Creating Web Services and Web Clients

Version 2024.2
2024-09-05

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

Creating Web Services and Web Clients
PDF generated on 2024-09-05
InterSystems IRIS® Version 2024.2

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

List of Tables

1

Introduction to Web Services and Web Clients in InterSystems IRIS

InterSystems IRIS® data platform supports SOAP 1.1 and 1.2 (Simple Object Access Protocol). This support is easy to use, efficient, and fully compatible with the SOAP specification. This support is built into InterSystems IRIS and is available on every platform supported by InterSystems IRIS.

1.1 Introduction to InterSystems IRIS Web Services

This section introduces InterSystems IRIS web services.

1.1.1 Creating InterSystems IRIS Web Services

In InterSystems IRIS, you can create a web service in any of the following ways:

- By converting an existing class to a web service with a few small changes. You also need to modify any object classes used as arguments so that they extend %XML.Adaptor and can be packaged in SOAP messages.
- By creating a new web service class from scratch.
- By reading an existing WSDL document and generating a web service class and all supporting type classes.

  This technique (*WSDL-first development*) applies if the WSDL has already been designed and it is now necessary to create a web service that complies with it.

1.1.2 InterSystems IRIS Web Service as Part of a Web Application

An InterSystems IRIS web service must run within a *web application* that you configure within the Management Portal. Specifically, before you can use a web service class, you must define a web application that uses the namespace that contains the class.

For details on defining and configuring this web application, see Defining Applications in the *Authorization Guide*.

1.1.3 The WSDL

When the class compiler compiles a web service, it generates a WSDL for the service and publishes that via your configured web server, for your convenience. This WSDL complies with the Basic Profile 1.0 established by the WS-I (Web Services

Interoperability Organization). In InterSystems IRIS, the WSDL document is served dynamically at a specific URL, and it automatically reflects any changes you make to the interface of your web service class (apart from header elements added at runtime). In most cases, you can use this document to generate web clients that interoperate with the web service.

For details and important notes, see Viewing the WSDL.

1.1.4 Web Service Architecture

To understand how an InterSystems IRIS web service works by default, it is useful to follow the events that occur when the web service receives a message it can understand: an HTTP request that includes a SOAP message.

First consider the contents of this HTTP request, which is directed to a specific URL:

- HTTP headers that indicate the HTTP version, character set, and other such information.

  The HTTP headers must include the SOAP action, which is a URI that indicates the intent of the SOAP HTTP request.

  For SOAP 1.1, the SOAP action is included as the `SOAPAction` HTTP header. For SOAP 1.2, it is included within the `Content-Type` HTTP header.

  The SOAP action is generally used to route the inbound SOAP message. For example, a firewall could use this header to appropriately filter SOAP request messages in HTTP. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable.
- A request line, which includes a HTTP method such as GET, POST, or HEAD. This line indicates the action to take.
- The message body, which in this case is a SOAP message that contains a method call. More specifically, this SOAP message indicates the name of the method to invoke and values to use for its arguments. The message can also include a SOAP header.

Now let us examine what occurs when this request is sent:

1. The request is received by a third-party web server.
2. Because the request is directed to a URL that ends with .cls, the web server forwards the request to the Web Gateway.
3. The Web Gateway examines the URL. It interprets part of this URL as the logical name of a web application. The Gateway forwards the request to the appropriate physical location (the page for the web service), within that web application.
4. When the web service page receives the request, it invokes its **OnPage()** method.
5. The web service checks whether the request includes an InterSystems IRIS SOAP session header and if so, resumes the appropriate SOAP session or starts a new one.

   Note: This step refers to SOAP sessions as supported by InterSystems IRIS SOAP support. The SOAP specification does not define a standard for sessions. However, InterSystems IRIS SOAP support provides a proprietary InterSystems IRIS SOAP session header that you can use to maintain a session between a web client and a web service, as described here.

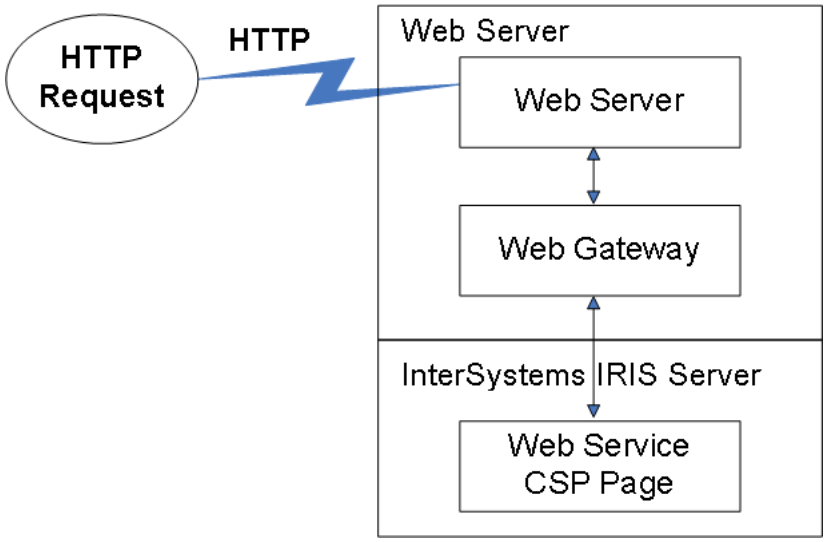
6. The web service unpacks the message, validates it, and converts all input parameters to their appropriate InterSystems IRIS representation. For each complex type, the conversion creates an object instance that represents the complex type and uses that object as input for the web method.

   The SOAP action from the HTTP header is used here to determine the method and hence the request object.

   When the web service unpacks the message, it creates a new request object and imports the SOAP message into that object. In this process, the web service uses a generated class (a web method handler class) that was created when you compiled the web service.

7. The web service executes the requested InterSystems IRIS method, packages up the reply, and constructs a SOAP response, including a SOAP header if appropriate.
8. The web service writes the SOAP response (an XML document) to the current output device.

The following figure shows the external parts of this flow:



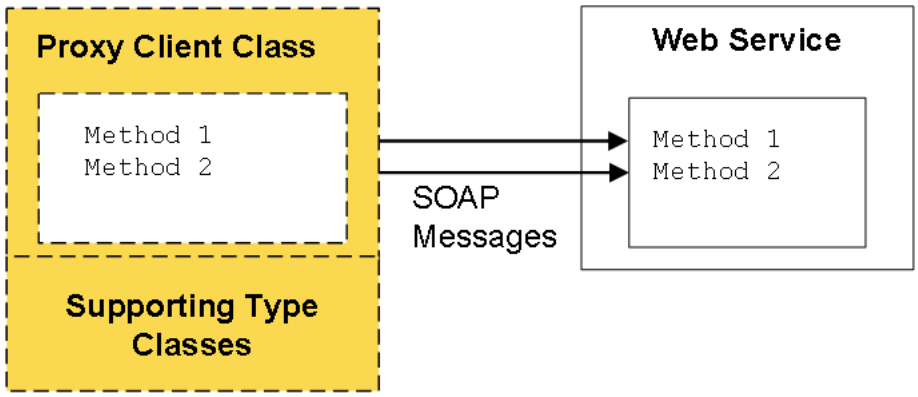
1.2 Introduction to InterSystems IRIS Web Clients

This section introduces InterSystems IRIS web clients.

1.2.1 Creating InterSystems IRIS Web Clients

In InterSystems IRIS, you create a web client by programmatically reading an existing WSDL document and generating a web client class and all supporting type classes.

The generated web client interface includes a client class that contains a proxy method for each method defined by the web service. Each proxy uses the same signature used by the corresponding web service method. The interface also includes classes to define any XML types needed as input or output for the methods.



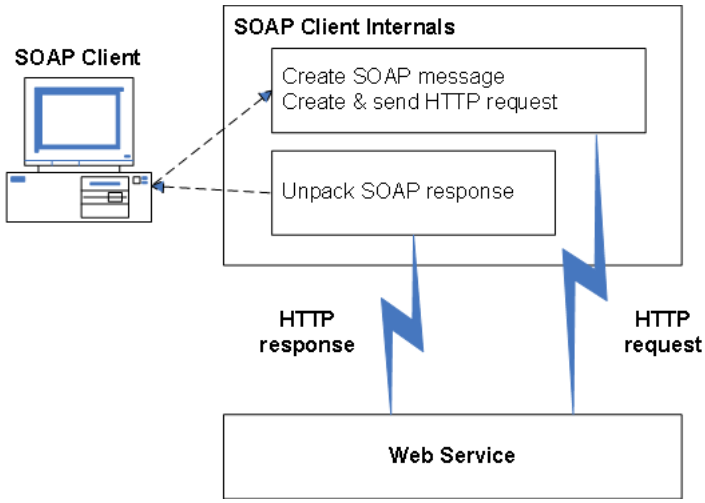
Typically you do not customize the generated classes. You instead create additional classes that control the behavior of your web client and invoke its proxy methods.

1.2.2 Web Client Architecture

To understand how an InterSystems IRIS web client works, we follow the events that occur when a user or other agent invokes a method within the web client.

1. First the web client creates a SOAP message that represents the method call and its argument values.
2. Next it creates an HTTP request that includes the SOAP message. The HTTP request includes a request line and HTTP headers, as described earlier.
3. It issues the HTTP request, sending it to the appropriate URL.
4. It waits for the HTTP response and determines the status.
5. It receives the SOAP response from the web service.
6. It unpacks the SOAP response.

The following figure shows this flow:



1.3 Additional Features

You can add the following features to your InterSystems IRIS web services and web clients:

- Session support. As noted earlier, although the SOAP specification does not define a standard for sessions, with InterSystems IRIS, you can create client-server SOAP sessions.
- Custom SOAP headers (including WS-Addressing headers), custom SOAP message bodies, and custom SOAP faults.
- MIME attachments.
- Use of MTOM (Message Transmission Optimization Mechanism).
- Authentication (user login) between a web client and a web service, as well as key parts of the WS-Security standard.
- Policies, which can control how the service or client do the following:
  - Specify the WS-Security header elements to use or to require.
  - Specify the use of MTOM.
  - Specify the use of WS-Addressing.

See *Securing Web Services*.

- Options to fine-tune the generated WSDL document to meet most format requirements.
- Use of transport other than HTTP between the web client and web service.

For details on supported standards, see the next section.

1.4 SOAP Standards

This section lists the basic standards and WSDL support details for InterSystems IRIS web services and web clients.

Also see XML Standards and SOAP Security Standards.

1.4.1 Basic Standards

InterSystems IRIS web services and clients support the following basic standards:

- SOAP 1.1 (see https://www.w3.org/TR/2000/NOTE-SOAP-20000508/), including encoded format.
- SOAP 1.2, including encoded format as specified in section 3 SOAP Version 1.2 Part 2: Adjuncts (https://www.w3.org/TR/soap12-part2/).
- MTOM (Message Transmission Optimization Mechanism) 1.0 (https://www.w3.org/TR/soap12-mtom/).
- WSDL 1.1. InterSystems IRIS web services produce WSDL documents that comply with the Basic Profile 1.0 established by the WS-I (Web Services Interoperability Organization). However, InterSystems IRIS web *clients* do work for more general WSDL documents.

  See WSDL Support in InterSystems IRIS.
- UDDI version 1.0 with client access only (no repository provided). See https://uddi.xml.org/
- Attachments handled as a multipart/related MIME message according to the SOAP with Attachments specification (https://www.w3.org/TR/SOAP-attachments).

  SOAP with Attachments is supported for SOAP 1.2 and SOAP 1.1.
- Transport via HTTP 1.1 or HTTP 1.0.
- Output from the web client is supported only in UTF-8.

1.4.2 WSDL Support in InterSystems IRIS

InterSystems IRIS does not support all possible WSDL documents. More flexibility is provided on the client side, because it is frequently necessary to create web clients that work with specific WSDLs that cannot be changed. This section discusses the details of the support.

1.4.2.1 Generated WSDL Documents

The WSDL documents generated by InterSystems IRIS web services do not include headers. Also, the web services that you can create in InterSystems IRIS do not reflect all possible variations.

Note that the SOAP specifications do not require a web service to *generate* a WSDL at all.

1.4.2.2 Consuming WSDLs

InterSystems IRIS cannot process all possible WSDL documents. In particular:

- It does not support the `<fault>` element. That is, if you include a `<fault>` element within the `<operation>` element of the binding, the `<fault>` element is ignored.
- For the response messages, one of the following must be true:
  - Each response message must be in the same namespace as the corresponding request message.
  - The response messages must all be in the same namespace as each other (which can be different from the namespaces used by request messages).
- InterSystems IRIS does not process headers of the WSDL.

InterSystems IRIS does allow the use of the MIME binding in a WSDL (https://www.w3.org/TR/wsdl#_Toc492291084). The MIME parts are ignored and the remainder of the WSDL is processed. When you create a web service or client based on a WSDL that contains MIME binding, you must add explicit ObjectScript code to support the MIME attachments; this task is beyond the scope of this documentation.

1.5 Key Points about the SAX Parser

The InterSystems IRIS SAX parser is used whenever InterSystems IRIS receives a SOAP message. It is useful to know its default behavior. Among other tasks, the parser does the following:

- It verifies whether the XML document is well-formed.
- It attempts to validate the document, using the given schema or DTD.

  Here it is useful to remember that a schema can contain `<import>` and `<include>` elements that refer to other schemas. For example:

  ```
  <xsd:import namespace="target-namespace-of-the-importing-schema"
                  schemaLocation="uri-of-the-schema"/>

  <xsd:include schemaLocation="uri-of-the-schema"/>
  ```

  The validation fails unless these other schemas are available to the parser. Especially with WSDL documents, it is sometimes necessary to download all the schemas and edit the primary schema to use the corrected locations.
- It attempts to resolve all entities, including all external entities. (Other XML parsers do this as well.) This process can be time-consuming, depending on their locations. In particular, Xerces uses a network accessor to resolve some URLs, and the implementation uses blocking I/O. Consequently, there is no timeout and network fetches can hang in error conditions, which have been rare in practice.

  If needed, you can create custom entity resolvers; see Customizing How the SAX Parser Is Used.

2

Creating Web Services

This topic describes the basics of how to create a web service in InterSystems IRIS® data platform.

Click here for a summary of the URLs related to your web service.

2.1 Overview of InterSystems IRIS Web Services

To create a web service in InterSystems IRIS, you create a class that extends %SOAP.WebService, which provides all the functionality required to make one or more methods callable via the SOAP protocol. In addition, this class automates the management of SOAP-related bookkeeping, such as maintaining a WSDL document that describes a service.

2.2 Basic Requirements

To create a web service in InterSystems IRIS, create and compile an InterSystems IRIS class that meets the following basic requirements:

- The class must extend %SOAP.WebService.
- The class must define the *SERVICENAME* parameter. InterSystems IRIS does not compile the class unless it defines this parameter.
- This class should define methods or class queries that are marked with the WebMethod keyword.

  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 (as described in later topics) 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 any web methods, make sure that each value in the method signature has an XML projection. For example, suppose that your method had the following signature:

  ```
  Method MyWebMethod(myarg as ClassA) as ClassB [ WebMethod ]
  ```

  In this case, both `ClassA` and `ClassB` must have an XML representation. In most cases, this means that their superclass lists must include %XML.Adaptor; see *Projecting Objects to XML*. InterSystems IRIS SOAP support provides special handling for collections and streams, as noted after this list.

The web method can specify the ByRef and Output keywords in the same way that ordinary methods do. (For information on these keywords, see Methods.)

- Consider the values that are likely to be carried within these arguments and return values. XML does not permit non-printing characters, specifically characters below ASCII 32 (except for carriage returns, line feeds, and tabs, which are permitted in XML).

  If you need to include any disallowed nonprinting character, specify the type as %Binary, %xsd.base64Binary (which is equivalent), or a subclass. This value is automatically converted to base–64 encoding on export to XML (or automatically converted from base–64 encoding on import).

- Do not rely on the method signature to specify the default value for an argument. If you do, the default value is ignored and a null string is used instead. For example, consider the following method:

  ```
  Method TestDefaults(val As %String = "Default String") As %String [ WebMethod ]
  ```

  When you invoke this method as a web method, if you do not supply an argument, a null string is used, and the value `"Default String"` is ignored.

  Instead, at the start of the method implementation, test for a value and use the desired default if applicable. One technique is as follows:

  ObjectScript

  ```
  if arg="" {
    set arg="Default String"
  }
  ```

  You can *indicate* the default value in the method signature as usual, but this is purely for informational purposes and does not affect the SOAP messages.

- For any required arguments in a web method, specify the *REQUIRED* property parameter within the method signature. For example:

  ```
  Method MyWebMethod(myarg as ClassA(REQUIRED=1)) as ClassB [ WebMethod ]
  ```

By default, any inherited methods are treated as ordinary methods, even if a superclass marks them as web methods (but see Subclassing an Existing InterSystems IRIS Web Services).

2.2.1 Input and Output Objects That Do Not Need %XML.Adaptor

In most cases, when you use an object as input or output to a web method, that object must extend %XML.Adaptor. The exceptions are as follows:

- If the object is %ListOfDataTypes, %ListOfObjects, %ArrayOfDataTypes, %ArrayOfObjects, or a subclass, the InterSystems IRIS SOAP support implicitly treats the object as if it included %XML.Adaptor. You do not need to subclass these classes. However:
  - You must specify *ELEMENTTYPE* within the method signature, as follows:

    Class Member

    ```
    Method MyMethod() As %ListOfObjects(ELEMENTTYPE="MyApp.MyXMLType") [WebMethod]
    {
       //method implementation
    }
    ```

    Or, in the case of an input argument:

Class Member

```
Method MyMethod(input As %ListOfObjects(ELEMENTTYPE="MyApp.MyXMLType")) [WebMethod]
{
   //method implementation
}
```

  - If the class that you name in *ELEMENTTYPE* is an object class, it must inherit from %XML.Adaptor.

- If the object is one of the stream classes, the InterSystems IRIS SOAP support implicitly treats the object as if it included %XML.Adaptor. You do not need to subclass the stream class.

  If it is a character stream, the InterSystems IRIS SOAP tools assume that the type is string. If it is a binary stream, the tools treat it as base-64–encoded data. Thus it is not necessary to supply type information.

2.2.2 Using Result Sets as Input or Output

You can use result sets as input or output, but your approach depends on the intended web clients.

- If both the web service and client are based on InterSystems IRIS or one is based on .NET, you can use the specialized result set class, %XML.DataSet, which is discussed in Using Datasets in SOAP Messages. Or you can use a class query as a web method. The XML representation is automatically the same as for %XML.DataSet.
- To output results of a query so that a Java-based web client can work with it, use a %ListOfObjects subclass.

2.3 Simple Example

This section shows an example web service, as well as an example of a request message that it can recognize and the corresponding response message.

First, the web service is as follows:

Class Definition

```
/// MyApp.StockService
Class MyApp.StockService Extends %SOAP.WebService
{

/// Name of the WebService.
Parameter SERVICENAME = "StockService";

/// TODO: change this to actual SOAP namespace.
/// SOAP Namespace for the WebService
Parameter NAMESPACE = "https://tempuri.org";

/// Namespaces of referenced classes will be used in the WSDL.
Parameter USECLASSNAMESPACES = 1;

/// This method returns tomorrow's price for the requested stock
Method Forecast(StockName As %String) As %Integer [WebMethod]
{
    // apply patented, nonlinear, heuristic to find new price
    Set price = $Random(1000)
    Quit price
}
}
```

When you invoke this method from a web client, the client sends a SOAP message to the web service. This SOAP message might look like the following (with line breaks and spaces added here for readability):

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
    <Forecast xmlns="https://tempuri.org">
     <StockName xsi:type="s:string">GZP</StockName>
    </Forecast>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Note that the message body (the `<SOAP-ENV:Body>` element) includes an element named `<Forecast>`, which is the name of the method that the client is invoking. The `<Forecast>` includes one element, `<StockName>`, whose name is based on the argument name of the web method that we are invoking. This element contains the actual value of this argument.

The web service performs the requested action and then sends a SOAP message in reply. The response message might look like the following:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
    <ForecastResponse xmlns="https://www.myapp.org">
      <ForecastResult>799</ForecastResult>
    </ForecastResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

These examples do not include the HTTP headers that precede the SOAP message itself.

2.4 Creating a Web Service

You can create web services in any of the following ways:

- By creating a new class or editing an existing class to follow the requirements described earlier in this topic
- By subclassing one or more InterSystems IRIS web services

You can also generate stub classes programmatically from a WSDL, using the same method via which you can generate web clients.

2.4.1 Subclassing an Existing InterSystems IRIS Web Service

You can create a web service by creating a subclass of an existing InterSystems IRIS web service class and then adding the *SOAPMETHODINHERITANCE* parameter to your class as follows:

Class Member

```
PARAMETER SOAPMETHODINHERITANCE = 1;
```

The default for this parameter is 0. If this parameter is 0, your class does not inherit the web methods as web methods. That is, the methods are available as ordinary methods but cannot be accessed as web methods within the web service defined by the subclass.

If you set this parameter to 1, then your class can use web methods defined in any superclasses that are web services.

2.5 Specifying Parameters of the Web Service

Make sure that your web service class uses appropriate values for the following parameters.

Note: If you generate a web service from an existing WSDL, do not modify any of these parameters.

SERVICENAME

Name of the web service. This name must start with a letter and must contain only alphanumeric characters.

InterSystems IRIS does not compile the class unless the class defines this parameter.

NAMESPACE

URI that defines the target namespace for your web service, so that your service, and its contents, do not conflict with another service. This is initially set to `"https://tempuri.org"` which is a temporary URI often used by SOAP developers during development.

If you do not specify this parameter, the target namespace is `"https://tempuri.org"`.

For an InterSystems IRIS web service, there is no way to put request messages in different namespaces. An InterSystems IRIS web client, however, does not have this limitation; see Namespaces for the Messages.

RESPONSENAMESPACE

URI that defines the namespace for the response messages. By default, this is equal to the namespace given by the *NAMESPACE* parameter.

For an InterSystems IRIS web service, there is no way to put response messages in different namespaces. An InterSystems IRIS web client, however, does not have this limitation; see Namespaces for the Messages.

TYPENAMESPACE

Namespace for the schema for the types defined by the web service. If you do not specify this parameter, the schema is in the target namespace of the web service (that is, either *NAMESPACE* or the default, which is `"https://tempuri.org"`).

For an InterSystems IRIS web service, there is no way to put the request message types in different namespaces. An InterSystems IRIS web client does not have this limitation; see Namespaces for Types.

RESPONSETYPENAMESPACE

URI that defines the namespace for types used by the response messages. By default, this is equal to the namespace given by the *TYPENAMESPACE* parameter.

This parameter is used only if SoapBindingStyle equals `"document"` (the default).

For either an InterSystems IRIS web service or an InterSystems IRIS web client, the types for the response messages must all be in the same namespace.

SOAPVERSION

Specifies the SOAP version or versions advertised in the WSDL of the web service. Use one of the following values:

- `""` — Use this value for SOAP 1.1 or 1.2.
- `"1.1"` — Use this value for SOAP 1.1. This is the default.

- `"1.2"` — Use this value for SOAP 1.2.

When the web service receives a SOAP request, the SoapVersion property of the web service is updated to equal the SOAP version of that request.

See also Restricting the SOAP Versions Handled by a Web Service.

For details on how these values affect the WSDL, see Details of the Generated WSDLs.

2.6 About the Catalog and Test Pages

When you compile a web service class, the class compiler generates a convenient catalog page that you can use to examine the web service. This catalog page provides a link to a simple, limited test page (also generated). These pages are disabled by default. Enable them *only* in a test environment.

2.6.1 Access to the Catalog and Test Pages

If there is no web application for the namespace you are using, you cannot access the catalog and test pages; see InterSystems IRIS Web Service as Part of a Web Application. Also, by default, these pages are inaccessible. To enable access to them, open the Terminal, go to the %SYS namespace, and enter the following commands:

```
set ^SYS("Security","CSP","AllowClass",webapplicationname,"%SOAP.WebServiceInfo")=1
set ^SYS("Security","CSP","AllowClass",webapplicationname,"%SOAP.WebServiceInvoke")=1
```

Where *webapplicationname* is the web application name with a trailing slash, for example, `"/csp/mynamespace/"`.

You can use these pages only if you are logged in as a user who has USE permission for the %Development resource.

2.6.2 Displaying the Catalog and Test Pages

For the catalog page, the URL has the following form, using the <baseURL> for your instance:

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

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*.) For example:

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

2.6.3 Notes on These Pages

The catalog page displays the class name, namespace, and service name, as well as the comments for the class and web methods. The **Service Description** link displays the generated WSDL; for information, see Viewing the WSDL. The page then lists the web methods, with links (if you have the suitable permissions). The link for a given method displays a test page where you can test that method in a limited fashion.

Notes about this test page:

- It does not enable you to see the SOAP request.
- It does not test the full SOAP pathway. This means, for example, it does not write to the SOAP log.
- It accepts only simple, literal inputs, so you cannot use it to call methods whose arguments are objects, collections, or datasets.

This documentation does not discuss this page further. To test your web service more fully, generate and use a web client as described in Creating Web Clients.

2.7 Viewing the WSDL

When you use %SOAP.WebService to define a web service, the system creates and publishes a WSDL document that describes this web service. Whenever you modify and recompile the web service, the system automatically updates the WSDL correspondingly. This section discusses the following:

- Viewing the WSDL and the URL at which the WSDL is published
- Methods you can use to generate the WSDL as a static document

Also see WSDL Support in InterSystems IRIS.

Important: By definition, a web service and its web clients are required to comply to a common interface, regardless of their respective implementations (and regardless of any underlying changes in technology). A WSDL is a standards-compliant *description* of this interface. It is important to note the following:

- In practice, a single SOAP interface can often be correctly described by multiple, slightly different WSDL documents.

  Accordingly, the WSDL generated by InterSystems IRIS may have a slightly different form depending on the version of InterSystems IRIS. It is beyond the scope of this documentation to describe any such differences. InterSystems can commit only to the interoperability of web services and their respective clients, as required in the W3C specifications.

- The W3C specifications *do not require* that either a web service or a web client be able to *generate* a WSDL to describe the interface with which it complies.

  The system generates the WSDL document and serves it at a specific URL, for convenience. However, if the containing web application requires password authentication or requires an SSL connection, you may find it impractical to access the WSDL in this way. In such cases, you should download the WSDL to a file and use the file instead. Also, as noted previously, the generated WSDL does not contain any information about SOAP headers added at runtime. If you need a WSDL document to contain information about SOAP headers added at runtime, you should download the WSDL to a file, modify the file as appropriate, and then use that file.

2.7.1 Viewing the WSDL

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*.)

Note: Any percent characters (%) in your class name are replaced by underscore characters (_) in this URL.

For example:

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

The browser displays the WSDL document, for example:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:SOAP-
  ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:s="http://www.w3.org/2001/XMLSchema" xmlns:s0="http://tempuri.org"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  targetNamespace="http://tempuri.org">
 - <types>
  - <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org">
   - <s:element name="Forecast">
    - <s:complexType>
     - <s:sequence>
        <s:element minOccurs="0" name="StockName" type="s:string" />
       </s:sequence>
      </s:complexType>
```

Important: Not all browsers display the schema correctly. You might need to view the page source to see the actual schema. For example, in Firefox, right-click and then select **View Source**.

2.7.2 Generating the WSDL

You can also generate the WSDL as a static document. The %SOAP.WebService class provides a method you can use for this:

FileWSDL()

```
ClassMethod FileWSDL(fileName As %String, includeInternalMethods As %Boolean = 1) As %Status
```

Where *fileName* is the name of the file, and *includeInternalMethods* specifies whether the generated WSDL includes any web methods that are marked as Internal.

2.7.3 Suppressing Internal Web Methods from the WSDL

If the web service has web methods that are marked as Internal, by default the WSDL includes these web methods. You can prevent these methods from being included in the WSDL. To do so, do either of the following:

- Use the **FileWSDL()** method of the web service to generate the WSDL; see the previous section. This method provides an argument that controls whether the WSDL includes internal web methods.
- Specify the *SOAPINTERNALWSDL* class parameter as 0 in the web service class. (The default for this class parameter is 1.)

3

SOAP Message Variations

This topic discusses the primary variations for SOAP messages and how they can be generated by InterSystems IRIS® data platform web services and clients.

For an InterSystems IRIS web service or client, several keywords and one parameter specify the message variation used by each web method. If you create a web service manually, the default values for these items are typically appropriate. If you create a web service or client programmatically, InterSystems IRIS sets the values as required by the WSDL. On some occasions, however, you may find it necessary to choose a specific message variation.

3.1 Overview

A SOAP message is in one of the following modes, determined formally by the WSDL:

- Document/literal — This is the default message mode in InterSystems IRIS web services and is the most commonly used mode.

  This message mode uses document-style binding and literal encoding format; bindings and encoding formats are discussed briefly in subsections.
- RPC/encoded — This is the second most common mode.
- RPC/literal — This mode is widely used by IBM.
- Document/encoded — This mode is extremely rare and is not recommended. It is also not in compliance with the WS-I Basic Profile 1.0.

Informally, document/literal messages can have an additional variation: they can be either *wrapped* (the default in InterSystems IRIS) or *unwrapped*. In a wrapped message, the message contains a single part that contains subparts. This is relevant in the case of methods that take multiple arguments. In a wrapped message, the arguments are subparts within this message. In an unwrapped message, the message consists of multiple parts, one per argument.

RPC messages can have multiple parts.

3.1.1 Binding Style

Each web method has a binding style for the inputs and outputs of the web method. A binding style is either document or RPC. The binding style determines how to translate a WSDL binding to a SOAP message. It also controls the format of the body of the SOAP messages.

3.1.2 Encoding Format

Each web method also has an encoding format, which is either literal or encoded (meaning SOAP-encoded). The encoding details are slightly different for SOAP 1.1 and SOAP 1.2. For details on the differences between literal format and SOAP-encoded format, see *Projecting Objects to XML*.

3.2 How Message Variation Is Determined

For an InterSystems IRIS web service or web client, the details of the service or client class control the message mode used by each web method. These details are as follows:

- The SoapBindingStyle class keyword and the SoapBindingStyle method keyword. The method keyword takes precedence.
- The SoapBodyUse class keyword and the SoapBodyUse method keyword. The method keyword takes precedence.
- The *ARGUMENTSTYLE* class parameter.

The following table summarizes how the message mode is determined for an InterSystems IRIS web method:

Message Mode	SoapBindingStyle	SoapBodyUse	ARGUMENTSTYLE
wrapped document/literal	`document` (default)	`literal` (default)	`wrapped` (default)
unwrapped document/literal	`document`	`literal`	`message`
rpc/encoded	`rpc`	`encoded`	*Ignored*
rpc/literal	`rpc`	`literal`	*Ignored*
document/encoded	`document`	`encoded`	*Ignored*

When you generate a web service or client class, InterSystems IRIS sets the values for these keywords and parameter as appropriate for the WSDL from which you started.

Important: If you modify the values, your web client or service may no longer work.

Also, for a web service that you create manually, the default values are usually suitable.

3.3 Examples of Message Variations

For reference, this section shows examples of the messages in the different modes (except for document/encoded, which is not recommended).

Also see <message> in Details of the Generated WSDLs.

3.3.1 Wrapped Document/Literal

This is the most common message style (and is the default message style for InterSystems IRIS web services).

XML

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
    <MyMethod xmlns="https://www.demoservice.org">
        <A>stringA</A>
        <B>stringB</B>
        <C>stringC</C>
    </MyMethod>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

3.3.2 Message/Unwrapped Document/Literal

This is a slight variation of the preceding style.

XML

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
      <A xmlns="https://www.demoservice.org">stringA</A>
      <B xmlns="https://www.demoservice.org">stringB</B>
      <C xmlns="https://www.demoservice.org">stringC</C>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

3.3.3 RPC/Encoded

This is the second most common style. The following shows an rpc/encoded message for SOAP 1.1:

XML

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'
                   xmlns:SOAP-ENC='https://schemas.xmlsoap.org/soap/encoding/'
                   xmlns:tns='https://www.demoservice.org'
                   xmlns:types='https://www.demoservice.org'>
  <SOAP-ENV:Body SOAP-ENV:encodingStyle='https://schemas.xmlsoap.org/soap/encoding/'>
    <types:MyMethod>
      <A>stringA</A>
      <B>stringB</B>
      <C>stringC</C>
    </types:MyMethod>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For SOAP 1.2, the rules for encoding are different, so the message is different:

XML

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://www.w3.org/2003/05/soap-envelope'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'
                   xmlns:SOAP-ENC='https://www.w3.org/2003/05/soap-encoding'
                   xmlns:tns='https://www.demoservice.org'
                   xmlns:types='https://www.demoservice.org'>
  <SOAP-ENV:Body>
    <types:MyMethod SOAP-ENV:encodingStyle="https://www.w3.org/2003/05/soap-encoding">
      <A>stringA</A>
      <B>stringB</B>
      <C>stringC</C>
    </types:MyMethod>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

3.3.4 RPC/Literal

The following shows an example of an rpc/literal message:

XML

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'
                   xmlns:tns='https://www.demoservice.org'>
  <SOAP-ENV:Body>
    <tns:MyMethod>
      <tns:A>stringA</tns:A>
      <tns:B>stringB</tns:B>
      <tns:C>stringC</tns:C>
    </tns:MyMethod>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4

Creating Web Clients

A web client is software that accesses a web service. A web client provides a set of proxy methods, each of which corresponds to a method of the web service. A proxy method uses the same signature as the web service method to which it corresponds, and it invokes the web service method when asked to do so. This topic describes how to create and use web clients in InterSystems IRIS® data platform.

For information on logging SOAP calls to your InterSystems IRIS web clients, see InterSystems IRIS SOAP Log.

Note: For an InterSystems IRIS web service, the automatically generated WSDL might not include information about the SOAP header elements:

- If you add SOAP headers manually by setting the HeadersOut property, be sure to follow the instructions in Specifying Supported Header Elements, in Adding and Using Custom Header Elements. If you do, the WSDL contains all the applicable information. Otherwise, it does not, and you must save the WSDL to a file and edit it manually as needed.
- If you add WS-Security header elements by setting the SecurityOut property (as described in *Securing Web Services*), the WSDL does not include all needed information. (This is because the WSDL is generated at compile time and the headers are added later, at runtime.) In this case, save the WSDL to a file and edit it manually as needed.

  For many reasons, it is simpler and easier to add WS-Security elements by using WS-Policy, as described in Creating and Using Policies. With WS-Policy, the generated WSDL includes all needed information.
- In other cases, the generated WSDL includes all needed information.

Note that the W3C specifications do not require a web service to provide a generated WSDL.

4.1 Generating the Client Classes Programmatically

You generate the client classes programmatically by using the %SOAP.WSDL.Reader class. To generate the client classes programmatically:

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

   Note that if the WSDL is at a location that uses SSL, %SOAP.WSDL.Reader will (by default) checks whether the certificate server name matches the DNS name used to connect to the server. If these names do not match, the connection

is not permitted. This default behavior prevents man in the middle attacks and is described in RFC 2818, section 3.1; also see RFC 2595, section 2.4.

To disable this check, set the *SSLCheckServerIdentity* property of the instance equal to 0.

3. If you need the ability to control the HTTP request in a way that is not directly supported by %SOAP.WSDL.Reader, do this:

   a. Create an instance of %Net.HttpRequest and set its properties as needed.

   b. For your instance of %SOAP.WSDL.Reader, set the *HttpRequest* property equal to the instance of %Net.HttpRequest that you just created.

   For example, you might do this if the server requires authentication. See Providing Login Credentials in Sending HTTP Requests.

4. Invoke the **Process()** method of your instance:

```
method Process(pLocationURL As %String, pPackage As %String = "") as %Status
```

   - *pLocationURL* 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.
   - *pPackage* is the name of the package in which to place the generated classes. If you do not specify a package, InterSystems IRIS uses the service name as the package name.

     Note: If this package is the same as an existing package, by default the tool overwrites any existing classes that have the same name. To prevent the tool from overwriting a class definition, add the following to that class definition:

     Class Member

```
Parameter XMLKEEPCLASS = 1;
```

The following shows an example Terminal session:

```
set r=##class(%SOAP.WSDL.Reader).%New()
GSOAP>set url="https://devsys/csp/mysamples/GSOP.AddComplexWS.CLS?WSDL=1"

GSOAP>d r.Process(url)
Compilation started on 11/09/2009 12:53:52 with qualifiers 'dk'
Compiling class AddComplex.ns2.ComplexNumber
Compiling routine AddComplex.ns2.ComplexNumber.1
Compilation finished successfully in 0.170s.

Compilation started on 11/09/2009 12:53:52 with qualifiers 'dk'
Compiling class AddComplex.AddComplexSoap
Compiling routine AddComplex.AddComplexSoap.1
Compiling class AddComplex.AddComplexSoap.Add
Compiling routine AddComplex.AddComplexSoap.Add.1
Compilation finished successfully in 0.363s.
```

The WSDL URL is part of a web application, which might be protected by password authentication. For details on using the WSDL in this case, to generate InterSystems IRIS web clients or third-party web clients, see Using a Password-Protected WSDL URL.

In all cases, it is also possible to retrieve the WSDL from a browser after supplying the required username and password, save it as a file, and use the file instead.

4.1.1 Properties to Control Class Generation and Compilation

%SOAP.WSDL.Reader includes the following properties which you can set in order to specify the types of classes to generate from the WSDL.

MakeClient

Specifies whether to generate client classes.

MakeService

Specifies whether to generate an InterSystems IRIS web service based on the WSDL.

CompileClasses

Specifies whether to compile classes after generating them.

If CompileClasses is 1, you can control the behavior of the compiler by specifying flags in the **CompileFlags** property. For more information, execute the following command:

ObjectScript

```
Do $System.OBJ.ShowFlags()
```

ClientPackage

Specifies the package name for the web client and any generated classes.

The default package name is the service name.

If you specify an existing package name, the tool overwrites any existing class that has the same name as a newly generated class by default.

MakeEnsembleClasses

Specifies whether to generate a business operation and related request and response message classes, which you can use in a production.

See *Creating Web Services and Web Clients in Productions*.

If MakeEnsembleClasses is 1, you must also specify the following propperties:

- OperationPackage — Package name for the business operation class.
- RequestPackage — Package name for the request message class.
- ResponsePackage — Package name for the response message class.

4.1.2 Properties to Control Class Details

%SOAP.WSDL.Reader includes the following properties to control details of the classes generated from the WSDL:

MakeNamespace

Specifies whether the generated classes should include the *NAMESPACE* class parameter set equal to the namespace of the web service.

- If the WSDL explicitly indicates the namespace to which a given type belongs, MakeNamespace should be 1. In this case, the generated type class includes the *NAMESPACE* class parameter set equal to that namespace.

- If the WSDL does not indicate the namespace for a given type, MakeNamespace can be either 1 or 0.

MakeMessageStyle

Specifies whether to use an unwrapped message format for the methods in the generated web client. This option affects only methods that have SoapBindingStyle equal to `"document"`.

If either of the following statements are true of the WSDL, specify MakeMessageStyle as 1:

- The `<message>` elements contain multiple parts.
- The types used by the response messages belong to multiple namespaces.

Alternatively, the code generation fails and displays an error message such as the following:

```
ERROR #6425: Element 'wsdl:binding:operation:msg:input' - message 'AddSoapOut'
Message Style must be used for document style message with 2 or more parts.
```

For more information, see Using InterSystems IRIS Web Client Classes.

NoArrayProperties

Specifies whether to generate array properties.

If this property is 1, the system does not generate array properties but instead generates another form. For more information, see Creation of Array Properties.

GenerateXMLNIL

Specifies whether to specify the *XMLNIL* property parameter for applicable properties in the generated classes.

This option applies to each property that corresponds to an XML element that is specified with `nillable="true"`. If this property is 1, the system adds `XMLNIL=1` to the property definition. Otherwise, it does not add this parameter.

For details on this property parameter, see Handling Empty Strings and Null Values.

GenerateXMLNILNOOBJECT

Specifies whether to specify the *XMLNILNOOBJECT* property parameter for applicable properties in the generated classes.

This option applies to each property that corresponds to an XML element that is specified with `nillable="true"`. If this property is 1, the system, the system adds `XMLNILNOOBJECT=1` to the property definition. Otherwise, it does not add this parameter. For details on this parameter, see Handling Empty Strings and Null Values.

NoSequence

Specifies whether to set the *XMLSEQUENCE* class parameter in the generated classes to `0`.

By default, the system sets this parameter to 1 in the generated classes, which ensures that the classes respect the order of elements as given in the schema in the WSDL. This value is useful when the schema has multiple elements of the same name within a given parent. For details, see Handling a Document with Multiple Tags with the Same Name.

IgnoreNull

Specifies whether the generated classes should include the *XMLIGNORENULL* class parameter set to 1.

If you select this option, the system adds `XMLIGNORENULL=1` to the class definitions, including the generated web client (or web service). Otherwise, it does not add this parameter.

For details on this class parameter, see Handling Empty Strings and Null Values.

BinaryAsStream

Specifies whether to generate a property of type %Stream.GlobalBinary for each element of type `xsd:base64Binary`.

If this property is 1, the generated properties are of type %Stream.GlobalBinary. Alternatively, the properties are of type %xsd.base64Binary.

%SOAP.WSDL.Reader ignores any *attributes* of type `xsd:base64Binary`.

SecurityInParameter

Specifies the value of the *SECURITYIN* class parameter in the generated client class.

If you are using Web-Services security, use `REQUIRE` or `ALLOW`, depending on whether you want the client to require the elements or simply validate them. Otherwise, `IGNORE` or `IGNOREALL` is generally suitable. For details, see Validating WS-Security Headers.

The *SECURITYIN* parameter is ignored if there is a security policy in an associated (and compiled) configuration class. See *Securing Web Services*.

4.2 Using the Generated Web Client Classes

As noted in the previous section, after you generate an InterSystems IRIS web client class, you do not usually edit the generated class. Instead you write code that creates an instance of that web client and that provides provide client-side error handling. In this code, do the following:

1. Create an instance of the web client class.
2. Set its properties. Here you can control items such as the following:
   - Endpoint of the web client (the URL of the web service it uses). To control this, set the Location property, which overrides the *LOCATION* parameter of the web client class.
   - Settings that designate a proxy server.
   - Settings that control HTTP Basic authentication.

   See the next section as well as Fine-Tuning a Web Client in InterSystems IRIS.
3. Invoke the methods of the web client as needed.
4. Perform client-side error handling. See SOAP Fault Handling.
5. Optionally examine the HTTP response received by the web client, as described later in this topic.

The following shows a simple example, from a session in the Terminal:

```
GSOAP>set client=##class(Proxies.CustomerLookupServiceSoap).%New()

GSOAP>set resp=client.GetCustomerInfo("137")

GSOAP>w resp

11@Proxies.CustomerResponse

GSOAP>w resp.Name
Smith,Maria
```

4.2.1 Example 1: Using the Client That Uses Wrapped Messages

In this example, we create a wrapper class for a web client that uses wrapped messages. To use the example `GSOAPClient.AddComplex.AddComplexSoap` shown previously, we could create a class like the following:

Class Definition

```
Class GSOAPClient.AddComplex.UseClient Extends %RegisteredObject
{

ClassMethod Add(arg1 As ComplexNumber, arg2 As ComplexNumber) As ComplexNumber
{
    Set client=##class(AddComplexSoap).%New()
    //uncomment the following to enable tracing
    //set client.Location="https://devsys:8080/csp/mysamples/GSOP.AddComplexWS.cls"
    Set ans=client.Add(arg1,arg2)
    Quit ans
}

}
```

The client application would invoke this method in order to execute the web method.

4.2.2 Example 2: Using the Client That Uses Unwrapped Messages

In this example, we create a wrapper class for a web client that uses unwrapped messages. To use the example `GSOAPClient.AddComplex.AddComplexSoap` shown previously, we could create a class like the following:

Class Definition

```
Class GSOAPClient.AddComplexUnwrapped.UseClient Extends %RegisteredObject
{

ClassMethod Add(arg1 As GSOAPClient.AddComplexUnwrapped.s0.ComplexNumber,
arg2 As GSOAPClient.AddComplexUnwrapped.s0.ComplexNumber)
As GSOAPClient.AddComplexUnwrapped.s0.ComplexNumber
{
    //create the Add message
    Set addmessage=##class(GSOAPClient.AddComplexUnwrapped.s0.Add).%New()
    Set addmessage.a = arg1
    Set addmessage.b = arg2

    Set client=##class(AddComplexSoap).%New()

    //send the Add message to client and get response
    Set addresponse=client.Add(addmessage)

    //get the result from the response message
    Set ans=addresponse.AddResult

    Quit ans

}

}
```

The method has the signature that would typically be expected; that is, it accepts two complex numbers and returns a complex number. The method creates the message that the web client expects. The elements of this message are the two complex numbers.

As you can see, when the web client uses unwrapped messages, it is necessary to write slightly more code to convert arguments in a user-friendly form into the message used by the web client.

4.3 Modifying the Generated Client Classes

After you generate an InterSystems IRIS web client class, you do not usually edit the class. Instead you write code that creates an instance of the web client and that provides provide client-side error handling. This section documents the notable exceptions where you do modify the generated client class.

Also see Additional Notes on Web Methods in the Generated Class and Fine-Tuning a Web Client in InterSystems IRIS.

Note: Do not create a subclass of the generated web client class. The compiler will not generate the supporting classes that it would need in order to run properly, and your subclass would not be usable.

4.3.1 Adjusting the Generated Classes for Extremely Long Strings

In rare cases, you might need to edit the generated client class to accommodate extremely long strings or binary values — values whose lengths exceed the string length limit.

When generating a web client from a WSDL, InterSystems IRIS assumes that any string-type input or output can be represented in InterSystems IRIS as %String, which is not always true. In rare cases, a string might exceed the string length limit. Similarly, InterSystems IRIS assumes that any input or output with XML type base64Binary can be represented in InterSystems IRIS as %xsd.base64Binary), which is not always true, due to the same string length limitation. In neither case does the WSDL contain any information to indicate that this input or output could exceed the string length limit.

When your web client encounters a string or a binary value that is too long, it throws one of the following errors:

- A <MAXSTRING> error
- A datatype validation error:

  ```
  ERROR #6232: Datatype validation failed for tag your_method_name ...
  ```

  (This error, of course, can also be caused by a datatype mismatch.)

The problem, however, is easy to correct: adjust the method signature in your generated web client class (specifically the class that inherits from %SOAP.WebClient) to use an appropriate stream class:

- Use %GlobalCharacterStream instead of %String.
- Use %GlobalBinaryStream instead of %xsd.base64Binary.

For example, consider a web service (`MyGiantStringService`) that has one method (`WriteIt`), which takes no arguments and returns a very long string. The web client class originally looks something like this:

Class Definition

```
Class GetGiantString.MyServiceSoap Extends %SOAP.WebClient
{

Method WriteIt() As %String
[Final,SoapBindingStyle=document,SoapBodyUse=literal,WebMethod]
{
 Quit ..WebMethod("WriteIt").Invoke($this,"https://tempuri.org/MyApp.MyGiantStringService.WriteIt")
}

}
```

In this case, there is only one adjustment to make. Change the return type of `WriteIt` as follows:

Class Member

```
Method WriteIt() As %GlobalCharacterStream
[Final,SoapBindingStyle=document,SoapBodyUse=literal,WebMethod]
{
 Quit ..WebMethod("WriteIt").Invoke($this,"https://tempuri.org/MyApp.MyGiantStringService.WriteIt")
}
```

When you compile this class, the system automatically regenerates the associated classes as needed.

You may also need to adjust property types within any generated type classes. For example, suppose the web service uses an element called `<Container>`, which includes an element `<ContainerPart>` of type string. When you generate the InterSystems IRIS web client classes, the system creates a `Container` class with a `ContainerPart` property of type %String. If the web service sends a string longer than the string length limit in the `<ContainerPart>` element, your web client throws an error. To avoid this error, change the type of the `ContainerPart` property to %GlobalCharacterStream.

4.3.2 Other Adjustments

If the WSDL does not specify the location of the web service, the generated web client does not specify the *LOCATION* parameter. This is a rare scenario. In this scenario, edit the web client class to include the *LOCATION* parameter. For example:

Class Member

```
Parameter LOCATION = "https://devsys/csp/mysamples/GSOP.AddComplexWS.cls";
```

Or specify the Location property of your web client instance as shown later in this topic.

You might need to adjust other parameters of the web client class to make other changes. See Fine-Tuning a Web Client in InterSystems IRIS for details.

4.4 Adjusting Properties of a Web Client Instance

When you use an instance of your web client classes, you can specify properties of that instance to control its behavior. This section discusses the properties that are most commonly set, as well as their default values.

4.4.1 Changing the Endpoint for the Web Client

When you generate web client, the class automatically specifies the *LOCATION* parameter as the URL of the web service with which it communicates.

To override this, set the Location property of your web client instance. If Location is null, then the *LOCATION* parameter is used.

A common usage is to set the Location property to use a different port, in order to enable tracing. For example, suppose that in the generated web client class, the endpoint is defined as follows:

Class Member

```
Parameter LOCATION = "https://devsys/csp/mysamples/GSOP.AddComplexWS.cls";
```

When you use this client, you can include the following line:

ObjectScript

```
   Set client.Location="https://devsys:8080/mysamples/mysamples/GSOP.AddComplexWS.cls"
```

Note: If the WSDL does not specify the location of the web service, the generated web client does not specify the *LOCATION* parameter. This is a rare scenario. In this scenario, either edit the web client class to include the *LOCATION* parameter or specify the Location property of your web client instance as shown here.

4.4.2 Configuring the Client to Use SSL

If the endpoint for a web client has HTTPS protocol, the web client must be configured to use SSL. Specifically:

- If you have not already done so, use the Management Portal to create an SSL/TLS configuration that contains the details of the needed SSL connection. For information, see About Configurations in the *TLS Guide*.
- Set the SSLConfiguration property of the web client equal to that SSL/TLS configuration name.

Note that if the client is connecting via a proxy server, you must also set the HttpProxySSLConnect property equal to 1 in the web client. For information on configuring an InterSystems IRIS web client to use a proxy server, see Fine-Tuning the Web Client.

4.4.3 Specifying the SOAP Version

The generated web client automatically specifies the SOAP version to use in request messages, based on the SOAP version in the WSDL of the web service. Specifically it sets the *SOAPVERSION* parameter.

To override this, set the SoapVersion property of your web client instance. Use one of the following values:

- `""` — The client sends SOAP 1.1 messages.
- `"1.1"` — The client sends SOAP 1.1 messages.
- `"1.2"` — The client sends SOAP 1.2 messages.

If SoapVersion is null, then the *SOAPVERSION* parameter is used.

4.4.4 Other Adjustments

You might need to set other properties of the web client instance to make other changes. See Fine-Tuning a Web Client in InterSystems IRIS for details.

4.5 Using the HTTP Response

By default, when you invoke a web client method, you do so via HTTP. The HTTP response is then available as the HttpResponse property of the web client instance. This property is an instance of %Net.HttpResponse, which in turn has properties like the following:

- Headers contains the headers of the HTTP response.
- Data is an InterSystems IRIS multidimensional array that contains any data in the HTTP response.
- StatusCode, StatusLine, and ReasonPhrase provide status information.

For details, see *Using Internet Utilities* or see the class documentation for %Net.HttpResponse.

5

SOAP Fault Handling

This topic describes how to handle faults within a web service and within a web client.

Also see Using Try-Catch.

Note that the *SOAPPREFIX* parameter affects the prefix used in any SOAP faults; see Specifying the SOAP Envelope Prefix.

5.1 Default Fault Handling in a Web Service

By default, when your InterSystems IRIS® data platform web service encounters an error, it returns a standard SOAP message containing a fault. The following shows an example (for SOAP 1.1). The SOAP envelope is omitted in this example:

```
<SOAP-ENV:Body>
 <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server</faultcode>
  <faultstring>Server Application Error</faultstring>
  <detail>
     <error xmlns='https://www.myapp.org' >
       <text>ERROR #5002: ObjectScript error: <DIVIDE>zDivide^FaultEx.Service.1</text>
     </error>
  </detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

5.2 Returning Custom SOAP Faults in an InterSystems IRIS Web Service

To create and return a custom SOAP fault, do the following within the appropriate area of your code that traps errors:

1. Create a fault object that contains the appropriate information. To do so, call one of the following methods of your web service: **MakeFault()**, **MakeFault12()**, **MakeSecurityFault()**, or **MakeStatusFault()**. These are discussed in the following subsection.

   Or create a fault object manually, as described in later in this topic.

2. Call the **ReturnFault()** method of the web service, passing the fault object as an argument. Note that **ReturnFault())** does not return to its caller; it just sends the fault and terminates processing of the web method.

The following shows an example:

Class Member

```
Method Divide(arg1 As %Numeric, arg2 As %Numeric) As %Numeric [ WebMethod ]
{
  Try {
    Set ans=arg1 / arg2
    } Catch {

        //<detail> element must contain element(s) or whitespace
        //specify this element by passing valid XML as string argument to MakeFault()
        set mydetail="<mymessage>Division error detail</mymessage>"

        set fault=..MakeFault($$$FAULTServer,"Division error",mydetail)

        // ReturnFault must be called to send the fault to the client.
        // ReturnFault will not return here.
        Do ..ReturnFault(fault)
      }
  Quit ans
}
```

5.2.1 Methods to Create Faults

MakeFault()

```
classmethod MakeFault(pFaultCode As %String,
                      pFaultString As %String,
                      pDetail As %String = "",
                      pFaultActor As %String = "") as %SOAP.Fault
```

Returns a fault object suitable for SOAP 1.1. Here:

- *pFaultCode* is used within the `<faultcode>` element of the SOAP fault. Set this property equal to one of the SOAP 1.1 macros listed in Macros for SOAP Fault Codes.
- *pFaultString* is used within the `<faultstring>` element of the SOAP fault. Specify a string that indicates the reason for the fault, as intended for users to see.
- *pDetail* is used within the `<detail>` element of the SOAP fault. Use this to specify information about the cause of the fault.

  If specified, this argument should be a string containing valid XML that can be used within the `<detail>` element. InterSystems IRIS does not verify that the string you provide is valid; it is the responsibility of your application to check this.
- *pFaultActor* specifies the URI of the SOAP node on the SOAP message path that caused the fault to happen.

  This is useful if the SOAP message travels through several nodes in the SOAP message path, and the client needs to know which node caused the error. It is beyond the scope of this documentation to discuss this advanced topic.

MakeFault12()

```
classmethod MakeFault12(pFaultCode As %String,
            pFaultString As %String,
            pDetail As %String = "",
            pFaultActor As %String = "") as %SOAP.Fault
```

Returns a fault object suitable for SOAP 1.2. Use this method only the SoapVersion property of the web service is `"1.2"`. For a discussion of how InterSystems IRIS handles the SOAP versions of request messages, see Specifying Parameters of the Web Service.

For details on the arguments, see **MakeFault()**.

MakeSecurityFault()

```
classmethod MakeSecurityFault(pFaultCode As %String,
                              securityNamespace As %String) as %SOAP.Fault
```

Returns a fault object appropriate for a security failure. Specify *FaultCode* as one of the following: `"FailedAuthentication"`, `"FailedCheck"`, `"InvalidSecurity"`, `"InvalidSecurityToken"`, `"SecurityTokenUnavailable"`, `"UnsupportedAlgorithm"`, or `"UnsupportedSecurityToken"`.

The namespace for this security fault is found in the SecurityNamespace property.

MakeStatusFault()

```
classmethod MakeStatusFault(pFaultCode As %String,
                            pFaultString As %String,
                            pStatus As %Status = "",
                            pFaultActor As %String = "") as %SOAP.Fault
```

Returns a fault object based on a value in a %Status object.

pStatus is the %Status object to use.

For details on the other arguments, see **MakeFault()**.

5.2.2 Macros for SOAP Fault Codes

The SOAP include file (%soap.inc) defines macros for some of the standard SOAP fault codes; these are listed in the following table. You can use these macros to specify SOAP fault codes. The table notes the version or versions of SOAP to which each macro applies.

Table 5–1: ObjectScript Macros for SOAP Fault Codes

Macro	SOAP Version(s)	When to Use This Macro
`$$$FAULTVersionMismatch`	1.1 and 1.2	When the web service receives a SOAP message that contained an invalid element information item instead of the expected envelope element information item. A mismatch occurs if either the namespace or the local name do not match.
`$$$FAULTMustUnderstand`	1.1 and 1.2	When the web service receives a SOAP message that contained an unexpected element that was marked with `mustUnderstand="true"`
`$$$FAULTServer`	1.1	When other server-side errors occur.
`$$$FAULTClient`	1.1	When the client made an incomplete or incorrect request.
`$$$FAULTDataEncodingUnknown`	1.2	When the arguments are encoded in a data encoding unknown to the receiver.
`$$$FAULTSender`	1.2	When the sender made an incomplete, incorrect, or unsupported request.
`$$$FAULTReceiver`	1.2	When the receiver cannot handle the message because of some temporary condition, for example, when it is out of memory.

5.3 Creating a Fault Object Manually

If you need more control than is given by the steps in the previous section, you can create and return a custom SOAP fault as follows:

1. Create a fault object manually.

   To do so, create an instance of %SOAP.Fault (for SOAP 1.1) or %SOAP.Fault12 (for SOAP 1.2) and then set its properties, as described in the following sections.

   Note: You can use %SOAP.Fault in all cases. If a web service receives a SOAP 1.2 request and needs to return a fault, the web service automatically converts the fault to SOAP 1.2 format.

2. Call the **ReturnFault()** method of the web service, passing the fault object as an argument. Note that **ReturnFault())** does not return to its caller; it just sends the fault and terminates processing of the web method.

5.3.1 SOAP 1.1 Faults

This section provides information on %SOAP.Fault, which represents SOAP 1.1 faults. This section includes the following:

- Example SOAP 1.1 fault
- Information on for %SOAP.Fault, which represents SOAP 1.1 faults

5.3.1.1 Example SOAP Fault

For reference, here is an example of a SOAP 1.1 fault, including the SOAP envelope:

XML

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema'
xmlns:flt="https://myfault.org" >
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
      <faultstring>Division error</faultstring>
      <detail><mymessage>Division error detail</mymessage></detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5.3.1.2 %SOAP.Fault Properties

InterSystems IRIS represents a SOAP 1.1 fault as an instance of %SOAP.Fault, which has the following properties:

detail

Used within the <detail> element of the SOAP fault. Use this to specify information about the cause of the fault.

If specified, this argument should be a string containing valid XML that can be used within the <detail> element. InterSystems IRIS does not verify that the string you provide is valid; it is the responsibility of your application to check this.

faultcode

Used within the `<faultcode>` element of the SOAP fault. Set this property equal to one of the SOAP 1.1 macros listed in Macros for SOAP Fault Codes.

faultstring

Used within the `<faultstring>` element of the SOAP fault. Specify a string that indicates the reason for the fault, as intended for users to see.

faultactor

Specifies the URI of the SOAP node on the SOAP message path that caused the fault to happen.

This is useful if the SOAP message travels through several nodes in the SOAP message path, and the client needs to know which node caused the error. It is beyond the scope of this documentation to discuss this advanced topic.

faultPrefixDefinition

Specifies a namespace prefix declaration that is added to the envelope of the SOAP fault. Use a value of the following form:

```
xmlns:prefix="namespace"
```

Where *prefix* is the prefix and *namespace* is the namespace URI.

For example:

ObjectScript

```
set fault.faultPrefixDefinition = "xmlns:FLT=""https://myfault.com"""
```

The %SOAP.Fault class also provides the **AsString()** method, which returns the fault object as a string.

5.3.2 SOAP 1.2 Faults

This section provides information on %SOAP.Fault12 and related classes, which represent SOAP 1.2 faults. This section includes the following:

- Example SOAP 1.2 fault
- %SOAP.Fault12, the main class that represents SOAP 1.2 faults
- %SOAP.Fault12.Code, a helper class
- %SOAP.Fault12.Text, another helper class

5.3.2.1 Example SOAP Fault

For reference, here is an example of a SOAP 1.2 fault, including the SOAP envelope:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="https://www.w3.org/2003/05/soap-envelope"
xmlns:flt="https://myfault.org">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <SOAP-ENV:Code>
        <SOAP-ENV:Value>SOAP-ENV:Receiver</SOAP-ENV:Value>
      </SOAP-ENV:Code>
      <SOAP-ENV:Reason>
        <SOAP-ENV:Text xml:lang="en">Division error</SOAP-ENV:Text>
        <SOAP-ENV:Text xml:lang="it">Errore di applicazione</SOAP-ENV:Text>
        <SOAP-ENV:Text xml:lang="es">Error del uso</SOAP-ENV:Text>
      </SOAP-ENV:Reason>
      <SOAP-ENV:Detail><mymessage>Division error detail</mymessage></SOAP-ENV:Detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

5.3.2.2 %SOAP.Fault12 Properties

The class %SOAP.Fault12 represents a SOAP 1.2 fault. This class has the following properties:

Code

An instance of %SOAP.Fault12.Code, discussed in the following section.

Detail

Used within the `<detail>` element of the SOAP fault. Use this to specify information about the cause of the fault.

If specified, this argument should be a string containing valid XML that can be used within the `<detail>` element. InterSystems IRIS does not verify that the string you provide is valid; it is the responsibility of your application to check this.

Node

Specifies the URI of the SOAP node on the SOAP message path that caused the fault to happen; optional for the destination node.

This is useful if the SOAP message travels through several nodes in the SOAP message path, and the client needs to know which node caused the error. It is beyond the scope of this documentation to discuss this advanced use of SOAP.

Reason

A list of instances of %SOAP.Fault12.Text, discussed in a following section. Each instance contains a reason string and a language code that indicates the language or locality of the reason string. These are used within the `<Reason>` element.

Role

Role that the node was operating in. See the preceding remarks for Node.

faultPrefixDefinition

Specifies a namespace prefix declaration that is added to the envelope of the SOAP fault. Use a value of the following form:

```
xmlns:prefix="namespace"
```

Where *prefix* is the prefix and *namespace* is the namespace URI.

For example:

ObjectScript

```
set fault.faultPrefixDefinition = "xmlns:FLT=""https://myfault.com"""
```

The %SOAP.Fault12 class also provides the **AsString()** method, which returns the fault object as a string.

5.3.2.3 %SOAP.Fault12.Code Properties

You use %SOAP.Fault12.Code as a value for the Code property of an instance of %SOAP.Fault12. The %SOAP.Fault12.Code class has the following properties:

Subcode

An optional subcode.

Value

The value you provide depends on whether you have provided a subcode:

- If you used a subcode, specify Value as a qname.
- If you did not use a subcode, specify Value as one of the SOAP 1.2 macros listed in Macros for SOAP Fault Codes.

5.3.2.4 %SOAP.Fault12.Text Properties

You use %SOAP.Fault12.Text as a list element in the Reason property of an instance of %SOAP.Fault12. The %SOAP.Fault12.Text class has the following properties:

Text

A string indicating the reason for the fault, as intended for users to see.

lang

A code that corresponds to the language or locality in which the fault text is phrased. For information, see the W3 web site (https://www.w3.org/).

5.4 Adding WS-Addressing Header Elements When Faults Occur

Your InterSystems IRIS web service can add WS-Addressing header elements when faults occur. To do this, include the following additional steps within the fault handling of your web service:

1. Choose a fault destination and a fault action to use in case of faults.
2. Using these as arguments, call the **GetDefaultResponseProperties()** class method of %SOAP.Addressing.Properties. This returns an instance of %SOAP.Addressing.Properties that is populated with values as typically needed.
3. Optionally set other properties of the instance of %SOAP.Addressing.Properties, as needed.

   For details, see the class documentation for %SOAP.Addressing.Properties.

4. Set the FaultAddressing property of your web service equal to the instance of %SOAP.Addressing.Properties.

5.5 Adding Other Header Elements When Faults Occur

In addition to or instead of the options discussed in the previous section, your InterSystems IRIS web service can add custom header elements when faults occur. To do this:

1. Create a subclass of %SOAP.Header. In this subclass, add properties to contain the additional data.

   See Adding and Using Custom Header Elements.

2. Within the fault handling of your web service (as described earlier in this topic), include the following additional steps:

   a. Create an instance of your header subclass.

      Note: Despite the name of this class, this object is really a SOAP header element, not an entire header. A SOAP message has one header, which contains multiple elements.

   b. Set its properties as needed.

   c. Insert this header element into the FaultHeaders array property of the web service. To do so, call the **SetAt()** of that property. The key that you provide is used as the main header element name.

For example, consider the following custom header class:

Class Definition

```
Class Fault.CustomHeader Extends %SOAP.Header
{

Parameter XMLTYPE = "CustomHeaderElement";

Property SubElement1 As %String;

Property SubElement2 As %String;

Property SubElement3 As %String;

}
```

We could modify the web method shown previously as follows:

Class Member

```
Method DivideAlt(arg1 As %Numeric, arg2 As %Numeric) As %Numeric [ WebMethod ]
{
  Try {
    Set ans=arg1 / arg2
    } Catch {

        //<detail> element must contain element(s) or whitespace
        //specify this element by passing valid XML as string argument to MakeFault()
        set mydetail="<mymessage>Division error detail</mymessage>"

        set fault=..MakeFault($$$FAULTServer,"Division error",mydetail)

        //Set fault header
        Set header=##class(CustomHeader).%New()
        Set header.SubElement1="custom fault header element"
        Set header.SubElement2="another custom fault header element"
        Set header.SubElement3="yet another custom fault header element"
        Do ..FaultHeaders.SetAt(header,"CustomFaultElement")

        // ReturnFault must be called to send the fault to the client.
        // ReturnFault will not return here.
        Do ..ReturnFault(fault)
```

```
      }
  Quit ans
}
```

When the web client invokes the **Divide()** web method and uses 0 as the denominator, the web service responds as follows:

```
<?xml version='1.0' encoding='UTF-8' standalone='no' ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema' xmlns:flt="https://myfault.org" >
  <SOAP-ENV:Header>
<CustomHeaderElement xmlns:hdr="https://www.mynamespace.org">
    <SubElement1>custom fault header element</SubElement1>
    <SubElement2>another custom fault header element</SubElement2>
    <SubElement3>yet another custom fault header element</SubElement3>
</CustomHeaderElement>
</SOAP-ENV:Header>
  <SOAP-ENV:Body>
...
```

Here line breaks were added for readability.

5.6 Handling SOAP Faults and Other Errors in an InterSystems IRIS Web Client

In an InterSystems IRIS web client, you can use the **TRY-CATCH** mechanism or the older **$ZTRAP** mechanism.

In either case, when an InterSystems IRIS web client receives an error, InterSystems IRIS sets the special variables **$ZERROR** and **%objlasterror**:

- If the error is a SOAP fault, the value of **$ZERROR** starts with <ZSOAP>, and **%objlasterror** contains the status error that is formed from the received SOAP fault.

  In addition, the client instance has a property named SoapFault, which is an instance of %SOAP.Fault or %SOAP.Fault12 (depending on the SOAP version used in the web service). You can use the information in this property. For more information on %SOAP.Fault and %SOAP.Fault12, see the previous sections.

- If the error is not a SOAP fault, use your normal error handling (typically using **$ZERROR**). It is your responsibility to specify how to proceed.

5.6.1 Example 1: Try-Catch

The following method uses **TRY-CATCH**:

Class Member

```
ClassMethod Divide(arg1 As %Numeric, arg2 As %Numeric) As %Numeric
{
    Set $ZERROR=""
    Set client=##class(FaultClient.DivideSoap).%New()

    Try {
        Set ans=client.Divide(arg1,arg2)
        }
    Catch {
        If $ZERROR["<ZSOAP>" {
            Set ans=%objlasterror
            }
            Else {
            Set ans=$$$ERROR($$$ObjectScriptError,$ZERROR)
        }
    }

    Quit ans
}
```

This method uses system macros defined in the %systemInclude include file, so the class that contains this method starts with the following:

```
Include %systemInclude
```

5.6.2 Example 2: $ZTRAP

The following example uses the older **$ZTRAP** mechanism. In this case, when an InterSystems IRIS web client receives an error, control is transferred to the label indicated by the **$ZTRAP** special variable (if that label is defined).

Class Member

```
ClassMethod DivideWithZTRAP(arg1 As %Numeric = 1, arg2 As %Numeric = 2) As %Numeric
{
    Set $ZERROR=""
    Set $ZTRAP="ERRORTRAP"
    Set client=##class(FaultClient.DivideSoap).%New()
    Set ans=client.Divide(arg1,arg2)
    Quit ans

    //control goes here in case of error
ERRORTRAP
    if $ZERROR["<ZSOAP>"
    {
         quit client.SoapFault.Detail
        }
    else
    {
        quit %objlasterror
       }
}
```

5.6.3 SSL Handshake Errors

If an InterSystems IRIS web client uses an SSL connection and a SSL handshake error has occurred, then the SSLError property of the client contains text that describes the SSL error.

6

Using MTOM for Attachments

You can include attachments in SOAP request and response messages. The preferred way to do this is to use InterSystems IRIS® data platform support for MTOM (Message Transmission Optimization Mechanism).

You can also specify MTOM use within a policy; see *Securing Web Services*.

You can also configure InterSystems IRIS® data platform web services and web clients to use gzip to compress their messages after performing any packaging; see Fine-Tuning a Web Service in InterSystems IRIS and Fine-Tuning a Web Client in InterSystems IRIS.

6.1 Attachments and SOAP Message Packaging

Attachments are generally used to carry binary data. InterSystems IRIS SOAP support provides three ways to package your SOAP messages. Before discussing detailed options, it is worthwhile to review these kinds of packaging.

- Package the message with all parts inline (without attachments). Use base-64 encoding for any binary data.

  This is the default behavior of InterSystems IRIS web services and web clients, except when a web service receives an MTOM request (in which case, the service responds with an MTOM response).

- Package the message according to the MTOM (Message Transmission Optimization Mechanism) specification, which results in a slightly more compact message than an all-inline message. This is now the preferred approach for SOAP messages.

  When you use this technique, the system automatically packages the SOAP messages appropriately. That is, the MIME parts are created as needed and are added to the message without your intervention.

  Also by default, when InterSystems IRIS creates an MTOM package, it outputs binary streams using an attachment, and it outputs binary strings (`%Binary` or `%xsd.base64Binary`) inline; you can control this behavior.

  For links to the specifications for MTOM, see SOAP Standards.

- Package the message according to the SOAP with Attachments specification, which results in a slightly more compact message than an all-inline message.

  When you use this technique, you must manually create MIME parts, populate them with data, specify the MIME headers as appropriate, and attach the parts to the SOAP message. This usually requires more work than the MTOM technique. See Using SOAP with Attachments.

6.1.1 SOAP Messages with All-Inline Parts (Default)

The default way to package a SOAP message is to include all its elements as inline parts (that is, without attachments). Any binary data is included inline as base-64–encoded data. For example (with line breaks and spaces added for readability):

```
HTTP/1.1 200 OK
Date: Wed, 19 Nov 2008 21:57:50 GMT
Server: Apache
SET-COOKIE: CSPSESSIONID-SP-8080-UP-csp-gsoap-=003000010000248
guobl000000K7opwlDlY$XbvrGR1eYZsA--; path=/csp/mysamples/;
CACHE-CONTROL: no-cache
EXPIRES: Thu, 29 Oct 1998 17:04:19 GMT
PRAGMA: no-cache
TRANSFER-ENCODING: chunked
Connection: close
Content-Type: text/xml; charset=UTF-8

1d7b
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema'>
<SOAP-ENV:Body>
  <DownloadResponse xmlns="https://www.filetransfer.org">
     <DownloadResult><Filename>sample.pdf</Filename>
     <IsBinary>true</IsBinary>
     <BinaryContents>
        [very long binary content not shown here]
     </BinaryContents></attachment></Upload>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that this packaging does not use MIME, and there are no message boundaries.

6.1.2 SOAP Messages with MTOM Packaging

Another way to package a SOAP message is to use MIME parts as described in the MTOM (Message Transmission Optimization Mechanism) specification. Binary data can be placed into separate MIME parts without base-64 encoding. The SOAP message includes references to the separate parts as needed. For example (with line breaks and spaces added for readability):

```
HTTP/1.1 200 OK
Date: Wed, 19 Nov 2008 21:54:57 GMT
Server: Apache
SET-COOKIE: CSPSESSIONID-SP-8080-UP-csp-gsoap-=003000010
000247guhlx000000NW1KN5UtWg$CWY38$bbTOQ--; path=/csp/mysamples/;
CACHE-CONTROL: no-cache
EXPIRES: Thu, 29 Oct 1998 17:04:19 GMT
MIME-VERSION: 1.0
PRAGMA: no-cache
TRANSFER-ENCODING: chunked
Connection: close
Content-Type: multipart/related; type="application/xop+xml";
boundary=--boundary388.5294117647058824932.470588235294118--;
start="<0.B1150656.EC8A.4B5A.8835.A932E318190B>"; start-info="text/xml"

1ddb
----boundary388.5294117647058824932.470588235294118--
Content-Type: application/xop+xml; type="text/xml"; charset="UTF-8"
Content-Transfer-Encoding: 8bit
Content-Id: <0.B1150656.EC8A.4B5A.8835.A932E318190B>

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema'>
<SOAP-ENV:Body>
<DownloadResponse xmlns="https://www.filetransfer.org">
<DownloadResult>
   <Filename>sample.pdf</Filename>
   <IsBinary>true</IsBinary>
   <BinaryContents>
      <xop:Include href="cid:1.B1150656.EC8A.4B5A.8835.A932E318190B"
              xmlns:xop="https://www.w3.org/2004/08/xop/include"/>
```

```
   </BinaryContents></DownloadResult></DownloadResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
----boundary388.5294117647058824932.470588235294118--
Content-Id: <1.B1150656.EC8A.4B5A.8835.A932E318190B>
Content-Transfer-Encoding: binary
CONTENT-TYPE: application/octet-stream


[very long binary content not shown here]
```

Notice the following differences compared to the default package:

- The message has MIME parts and thus includes boundaries.
- The MIME part has a `Content-ID` attribute.
- In the SOAP body, the element `BinaryContents` consists of a reference to that content ID.

6.1.3 SOAP with Attachments

A third way to package SOAP messages is to use the SOAP with Attachments specification, which also uses MIME parts, but packages the message somewhat differently from MTOM. An example follows (with line breaks and spaces added for readability):

```
HTTP/1.1 200 OK
Date: Mon, 09 Nov 2009 17:47:36 GMT
Server: Apache
SET-COOKIE: CSPSESSIONID-SP-8080-UP-csp-gsoap-=
000000010000213eMwn70000004swjTo4cGuInLMU1n7jaPg--; path=/csp/mysamples/;
CACHE-CONTROL: no-cache
EXPIRES: Thu, 29 Oct 1998 17:04:19 GMT
MIME-VERSION: 1.0
PRAGMA: no-cache
TRANSFER-ENCODING: chunked
Connection: close
Content-Type: multipart/related; type="text/xml";
 boundary=--boundary2629.3529411764705883531.411764705882353--

1ca2
----boundary2629.3529411764705883531.411764705882353--
Content-Type: text/xml; charset="UTF-8"
Content-Transfer-Encoding: 8bit

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
xmlns:s='https://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body><DownloadBinaryResponse xmlns="https://www.filetransfer.org">
<DownloadBinaryResult>MQ==</DownloadBinaryResult></DownloadBinaryResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
----boundary2629.3529411764705883531.411764705882353--
Content-Transfer-Encoding: binary
Content-Type: application/octet-stream

%PDF-1.4
%âãÏÓ
86 0 obj
<</Length 87 0 R
/Filter /FlateDecode
>>
stream
[stream not shown]
```

As with MTOM, there is a boundary string and the attachment is a MIME part. However, in contrast to MTOM, the MIME part does not have a content ID, and the SOAP body does not include any references to the MIME part.

6.2 Default Behavior of InterSystems IRIS Web Services and Web Clients

By default, an InterSystems IRIS web service behaves as follows:

- If it receives a request in an MTOM package, the web service sends the response as an MTOM package.

  Also, the IsMTOM property of the web service instance is set to 1.

- If it receives a request not in an MTOM package, the web service sends the response not in an MTOM package.

By default, an InterSystems IRIS web client behaves as follows:

- It does not send requests as MTOM packages.
- It processes the response regardless of whether the response is in an MTOM package.

  If the response is in an MTOM package, the IsMTOM property of the web client instance is set to 1. If the response is not in an MTOM package, the IsMTOM property is not changed.

6.3 Forcing Responses as MTOM Packages

You can force an InterSystems IRIS web service to send every response as an MTOM package. To do, do any of the following:

- In your InterSystems IRIS web service class, set the *MTOMREQUIRED* parameter to 1.
- In your InterSystems IRIS web service instance, set the MTOMRequired property to 1. You can do this within the web method or within the OnPreWebMethod() callback. For an introduction to this callback, see Customizing Callbacks of the Web Service.
- Attach a policy statement for the web service to send MTOM packages. To do so, you create and compile a configuration class that refers to the web service class; in this policy, enable use of MTOM. See *Securing Web Services*.

  If you attach such a policy statement, your values for *MTOMREQUIRED* is ignored, and MTOMRequired is set equal to 1.

6.3.1 Effect on the WSDL

MTOMREQUIRED and MTOMRequired do not affect the WSDL of the web service.

A policy statement that refers to MTOM *does* affect the WSDL; if you add a policy statement, it is necessary to regenerate any web clients. For an InterSystems IRIS web client, you can simply attach an MTOM policy statement to the client instead of regenerating the client classes.

6.4 Forcing Requests as MTOM Packages

You can force an InterSystems IRIS web client to send every request as an MTOM package. To do, do either of the following:

- In your InterSystems IRIS web client class, set the *MTOMREQUIRED* parameter to 1.

- In your InterSystems IRIS web client instance, set the MTOMRequired property to 1.
- Attach a policy statement to the web client to send MTOM packages. To do so, you create and compile a configuration class that refers to the web service client; in this policy, enable use of MTOM. See *Securing Web Services*.

  If you attach such a policy statement, your values for *MTOMREQUIRED* is ignored, and MTOMRequired is set equal to 1.

6.4.1 Effect on the WSDL

MTOMREQUIRED and MTOMRequired do not assume any change in the WSDL of the web service used by this web client.

A policy statement that refers to MTOM *does* affect the WSDL. That is, you would add an MTOM policy statement to a client only if the web service required it.

6.5 Controlling the MTOM Packaging

By default, when InterSystems IRIS creates an MTOM package, it uses the following rules:

- It outputs binary strings (%Binary or %xsd.base64Binary) inline.
- It outputs binary streams using an attachment.

You can use the *MTOM* property parameter to change this default:

- 1 means output this property as an attachment.
- 0 means output this property inline.

The *MTOM* property parameter has no effect when a web service or web client is not using MTOM.

Also, this property parameter has no effect on the WSDL of a web service.

6.6 Example

This example shows an InterSystems IRIS web service that receives a binary file and sends it back to the caller.

The corresponding web client sends a file with a hardcoded filename, receives the same file from the web service, and then saves it with a new name to prove that it has been successfully sent.

6.6.1 Web Service

The web service is as follows:

Class Definition

```
/// Receive an attachment and send it back
Class MTOM.RoundTripWS Extends %SOAP.WebService
{

///  Name of the web service.
Parameter SERVICENAME = "RoundTrip";

///  SOAP namespace for the web service
Parameter NAMESPACE = "https://www.roundtrip.org";

///  Receive an attachment and send it back
Method ReceiveFile(attachment As %GlobalBinaryStream) As %GlobalBinaryStream [ WebMethod ]
{
  Set ..MTOMRequired=1
  Quit attachment
}

}
```

6.6.2 Web Client

The generated web client (`MTOMClient.RoundTripSoap`) contains the method `ReceiveFile()`, which invokes the web method of the same name. This method is originally as follows:

Class Member

```
Method ReceiveFile(attachment As %xsd.base64Binary) As %xsd.base64Binary
[ Final, SoapBindingStyle = document, SoapBodyUse = literal, WebMethod ]
{
 Quit ..WebMethod("ReceiveFile").Invoke($this,"https://www.roundtrip.org/MTOM.RoundTripWS.ReceiveFile",

 .attachment)
}
```

Because the files we send might exceed the string length limit, we adjust the method signature as follows:

Class Member

```
Method ReceiveFile(attachment As %GlobalBinaryStream) As %GlobalBinaryStream
[ Final, SoapBindingStyle = document, SoapBodyUse = literal, WebMethod ]
{
 Quit ..WebMethod("ReceiveFile").Invoke($this,"https://www.roundtrip.org/MTOM.RoundTripWS.ReceiveFile",

 .attachment)
}
```

MTOM is not required by default in the web client; that is, the *MTOMREQUIRED* parameter is not defined.

To use this proxy client, we create the following class:

Class Definition

```
Include %systemInclude

Class MTOMClient.UseClient
{

/// For this example, hardcode what we are sending
ClassMethod SendFile() As %GlobalBinaryStream
{
  Set client=##class(MTOMClient.RoundTripSoap).%New()
  Set client.MTOMRequired=1

  //reset location to port 8080 to enable tracing
  Set client.Location="https://devsys:8080/csp/mysamples/MTOM.RoundTripWS.cls"

  //create file
  Set filename="c:\sample.pdf"
  Set file=##class(%Library.FileBinaryStream).%New()
  Set file.Filename=filename

  //create %GlobalBinaryStream
```

```
  Set attachment=##class(%GlobalBinaryStream).%New()
  Do attachment.CopyFrom(file)

  //call the web service
  Set answer=client.ReceiveFile(attachment)

  //save the received file to prove we made the round trip successfully
  Set newfilename="c:\roundtrip"_$h_"sample.pdf"
  Set newfile=##class(%Library.FileBinaryStream).%New()
  Set newfile.Filename=newfilename
  Do newfile.CopyFromAndSave(answer)

  Quit answer
}

}
```

7

Using SOAP with Attachments

In your InterSystems IRIS® data platform web clients and web services, you can add and use attachments to SOAP messages by using the InterSystems IRIS support for SOAP with Attachments, instead of using the InterSystems IRIS MTOM support, as described in the previous topic.

This method requires more work than using MTOM because your code must directly manage the MIME parts used as attachments.

For the specifications for the SOAP with Attachments standard, see SOAP Standards.

7.1 Sending Attachments

When you use the InterSystems IRIS support for the SOAP with Attachments standard, you use the following process to send attachments:

1. Create the attachments. To create an attachment:
   a. Use a stream object to represent the attachment data. The class you use depends on the exact interface you need to obtain the stream data. For example, you might use %Library.FileCharacterStream to read the contents of a file into a stream.
   b. Create a MIME part, which is an instance of %Net.MIMEPart.
   c. For the MIME part:
      - Set the Body property equal to your stream object. Or set the Parts property, which must be a list of instances of %Net.MIMEPart.
      - Call the **SetHeader()** method to set the `Content-Transfer-Encoding` header of the MIME part. Be sure to set this appropriately for the type of data you are sending.

2. Add the attachments to the web service or web client. To add a given attachment, you insert the MIME part into the appropriate property as follows:
   - If you are sending an attachment from a web client, update the Attachments property of your web client.
   - If you are sending an attachment from a web service, update the ResponseAttachments property of the web service.

   Each of these properties is a list with the usual list interface (for example, **SetAt()**, **Count()**, and **GetAt()** methods).
3. Update the appropriate properties of the web client or the web service to describe the attachment contents:

- ContentId
- ContentLocation

7.2 Using Attachments

When an InterSystems IRIS web service or web client receives a SOAP message that has attachments (as specified by the SOAP with Attachments specification), the following happens:

- The attachments are inserted into the appropriate property:
  - For a web service, the inbound attachments are placed in the Attachments property.
  - For a web client, the inbound attachments are placed in the ResponseAttachments property.

  Each of these properties is a list with the usual list interface (for example, **SetAt()**, **Count()**, and **GetAt()** methods). Each list element is an instance of %Net.MIMEPart. Note that a MIME part can in turn contain other MIME parts. Your code is responsible for determining the structure and contents of the attachments.
- The ContentID and ContentLocation properties are updated to reflect the `Content-ID` and `Content-Location` headers of the inbound SOAP message.

The web service or web client can access these properties and thus access the attachments.

7.3 Example

This section provides an example web service and web client that send attachments to each other.

7.3.1 Web Service

The web service provides two methods:

- `UploadAscii()` receives an ASCII attachment and saves it.
- `DownloadBinary()` sends a binary attachment to the requestor.

The class definition is as follows:

Class Definition

```
Class GSOAP.FileTransferWS Extends %SOAP.WebService
{

///  Name of the web service.
Parameter SERVICENAME = "FileTransfer";

///  SOAP namespace for the web service
Parameter NAMESPACE = "https://www.filetransfer.org";

/// Namespaces of referenced classes will be used in the WSDL.
Parameter USECLASSNAMESPACES = 1;

///  Receive an attachment and save it
Method UploadAscii(filename As %String = "sample.txt") As %Status [WebMethod]
{
  //assume 1 attachment; ignore any others
  Set attach=..Attachments.GetAt(1)
```

```
  Set file=##class(%FileCharacterStream).%New()
  Set file.Filename="c:\from-client"_$H_filename

  //copy attachment into file
  Set status=file.CopyFrom(attach.Body)
  If $$$ISERR(status) {do $System.Status.DisplayError(status)}
  Set status= file.%Save()
  Quit status
}

///  Create an attachment and send it in response to the web client call
Method DownloadBinary(filename As %String = "sample.pdf") As %Status [WebMethod]
{
  //use a file-type stream to read file contents
  Set file=##class(%Library.FileBinaryStream).%New()
  Set file.Filename="c:\"_filename

  //create MIMEpart and add file to it
  Set mimepart=##class(%Net.MIMEPart).%New()
  Set mimepart.Body=file

  //set header appropriately for binary file
  Do mimepart.SetHeader("Content-Type","application/octet-stream")
  Do mimepart.SetHeader("Content-Transfer-Encoding","binary")

  //attach
  Set status=..ResponseAttachments.Insert(mimepart)
  Quit status
}

}
```

7.3.2 Web Client

The web client application provides two methods:

- `UploadAscii()` sends an ASCII file to the web service.
- `DownloadBinary()` calls the web service and receives an binary file in response.

The generated web client class (`GSOAPClient.FileTransfer.FileTransferSoap`) includes the methods `UploadAscii()` and `DownloadBinary()`, which invoke the corresponding methods of the preceding web service. This class is not shown.

The web client application also includes the following class, which uses this generated web client class:

Class Definition

```
Include %systemInclude

Class GSOAPClient.FileTransfer.UseClient
{

ClassMethod DownloadBinary(filename As %String = "sample.pdf") As %Status
{
  Set client=##class(GSOAPClient.FileTransfer.FileTransferSoap).%New()

  //call web method
  Set ans=client.DownloadBinary(filename)

  //get the attachment (assume only 1)
  Set attach=client.ResponseAttachments.GetAt(1)

  //create a file and copy stream contents into it
  Set file=##class(%FileBinaryStream).%New()
  //include $H in the filename to make filename unique
  Set file.Filename="c:\from-service"_$H_filename
  Set status=file.CopyFrom(attach.Body)
  If $$$ISERR(status) {do $System.Status.DisplayError(status)}
  Set status= file.%Save()
  Quit status
}

ClassMethod UploadAscii(filename As %String = "sample.txt") As %Status
{
  Set client=##class(GSOAPClient.FileTransfer.FileTransferSoap).%New()
```

```
  //use a file-type stream to read file contents
  Set file=##class(%Library.FileCharacterStream).%New()
  Set file.Filename="c:\"_filename

  //create MIME part, add file as Body, and set the header
  Set mimepart=##class(%Net.MIMEPart).%New()
  Set mimepart.Body=file
  Do mimepart.SetHeader("Content-Transfer-Encoding","7bit")

  //attach to client and call web method
  Do client.Attachments.Insert(mimepart)
  Set status=client.UploadAscii(filename)
  Quit status
}


}
```

8

Adding and Using Custom Header Elements

This topic describes how to add and use custom SOAP header elements.

For information on adding header elements when faults occur, SOAP Fault Handling.

WS-Addressing header elements are described elsewhere. For information on WS-Security header elements, see *Securing Web Services*.

8.1 Introduction to SOAP Header Elements in InterSystems IRIS

A SOAP message can include a header (the `<Header>` element), which contains a set of *header elements*. For example:

```
<SOAP-ENV:Envelope>
   <SOAP-ENV:Header>
      <MyHeaderElement>
         <Subelement1>abc</Subelement1>
         <Subelement2>def</Subelement2>
      </MyHeaderElement>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
...
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Informally, each header element is often referred to as a header. This is not strictly accurate, because the message itself can contain at most one header, which is always `<Header>`, with an appropriate namespace prefix. The header can contain WS-Security header elements, WS-Addressing header elements, and your own custom header elements.

A header element carries additional information for possible use by the web service or web client that receives the SOAP message. In the example shown here, this information is carried within XML elements. A header element can also include XML attributes, although none are shown in the previous example. The SOAP standard specifies three standard attributes (`mustUnderstand`, `actor`, and `encodingStyle`) to indicate how a recipient should process the SOAP message.

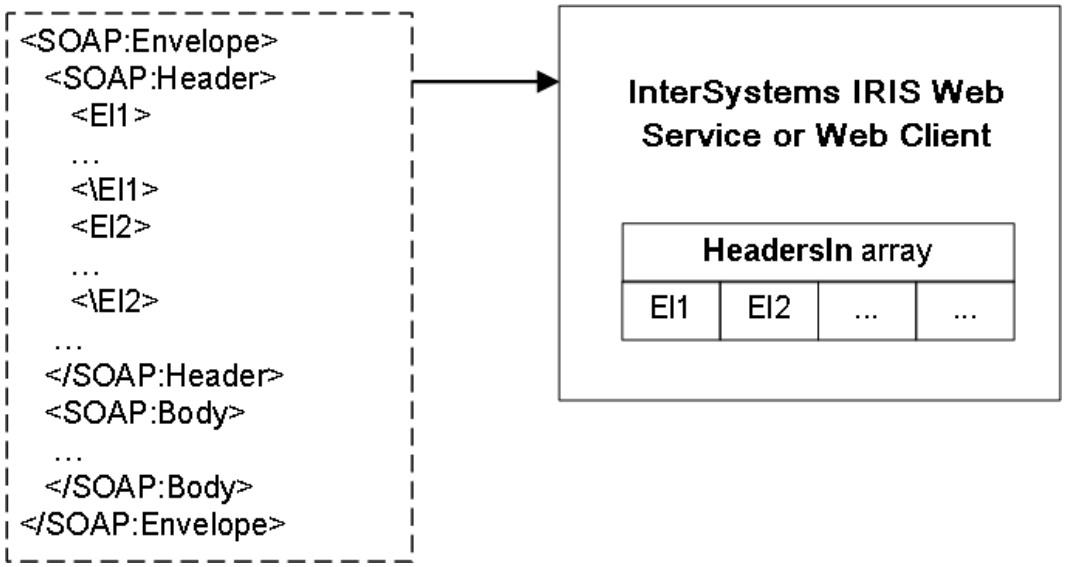
8.1.1 How InterSystems IRIS Represents SOAP Headers

InterSystems IRIS® data platform represents each header element as an instance of %SOAP.Header or one of its subclasses. %SOAP.Header is an XML-enabled class with properties that correspond to the standard header element attributes (`mustUnderstand`, `actor`, and `encodingStyle`).

InterSystems IRIS provides specialized subclasses of %SOAP.Header for use with WS-Addressing and WS-Security. To represent custom header elements, you create your own subclasses of %SOAP.Header.

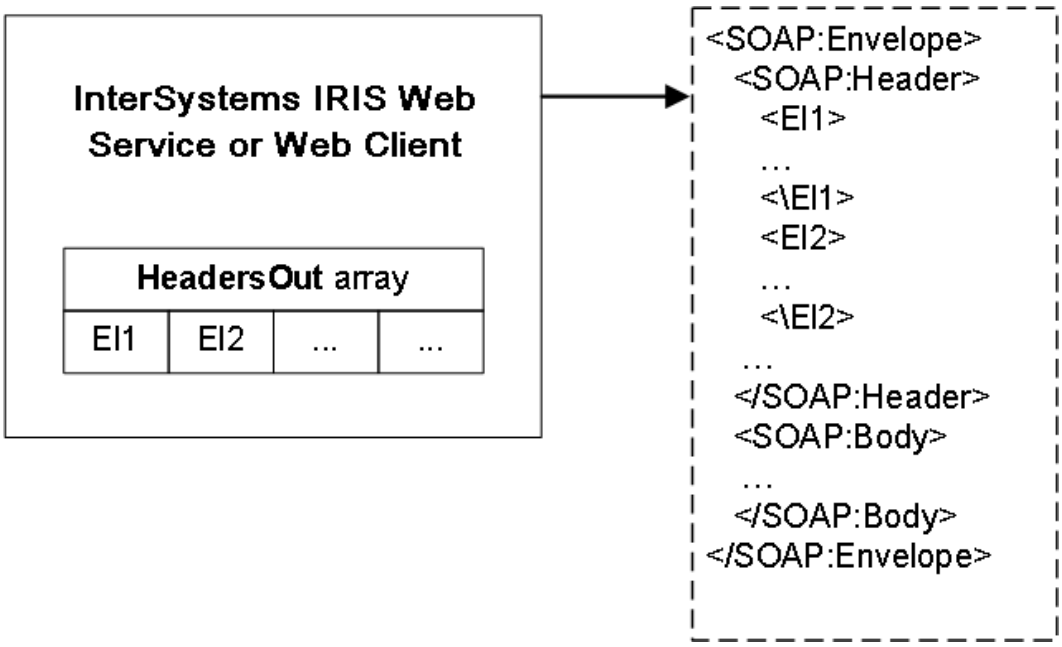
When an InterSystems IRIS web service or client receives a SOAP message, it imports and processes the message. During this step, if the message contains a header with custom header elements, InterSystems IRIS compares the header elements to the list of supported header elements (discussed in the next subsection).

Then the service or client creates an instance of each applicable header element class, inserts those into an array, and places that array in its own HeadersIn property:



To use these header elements, your InterSystems IRIS web service or client can access the HeadersIn property. If the SOAP message did not include a `<Header>` element, the **Count()** of the HeadersIn property is 0.

Similarly, before your InterSystems IRIS web service or client sends a SOAP message, it must update the HeadersOut property so that it contains any custom elements you want to include in the outbound message. If the HeadersOut **Count()** is 0, the outbound SOAP message does not include a `<Header>` element.



For custom header elements, you always use the HeadersIn and HeadersOut properties.

The details are different for other (non-custom) header elements:

- For WS-Addressing, use the AddressingIn and AddressingOut properties rather than the HeadersIn and HeadersOut properties. See Adding and Using WS-Addressing Header Elements.
- For WS-Security header elements, use the WS-Policy features, described in *Securing Web Services*.

  Or directly use the SecurityIn and SecurityOut properties, discussed in *Securing Web Services*. This is generally more work.

  (Note that the WS-Security header elements are also contained in the HeadersIn and HeadersOut properties, b´ut it is not recommended to access them or to set them via those properties.)
- InterSystems IRIS SOAP session support uses the HeadersIn and HeadersOut properties. See SOAP Session Management.

8.1.2 Supported Header Elements

InterSystems IRIS web services and clients automatically support WS-Addressing and WS-Security headers, but do not automatically support other headers.

To specify the supported header elements in an InterSystems IRIS web service or client, you add an XData block to the class and specify the class parameter *USECLASSNAMESPACES*. The XData block lists the supported elements. The class parameter causes the WSDL to include the applicable types. See Specifying the Supported Header Elements.

8.1.3 Header Elements and the WSDL

The WSDL for a web service advertises the header elements supported by that web service and permitted by web clients that communicate with that web service.

For an InterSystems IRIS web service, the generated WSDL might not include information about the SOAP header elements:

- If you add SOAP headers manually by setting the HeadersOut property, be sure to declare them in an XData block as described in Specifying Supported Header Elements. Also specify the class parameter *USECLASSNAMESPACES* as 1 in the web service class.

If you follow these steps, the WSDL contains all the applicable information. Otherwise, it does not, and you must save the WSDL to a file and edit it manually as needed.

- If you add WS-Security headers by setting the SecurityOut property (as described in *Securing Web Services*), the WSDL does not include all needed information. (This is because the WSDL is generated at compile time and the headers are added later, at runtime.) In this case, save the WSDL to a file and edit it manually as needed.

  For many reasons, it is simpler and easier to add WS-Security elements by using WS-Policy. With WS-Policy, the generated WSDL includes all needed information.

- In other cases, the generated WSDL includes all needed information.

Note that the W3C specifications do not require a web service to provide a generated WSDL.

8.1.4 Required Header Elements

If a given header element specifies `mustUnderstand=1`, the element is considered mandatory, and the recipient must support it. The recipient cannot process the message unless it recognizes all mandatory header element.

Following the SOAP standard, InterSystems IRIS rejects SOAP messages that contain required but unsupported header elements. Specifically, if InterSystems IRIS web service or client receives a message that contains a header element that includes `mustUnderstand=1`, and if that service or client does not support that header element, the service or client issues a SOAP fault and then ignores the message.

8.2 Defining Custom Header Elements

If you generate an InterSystems IRIS web service or web client based on a given WSDL, the system generates classes to represent any header elements as needed.

If you create a web service or client manually, you must manually define classes to represent any custom header elements. To do so:

1. For each custom header element, create a subclass of %SOAP.Header.
2. Specify the *NAMESPACE* parameter to indicate the namespace of the header element.
3. Specify the *XMLNAME* parameter to indicate the name of the header element.
4. In the subclass, define properties to contain the header information you need. By default, your properties are projected to elements within your `<Header>` element.
5. Optionally specify the *XMLFORMAT* parameter, which controls the format of this header element. By default, the header elements are always in literal format (rather than SOAP-encoded).

For example:

Class Definition

```
Class Scenario1.MyHeaderElement Extends %SOAP.Header
{

Parameter NAMESPACE = "https://www.myheaders.org";

Parameter XMLNAME = "MyHeader";

Property Subelement1 As %String;

Property Subelement2 As %String;

}
```

This header element appears as follows within a SOAP message:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope [parts omitted]>
  <SOAP-ENV:Header>
    <MyHeader xmlns="https://www.myheaders.org"
                    xmlns:hdr="https://www.myheaders.org">
      <Subelement1>abc</Subelement1>
      <Subelement2>def</Subelement2>
    </MyHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
   [omitted]
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For details on customizing the XML projection of any given object class, see *Projecting Objects to XML*.

8.3 Adding a Custom Header Element to a SOAP Message

To add custom header elements to a SOAP message (from either the web service or the web client), do the following before sending the SOAP message.

1. Create an instance of your header object.
2. Set the properties of that object as appropriate, optionally including the actor and mustUnderstand properties.
3. Add the new header to the outbound header array, the HeadersOut property. This property is an array with the usual array interface (for example, the **SetAt()**, **Count()**, and **GetAt()** methods).

Note: If you perform these steps in a utility method, note that the method must be an instance method and must be a member of an instantiable class (not an abstract class, for example).

Then within your web service or client class, you could have a utility method that adds a header element:

Class Member

```
Method AddMyHeaderElement(mustUnderstand=0)
{
    Set h=##class(MyHeaderElement).%New()
    Set h.Subelement1 = "abc"
    Set h.Subelement2 = "def"
    If mustUnderstand {Set h.mustUnderstand=1}
    Do ..HeadersOut.SetAt(h,"MyHeaderElement")
}
```

Finally, you could call this utility method from each web method where you wanted to use it. For example:

Class Member

```
/// Divide arg1 by arg2 and return the result
Method Divide(arg1 As %Numeric, arg2 As %Numeric) As %Numeric [ WebMethod ]
{
    //main method code here
    //...

    do ..AddMyHeaderElement()

    Quit ans
}
```

When you invoke this web method, the header is added to the SOAP response.

8.4 Specifying Supported Header Elements

InterSystems IRIS web services and clients automatically support WS-Addressing and WS-Security header elements, but do not automatically support other header elements.

To specify the header elements supported by an InterSystems IRIS web service or web client, do the following:

- Define classes to represent these header elements as described in Defining Custom Header Elements.
- Associate the header element classes with header elements for the web service or web client.

  You can do this in either of two ways:

  – Add an XData block to the web service or client class. In this XData block, specify an association between specific header elements and the corresponding header element classes.

    If the service or client class also sets the class parameter *USECLASSNAMESPACES* to 1 (the recommended value), then this header information is used in the generated WSDL.

  – In your web service or web client class, specify the *SOAPHEADERS* parameter. In this parameter, specify an association between specific header elements and the corresponding header element classes.

    Note: This technique is less flexible, does not affect the generated WSDL, and is now deprecated.

  The following sections give the details.

8.5 Specifying the Supported Header Elements in an XData Block

If you generate an InterSystems IRIS web service or web client based on a given WSDL, the generated class includes an XData block to represent any header elements supported in its SOAP messages.

If you create a web service or client manually, you must manually specify this XData block.

The following is a simple example:

```
XData NewXData1
{
<parameters xmlns="https://www.intersystems.com/configuration">
   <request>
      <header name="ServiceHeader" class="NewHeaders.MyCustomHeader"/>
   </request>
   <response>
      <header name="ExpectedClientHeader" class="NewHeaders.MyCustomHeader"/>
   </response>
</parameters>
}
```

8.5.1 Details

The requirements for this XData block are as follows:

- The XData block can have any name. The name (`NewXData1` in this case) is not used.
- The top level element must be `<parameters>`

- The <parameters> element and all its child elements (and their children) must be in the namespace "https://www.intersystems.com/configuration"
- The <parameters> element can have the following children:
  - <request> — Determines the header elements associated with all request messages, for any header elements that should be the same in all request messages.

    This element should have a child element <header> for each applicable header element.
  - <response> — Determines the header elements associated with all response messages, for any header elements that should be the same in all response messages.

    This element should have a child element <header> for each applicable header element.
  - <methodname> — Determines the header elements associated with the web method whose name is *methodname*.

    This element can have the following children:

    - <header> — Determines the header elements associated with request and response messages for this web method, for any header elements that should be the same in both cases.
    - <request> — Determines the header elements associated with request messages for this web method.

      This element should have a child element <header> for each applicable header element.
    - <response> — Determines the header elements associated with response messages for this web method.

      This element should have a child element <header> for each applicable header element.

- In this XData block, each <header> element associates a header element with the InterSystems IRIS class that should be used to represent it. This element includes the following attributes:

Attribute	Purpose
name	Name of the header element.
class	InterSystems IRIS class that represents this header element.
alias	(Optional) Key for this header element in the HeadersIn array of the web service or web client. The default is the value given for the name attribute.

  The position of a <header> element within the XData block indicates the messages to which it applies.

8.5.2 Inheritance of Custom Headers

If you create a subclass of this web service, that subclass inherits the header information that is not specific to a method — the header information contained in the <request> or <response> elements that are direct child elements of <parameters>. This is true even if *SOAPMETHODINHERITANCE* is 0.

8.5.3 Examples

Another example is as follows:

Class Member

```
XData service
{
<parameters xmlns="https://www.intersystems.com/configuration">
  <response>
    <header name="Header2" class="User.Header4" alias="Header4"/>
    <header name="Header3" class="User.Header3"/>
  </response>
```

```
  <method name="echoBase64">
    <request>
      <header name="Header2" class="User.Header4" alias="Header4"/>
      <Action>https://soapinterop.org/Round2Base.Service.echoBase64Request</Action>
    </request>
    <response>
      <header name="Header2" class="User.Header2" alias="Header2"/>
      <header name="IposTransportHeader" class="ipos.IposTransportHeader"/>
      <Action>https://soapinterop.org/Round2Base.Service.echoBase64Result</Action>
    </response>
  </method>
  <method name="echoString">
    <request>
      <Action>https://soapinterop.org/Round2Base.Service.echoStringRequest</Action>
    </request>
    <response>
      <Action>https://soapinterop.org/Round2Base.Service.echoStringAnswer</Action>
    </response>
  </method>
</parameters>
}
```

8.6 Specifying the Supported Header Elements in the SOAPHEADERS Parameter

The older way to specify supported header elements is to include the *SOAPHEADERS* parameter in the web service or web client class.

This parameter must equal a comma-separated list of header specifications. Each header specification has the following form:

```
headerName:headerPackage.headerClass
```

Where *headerName* is the element name of the supported header and *headerPackage.headerClass* is the complete package and class name of a class that represents that header. For example:

```
Parameter SOAPHEADERS = "MyHeaderElement:Scenario1Client.MyHeaderElement"
```

This list identifies all headers supported in the SOAP requests to this web service or client and indicates the class to which each one is mapped.

If you use this older technique, note the following points:

- For a web service, this technique does not affect the generated WSDL.
- It is not possible to specify different header elements for specific web methods.
- This technique is deprecated.

8.6.1 Inheritance of Custom Headers

If you create a subclass of this web service, that subclass inherits the *SOAPHEADERS* parameter. This is true even if *SOAPMETHODINHERITANCE* is 0.

8.7 Using Header Elements

To use specific SOAP header elements after receiving a request message, use the HeadersIn property of the service or client.

For each supported header element, the service or client creates an instance of the appropriate header class and adds the header to the inbound header array, which is the HeadersIn property. This property is an array with the usual array interface (for example, **SetAt()**, **Count()**, and **GetAt()** methods). The web service or web client can then act on these headers as appropriate.

Note: The header element namespace is not used for matching the header element in the list. However, the header element namespace in the SOAP message must be the same as specified by the *NAMESPACE* parameter in your header element subclass; otherwise, an error occurs when the message is imported.

9

Adding and Using WS-Addressing Header Elements

This topic describes how to add and use WS-Addressing header elements.

For details about this standard, see Standards Supported by InterSystems IRIS.

Also see Adding WS-Addressing Header Elements When Faults Occur.

9.1 Overview

You can add WS-Addressing header elements to your SOAP messages, as specified by the WS-Addressing standards for SOAP 1.1 and SOAP 1.2. To do so, do one of the following:

- Specify the *WSADDRESSING* parameter of your web service or client as `"AUTO"`. This option adds a default set of WS-Addressing header elements, discussed in a following subsection.
- Specify *WSADDRESSING* as `"OFF"` (the default) and add WS-Addressing header elements manually, as discussed in a following subsection.
- Create a policy for the web service or client to include WS-Addressing header elements. To do so, you create and compile a configuration class that refers to the web service or client; in this policy, enable WS-Addressing. See *Securing Web Services*.

  If you attach such a policy, InterSystems IRIS® data platform uses the same set of default WS-Addressing header elements by default. You can create and add WS-Addressing header elements manually instead.

  If you attach such a policy, your value for *WSADDRESSING* is ignored.

9.2 Effect on the WSDL

For a web service, the *WSADDRESSING* parameter does not affect the generated WSDL. Similarly, if you specify this for a web client, it is not necessary for the WSDL to change.

A policy statement that refers to WS-Addressing does affect the WSDL; if you add a policy statement, it is necessary to regenerate any web clients. For an InterSystems IRIS web client, you can simply attach a WS-Addressing policy statement to the client instead of regenerating the client classes.

9.3 Default WS-Addressing Header Elements

This section describes and shows examples of the default WS-Addressing header elements.

9.3.1 Default WS-Addressing Header Elements in Request Messages

If you enable WS-Addressing as described previously in this section, the web client includes the following WS-Addressing header elements in its request messages:

- To:destination address
- Action: SoapAction
- MessageID: unique uuid
- ReplyTo: anonymous

For example:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'
                   xmlns:wsa='https://www.w3.org/2005/08/addressing'>
   <SOAP-ENV:Header>
      <wsa:Action>https://www.myapp.org/GSOAP.DivideAddressingWS.Divide</wsa:Action>
      <wsa:MessageID>urn:uuid:91576FE2-4533-43CB-BFA1-51D2B631453A</wsa:MessageID>
      <wsa:ReplyTo>
        <wsa:Address xsi:type="s:string">https://www.w3.org/2005/08/addressing/anonymous</wsa:Address>

      </wsa:ReplyTo>
      <wsa:To>https://devsys:8080/csp/mysamples/GSOP.DivideAddressingWS.cls</wsa:To>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <Divide xmlns="https://www.myapp.org">
         <arg1 xsi:type="s:decimal">1</arg1>
         <arg2 xsi:type="s:decimal">7</arg2>
      </Divide>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

9.3.2 Default WS-Addressing Header Elements in Response Messages

If you enable WS-Addressing as described previously in this section and if the request message includes WS-Addressing header elements, the web service includes the following WS-Addressing header elements in its response messages:

- To: anonymous
- Action: SoapAction_"Response"
- MessageID: unique uuid
- RelatesTo: MessageID of request

For example:

XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
                   xmlns:xsi='https://www.w3.org/2001/XMLSchema-instance'
                   xmlns:s='https://www.w3.org/2001/XMLSchema'
                   xmlns:wsa='https://www.w3.org/2005/08/addressing'>
   <SOAP-ENV:Header>
      <wsa:Action>https://www.myapp.org/GSOAP.DivideAddressingWS.DivideResponse</wsa:Action>
      <wsa:MessageID>urn:uuid:577B5D65-D7E3-4EF7-9BF1-E8422F5CD739</wsa:MessageID>
      <wsa:RelatesTo>urn:uuid:91576FE2-4533-43CB-BFA1-51D2B631453A</wsa:RelatesTo>
      <wsa:To>https://www.w3.org/2005/08/addressing/anonymous</wsa:To>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <DivideResponse xmlns="https://www.myapp.org">
         <DivideResult>.14285714285714285</DivideResult>
      </DivideResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

9.4 Adding WS-Addressing Header Elements Manually

Instead of using the default WS-Addressing header elements, you can create and add your own elements manually. To do so:

1. Create an instance of %SOAP.Addressing.Properties and specify its properties as needed. For details, see the class reference.
2. Set the AddressingOut property of the web service or client equal to this instance of %SOAP.Addressing.Properties.

Note: If you set the AddressingOut property, the web service or web client uses the WS-Addressing header elements in this property rather than any WS-Addressing elements specified in an attached policy.

9.5 Handling WS-Addressing Header Elements

When an InterSystems IRIS web service or client receives a message that includes WS-Addressing header elements, the AddressingIn property of the service or client is updated to equal an instance of %SOAP.Addressing.Properties. Your web service or client can then examine the details of its AddressingIn property.

For details on %SOAP.Addressing.Properties, see the class reference.

10

SOAP Session Management

SOAP web services are inherently stateless and thus do not maintain sessions. However, it is often useful to maintain a session between a web client and the web service that it uses. The InterSystems IRIS® data platform SOAP support provides a way for you to do this.

Also see Specifying Custom HTTP Requests.

And see WSDL Differences for InterSystems IRIS Sessions in Details of the Generated WSDLs.

10.1 Overview of SOAP Sessions

You can maintain a session between a web client and an InterSystems IRIS web service. This support consists of the following tools:

- Web session management (managed by InterSystems IRIS and by the InterSystems Web Gateway).
- The InterSystems IRIS SOAP session header, which is a simple proprietary header.

The overall flow is as follows:

1. The web client sends an initial message to the web service. This message does not include the InterSystems IRIS SOAP session header.
2. The web service receives the message and starts a new web session.
3. When the web service sends its reply, it adds the InterSystems IRIS SOAP session header to the message.
4. When the web client receives the reply, it must detect the SOAP session header and extract the session cookie. Then when the web client sends another message, it must use the cookie to create a SOAP session header in that message.

   Note:

   - If the client is an InterSystems IRIS web client, the session cookie is saved automatically in the SessionCookie property of web client. Also, the client instance automatically creates the SOAP session header and includes it in all messages that it sends.
   - This step also happens automatically for .NET web clients, if the same client instance is used for all SOAP messages in the session. You may need further code for other client platforms.

5. The web service receives the next reply, continues the web session, and includes the SOAP session header again when it responds.

It is not necessary to include a method to log out. The web session times out after a brief interval (the timeout period for the web application).

10.2 Enabling Sessions

In order to use InterSystems IRIS support for SOAP sessions, you must be using an InterSystems IRIS web service.

- If the web client is based on InterSystems IRIS, only one step is needed to enable SOAP session support. In your web service class, set the *SOAPSESSION* parameter equal to 1.
- If you are using a third-party tool to create the web client, you are responsible for detecting the InterSystems IRIS SOAP session header element in the initial response and ensuring that the web client includes this header element in all requests during the life of the session. This header element has the following format:

  XML

  ```
  <csp:CSPCHD xmlns:csp="https://www.intersystems.com/SOAPheaders"><id>value of CSPCHD
  token</id></csp:CSPCHD>
  ```

10.3 Using Session Information

When sessions are enabled, the web service can use the variable *%session*, which is an instance of %CSP.Session. Properties of this object contain system information and any information you choose to add. Some of the commonly used properties are as follows:

- SessionID — Unique identifier of this session.
- EndSession — Normally this equals 0. Set this property to 1 in order to end the session.
- Data — InterSystems IRIS multidimensional array intended to hold any custom data.
- NewSession — Equals 1 if this is new session.
- AppTimeout — Specifies the timeout value for the session, in seconds.

The *%session* object provides many other properties, as well as some methods for tasks related to sessions. For further details, see the class documentation for %CSP.Session.

11

Using the InterSystems IRIS Binary SOAP Format

InterSystems IRIS® data platform SOAP support provides an optional proprietary binary SOAP format, which is useful when you send and receive large SOAP messages and want to minimize message size.

An InterSystems IRIS web service can receive SOAP requests either in InterSystems IRIS binary SOAP format or in the usual SOAP format. No parameter is needed to enable this behavior. An InterSystems IRIS web client uses binary SOAP format only if it is configured to do so.

Also see WSDL Differences for InterSystems IRIS Binary SOAP Format in Details of the Generated WSDLs.

Note: If an InterSystems IRIS web service or web client uses this proprietary binary SOAP format, you cannot use WS-Security or WS-Policy features with this web service or client. See *Securing Web Services*.

11.1 Introduction

InterSystems IRIS binary SOAP is carried over HTTP messages as follows:

- The message uses the POST method.
- Content-Type is always `"application/octet-stream"`.
- The body is a binary representation of objects using a proprietary protocol.
- A binary SOAP request includes an HTTP ISCSoap header of the following form:

  ```
  ISCSoap: NAMESPACE/Package.Class.Method
  ```

- SOAP sessions are supported. The session information is maintained by using the normal web session cookie. However, the SessionCookie property for SOAP web clients and web services is not supported, because binary SOAP does not use the CSPCHD proprietary SOAP header.

The following example shows a binary SOAP request:

```
POST /csp/mysamples/GSOP.WebServiceBinary.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; InterSystems IRIS;)
Host: devsys:8080
Connection: Close
ISCSoap: https://www.myapp.org/GSOAP.WebServiceBinary.Divide
Content-Type: application/octet-stream
Content-Length: 90

00085hdBinaryClient.MyAppSoap.Dividearg1arg2t
```

Notice that only the SOAP envelope and its contents are affected. The HTTP header is not affected.

11.2 Extending the WSDL for an InterSystems IRIS Web Service

Any InterSystems IRIS web service can receive SOAP requests either in InterSystems IRIS binary SOAP format or in the usual SOAP format. If an InterSystems IRIS web service receives a binary request, it sends a binary response. Otherwise, it sends the usual response. No parameter is needed to enable this behavior.

You can extend the WSDL of a web service so that:

1. The WSDL publicly states that the web service supports the InterSystems IRIS binary SOAP format in addition to the usual SOAP format.
2. The WSDL includes information on using the InterSystems IRIS binary SOAP format.

This allows any InterSystems IRIS web client to correctly send messages in this format if wanted.

To extend the WSDL of an InterSystems IRIS web service in this way, set the *SOAPBINARY* parameter to 1 for the web service.

For details on the changes, see WSDL Differences for InterSystems IRIS Binary SOAP Format in Details of the Generated WSDLs.

11.3 Redefining an InterSystems IRIS Web Client to Use Binary SOAP

You can redefine an existing InterSystems IRIS web client so that it uses InterSystems IRIS binary SOAP format. To do so, set the *SOAPBINARY* parameter or the SoapBinary property to 1 for the web client. You may need to make additional changes; see WSDL Differences for InterSystems IRIS Binary SOAP Format in Details of the Generated WSDLs.

11.4 Specifying the Character Set

The SoapBinaryCharset property of the web client specifies the InterSystems IRIS character set (for example: Unicode, Latin1) of the web service. If the character set of the client machine and service machine are the same, strings are sent RAW; otherwise they are sent encoded as UTF8.

The SoapBinaryCharset property defaults to the *SOAPBINARYCHARSET* parameter, which defaults to null, which always converts strings to UTF8.

11.5 Details on the InterSystems IRIS Binary SOAP Format

The API for binary SOAP is different from XML SOAP as follows:

- For the InterSystems IRIS server:
  - Binary SOAP is denoted by the presence of the ISCSoap HTTP header.
  - The **Initialize()** method of the web service is not called.
  - A normal %request.Content stream is used in the initial implementation.
  - Login is via IRISUsername and IRISPassword query parameters attached to the URL. No login page is ever returned for binary SOAP.
  - If an invalid login occurs, then an instance of %SOAP.Fault is returned.
- For %Net.HttpRequest responses:
  - A binary SOAP request is indicated by setting the SoapBinary property of the web client class for the method being called.
  - The request is sent using a normal EntityBody stream.
  - The response is returned in the Data property of HttpResponse.

12

Using Datasets in SOAP Messages

This topic discusses **%XML.DataSet**, which is an XML-enabled dataset that you can use in SOAP messages when both the web service and client are based on InterSystems IRIS® data platform or when one side uses .NET.

Important: Other SOAP vendors do not support datasets and cannot process a WSDL generated by a web service that uses them.

12.1 About Datasets

A *dataset* is an XML-format result set defined by Microsoft (and used for .NET) and also supported in InterSystems IRIS. If both the web service and client are based on InterSystems IRIS or when one side uses .NET, you can use datasets as input to or output from a web method. Other SOAP technologies do not understand this format.

When you work with web services or clients in InterSystems IRIS, you use %XML.DataSet (or a custom subclass) to represent a dataset. For details on using %XML.DataSet, see the class reference.

Note: The %XML.DataSet class supports only a single table. That is, the query it uses can return only a single table.

The sample web service SOAP.Demo (in the SAMPLES namespace) demonstrates datasets in InterSystems IRIS. Specifically, the following web methods use datasets:

- **GetByName()**
- **GetDataSetByName()**
- **QueryByName()**

To output results of a query so that a Java-based web client can work with it, use a %ListOfObjects subclass; SOAP.Demo shows an example.

12.2 Defining a Typed Dataset

Any dataset uses a query that specifies the data to retrieve. If the query is known at compile time, the dataset is *typed*; otherwise it is *untyped*. Typed datasets are convenient in many cases; for example, in .NET, a typed dataset allows code completion in Microsoft IDEs.

To define a typed dataset, create a subclass of %XML.DataSet and specify the *QUERYNAME* and *CLASSNAME* parameters. Together, these parameters refer to a specific SQL query. When it generates the schema for the dataset, %XML.DataSet considers the class and property metadata such as any **LogicalToXSD()** methods in custom data types.

Note: If you use %XML.DataSet as the return value for a method, the XML type for that value is `DataSet`. On the other hand, if you use a subclass of %XML.DataSet as the return value, the XML type for the value is the name of that subclass. This behavior is the same as that of other XML-enabled classes, and it affects the XML types that are described in the WSDL. See Controlling the Projection to XML Types

12.3 Controlling the Dataset Format

By default, a dataset is written in Microsoft DiffGram format and is preceded by its XML schema. The following shows an example:

```
<SOAP-ENV:Body>
  <Get0Response xmlns="https://www.myapp.org">
    <Get0Result>
    <s:schema id="DefaultDataSet" xmlns=""
    attributeFormDefault="qualified"
    elementFormDefault="qualified"
    xmlns:s="https://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
      <s:element name="DefaultDataSet" msdata:IsDataSet="true">
        <s:complexType>
          <s:choice maxOccurs="unbounded">
            <s:element name="GetPeople">
              <s:complexType>
                <s:sequence>
                  <s:element name="Name" type="s:string" minOccurs="0" />
                  <s:element name="DOB" type="s:date" minOccurs="0" />
                </s:sequence>
              </s:complexType>
            </s:element>
          </s:choice>
        </s:complexType>
      </s:element>
    </s:schema>

    <diffgr:diffgram
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
      <DefaultDataSet xmlns="">
        <GetPeople diffgr:id="GetPeople1" msdata:rowOrder="0">
          <Name>Quine,Howard Z.</Name>
          <DOB>1965-11-29</DOB>
        </GetPeople>
...
      </DefaultDataSet>
    </diffgr:diffgram>
   </Get0Result>
 </Get0Response>
</SOAP-ENV:Body>
```

The %XML.DataSet class provides the following options for controlling this format:

- The *DATAONLY* parameter and the DiffGram property control whether the output is in DiffGram format. By default, the output is in DiffGram format, which is shown above. If you subclass %XML.DataSet and set *DATAONLY* equal to

1, or if you set the DiffGram equal to 0, the output is not in DiffGram format. The body of the XML dataset is as follows instead:

```
<SOAP-ENV:Body>
  <Get0Response xmlns="https://www.myapp.org">
    <Get0Result>
      <GetPeople xmlns="">
        <Name>Quine,Howard Z.</Name>
        <DOB>1965-11-29</DOB>
      </GetPeople>
      <GetPeople xmlns="">
...
    </Get0Result>
  </Get0Response>
</SOAP-ENV:Body>
```

In contrast to DiffGram format, notice that the schema is not output by default and that the output does not include the `<diffgram>` element.

- The NeedSchema property controls whether the output includes the XML schema. If you are using DiffGram format, the default is to output the schema; if you are not using DiffGram format, the default is not to output the schema. To force output of the schema, set NeedSchema equal to 1, or to suppress output of the schema, set it equal to 0.
- If you use DiffGram format, the WriteEmptyDiffgram property controls the contents of the `<diffgram>` element in the case when the dataset has no rows. By default (or if WriteEmptyDiffgram equals 0), the `<diffgram>` element contains an empty element as follows:

```
...
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
   <DefaultDataSet xmlns="">
   </DefaultDataSet>
</diffgr:diffgram>
...
```

In contrast, if WriteEmptyDiffgram equals 1, the `<diffgram>` element contains nothing:

```
...
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
</diffgr:diffgram>
...
```

This property has no effect if you are not using DiffGram format.

- If you use DiffGram format, the DataSetName property controls the name of the element within the `<diffgram>` element. By default, this element is named `<DefaultDataSet>`, as you can see in the example above. This property has no effect if you are not using DiffGram format.

%XML.DataSet also provides the CaseSensitive property, which corresponds to the Microsoft dataset property of the same name. The default is false, for compatibility reasons.

12.4 Viewing the Dataset and Schema as XML

A dataset that extends %XML.DataSet has utility methods that you can use to generate XML. All of these methods write to the current device:

- **WriteXML()** writes the dataset as XML, optionally preceded by the XML schema. This method has optional arguments to control the name of the top-level element, the use of namespaces, treatment of nulls, and so on. By default, this method considers the format of the dataset, as specified by the settings in the previous section. You can override that result by providing values for optional arguments that control whether the output is in DiffGram format and so on. For details, see the class documentation for %XML.DataSet.

- **XMLExport()** writes the XML schema for the dataset, followed by the dataset as XML.
- **WriteSchema()** writes just the XML schema for the dataset.
- **XMLSchema()** writes the Microsoft proprietary XML representation of its dataset class.

For information on generating XML schemas from XML-enabled objects, see *Using XML Tools*.

12.5 Effect on the WSDL

If an InterSystems IRIS web service uses %XML.DataSet as input or output to a web method, that affects the WSDL so that clients other than InterSystems IRIS and .NET have difficulty consuming the WSDL.

For a typed dataset, the WSDL includes the following elements (within the `<types>` section):

```
<s:element name="GetDataSetByNameResponse">
   <s:complexType>
      <s:sequence>
        <s:element name="GetDataSetByNameResult" type="s0:ByNameDataSet" />
      </s:sequence>
   </s:complexType>
</s:element>
<s:complexType name="ByNameDataSet">
   <s:sequence>
     <s:any namespace="https://tempuri.org/ByNameDataSet" />
   </s:sequence>
</s:complexType>
```

For an untyped dataset, the WSDL includes the following:

```
<s:element name="GetByNameResponse">
   <s:complexType>
      <s:sequence>
         <s:element name="GetByNameResult" type="s0:DataSet" />
      </s:sequence>
   </s:complexType>
</s:element>
<s:complexType name="DataSet">
   <s:sequence>
      <s:element ref="s:schema" />
   <s:any />
   </s:sequence>
</s:complexType>
```

In the latter case, if you attempt to generate a web client within a tool other than InterSystems IRIS or .NET, an error occurs because there is not enough information for that tool. For Metro, it is possible to load additional schema information before attempting to consume the WSDL. To do so, you can use a command-line tool called wsimport. This technique can provide enough information to generate the client.

In all cases, however, considerable work is necessary to write code so that the client can either interpret or generate messages in the appropriate form.

13

Troubleshooting SOAP Problems in InterSystems IRIS

This topic provides information to help you identify causes of SOAP problems in InterSystems IRIS® data platform.

For information on problems that are obviously related to security, see Troubleshooting Security Problems. In the rare case that your SOAP client is using HTTP authentication, note that you can enable logging for the authentication; see Providing Login Credentials in Sending HTTP Requests.

13.1 Information Needed for Troubleshooting

To identify the cause of a SOAP problem, you typically need the following information:

- The WSDL and all external documents to which it refers.
- (In the case of message-related problems) Some form of message logging or tracing. You have the following options:

Option	Usable with SSL/TLS?	Shows HTTP headers?	Comments
InterSystems IRIS SOAP log	Yes	Optionally	For security errors, this log shows more detail than is contained in the SOAP fault.
Web Gateway trace	Yes	Yes	For problems with SOAP messages that use MTOM (MIME attachment), it is crucial to see HTTP headers.
Third-party tracing tools	No	Depends on the tool	Some tracing tools also show lower-level details such as the actual packets being sent, which can be critical when you are troubleshooting.

  These options are discussed in the following subsections.

It is also extremely useful to handle faults correctly so that you receive the best possible information. See SOAP Fault Handling.

13.1.1 InterSystems IRIS SOAP Log

To log the SOAP calls made to or from an InterSystems IRIS namespace, enable SOAP logging as described here.

Important: The SOAP log is voluminous, so it is important to enable it only when needed and disable it as soon as possible. See the notes below for details.

To log SOAP calls for a namespace, set nodes of the *^ISCSOAP* global in that namespace as follows:

ObjectScript

```
Set ^ISCSOAP("LogFile")=filename
Set ^ISCSOAP("Log")=optionString
Set ^ISCSOAP("LogMaxFileSize")=optionalMaxLogSize
```

You can also set additional nodes to further fine-tune what is logged:

ObjectScript

```
Set ^ISCSOAP("LogURL",optionalUrlMatch)=""
Set ^ISCSOAP("LogJob",optionalJobId)=""
Set ^ISCSOAP("LogClass",optionalClassname)=""
```

Where:

- *optionString* specifies the type of data to include in the log. Use a combination of the following case-sensitive values:
  - `i` — Log inbound messages.
  - `o` — Log outbound messages.
  - `s` — Log security information. Note that this option provides more detail than is generally contained in the SOAP fault, which is intentionally vague to prevent follow-on security attacks.
  - `h` — Log only SOAP headers. You must combine `h` with `i` and/or `o`. When you use `h` with `i`, the log includes only the SOAP Envelope and Header elements for inbound messages. Similarly, when you use `h` with `o`, the log includes only the SOAP Envelope and Header elements for outbound messages. The corresponding SOAP Body elements are not logged.
  - `H` — Log HTTP headers. You must combine `H` with `i` and/or `o`. When you use `H` with `i`, the log includes HTTP headers for inbound messages. Similarly, when you use `H` with `o`, the log includes HTTP headers for outbound messages. HTTP headers are logged in addition to any SOAP data.

    Note that this option also logs HTTP headers for SOAP messages received by web services within Interoperability productions (that is, received by subclasses of EnsLib.SOAP.Service),

  You can use a string that contains any combination of these values, for example: `"iosh"`
- *filename* is the complete path and filename of the log file to create.
- *optionalMaxLogSize* is the maximum size of the log file, in bytes. When this maximum is reached, no more entries are written to the log file. This setting is optional.
- *optionalUrlMatch* is a string that specifies the URL pattern to match. If this option is specified, only traffic using this URL pattern is written to the log. This string can be an exact URL or can include a leading * wildcard, a trailing * wildcard, or both.
  - Example with exact URL: http://localhost:8080/instance/csp/custom/WebService.cls?CfgItem=SoapLogIn
  - Example with leading wildcard : */instance/csp/custom/WebService.cls?CfgItem=SoapLogIn
  - Example with trailing wildcard: http://localhost:8080/instance/csp/custom/WebService.cls

– Example with both leading and trailing wildcards: /instance/csp/custom/WebService.cls

Note: To match in all scenarios (client and server), use both leading and trailing wildcards.

Example:

ObjectScript

```
Set ^ISCSOAP("LogURL","/instance/csp/custom/WebService.cls")=""
```

- *optionalJobId* is the ID of a job (process). This can be the ID of any of the following:
  – The web client process
  – The Web Gateway process running the web service (whether the service is part of a production or not)

    Note that the Web Gateway operates a pool of processes, any one of which might service a request. Also, the Web Gateway may recycle and open new connections and the LogJob setting would need to be updated to follow.
  – The production process, in the case of a web service that is part of a production

  These job IDs can be seen in the Management Portal. If a job ID is no longer valid due to that process stopping, the logging will not automatically delete obsolete job ID entries.

  For example:

  ObjectScript

  ```
  Set ^ISCSOAP("LogJob",15712)=""
  Set ^ISCSOAP("LogJob",16108)=""
  ```

- *optionalClassname* is the name of a class. For example:

  ObjectScript

  ```
  Set ^ISCSOAP("LogClass", "WSDL.SoapLog.Service" )=""  // SOAP Client classname
  Set ^ISCSOAP("LogClass", "WSDL.SoapLog.Client.SoapLogServiceSoap" )=""  // SOAP Service classname
  ```

The log indicates the sender or the recipient as appropriate, so that you can see which web service or client participated in the exchange.

The following shows a partial example of a log file with line breaks added for readability:

```
01/05/2022 13:27:02 ********************
Output from web client with SOAP action = https://www.mysecureapp.org/GSOAP.AddComplexSecureWS.Add
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='https://schemas.xmlsoap.org/soap/envelope/'
...
  <SOAP-ENV:Header>
      <Security
xmlns="https://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">

  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
...
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>


**** Output HTTP headers for Web Client
User-Agent: Mozilla/4.0 (compatible; InterSystems IRIS;)
Host: hostid
Accept-Encoding: gzip


**** Input HTTP headers for Web Client
HTTP/1.1 200 OK
CACHE-CONTROL: no-cache
```

```
CONTENT-ENCODING: gzip
CONTENT-LENGTH: 479
CONTENT-TYPE: application/soap+xml; charset=UTF-8
...


01/05/2022 13:27:33 ********************
Input to web client with SOAP action = https://www.mysecureapp.org/GSOAP.AddComplexSecureWS.Add

ERROR #6059: Unable to open TCP/IP socket to server devsys:8080
string
```

Note the following points:

- With InterSystems IRIS XML tools, you can validate signatures of signed XML documents and decrypt encrypted XML documents. If you perform these tasks in this namespace, the log contains details for them as well. See *Using XML Tools*.
- The InterSystems IRIS SOAP log captures SOAP calls even when no message is sent on the wire (that is, when the service and client are both on a single machine).
- If a severe error occurs, the system stops writing to the SOAP log. See the messages log instead. For information, see Monitoring Log Files.
- The Task Manager `CheckLogging` task, which runs every night, creates an alert if SOAP logging is left on for too long (by default, 2 days). Because the SOAP log is voluminous, it is important to pay attention to this alert.
- The **SOAPLogContains()** method of %SOAP.WebBase reads through the current SOAP log trying to match lines that meet criteria specified by arguments to the method.

13.1.2 HTTP Trace in the Web Gateway

The Web Gateway management page enables you to trace HTTP requests and responses. See Using the HTTP Trace Facility.

13.1.3 Third-Party Tracing Tools

To test your web service, you can use third-party tracing tools.

Tracing tools enable you to see the actual method call, as well as the response. A tracing session listens on a certain port, shows you the messages it receives there, forwards those messages to a destination port, shows the responses, and forwards the responses to the listening port.

For example, suppose you have a web service at `https://devsys/csp/mysamples/GSOP.Divide.CLS`

And suppose you have a web client that you created to talk to that service. The web client has a *LOCATION* parameter equal to `"https://devsys/csp/mysamples/GSOP.Divide.CLS"`

To trace messages between the client and service, you need to do two things:

- In the tracing tool, start a tracing session that listens on port 8080 (for example) and that uses the destination port 52773.
- In the web client, edit the *LOCATION* parameter to use port 8080 instead of 52773. Then recompile.

  Or, in your code that invokes the web client, change the Location property of the web client:

  ObjectScript

  ```
  //reset location to port 8080 to enable tracing
  set client.Location="https://devsys:8080/csp/mysamples/GSOP.DivideWS.cls"
  ```

Now when you use the web client, the tracing tool intercepts and displays messages between the client and the web service, as shown in the following example:

```
POST /csp/gsop/GSOP.Divide.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; IRIS;)
Host: localhost:8080
Connection: Close
SOAPAction: http://www.mynamespace.org/GSOP.Divide.Divide
Content-Length: 401
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
```

```
HTTP/1.1 200 OK
Date: Fri, 18 Apr 2008 15:52:33 GMT
Server: Apache
SET-COOKIE:
CSPSESSIONID-SP-8080-UP-csp-gsop-=00000001000026fgoIxI000000nRuHG3uXQjvSOYWJ7z2ZVw--;
path=/csp/gsop/;
CACHE-CONTROL: no-cache
EXPIRES: Thu, 29 Oct 1998 17:04:19 GMT
PRAGMA: no-cache
CONTENT-LENGTH: 378
Connection: close
Content-Type: text/xml; charset=UTF-8

<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:s='http://www.w3.org/2001/XMLSchema'>
  <SOAP-ENV:Body>
<DivideResponse
xmlns="http://www.mynamespace.org"><DivideResult>.5</DivideResult></DivideResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The top area shows the request sent by the client. The bottom area shows the response sent by the web service.

13.2 Problems Consuming WSDLs

You may see an error while generating client classes for several reasons:

- The WSDL URL that you supplied may require authentication with an SSL certificate, and you did not specify an SSL configuration or you specified an incorrect SSL configuration. If so, an error message appears, for example:

  ```
  ERROR #6301: SAX XML Parser Error: invalid document structure
  while processing Anonymous Stream at line 1 offset 1
  ```

  To correct this error, you can specify the appropriate SSL configuration.

- The WSDL URL that you supplied requires authentication with a username and password. If so, an error message appears, for example:

  ```
  ERROR #6301: SAX XML Parser Error: Expected entity name for reference
  while processing Anonymous Stream at line 10 offset 27
  ```

Note: While the line and the offset values may vary, they will differ from those in the previous scenario.

To correct this error, you can specify a username and password as described in Generating the Client Classes, and Using a Password-Protected WSDL URL.

- The WSDL may contain references to externally defined entities that could not be resolved before 10-second timeout period. If so, an error message appears, for example:

```
ERROR #6416: Element 'wsdl:definitions' - unrecognized wsdl element 'porttype'
```

To correct this error, you can check the WSDL for `<import>` and `<include>` directives, for example:

```
<import namespace="https://example.com/stockquote/definitions"
           location="https://example.com/stockquote/stockquote.wsdl"/>
```

If you find such directives, take the following steps:

1. Download the primary WSDL to a file.
2. Download the referenced WSDL to a file.
3. Edit the primary WSDL to refer to the new location of the referenced WSDL.

Similarly, you can check whether the WSDL refers to other documents using a relative URL, for example:

```
xmlns:acme="urn:acme.com.:acme:service:ServiceEndpointInterface"
```

If you downloaded the WSDL to a file, you cannot use relative references. Instead, you must also download the referenced document and edit the WSDL to point to its new location.

- The WSDL contains `<message>` elements with multiple parts and uses document-style binding. If so, an error message appears, for example:

```
ERROR #6425: Element 'wsdl:binding:operation:msg:input' - message 'AddSoapOut'
Message Style must be used for document style message with 2 or more parts.
```

- The WSDL is invalid. If so, an error message appears, for example:

```
ERROR #6419: Element 'wsdl:binding:operation' - inconsistent
soap:namespace for operation getWidgetInfo
```

The error message specifies the problem with the WSDL. In this example, the `<operation>` element in the following WSDL excerpt produced the error:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="https://acme.acmecorp.biz:9999/widget/services"
                  xmlns="https://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdl="https://schemas.xmlsoap.org/wsdl/" =
                 [parts omitted]>
  <wsdl:message name="getWidgetInfoRequest">
  </wsdl:message>
  <wsdl:message name="getWidgetInfoResponse">
    <wsdl:part name="getWidgetInfoReturn" type="xsd:string"/>
  </wsdl:message>
  <wsdl:portType name="Version">
    <wsdl:operation name="getWidgetInfo">
      <wsdl:input message="impl:getWidgetInfoRequest" name="getWidgetInfoRequest"/>
      <wsdl:output message="impl:getWidgetInfoResponse" name="getWidgetInfoResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="VersionSoapBinding" type="impl:Version">
    <wsdlsoap:binding style="rpc" transport="https://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getWidgetInfo">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="getWidgetInfoRequest">
        <wsdlsoap:body encodingStyle="https://schemas.xmlsoap.org/soap/encoding/"
                       namespace="https://acmesubsidiary.com"
                       use="encoded"/>
      </wsdl:input>
```

```
      <wsdl:output name="getWidgetInfoResponse">
        <wsdlsoap:body encodingStyle="https://schemas.xmlsoap.org/soap/encoding/"

                       namespace="https://acme.acmecorp.biz:9999/widget/services"
                       use="encoded"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  [parts omitted]
```

In this case, the problem is that the `<input>` part of `<operation>` states that the request message (`getVersionRequest`) is in the namespace `"https://acmesubsidiary.com"`, but the earlier part of the WSDL shows that this message is the target namespace of the web service: `"https://acme.acmecorp.biz:9999/widget/services"`.

Note that an invalid WSDL document can be a valid XML document, so using a pure XML tool to validate the WSDL is not a sufficient test. Some third-party WSDL validation tools are available.

- The WSDL contains features that are not supported in InterSystems IRIS. For more information, see Consuming WSDLs.

13.3 Problems Sending Messages

If you have problems when sending SOAP messages to or from an InterSystems IRIS web service or client, consider this list of common scenarios:

- The SOAP message might include an extremely long string or binary value, exceeding the string length limit. In this case, InterSystems IRIS throws one of the following errors:
  - A <MAXSTRING> error
  - A datatype validation error (which can have other causes as well):

    ```
    ERROR #6232: Datatype validation failed for tag your_method_name ...
    ```

  When generating web client or web service classes, InterSystems IRIS assumes that any string-type input or output can be represented in InterSystems IRIS as %String. Similarly, it assumes that any input or output with XML type base64Binary can be represented in InterSystems IRIS as %xsd.base64Binary). There is no information in a WSDL to inform InterSystems IRIS that this input or output could exceed the string length limit.

  See Adjusting the Generated Classes for Extremely Long Strings; this information applies to both web clients and services.

- The web service or client might receive WS-Security headers, but not yet be configured to recognize them. This can result in a generic error like the following:

  ```
  <ZSOAP>zInvokeClient+269^%SOAP.WebClient.1
  ```

  An error like this can also have other causes. If you receive an error like this, first check whether the messages include WS-Security headers; if so, add the following to the web service or client and recompile it:

  ### Class Member

  ```
  Parameter SECURITYIN="REQUIRE";
  ```

  Also, if InterSystems IRIS generated a security policy (in a configuration class), you might need to edit that policy to provide missing details; see Editing the Generated Policy. If you do not do so, you can receive generic errors as given above.

- The web service or client might require a more specific message form than should be required, according to the SOAP specifications. (This can occur for a service or client that is not in InterSystems IRIS.) InterSystems has encountered the following scenarios, listed here from (approximately) most common to least common:
  - The web service or client requires the message to specify the `xsi:type` attribute for all elements in the message. To specify use of this attribute, see Controlling Use of the xsi:type Attribute, which applies to both web services and clients.
  - For a null string value, the web service or client requires a null element (rather than omitting the element). To work around this, you can control the form of null string arguments, see Controlling the Form of Null String Arguments, which applies to both web services and clients.
  - The web service or client requires specific namespace prefixes. InterSystems IRIS does not provide a way to specify the namespace prefixes in general.

    For the SOAP envelope, however, you can specify the prefix to use. See Specifying the SOAP Envelope Prefix, which applies to both web services and clients.
  - The web client requires the SOAP action to be quoted. To work around this, see Quoting the SOAP Action (SOAP 1.1 Only).
  - The web service or client requires a BOM (byte-order mark) at the start of each SOAP message. The BOM should not be needed because a SOAP message is encoded as UTF-8, which does not have byte order issues. See Adding a Byte-Order Mark to the SOAP Messages, which applies to both web services and clients.

  The symptoms of these problems depend upon the third-party product in use.
- The web service or client might not comply with the WSDL. This should not be possible for an InterSystems IRIS web service or client, but can occur in other scenarios. InterSystems has seen the following scenarios:
  - An element in the message is not in the namespace required by the WSDL.
  - The message does not have elements in the same order as the WSDL.

  To determine whether the service or client complies with the WSDL, compare the messages to the WSDL.

  Or, for a third-party web service, to determine whether the web service complies with the WSDL, it is useful to do the following:

  1. Generate a web client using a third-party tool.
  2. Send messages from that web client:
     - If this is successful, it is likely that the web service does expect and send messages that are consistent with its WSDL, and the cause of the problem is elsewhere. In this case, compare the messages sent by this client to the messages sent by the InterSystems IRIS client.
     - If this is not successful, it is likely that the web service does not expect or send messages that are consistent with its WSDL.

- The web service or client might send messages of a form not supported in InterSystems IRIS. It is useful to examine the WSDL in use and make sure it is supported in InterSystems IRIS; see Consuming WSDLs. Note that these details have changed in InterSystems IRIS over time.

A

Summary of Web Service URLs

This topic summarizes the URLs related to an InterSystems IRIS® data platform web service.

A.1 Web Service URLs

The URLs related to an InterSystems IRIS web service are as follows:

end point for the web service

```
https://<baseURL>/csp/namespace/web_serv.cls
```

Where:

- <baseURL> is the base URL for your instance.
- */csp/namespace* is the name of the web application in which the web service resides.
- *web_serv* is the class name of the web service.

For example:

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

WSDL

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

For example:

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

Note that both of these URLs are part of the */csp/namespace* web application.

A.2 Using a Password-Protected WSDL URL

You can use the WSDL URL of an existing InterSystems IRIS web service to create a web client in InterSystems IRIS or a third-party tool. However, if the parent web application for the web service requires password authentication, you must

supply a valid username and password in the WSDL URL to access the WSDL. To do so, you append `&IRISUsername=`*`username`*`&IRISPassword=`*`password`* to the URL, for example:

```
https://devsys/csp/mysamples/MyApp.StockService.cls?WSDL&IRISUsername=
_SYSTEM&IRISPassword=SYS
```

Additionally, if you use a third-party tool to create the web client and the tool uses URL redirection after logins, you must append `&IRISNoRedirect=1`. For example, after a login, .NET performs a URL redirect. Consequently, the WSDL URL format for a .NET web client is as follows:

```
https://devsys/csp/mysamples/MyApp.StockService.cls?WSDL&IRISUsername=
_SYSTEM&IRISPassword=SYS&IRISNoRedirect=1
```

If you are unable to generate a web client from a password-protected WSDL URL after several attempts, consider the following alternatives:

- Retrieve the WSDL from a browser by supplying a valid username and password, save the WSDL as a file, and use the file to generate the web client.
- If the web service must provide continuous access to the WSDL, create a web application that is not password protected to serve the WSDL.
- If there is a legacy application that uses CSP/ZEN to serve the WSDL and its **Prevent login CSRF attack** setting is enabled, then temporarily disable the setting if you determine that it is safe to do so. For more information, see content on settings in this type of legacy application.

B

Details of the Generated WSDLs

For reference, this topic shows the parts of a sample WSDL document for an InterSystems IRIS® data platform web service, along with information about how keywords and parameters affect these parts.

The signatures of your web methods also affect the WSDL, but this topic does not discuss the details.

The WSDL is also affected by the XML projections of all XML-enabled classes used by the web service. See *Projecting Objects to XML*.

Note: If the web service has a compiled policy configuration class, the `<binding>` section also includes elements of the form `<wsp:Policy>`. This documentation does not discuss how policies affect the WSDL, because the effects are determined by the WS-SecurityPolicy and other specifications.

For information on policy configurations, see *Securing Web Services*.

The system generates WSDL documents for convenience, but this is not required by the W3C specifications. For important notes on this topic, see Viewing the WSDL.

B.1 Overview of WSDL Documents

A web service has a *WSDL document*, a machine-readable interface definition. A WSDL document is written in XML, following the standard for the Web Services Description Language. It defines the contract for how the web service and its clients interact.

A WSDL has a root `<definitions>` element that contains additional elements that define the following:

- Definition of any XML types or elements needed for inputs or outputs of the web service, defined in terms of base XML types. The `<types>` element includes one or more `<schema>` elements, which define the XML types, elements, or both as needed.
- Definition of *messages* used by the web service. Each web method requires one or two messages: a request message to call the web method, and a response message to use in reply. Each message is defined in terms of XML types or elements.
- Definition of *port types* used by the web service. Each port defines one or more *operations*. An operation corresponds to a web method and uses the corresponding message or messages.

  In general, a WSDL can contain multiple `<portType>` elements, but the WSDL for an InterSystems IRIS web service contains only one.

- The *bindings* of the web service, which defines the message format and protocol details for operations and messages defined by a particular port type.

  In general, a WSDL can contain multiple `<binding>` elements, but the WSDL for an InterSystems IRIS web service contains only one.

- Formal definition of the web *service*, in terms of the preceding components. This includes a URL for invoking the web service.

The service, any schemas, and the messages are all associated with XML namespaces; these can all be in a single namespace or can be in different namespaces. Note that InterSystems IRIS support for SOAP does not support all possible variations. See Standards Supported by InterSystems IRIS.

B.2 Sample Web Service

This topic shows parts of the WSDL of the following sample web service:

Class Definition

```
Class WSDLSamples.BasicWS Extends %SOAP.WebService
{

Parameter SERVICENAME = "MyServiceName";

Parameter NAMESPACE = "https://www.mynamespace.org";

Parameter USECLASSNAMESPACES = 1;

///  adds two complex numbers
Method Add(a As ComplexNumber, b As ComplexNumber) As ComplexNumber [ WebMethod ]
{
    Set sum = ##class(ComplexNumber).%New()
    Set sum.Real = a.Real + b.Real
    Set sum.Imaginary = a.Imaginary + b.Imaginary

    Quit sum
}
}
```

This web service refers to the following class:

Class Definition

```
///  A complex number
Class WSDLSamples.ComplexNumber Extends (%RegisteredObject, %XML.Adaptor)
{

/// real part of the complex number
Property Real As %Double;

/// imaginary part of the complex number
Property Imaginary As %Double;

}
```

Except where noted, the topic shows parts of the WSDL for this web service. (In some cases, the topic uses variations of this web service.)

B.3 Namespace Declarations

Before we examine the rest of the WSDL in detail, it is useful to see the namespace declarations used by the rest of the WSDL. The <definitions> element contains one namespace declaration for each namespace used in the WSDL. For the sample web service shown earlier in this topic, these declarations are as follows:

```
<definitions xmlns="https://schemas.xmlsoap.org/wsdl/"
xmlns:SOAP-ENC="https://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="https://schemas.xmlsoap.org/wsdl/mime/"
xmlns:s="https://www.w3.org/2001/XMLSchema"
xmlns:s0="https://www.mynamespace.org"
xmlns:soap="https://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="https://schemas.xmlsoap.org/wsdl/"
xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
targetNamespace="https://www.mynamespace.org">
```

The following parameters affect the namespace declarations:

- The *NAMESPACE* parameter in the web service.

  This parameter is used for the `targetNamespace` attribute, which indicates the target namespace of the web service.

  If the *NAMESPACE* parameter is not specified, `targetNamespace` is `"https://tempuri.org"`

- The *SOAPVERSION* parameter in the web service.

  This affects the SOAP namespaces that are automatically included.

  By default, *SOAPVERSION* is 1.1.

  For *SOAPVERSION* equal to 1.2, the WSDL would instead include the following:

  ```
  <definitions ...
  xmlns:soap12="https://schemas.xmlsoap.org/wsdl/soap12/"
  ...
  ```

  For *SOAPVERSION* equal to `""`, the WSDL would instead include the following:

  ```
  <definitions
  xmlns:soap="https://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soap12="https://schemas.xmlsoap.org/wsdl/soap12/"
  ...
  ```

- Other namespace keywords and parameters in the web service and in any XML-enabled classes used by the web service.

  These items are discussed in the following sections.

  These namespaces are declared as needed, for consistency with the rest of the WSDL.

Also, other namespaces (such as `https://schemas.xmlsoap.org/wsdl/soap/`) are included automatically as appropriate.

The namespace prefixes are all chosen automatically and cannot be customized.

B.4 <service>

When you examine a WSDL, it is useful to read it from the end to the beginning.

The final element within a WSDL is the <service> element, which defines the web service. For the sample web service shown earlier in this topic, this element is as follows:

XML

```
<service name="MyServiceName">
    <port name="MyServiceNameSoap" binding="s0:MyServiceNameSoap">
        <soap:address location="https://devsys/csp/mysamples/WSDLSamples.BasicWS.cls"/>
    </port>
</service>
```

This element is specified as follows:

- The *SERVICENAME* parameter of the web service is used as the `name` attribute of the `<service>` element. See Specifying the Service Name and Namespaces of the Web Service.

  This parameter also affects the `name` and `binding` attributes of the `<port>` element; no separate control is provided.

- The `binding` attribute refers to a binding in the `s0` namespace, which is listed in the namespace declarations. This namespace is specified by the *NAMESPACE* parameter of the web service.

- The URL of the web service class controls the `location` attribute of the `<soap:address>` element.

B.5 <binding>

Before the `<service>` element, the WSDL contains `<binding>` elements, each of which defines message format and protocol details for operations and messages defined by a particular `<portType>` element.

In general, a WSDL can contain multiple `<binding>` elements, but the WSDL for an InterSystems IRIS web service contains only one.

For the sample web service shown earlier in this topic, this element is as follows:

XML

```
<binding name="MyServiceNameSoap" type="s0:MyServiceNameSoap">
    <soap:binding transport="https://schemas.xmlsoap.org/soap/http"
                  style="document"/>
    <operation name="Add">
        <soap:operation soapAction="https://www.mynamespace.org/WSDLSamples.BasicWS.Add"
                        style="document"/>
        <input>
            <soap:body use="literal"/>
        </input>
        <output>
            <soap:body use="literal"/>
        </output>
    </operation>
</binding>
```

This element is specified as follows:

- The `name` attribute of the `<binding>` element is automatically kept consistent with the `<service>` element (and would not be meaningful to change).

  ```
  <binding name="MyServiceNameSoap"  ...
  ```

- The `type` attribute of the `<binding>` element refers to a `<portType>` element in the `s0` namespace, which is listed in the namespace declarations. This namespace is specified by the *NAMESPACE* parameter of the web service.

- The `name` attribute of each `<operation>` element is based on the name of the web method (and would not be meaningful to change).

  ```
  <operation name="Add"> ...
  ```

- If you specify the SoapAction keyword for the web method, that value is used for the `soapAction` attribute of the operation. For example:

```
...
<operation name="Add">
    <soap:operation soapAction="mysoapaction" style="document"/>
...
```

- If the return type of a method is defined as %SOAP.OneWay, the affects this element as described in WSDL Differences for One-Way Web Methods.
- If the *SOAPBINARY* parameter is 1 for the web service, that affects this element as described in WSDL Differences for InterSystems IRIS Binary SOAP Format.
- If the *SOAPSESSION* parameter is 1 for the web service, that affects this element as described in WSDL Differences for InterSystems IRIS SOAP Sessions.

  The SoapBindingStyle class keyword, SoapBindingStyle method keyword, and SoapBindingStyle query keyword affect the `<binding>` element as described in the *Class Definition Reference*. These keywords can have the values `document` and `rpc`.
- The SoapBodyUse class keyword, SoapBodyUse method keyword, and SoapBodyUse query keyword affect the `<binding>` element as described in the *Class Definition Reference*. These keywords can have the values `literal` and `encoded`.

Note: If the web service has a compiled policy configuration class, the `<binding>` section also includes elements of the form `<wsp:Policy>`. This documentation does not discuss how policies affect the WSDL, because the effects are determined by the WS-SecurityPolicy and other specifications.

For information on policy configurations, see *Securing Web Services*.

B.6 <portType>

Before the `<binding>` section, a WSDL contains `<portType>` elements, each of which defines an individual endpoint by specifying a single address for a `<binding>` element. A `<portType>` element is a named set of abstract operations and the abstract messages involved.

In general, a WSDL can contain multiple `<portType>` elements, but the WSDL for an InterSystems IRIS web service contains only one.

For the sample web service shown earlier in this topic, the `<portType>` element is as follows:

XML

```
<portType name="MyServiceNameSoap">
    <operation name="Add">
        <input message="s0:AddSoapIn"/>
        <output message="s0:AddSoapOut"/>
    </operation>
</portType>
```

All aspects of this element are automatically kept consistent with other parts of the WSDL; there is no independent control of it.

B.7 <message>

Before the <portType> element, the <message> elements define the messages used in the operations. The WSDL typically contains two <message> elements for each web method. For the sample web service shown earlier in this topic, these elements are as follows:

```
<message name="AddSoapIn">
    <part name="parameters" element="s0:Add"/>
</message>
<message name="AddSoapOut">
    <part name="parameters" element="s0:AddResponse"/>
</message>
```

This element is specified as follows:

- The name attribute of a <message> element is based on the name of the web method.
- The binding style of a method is determined by the <binding> element shown previously. This determines whether a message can have multiple parts:
  - If the binding style is "document", the message has only one part by default. For example:

    ```
    <message name="AddSoapIn">
        <part name="parameters" .../>
    </message>
    ```

    If the *ARGUMENTSTYLE* parameter is "message", then the message can have multiple parts. For example:

    ```
    <message name="AddSoapIn">
       <part name="a" .../>
       <part name="b" .../>
    </message>
    ```

  - If the binding style is "rpc", the message can have multiple parts. For example:

    ```
    <message name="AddSoapIn">
        <part name="a" .../>
        <part name="b" .../>
    </message>
    ```

- The use attribute of the <soap:body> element, as specified in the <binding> element shown previously, determines the contents of a message <part> element:
  - If the use attribute is "literal", the <part> element includes an element attribute. For example:

    ```
    <part name="parameters" element="s0:Add"/>
    ```

    For another example:

    ```
    <part name="b" element="s0:b"/>
    ```

  - If the use attribute is "encoded", the <part> element includes a type attribute rather than an element attribute. For example:

    ```
    <part name="a" type="s0:ComplexNumber"/>
    ```

- The names of the elements or types to which the messages refer are determined as described in the next section.
- The namespaces to which those elements and types belong are determined as described in the next section.
- Also see WSDL Differences for One-Way Web Methods.

- If the *SOAPSESSION* parameter is 1 for the web service, that affects this element as described in WSDL Differences for InterSystems IRIS SOAP Sessions.

B.8 <types>

Before the <message> elements, the WSDL includes a <types> element, which defines the schema or schemas used by the messages. The <types> element includes one or more <schema> elements, and these define the elements, types, or both used by the web service and its clients. For the sample web service shown earlier in this topic, this element is as follows:

XML

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="https://www.mynamespace.org">
        <s:element name="Add">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="0" name="a" type="s0:ComplexNumber"/>
                    <s:element minOccurs="0" name="b" type="s0:ComplexNumber"/>
                </s:sequence>
            </s:complexType>
        </s:element>
        <s:complexType name="ComplexNumber">
            <s:sequence>
                <s:element minOccurs="0" name="Real" type="s:double"/>
                <s:element minOccurs="0" name="Imaginary" type="s:double"/>
            </s:sequence>
        </s:complexType>
        <s:element name="AddResponse">
            <s:complexType>
                <s:sequence>
                    <s:element name="AddResult" type="s0:ComplexNumber"/>
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:schema>
</types>
```

The following subsections discuss the primary variations:

- Name attributes in <types>
- Use of namespaces in <types>
- Other possible variations for the <types> section

Note: The <types> section is also influenced by the XML projections defined for all XML-enabled classes used by the web service. The XML projections determine issues such as namespace use, null handling, and handling of special characters. See *Projecting Objects to XML*.

The SoapBindingStyle and SoapBodyUse keywords affect other parts of the WSDL, which in turn determine the structure of the <types> section.

B.8.1 Name Attributes

Each <schema> element can consist of elements, types, or both, depending on the message style. Each of element or type has a name attribute, which is specified as follows:

- If the item corresponds to the web method, its name attribute equals the name of that web method (for example, Add) and cannot be changed.
- If the item corresponds to an XML-enabled class used as an argument or return value, its name attribute is determined by the XML projection of that class. For details, see *Projecting Objects to XML*.

- If the item corresponds to the response message, by default its `name` attribute has the form *method_name*`Response` (for example, `AddResponse`).

  For web methods that use document-style binding, you can override this by specifying the SoapMessageName keyword of the web method.

- For lower-level items within `<schema>`, the `name` attributes are set automatically and cannot be independently controlled.

For example, suppose that we edited the sample web method as follows:

Class Member

```
Method Add(a As ComplexNumber, b As ComplexNumber)
As ComplexNumber [ WebMethod, SoapMessageName = MyResponseMessage]
{
    Set sum = ##class(ComplexNumber).%New()
    Set sum.Real = a.Real + b.Real
    Set sum.Imaginary = a.Imaginary + b.Imaginary

    Quit sum
}
```

In this case, the `<types>` section would be as follows:

XML

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="https://www.mynamespace.org">
        <s:element name="Add">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="0" name="a" type="s0:ComplexNumber"/>
                    <s:element minOccurs="0" name="b" type="s0:ComplexNumber"/>
                </s:sequence>
            </s:complexType>
        </s:element>
        <s:complexType name="ComplexNumber">
            <s:sequence>
                <s:element minOccurs="0" name="Real" type="s:double"/>
                <s:element minOccurs="0" name="Imaginary" type="s:double"/>
            </s:sequence>
        </s:complexType>
        <s:element name="MyResponseMessage">
            <s:complexType>
                <s:sequence>
                    <s:element name="AddResult" type="s0:ComplexNumber"/>
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:schema>
</types>
```

For more information, see Controlling the Message Name of the SOAP Response. Also see *Projecting Objects to XML*.

B.8.2 Namespaces in <types>

The following parameters of the web service affect the use of namespaces within the `<types>` section:

- If it is specified, *TYPENAMESPACE* controls the `targetNamespace` attribute of the `<schema>` element.
- If *TYPENAMESPACE* is not specified, the `targetNamespace` attribute is specified by the *NAMESPACE* parameter.
- *RESPONSETYPENAMESPACE* controls the `targetNamespace` attribute of the type used by the response.
- *USECLASSNAMESPACES* controls whether `<types>` also uses the namespaces specified in the supporting type classes.

The *NAMESPACE* parameter of each XML-enabled class also affects the `<types>` element of the WSDL.

Consider the following variation of the web service shown earlier:

Class Definition

```
Class WSDLSamples.Namespaces Extends %SOAP.WebService
{

Parameter SERVICENAME = "MyServiceName";

Parameter NAMESPACE = "https://www.mynamespace.org";

Parameter RESPONSENAMESPACE = "https://www.myresponsenamespace.org";

Parameter TYPENAMESPACE = "https://www.mytypes.org";

Parameter RESPONSETYPENAMESPACE = "https://www.myresponsetypes.org";

Parameter USECLASSNAMESPACES = 1;

///  adds two complex numbers
Method Add(a As ComplexNumberNS, b As ComplexNumberNS) As ComplexNumberNS [ WebMethod ]
{
    Set sum = ##class(ComplexNumberNS).%New()
    Set sum.Real = a.Real + b.Real
    Set sum.Imaginary = a.Imaginary + b.Imaginary

    Quit sum
}

}
```

The class `WSDLSamples.ComplexNumberNS` is as follows:

Class Definition

```
///  A complex number
Class WSDLSamples.ComplexNumberNS Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter NAMESPACE = "https://www.complexnumbers.org";

Property Real As %Double;

Property Imaginary As %Double;

}
```

For the WSDL of this web service, the `<types>` part is as follows:

XML

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="https://www.mytypes.org">
        <s:import namespace="https://www.complexnumbers.org"/>
        <s:element name="Add">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="0" name="a" type="ns2:ComplexNumberNS"/>
                    <s:element minOccurs="0" name="b" type="ns2:ComplexNumberNS"/>
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:schema>
    <s:schema elementFormDefault="qualified" targetNamespace="https://www.complexnumbers.org">
        <s:complexType name="ComplexNumberNS">
            <s:sequence>
                <s:element minOccurs="0" name="Real" type="s:double"/>
                <s:element minOccurs="0" name="Imaginary" type="s:double"/>
            </s:sequence>
        </s:complexType>
    </s:schema>
    <s:schema elementFormDefault="qualified" targetNamespace="https://www.myresponsetypes.org">
        <s:import namespace="https://www.complexnumbers.org"/>
        <s:element name="AddResponse">
            <s:complexType>
                <s:sequence>
                    <s:element name="AddResult" type="ns2:ComplexNumberNS"/>
                </s:sequence>
```

```
            </s:complexType>
        </s:element>
    </s:schema>
</types>
```

B.8.3 Other Possible Variations

The following additional parameters also affect the <types> element:

- If the *INCLUDEDOCUMENTATION* parameter is 1 in the web service. the <types> section includes <annotation> elements that contain any comments that you include in the type classes. (These comments must be preceded by three slashes.)

  By default, *INCLUDEDOCUMENTATION* is 0.

  For example, suppose that we edit the sample web service to add the following:

  Class Member

  ```
  Parameter INCLUDEDOCUMENTATION = 1;
  ```

  In this case, the <types> section includes the following:

  ```
  ...
  <s:complexType name="ComplexNumber">
      <s:annotation>
          <s:documentation>A complex number</s:documentation>
      </s:annotation>
      <s:sequence>
          <s:element minOccurs="0" name="Real" type="s:double">
              <s:annotation>
                  <s:documentation>real part of the complex number</s:documentation>
              </s:annotation>
          </s:element>
          <s:element minOccurs="0" name="Imaginary" type="s:double">
              <s:annotation>
                  <s:documentation>imaginary part of the complex number</s:documentation>
              </s:annotation>
          </s:element>
      </s:sequence>
  </s:complexType>
  ...
  ```

- If the *SOAPBINARY* parameter is 1 for the web service, that affects the <types> element as described in WSDL Differences for InterSystems IRIS Binary SOAP Format.
- If the *SOAPSESSION* parameter is 1 for the web service, that affects this element as described in WSDL Differences for InterSystems IRIS SOAP Sessions.
- If specified, the SoapTypeNameSpace keyword affects this part of the WSDL. See the *Class Definition Reference*.
- If the *REQUIRED* parameter is 1 for an argument, the WSDL includes `minOccurs=1` for that argument. For information on this parameter, see Basic Requirements.
- If the SoapRequestMessage keyword is specified for a method, the name of the corresponding element is the value of the SoapRequestMessage keyword, rather than the name of the method.
- For information on the effect of the *ALLOWREDUNDANTARRAYNAME* parameter, see Projection of Collection Properties to XML Schemas in Controlling the Projection to XML Schemas.

B.9 WSDL Variations Due to Method Signature Variations

This section shows some WSDL variations caused by variations in the method signature.

B.9.1 Returning Values by Reference or as Output Parameters

To return values by reference or as output parameters, use the ByRef or Output keyword, as appropriate, within the signature of the web method. This change affects the schema and the SOAP response message.

For example, consider the following web method signatures, from methods in two different web services:

```
//from web service 1
Method HelloWorld() As %String [ WebMethod ]

//from web service 2
Method HelloWorld(ByRef myarg As %String) [ WebMethod ]
```

For the first web service, the <types> section is as follows:

XML

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="https://www.helloworld.org">
        <s:element name="HelloWorld1">
            <s:complexType>
                <s:sequence/>
            </s:complexType>
        </s:element>
        <s:element name="HelloWorld1Response">
            <s:complexType>
                <s:sequence>
                    <s:element name="HelloWorld1Result" type="s:string"/>
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:schema>
</types>
```

For the second web service, which returns the value by reference, the <types> section has a variation for the type that corresponds to the response message:

```
<types>
...
        <s:element name="HelloWorld2Response">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="0" name="myarg" type="s:string"/>
...
```

This indicates that the element contained in the <HelloWorld2Response> message is <myarg>, which corresponds to the name of the argument in the message signature. In contrast, this element is usually <methodnameResult>.

If you use the ByRef keyword instead of Output, that has the same effect on the WSDL.

For information on these keywords, see Methods.

B.10 Other WSDL Variations for InterSystems IRIS Web Services

This section discusses other possible variations for WSDLs for InterSystems IRIS web services.

B.10.1 WSDL Differences for InterSystems IRIS SOAP Sessions

If the *SOAPSESSION* parameter is 1 for the web service, that affects the WSDL as follows:

- Within the <binding> element, the <input> and <output> elements of each <operation> include the following additional subelement:

  XML

  ```
  <soap:header message="s0:IRISSessionHeader" part="CSPCHD" use="literal"/>
  ```

  For example:

  ```
  <operation name="Add">
      <soap:operation soapAction="https://www.mynamespace.org/WSDLSamples.BasicWS.Add"
  style="document"/>
      <input>
          <soap:body use="literal"/>
          <soap:header message="s0:IRISSessionHeader" part="CSPCHD" use="literal"/>
      </input>
      <output>
          <soap:body use="literal"/>
          <soap:header message="s0:IRISSessionHeader" part="CSPCHD" use="literal"/>
      </output>
  </operation>
  ```

- The WSDL includes the following additional <message> element:

  XML

  ```
  <message name="IRISSessionHeader">
      <part name="CSPCHD" element="chead:CSPCHD"/>
  </message>
  ```

- The <types> element includes the following additional item:

  XML

  ```
  <s:schema elementFormDefault="qualified" targetNamespace="https://www.intersystems.com/SOAPheaders">

      <s:element name="CSPCHD">
          <s:complexType>
              <s:sequence>
                  <s:element name="id" type="s:string"/>
              </s:sequence>
          </s:complexType>
      </s:element>
  </s:schema>
  ```

- The namespace declarations include the following additional item:

  ```
  xmlns:chead="https://www.intersystems.com/SOAPheaders"
  ```

B.10.2 WSDL Differences for InterSystems IRIS Binary SOAP Format

For an InterSystems IRIS web service that has the *SOAPBINARY* parameter specified as 1, the WSDL is enhanced as follows:

- The <binding> element includes an extension child element that indicates SOAP binary support:

  ```
  <isc:binding charset="isc_charset">
  ```

  Where *isc_charset* is the InterSystems IRIS character set (for example: Unicode, Latin1) of the InterSystems IRIS namespace for the web service.

  For example:

  ```
  <isc:binding charset="Unicode">
  ```

- Within the <types> section of the WSDL, each <complexType> element includes an extension attribute as follows:

  ```
  <complexType isc:classname="service_name:class_name" ...>
  ```

  Where *service_name* is the web service name and *class_name* is the InterSystems IRIS class name that corresponds to this complex type. For example:

  ```
  <s:complexType isc:classname="AddComplex:GSOAP.ComplexNumber" name="ComplexNumber">
    <s:sequence>
      <s:element minOccurs="0" name="Real" type="s:double"/>
      <s:element minOccurs="0" name="Imaginary" type="s:double"/>
    </s:sequence>
  </s:complexType>
  ```

- Also within the <types> section, the <element> and <simpleContent> elements include an extension attribute of the following form (as needed):

  ```
  <element isc:property="property_name" ....>
  <simpleContent isc:property="property_name" ....>
  ```

  Where *property_name* is the name of the InterSystems IRIS property that maps to that element. (Note that the WSDL includes a <simpleContent> element for a property with *XMLPROJECTION* equal to "content".)

  For example:

  ```
  <s:element minOccurs="0" name="RealType" isc:property="Real" type="s:double"/>
  ```

  This attribute is set only when the property uses the *XMLNAME* property parameter, which enables the XML name to be different from the property name. The preceding example is taken from a WSDL for a web service that uses a class that contains the following property:

  ```
  Property Real As %Double (XMLNAME = "RealType");
  ```

  The isc:property attribute allows the property names to be the same in the generated client classes as in the service classes. Current exceptions are any choice or substitutionGroup usage or wrapped elements where the type class has an *XMLNAME* parameter.

- The namespace declarations include the following additional item:

  ```
  xmlns:isc="https://www.intersystems.com/soap/"
  ```

These WSDL extensions are valid according to the XML Schema, WSDL, and WS-I Basic Profile specifications and are expected to be ignored by all conforming web client toolkits.

Note: If an InterSystems IRIS web service or web client uses the InterSystems IRIS binary SOAP format, you cannot use WS-Security or WS-Policy features with this web service or client. See *Securing Web Services*.

B.10.3 WSDL Differences for One-Way Web Methods

If the return type of a method is defined as %SOAP.OneWay, the WSDL is different from the default in the following ways:

- Within the <binding> element, the <operation> element for that method does not include an <output> element.
- Within the <portType> element, the <operation> element for that method does not include an <output> element.
- The WSDL does not include a <message> element for the response message.

C

Details of the Generated Web Service and Client Classes

For reference, this topic provides information on the classes generated by %SOAP.WSDL.Reader.

C.1 Overview of the Generated Classes

%SOAP.WSDL.Reader generates classes as follows:

- It generates the web client class, the web service class, or both, according to how you use %SOAP.WSDL.Reader. If created, the web client class extends %SOAP.WebClient. If created, the web service class extends %SOAP.WebService.

  In each of these classes, there is one web method for each web method defined in the WSDL. For a web client, the method looks like the following example:

  Class Member

  ```
  Method DemoMethod() As %String [ Final, SoapBindingStyle = document,
  SoapBodyUse = literal, WebMethod ]
  {
   Quit ..WebMethod("DemoMethod").Invoke($this,"https://tempuri.org/Demo.MyService.DemoMethod")
  }
  ```

  For a web service, the method looks like the following:

  Class Member

  ```
  Method DemoMethod() As %String [ Final,
  SoapAction = "https://tempuri.org/Demo.MyService.DemoMethod",
  SoapBindingStyle = document, SoapBodyUse = literal, WebMethod ]
  {
   // Web Service Method Implementation Goes Here.
  }
  ```

- For each complex type that is used as input or output to a web method, there is an XML-enabled class.
- For each complex type that is a component of the preceding types, there is an XML-enabled class.

  %SOAP.WSDL.Reader does this recursively so that the properties of the least complex type are simple datatype properties, which correspond directly to XSD types.

In these classes, the system specifies the class and method keywords and parameters as needed to specify encoding and binding style, namespace assignment, and other items.

C.2 Keywords That Control Encoding and Binding Style

The generated web client and web service classes contain settings for the following keywords, which control the encoding and message style needed to work with the given WSDL:

- SoapBodyUse class keyword
- SoapBodyUse method keyword
- SoapBindingStyle class keyword
- SoapBindingStyle method keyword

You should not modify these keywords, because the web client or web service would no longer obey the WSDL. For details on them, see the *Class Definition Reference*.

C.3 Parameters and Keywords That Control Namespace Assignment

The generated classes contain parameters and keywords to control namespace assignments. The following subsections discuss namespaces for messages and namespaces for types.

You should not modify these values, because the web client or web service would no longer obey the WSDL. For details on SoapNameSpace and SoapTypeNameSpace, see the *Class Definition Reference*.

C.3.1 Namespaces for the Messages

The following values control the namespaces used for the SOAP messages:

Table III–1: Namespaces for SOAP Messages Sent by Web Client or Service

Item	Value Given by %SOAP.WSDL.Reader
NAMESPACE (class parameter)	Namespace of the request messages, if all request messages use the same namespace.
SoapNameSpace (method keyword)	Namespace of a given request message, if request messages use different namespaces.
RESPONSENAMESPACE (class parameter)	Namespace of the response messages. If this is not specified, the response messages are in the namespace given by the *NAMESPACE* parameter. Note that the SoapNameSpace keyword has no effect on the namespaces of the response messages.

C.3.2 Namespaces for the Types

The message types are assigned to namespaces as follows:

Table III–2: Namespaces for Types Used By Web Clients and Web Services

Item	Value Given by %SOAP.WSDL.Reader
TYPENAMESPACE (class parameter)	%SOAP.WSDL.Reader sets this parameter if all methods refer to types in the same namespace.
RESPONSETYPENAMESPACE (class parameter)	%SOAP.WSDL.Reader sets this parameter if the WSDL uses document-style binding and the response messages use types in a different namespace than the request messages. This parameter applies to all methods in the class. Note that all response types are assumed to be in the same namespace as each other.
SoapTypeNameSpace (method keyword)	Value of the `targetNamespace` attribute of the `<s:schema>` element. %SOAP.WSDL.Reader sets this keyword per method if methods use types from different namespaces. This keyword does not override the *RESPONSETYPENAMESPACE* parameter.

C.4 Creation of Array Properties

By default, %SOAP.WSDL.Reader creates array-type properties in certain scenarios. You can use the **NoArrayProperties** option to create properties with a different structure if needed.

Specifically, consider a WSDL that includes the following types:

```
<s:complexType name="Obj">
    <s:sequence>
        <s:element minOccurs="0" name="MyProp" type="test:Array"/>
    </s:sequence>
</s:complexType>
<s:complexType name="Array">
    <s:sequence>
        <s:element maxOccurs="unbounded" minOccurs="0" name="Item" nillable="true" type="test:Item"/>
    </s:sequence>
</s:complexType>
<s:complexType name="Item">
    <s:simpleContent>
        <s:extension base="s:string">
<s:attribute name="Key" type="s:string" use="required"/>
        </s:extension>
    </s:simpleContent>
</s:complexType>
```

By default, %SOAP.WSDL.Reader generates the following class:

Class Definition

```
Class testws.Obj Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter ELEMENTQUALIFIED = 1;

Parameter NAMESPACE = "https://testws.org";

Parameter XMLNAME = "Obj";

Parameter XMLSEQUENCE = 1;

Property MyProp As array Of %String(MAXLEN = "", XMLITEMNAME = "Item",
XMLKEYNAME = "Key", XMLNAME = "MyProp", XMLPROJECTION = "COLLECTION");

}
```

If you specify **NoArrayProperties** as 1 when you generate the classes, the property `MyProp` in generated `Obj` class is defined as follows instead:

Class Member

```
Property MyProp As list Of testws.Item(XMLITEMNAME = "Item", XMLNAME = "MyProp", XMLPROJECTION =
"COLLECTION");
```

This property refers to the following class, which %SOAP.WSDL.Reader also generates:

Class Definition

```
Class testws.Item Extends (%SerialObject, %XML.Adaptor)
{

Parameter ELEMENTQUALIFIED = 1;

Parameter NAMESPACE = "https://testws.org";

Parameter XMLNAME = "Item";

Parameter XMLSEQUENCE = 1;

Property content As %String(MAXLEN = "", XMLNAME = "content", XMLPROJECTION = "CONTENT");

Property Key As %String(MAXLEN = "", XMLNAME = "Key", XMLPROJECTION = "ATTRIBUTE")
[ Required, SqlFieldName = _Key ];

}
```

C.5 Additional Notes on Web Methods in the Generated Class

This section contains additional notes on the web methods in the generated web client classes.

- InterSystems IRIS® data platform ensures that each method name is shorter than the limit on method names *and* is unique. (For information on the length of method names, see General System Limits,) This means that if the method names in the WSDL are longer than this limit, the names of the generated web methods are not the same as in the WSDL.

  The algorithm used is not documented and is subject to change without notice.

- If the `soapAction` attribute equals `""` for a given method in the WSDL (which is valid), it is necessary to make one of the following changes to the generated client class so that this method will work:

  – Set the *SOAPACTIONQUOTED* parameter equal to 1.

  – Edit the web method in the generated client class. Originally it includes the following contents:

  ```
  Quit ..WebMethod("HelloWorld").Invoke($this,"")
  ```

  Edit this to the following:

  ```
  Quit ..WebMethod("HelloWorld").Invoke($this,"""")
  ```

  Alternatively, edit the WSDL before generating the web client. If you do so, edit the `soapAction` attribute to equal `""""` instead of `""`