Configuring Productions.

4. Run the production.

3 Creating a Web Client in a Production

This topic describes how to create an InterSystems IRIS® Interoperability web client. At a high level, your InterSystems IRIS Interoperability web client receives InterSystems IRIS Interoperability requests from elsewhere within the production, converts them to SOAP requests, and sends them to the appropriate web service. Similarly, it receives SOAP responses, converts them into InterSystems IRIS responses, and sends them back within the production.

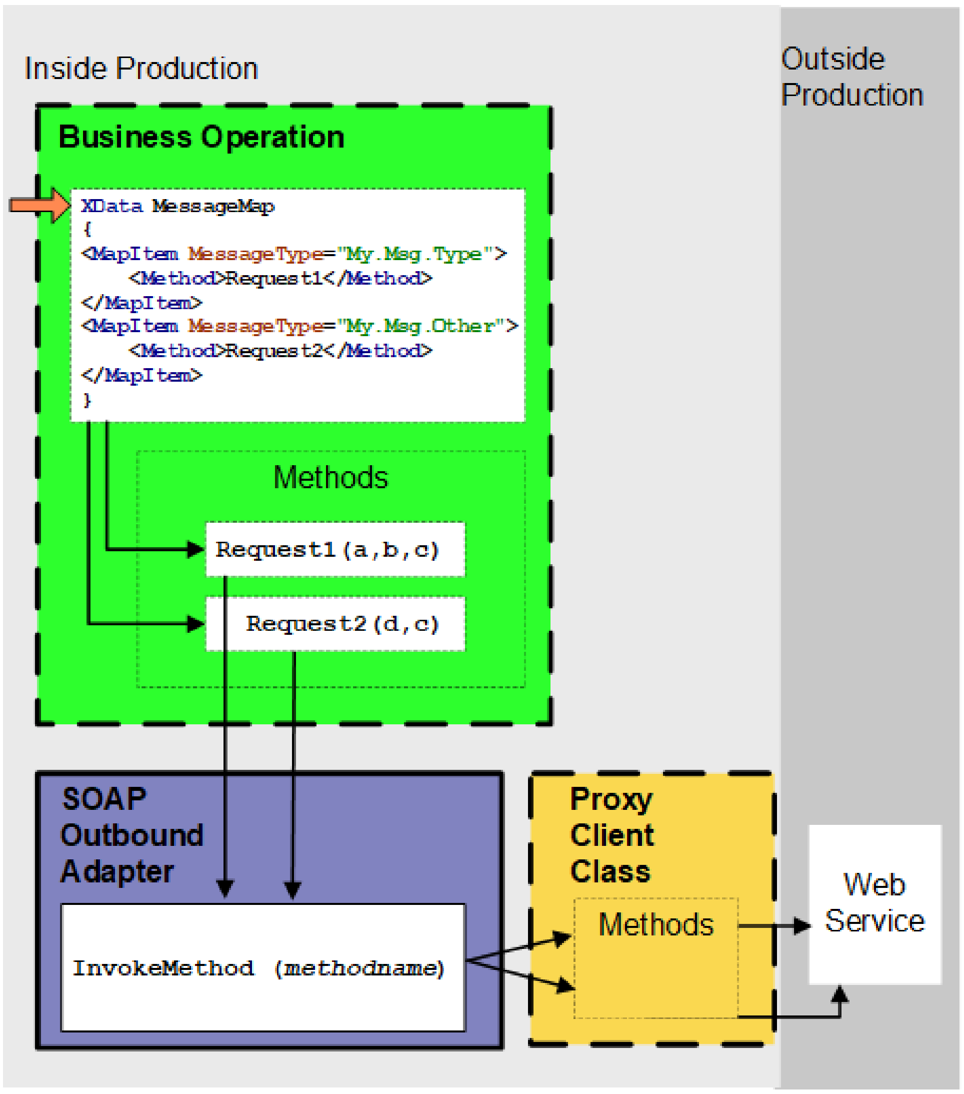
Tip: InterSystems IRIS also provides specialized business service classes that use SOAP, and one of those might be suitable for your needs. If so, no programming would be needed. See Connectivity Options.

3.1 Overview

An InterSystems IRIS Interoperability web client consists of the following parts:

- A *proxy client* class that defines a *proxy method* for each method defined by the web service. Each proxy method uses the same signature used by the corresponding web service method and invokes that method when requested.
- A business operation that uses the InterSystems IRIS Interoperability SOAP outbound adapter to invoke the proxy client.
- Supporting classes as needed to define XML types and production messages.

The following figure shows how the business operation, adapter, and proxy client class work together. Supporting classes are not shown here.



```
XData MessageMap
{
<MapItem MessageType="My.Msg.Type">
    <Method>Request1</Method>
</MapItem>
<MapItem MessageType="My.Msg.Other">
    <Method>Request2</Method>
</MapItem>
}
```

In the preceding figure, all items within dashed lines can be generated programmatically. You can then edit this code as needed.

For a more detailed look at these parts, see InterSystems IRIS Support for Web Clients.

3.2 Basic Steps

This section outlines the basic steps to create an InterSystems IRIS Interoperability web client.

To create an InterSystems IRIS Interoperability web client:

1. Programmatically generate business operation class, proxy client class, and supporting classes, as described below.
2. Check whether you need to adjust the types of the inputs and outputs of the methods, as described below.

3.2.1 Generating the Client Classes

To generate a set of web client classes for a given WSDL:

1. Create an instance of %SOAP.WSDL.Reader.
2. Optionally set properties to control the behavior of your instance.

Property	Purpose
CompileFlags	Specifies the flags to use when compiling the generated classes. The initial expression is `"dk"`; use **$System.OBJ.ShowFlags()** to get information on the available flags.
MakePersistent	If this property is 1, the proxy client is a persistent object; otherwise, it is a registered object. The initial expression is 0.
MakeSerial	If this property is 1 and if MakePersistent is 1, the proxy client is a serial class. The initial expression is 0.
OutputTypeAttribute	Controls how the WSDL reader sets the *OUTPUTTYPEATTRIBUTE* parameter in the proxy client that it generates; which controls the use of the xsi:type attribute in the SOAP messages. See Creating Web Services and Web Clients.
MakeBusinessOperation	Specifies whether to generate a business operation and related request and response objects. The *ADAPTER* setting for this business operation is EnsLib.SOAP.OutboundAdapter. This option works only if you are working within an interoperability-enabled namespace.

   For other properties, see the class documentation for %SOAP.WSDL.Reader.

3. Invoke the **Process()** method, providing the following arguments:
   - The first argument must be the URL of the WSDL of the web service or the name of the WSDL file (including its complete path). Depending on the configuration of the web service, it may be necessary to append a string that provides a suitable username and password; see the examples.
   - The optional second argument is the name of the package in which the reader should place the generated classes. If you do not specify a package, InterSystems IRIS uses the service name as the package name.

3.2.2 Generated Classes and XMLKEEPCLASS

When you generate web client classes, you specify the package for the classes. If this package is the same as an existing package, by default the tool will overwrite any existing classes that have the same name. To prevent InterSystems IRIS from overwriting a class definition, add the *XMLKEEPCLASS* parameter to that class and set this parameter equal to 1.

3.3 Generated Classes for an InterSystems IRIS Interoperability Web Client

This section provides information about the generated web client classes.

Consider a web service named `MyService` that has the following details:

- It is hosted at `https://devsys/csp/mysamples/MyApp.AddService.CLS`
- The target XML namespace for the web service is `http://www.myapp.org`

- It defines a web method named `AddService`, which accepts two complex numbers as arguments and returns the result.

Suppose that you generate a web client client for this web service. If you specify the package for the client classes as `MyClient`, InterSystems IRIS generates the following classes:

- It generates the `MyClient.BusOp.MyServiceSoap` class, which defines the business operation.

```
Class MyClient.BusOp.MyServiceSoap Extends Ens.BusinessOperation
{

Parameter ADAPTER = "EnsLib.SOAP.OutboundAdapter";

Method Add(pRequest As MyClient.BusOp.AddRequest,
Output pResponse As MyClient.BusOp.AddResponse) As %Library.Status
{
   Set ..Adapter.WebServiceClientClass = "MyClient.MyServiceSoap"
   Set tSC = ..Adapter.InvokeMethod("Add",.AddResult,
pRequest.a,pRequest.b) Quit:$$$ISERR(tSC) tSC

   Set tSC = pRequest.NewResponse(.pResponse) Quit:$$$ISERR(tSC) tSC
   Set pResponse.AddResult=AddResult
   Quit $$$OK
}

XData MessageMap
{
<MapItems>
    <MapItem MessageType="MyClient.BusOp.AddRequest">
        <Method>Add</Method>
    </MapItem>
</MapItems>
}

}
```

- It generates the `MyClient.AddServiceSOAP` class, the proxy client class:

Class Definition

```
Class MyClient.AddServiceSoap Extends %SOAP.WebClient
{

/// This is the URL used to access the web service.
Parameter LOCATION = "https://devsys/csp/mysamples/MyApp.AddService.cls";

/// This is the namespace used by the Service
Parameter NAMESPACE = "http://www.myapp.org";

/// Use xsi:type attribute for literal types.
Parameter OUTPUTTYPEATTRIBUTE = 1;

/// This is the name of the Service
Parameter SERVICENAME = "AddService";

Method Add(a As MyClient.ComplexNumber, b As MyClient.ComplexNumber)
   As MyClient.ComplexNumber [ Final,
   SoapBindingStyle = document, SoapBodyUse = literal, WebMethod ]
{
 Quit ..WebMethod("Add").Invoke($this,
      "http://www.myapp.org/MyApp.AddService.Add",.a,.b)
}

}
```

- It generates the request and response message classes needed by the business operation. The request class is as follows:

Class Definition

```
Class MyClient.BusOp.AddRequest Extends Ens.Request
{

Parameter RESPONSECLASSNAME = "MyClient.BusOp.AddResponse";

Property a As MyClient.ComplexNumber;

Property b As MyClient.ComplexNumber;

}
```

The response class is as follows:

Class Definition

```
Class MyClient.BusOp.AddResponse Extends Ens.Response
{

Property AddResult As MyClient.ComplexNumber;

}
```

- Finally, it generates the `MyClient.ComplexNumber` class, which defines a complex number and which is used by the other classes.

Class Definition

```
/// Created from: http://devsys/csp/mysamples/MyApp.AddService.CLS?WSDL=1
Class MyClient.ComplexNumber Extends (%RegisteredObject, %XML.Adaptor)
{
Parameter XMLNAME = "ComplexNumber";

Parameter XMLSEQUENCE = 1;

Property Real As %xsd.double(XMLNAME = "Real") [ SqlFieldName = _Real ];

Property Imaginary As %xsd.double(XMLNAME = "Imaginary");

}
```

When you compile these classes, the compiler also generates a class for each method defined in the web service. These classes are *not* automatically added to your project and their internal details are subject to change. These classes extend %SOAP.ProxyDescriptor, which you should never subclass yourself.

3.4 Creating a Business Operation Class Manually

Rather than using the generated business operation class, you can create your own class. This section describes how to do so. It discusses the following:

- Basic requirements of the business operation class
- Basic requirements of the methods
- Specific techniques, with examples, for calling the proxy methods
- Reference information for the adapter property and methods used here

3.4.1 Basic Requirements of the Class

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.SOAP.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 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 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 one method for each method of the proxy client, as described in the following section.
- Your class should define a *message map* that includes one entry for each method. A message map is an XData block entry that has the following structure:

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

You will also need to define the production message classes that the business operation uses.

3.4.2 Basic Requirements of the Methods

Within your business operation class, your methods should invoke the proxy methods. Here the general requirements are as follows:

1. The method should have the same signature as the proxy method that it is invoking.
2. The method should be marked with the WebMethod keyword.
3. The method should set the SoapBindingStyle and SoapBodyUse keywords as expected by the proxy client. (That is, use the same values as in the signature of the corresponding proxy method.)
4. The method should set the WebServiceClientClass property of the adapter. When this property is set, a proxy client instance is created and placed in the %Client property of the adapter.
5. The method should call the corresponding proxy method, using one of the techniques in the next section.
6. The method should check the status.
7. Then:
   - In the case of success, create a new response message (via the **NewResponse()** method of the request) and set its properties as appropriate.
   - In the case of failure, quit with the error.

3.4.3 Ways to Execute the Proxy Methods

Within your business operation, your methods should execute the proxy methods of the proxy client class. You can do this in multiple ways, which are best shown via an example. This section uses an example web service that has a web method named `GetStock` that accepts a stock symbol (a string) and returns a number. Suppose that you have used generated a proxy client (`GetStock.StockServiceSoap`), which contains a method called `GetStock`.

Also suppose that you have created message classes as follows:

Class Definition

```
Class GetStock.BusOp.GetQuoteRequest Extends Ens.Request
{

Parameter RESPONSECLASSNAME = "GetStock.BusOp.GetQuoteResponse";

Property StockName As %String;

}
```

And

Class Definition

```
Class GetStock.BusOp.GetQuoteResponse Extends Ens.Response
{

Property StockValue As %Numeric;

}
```

To execute the proxy method `GetStock`, your business operation class can do any of the following:

- Call the **InvokeMethod()** method of the adapter and specify the name of the proxy method to run, as well as any number of arguments. In this case, there is only one argument (which we specify as `pRequest.StockName`). For example:

  #### Class Member

  ```
  Method GetQuote1(pRequest As GetStock.BusOp.GetQuoteRequest,
  Output pResponse As GetStock.BusOp.GetQuoteResponse) As %Status
  {
   set ..Adapter.WebServiceClientClass = "GetStock.StockServiceSoap"

   set status = ..Adapter.InvokeMethod("GetQuote",.answer,pRequest.StockName)
   if $$$ISERR(status) quit status

   set pResponse=##class(GetStock.BusOp.GetQuoteResponse).%New()
   set pResponse.GetQuoteResult=answer
   quit $$$OK
  }
  ```

- Access the %Client property of the adapter, which gives you an instance of the proxy client class, and execute the proxy method of that property. The %Client property is set when you set the WebServiceClientClass property. In this case, %Client has a method named `GetQuote`, which accepts a string stock symbol. For example:

  #### Class Member

  ```
  Method GetQuote2(pRequest As GetStock.BusOp.GetQuoteRequest,
  Output pResponse As GetStock.BusOp.GetQuoteResponse) As %Status
  {
   set ..Adapter.WebServiceClientClass = "GetStock.StockServiceSoap"

   set client=..Adapter.%Client
   set answer=client.GetQuote("GRPQ")

   set pResponse=##class(GetStock.BusOp.GetQuoteResponse).%New()
   set pResponse.GetQuoteResult=answer
   quit $$$OK
  }
  ```

  Note that with this technique, you do not have access to the retry logic of InterSystems IRIS.

- Create a proxy method object by calling the **WebMethod()** method of the adapter. Set properties of this object as appropriate (one property per named argument). In this case, **WebMethod()** returns an object with one property: `StockName`. After setting properties as needed, call the **Invoke()** method of that object. For example:

  Class Member

  ```
  Method GetQuote3(pRequest As GetStock.BusOp.GetQuoteRequest,
  Output pResponse As GetStock.BusOp.GetQuoteResponse) As %Status
  {
   set ..Adapter.WebServiceClientClass = "GetStock.StockServiceSoap"


   set proxymethod=..Adapter.WebMethod("GetQuote")
   set proxymethod.StockName=pRequest.StockName


   set status=..Adapter.Invoke(proxymethod)
   if $$$ISERR(status) quit status


   set pResponse=##class(GetStock.BusOp.GetQuoteResponse).%New()
   set pResponse.GetQuoteResult=proxymethod.%Result
   quit $$$OK
  }
  ```

  In this case, you can provide any number of arguments.

3.4.4 Reference Information

This section provides reference information for the adapter property and methods mentioned in the previous section.

%Client property

```
%SOAP.WebClient
```

The associated instance of the proxy client (an instance of %SOAP.WebClient). This property is set when you set the WebServiceClientClass property of the adapter.

InvokeMethod() method

```
Method InvokeMethod(pMethodName As %String,
       Output pResult As %RegisteredObject,
       pArgs...) As %Status
```

Calls the specified method of the proxy client class, passing all the arguments and returns the status. The output is returned by reference as the second argument.

WebMethod() method

```
Method WebMethod(pMethodName As %String) As %SOAP.ProxyDescriptor
```

Returns an object that corresponds to the specified method. This object has one property corresponding to each method argument; you should set this properties before using the **Invoke()** method. For details on %SOAP.ProxyDescriptor, see the class reference.

Invoke() method

```
Method Invoke(pWebMethod As %SOAP.ProxyDescriptor) As %Status
```

Calls the given method and returns the status.

3.5 Adding and Configuring the Web Client

To add your InterSystems IRIS Interoperability web client to a production, use the Management Portal to do the following:

1. Add an instance of your custom business operation class to the production, specifically the generated business operation class.
2. Enable the business operation.
3. Specify appropriate values for the runtime settings of the associated adapter, as discussed below.
4. Run the production.

The following subsections describes the runtime settings for your IRIS Interoperability web client, which fall into several general groups:

- Basic settings
- Settings related to credentials
- The setting that controls use of TLS
- Settings that control the use of a proxy server

For settings not listed here, see Settings in All Productions.

3.5.1 Specifying Basics

The following settings specify the basic information for the InterSystems IRIS Interoperability web client:

- Web Service URL
- Web Service Client Class
- Response Timeout

3.5.2 Specifying Credentials

The web service that you are accessing might require a username and password. In general, the InterSystems IRIS Interoperability SOAP client can log in to a web service in either of the following ways:

- You can use WS-Security user authentication. In this case, you include a WS-Security header in the SOAP request; this header includes the username and password. The proxy client automatically does this if you specify a value for the SOAP Credentials setting.

  CAUTION: Ensure that you are using TLS between the web client and the web service. The WS-Security header is sent in clear text, so this technique is not secure unless TLS is used.

- You can use basic HTTP user authentication, which is less secure than WS-Security but is sometimes required. In this case, you include the username and password in the HTTP header of the SOAP request. The proxy client automatically does this if you specify a value for the Credentials setting.

Use the technique that is appropriate for the web service you are using.

3.5.3 Specifying a TLS Configuration

If your web server supports it, you can connect with TLS. To do so, specify a value for the SSL Configuration setting.

Note: You must also ensure the web service is at a URL that uses `https://`. The web service location is determined by the Web Service URL setting; if this is not specified, the InterSystems IRIS Interoperability web client assumes the web service is at the URL specified by the *LOCATION* parameter in the proxy client class.

3.5.4 Specifying a Proxy Server

You can communicate with the web service via a proxy server. To set this up, use Proxy Server and other settings in the **Proxy Settings** group.

A

Configuring a Production for SOAP Services

This topic briefly discusses how to configure your system so that you can use HTTP and SOAP services through the Web port. This information is intended to help you set up a development or test system for these services. Complete information about these topics is provided in the documentation. See Configuring System-Wide Settings for more details.

To set up an InterSystems IRIS® development or test system for HTTP or SOAP services, follow these steps:

1. If you are not using an existing namespace, create a new namespace.
2. Create an empty role and assign it to the unknown user:
   a. Select **System Administration**, **Security**, and **Roles** to display the Roles portal page.
   b. Click the **Create New Role** button and name the role, for example, Services_Role, and click the **Save** button.
   c. Select the **Members** tab, select the Unknown User, click the right arrow, and click the **Assign** button.
3. Define a web application that will handle calls to the Web port. The web application name defines the root of the URL that will call the service. A single web application can support multiple business services but they must all have a class that is the same or a subclass of the web application dispatch class.
   a. Select **System Administration**, **Security**, **Applications**, and **Web Applications** to display the **Web Applications** portal page. Click the **Create New Web Application** button.
   b. Name the web application, such as `/weatherapp` or `/math/sum`. You must start the name with a / (slash) character.
   c. Set the **Namespace** to the namespace that the production is running in, such as SERVICESNS. Leave the Namespace Default Application unchecked.
   d. You can check the Application, CSP/ZEN, and Inbound Web Services check boxes.
   e. Leave the Resource Required and Group By ID fields empty.
   f. Check the Unauthenticated check box on the **Allowed Authentication Methods** line.

      Or for an alternative, see the tip after this list.
   g. Click **Save**.
   h. Select the **Matching Roles** tab.
   i. In the **Select a Matching Role:** field, select the role that you created in the previous step.

j. In the **Select target roles to add to the selected matching role** field, select the role or roles associated with the namespace globals and routines. The globals and routines may be in the same database or in separate databases. If your service accesses another InterSystems IRIS database, you should also select its role. You can select multiple roles while holding the **Ctrl** key.

   Note: The globals database also may have a secondary database and a corresponding role, such as %DB_GDBSECONDARY. This secondary database is used to store passwords. You don't need access to this database for pass-through services and operations, but if you create a custom web service that uses password access, you should also add the secondary database role to the target database.

k. After the roles are highlighted, click the right-arrow key to move them to the **Selected** text box.

l. Then click the **Assign** button.

Tip: If you want to authenticate users:

1. Create a new class, extending EnsLib.SOAP.GenericService.
2. In this class, override the parameter *SECURITYIN*, setting it to either ALLOW or REQUIRE.
3. Use this new class when configuring the business service.
4. Use the WS-Security Username Token when invoking the service.
5. In the web application configuration, clear the **Unauthenticated** option.

B
Using the SOAP Inbound Adapter

This topic briefly discusses the class EnsLib.SOAP.InboundAdapter, which you can use as an alternative to EnsLib.SOAP.Service (described in Creating a Web Service in a Production).

The standard way to create a web service in an interoperability production is to create a subclass of EnsLib.SOAP.Service and set up a web server to be your production web server. In this way, your system will be able to utilize all the SOAP and security features provided by a commercial web server and the InterSystems IRIS® SOAP framework. In contrast, using the EnsLib.SOAP.InboundAdapter is easier to configure and lighter weight but doing so bypasses the above mentioned formal web support machinery. Also, the adapter does not expose the WSDL and test page the way the standard InterSystems IRIS SOAP framework does.

B.1 Notes

The SOAP inbound adapter (EnsLib.SOAP.InboundAdapter) does not require web server software. Instead it spawns a TCP listener job using the InterSystems IRIS super server. This lets you run your service in a foreground window, which is useful for debugging. (To do this, you must be running the service locally. Also make sure the PoolSize setting is 1 and the JobPerConnection setting is false.) It also supports TLS.

The EnsLib.SOAP.InboundAdapter listens for HTTP input on a given port. When the adapter receives input, the following occurs:

1. It extracts the HTTP `SOAPaction` header.
2. It creates a stream (%Library.GlobalBinaryStream) that contains the body of the input.
3. It calls the web method that corresponds to the given `SOAPaction`.

This adapter provides an advantage in that it supports persistent connections for successive SOAP calls. Also, if you use it with JobPerConnection=0, it can retain expensively instantiated resources such as XPath parsers even across successive connections that encompass individual SOAP service calls.

B.2 Development Tasks

To use the InterSystems IRIS SOAP inbound adapter, write and compile a new business service class in an IDE. The following list describes the basic requirements:

- Your class should extend EnsLib.SOAP.Service. This class extends both Ens.BusinessService (so that it is a business service) and %SOAP.WebService (so that it can act as a web service as well).
- Your class should provide values for *SERVICENAME* and other parameters, as described in Basic Requirements.
- The class should define web methods, as described in Defining Web Methods for Use in InterSystems IRIS.

Optionally, to disable support for calls via the adapter in your service, add this to your class:

```
Parameter ADAPTER="";
```

B.3 Configuration Tasks

Use the Management Portal to do the following:

1. Add an instance of your custom class to the production.

   Important: Ensure that the configuration name is the same as the full class name, including package. This is a requirement for running a production web service.

2. Enable the business service.
3. Set the PoolSize setting to 1 so that the adapter can use its TCP listener.
4. Set the StayConnected setting to 0. Otherwise, clients may hang for their timeout period while waiting for the server to drop the connection.
5. Specify other settings as needed; see Settings for the SOAP Inbound Adapter.
6. Run the production.

C
Older Web Service Variation

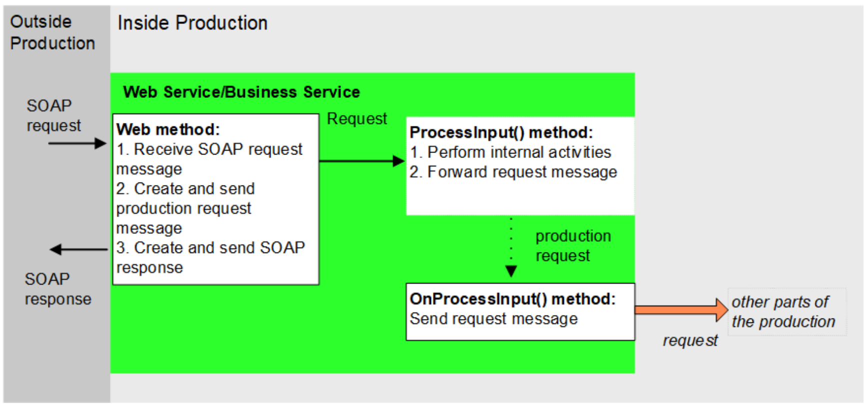
In previous releases, an InterSystems IRIS® web method could not directly call **SendRequestSync()**, **SendRequestAsync()**, or **SendDeferredResponse()**. An alternative approach was needed. This topic provides the details, for the benefit of anyone who is maintaining code that uses this alternative approach.

C.1 Overview

In previous releases, an InterSystems IRIS web method could not directly call **SendRequestSync()**, **SendRequestAsync()**, or **SendDeferredResponse()**. Instead, there were two requirements for a production web service, in addition to the basic requirements:

- Each web method had to invoke the internal **ProcessInput()** method as appropriate, passing to it the appropriate request production message and receiving a response message.
- The web service class had to define the **OnProcessInput()** callback method. In *this* method, you would call **SendRequestSync()**, **SendRequestAsync()**, or **SendDeferredResponse()**.

The following figure shows the overall flow of request messages in this scenario:


Outside Production
Inside Production
Web Service/Business Service
SOAP request
SOAP response
Web method:
1. Receive SOAP request message
2. Create and send production request message
3. Create and send SOAP response
Request
ProcessInput() method:
1. Perform internal activities
2. Forward request message
production request
OnProcessInput() method:
Send request message
request
other parts of the production

C.2 Implementing the OnProcessInput() Method

The **OnProcessInput()** method has the following signature:

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

Here:

1. *pInput* is the request message that you are sending.
2. *pOutput* is the response message that is sent in return.
3. *pHint* is an optional string that you can use to decide how to handle the InterSystems IRIS request; see Using the pHint Argument.

The following shows an example:

Class Member

```
Method OnProcessInput(pInput As %RegisteredObject, ByRef pOutput As %RegisteredObject) As %Status
{
    set sc= ..SendRequestSync("Lookup",pInput,.pOutput)
    Quit sc
}
```

C.2.1 Using the pHint Argument

If a web service defined multiple methods, and you wanted to send them to different destinations within the production, you used the optional hint argument of the internal **ProcessInput()** method and the **OnProcessInput()** callback method, as follows:

1. When invoking **ProcessInput()**, you used a value for the hint argument to indicate which web method is making this call. For example:

   ```
   Method GetCustomerInfo(ID As %Numeric) As ESOAP.SOAPResponse [ WebMethod ]
   {
       //create request message with given ID
       set request=##class(ESOAP.CustomerRequest).%New()
       set request.CustomerID=ID

       //send request message
       //ProcessInput() calls OnProcessInput(), which actually
       //sends the message
       set sc=..ProcessInput(request,.response,"GetCustomerInfo")
   ...

       quit soapresponse
   }
   ```

2. Within **OnProcessInput()**, you used the hint argument to determine the flow. For example:

   #### Class Member

   ```
   Method OnProcessInputAlt(pInput As %RegisteredObject,
   ByRef pOutput As %RegisteredObject, pHint As %String) As %Status
   {
       if pHint="GetCustomerInfo"{
           set sc= ..SendRequestSync("GetCustomerInfoBO",pInput,.pOutput)
           }
       elseif pHint="GetStoreInfo" {
               set sc= ..SendRequestSync("GetStoreInfoBO",pInput,.pOutput)
       }
       Quit sc
   }
   ```

SOAP Adapter Settings

This section provides reference information for the SOAP adapters.

Also see Settings in All Productions.

Settings for the SOAP Inbound Adapter

Provides reference information for settings of the SOAP inbound adapter, EnsLib.SOAP.InboundAdapter. Also see Creating a Web Service in a Production, which does not require this adapter.

Summary

The inbound SOAP adapter has the following settings:

Group	Settings
Basic Settings	Port, Call Interval
Connection Settings	Enable Standard Requests, Adapter URL, Job Per Connection, Allowed IP Addresses, OS Accept Connection Queue Size, Stay Connected, Read Timeout, SSL Configuration, Local Interface
Delayed Response Support	Support Delayed Sync Request, Override Client Response Wait Timeout, Gateway Timeout
Additional Settings	Generate SuperSession ID

The remaining settings are common to all business services. For information, see Settings for All Business Services.

Adapter URL

A specific URL for the service to accept requests on. For SOAP services invoked through the SOAP inbound adapter on a custom local port, this setting allows a custom URL to be used instead of the standard csp/namespace/classname style of URL.

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. 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.

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.

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.

Call Interval

Specifies the number of seconds that the adapter will listen for incoming data from its configured source, before checking for a shutdown signal from the production framework.

If the adapter finds input, it acquires the data and passes it to the business service. The business service processes the data, and then the adapter immediately begins waiting for new input. This cycle continues whenever the production is running and the business service is enabled and scheduled to be active.

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

Enable Standard Requests

If this setting is true, the adapter can also receive SOAP requests in the usual way (that is, via the Web Gateway port, rather than the TCP connection). The default is false.

Note: Note that if the adapter uses the Web Gateway port, First-In-First-Out processing for messages is not supported.

Gateway Timeout

Specifies the expected timeout (in seconds) of the external TCP socket system.

The default value is "IRIS" which allows the adapter to use the IRIS web gateway timeout value found in the request headers as long as the incoming request comes through an IRIS web application. Otherwise, the default value will be 60 seconds.

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 SuperSession 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. When false, it 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.

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.

Override Client Response Wait Timeout

Specifies the expected response wait time (in seconds) for the client.

The default is 0 which uses the timeout set by the client.

If the client is an IRIS Interoperability production SOAP operation, the client response time will be included in the HTTP headers.

Port

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

Read Timeout

Specifies the number of seconds to wait for each successive incoming TCP read operation, following receipt of initial data from the remote TCP port. The default is 5 seconds. The range is 0–600 seconds (a maximum of 10 minutes).

SSL Config

The name of an existing TLS configuration to use to authenticate this connection. This should be a server configuration.

To create and manage TLS configurations, use the Management Portal. See InterSystems TLS Guide. The first field on the **Edit SSL/TLS Configuration** page is **Configuration Name**. Use this string as the value for the **SSLConfig** setting.

Stay Connected

Specifies whether to keep the connection open between requests.

- If this setting is 0, the adapter will disconnect immediately after every event.
- If this setting is -1, the adapter auto-connects on startup and then stays connected.
- This setting can also be positive (which specifies the idle time, in seconds), but such a value is not useful for this adapter, which works by polling. If the idle time is longer than the polling interval (that is, the CallInterval) the adapter stays connected all the time. If the idle time is shorter than the polling interval, the adapter disconnects and reconnects at every polling interval.

Support Delayed Sync Request

Specifies whether delayed sync requests are enabled. The default is disabled.

If the client provides unique `ClientRequestKey` and `ClientRetryRequestKey`, then the SendRequestSync carried out by this service will attempt to prevent processing the same request multiple times if the client times out waiting for the initial response and retries.

Settings for the SOAP Outbound Adapter

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

Summary

The outbound SOAP adapter has the following settings:

Group	Settings
Basic Settings	Web Service URL, Web Service Client Class, SOAP Credentials, Credentials
Connection Settings	SSL Configuration, SSL Check Server Identity
Proxy Settings	Proxy Server, Proxy Port, Proxy HTTPS, ProxyHttpTunnel, ProxyHttpSSLConnect
Delayed Response Support	Include Request Key In HTTP Header
Additional Settings	ResponseTimeout, HttpVersion, ConnectTimeout, SendSuperSession

The remaining settings are common to all business operations. For information, see Settings for All Business Operations.

ConnectTimeout

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

Specify the ID of the *production credentials* that contain the username and password to be used in the HTTP header. See Defining Production Credentials.

HttpVersion

Specifies the HTTP version that the adapter should report in the HTTP request it sends to the server.

Include Request Key in HTTP Header

Specifies whether the request key should be included in the HTTP headers. The default is disabled.

If checked, the key will either be `VND.InterSystems.IRIS.RequestKey` or `VND.InterSystems.IRIS.RetryRequestKey` and have a value of a System GUID, the current namespace, and the current request header ID.

The HTTP header will also include `VND.InterSystems.IRIS.ResponseTimeout` and have a value of ResponseTimeout.

This setting is intended to be used in conjunction with an IRIS Interoperability production SOAP service with delayed response support enabled.

This setting should not be enabled if the request payload is changed by the host class on retrying and has a different expected response from the original request.

ProxyHTTPS

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

ProxyHttpTunnel

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 SSL Connect is true, then once the tunnel is established, InterSystems IRIS® negotiates the TLS connection. The default value is false.

ProxyPort

Specifies the proxy server port on which to send HTTP requests, if using a proxy server. The default value is 80.

ProxyServer

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

ProxyHttpSSLConnect

Specifies whether the adapter should use a proxy TLS connection to the proxy. Note that the use of TLS to the eventual endpoint is determined by the protocol part of web service's location URL.

ResponseTimeout

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

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 a SOAP message and enters a different production, the production receiving the message assigns a new SessionId.

If **SendSuperSession** is selected, the SOAP 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 SOAP 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.

SOAPCredentials

Specify the ID of the *production credentials* that contain the username and password to be used in the WS-Security header of the SOAP request. See Defining Production Credentials. For more information on WS-Security support, see Securing Web Services.

SSLCheckServerIdentity

Specifies that when making a TLS 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 TLS certificate does not match the DNS name.

SSLConfig

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

To create and manage TLS configurations, use the Management Portal. See InterSystems TLS Guide. The first field on the **Edit SSL/TLS Configuration** page is **Configuration Name**. Use this string as the value for the **SSLConfig** setting.

Note: You must also ensure the web service is at a URL that uses `https://`. The web service location is determined by the WebServiceURL setting; if this is not specified, the InterSystems IRIS web client assumes the web service is at the URL specified by the *LOCATION* parameter in proxy client class.

WebServiceClientClass

Specifies the full name (including package) of the proxy client class, specifically the class that actually sends and receives SOAP messages to the web service.

WebServiceURL

Specifies the URL where the web service is located. If this setting is not given, the adapter uses the default location (the *LOCATION* parameter) declared in the proxy client class; see the WebServiceClientClass setting. Note that TLS will only work if this URL uses `https://`