Creating Web Services and Web Clients in Caché

Version 2018.1
2020-11-13

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

Creating Web Services and Web Clients in Caché
Caché Version 2018.1 2020-11-13

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

About This Book

This book describes, to programmers, how to create Caché web services and web clients. It includes the following sections:

- Introduction to Caché Web Services and Web Clients
- Creating Web Services
- SOAP Message Variations
- Creating Web Clients
- SOAP Fault Handling
- Using MTOM for Attachments
- Using SOAP with Attachments
- Adding and Using Custom Header Elements
- Adding and Using WS-Addressing Header Elements
- SOAP Session Management
- Using the Caché Binary SOAP Format
- Using Datasets in SOAP Messages
- Fine-Tuning a Caché Web Service
- Fine-Tuning a Caché Web Client
- Troubleshooting Caché SOAP Problems
- Summary of Web Service URLs
- Details of the Generated WSDLs
- Details of the Generated Classes

For a detailed outline, see the table of contents.

For more information, try the following sources:

- *Securing Caché Web Services* describes how to add security elements to Caché web services and web clients.
- *Projecting Objects to XML* describes how to project Caché objects to XML and how to control that projection.
- *Using Caché XML Tools* describes how to use Caché tools to work with XML-enabled objects and with general XML documents and DOMs (Document Object Model).
- *Using Caché Server Pages (CSP)* describes how to create web applications that consist of CSP pages.
- *Using Caché Internet Utilities* includes information on using HTTP responses.

For general information, see the *InterSystems Documentation Guide*.

1

Introduction to Caché Web Services and Web Clients

Caché 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 Caché and is available on every platform supported by Caché.

This chapter introduces the following:

- An introduction to Caché web services
- An introduction to Caché web clients
- Additional features you can add to your web services and clients
- Standards supported by Caché web services and clients
- Key points about the SAX parser

1.1 Introduction to Caché Web Services

This section introduces Caché web services.

1.1.1 Creating Caché Web Services

In Caché, 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 using the Caché SOAP Wizard to read an existing WSDL document and generate 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 Web Service as Part of a Web Application

A Caché web service class inherits from the %SOAP.WebService class, which in turn inherits from %CSP.Page.

Because of this fact, a Caché web service resides within a *web application* that you configure within the Management Portal. For details, see the chapter "Applications" in the *Caché Security Administration 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 a web server, for your convenience. This WSDL complies with the Basic Profile 1.0 established by the WS-I (Web Services Interoperability Organization). In Caché, 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," in the next chapter.

1.1.4 Web Service Architecture

To understand how a Caché 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 CSP Gateway.
3. The CSP 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 a Caché 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 Caché SOAP support. The SOAP specification does not define a standard for sessions. However, Caché SOAP support provides a proprietary Caché 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 Caché 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.

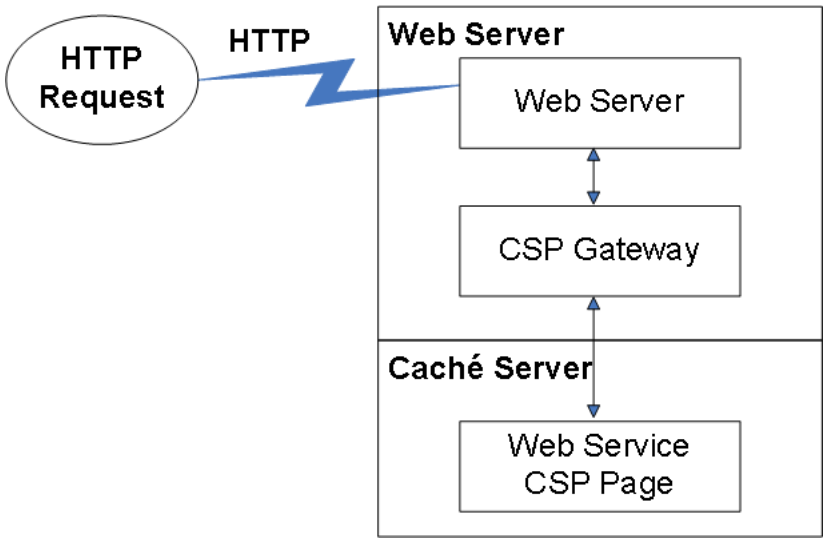
   The SOAP action from the HTTP header is used here to determine the method and hence the request object.

   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 Caché 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:



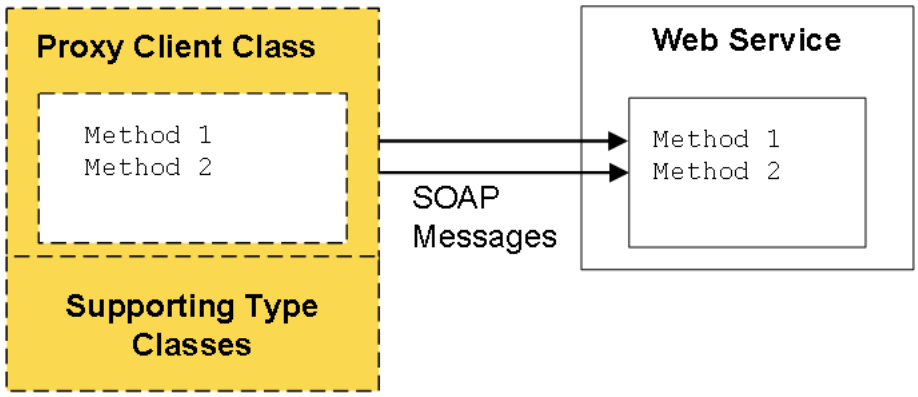
1.2 Introduction to Caché Web Clients

This section introduces Caché web clients.

1.2.1 Creating Caché Web Clients

In Caché, you create a web client by using the Caché SOAP Wizard to read an existing WSDL document. The wizard generates 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.



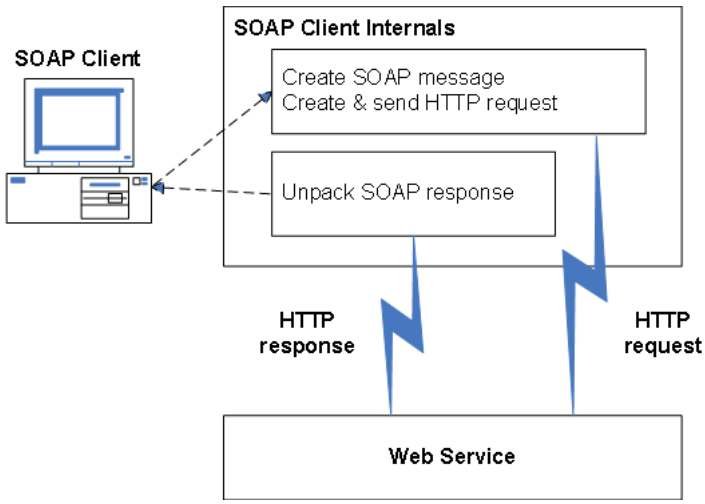
Typically you do not customize the generated classes. You instead create additional classes that control the behavior of your web client and invoke its proxy methods.

1.2.2 Web Client Architecture

To understand how a Caché 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 Caché web services and web clients:

- Session support. As noted earlier, although the SOAP specification does not define a standard for sessions, you can create client-server SOAP sessions by using the CSP infrastructure and the session support provided by the %SOAP package.
- 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 Caché 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 Standards Supported in Caché

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

Additional standards are listed in *Securing Caché Web Services*.

1.4.1 Basic Standards

Caché 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. Caché web services produce WSDL documents that comply with the Basic Profile 1.0 established by the WS-I (Web Services Interoperability Organization). However, Caché web *clients* do work for more general WSDL documents.

  See "WSDL Support in Caché."
- UDDI version 1.0 with client access only (no repository provided). See http://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.

For information on XML standards supported in Caché, see the book *Using Caché XML Tools*.

1.4.2 WSDL Support in Caché

Caché 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 Caché web services do not include headers. Also, the web services that you can create in Caché 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

The Caché SOAP Wizard 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).
- The Caché SOAP Wizard does not process headers of the WSDL.

The SOAP Wizard 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 book.

1.5 Key Points about the SAX Parser

The Caché SAX parser is used whenever Caché 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" in *Using Caché XML Tools*.

2

Creating SOAP Web Services

This chapter describes the basics of how to create a web service in Caché. It includes the following topics:

- Overview of Caché web services
- Basic requirements for web services
- Example
- How to create web services
- How to specify parameters of the web service
- Caché catalog and test pages for the web service
- How to view the generated WSDL

See the first appendix for a table that summarizes the URLs related to your web service.

2.1 Overview of Caché Web Services

To create a web service in Caché, 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.

This class is derived from the %CSP.Page class. Because of this, every web service class can respond to HTTP requests. Thus, the %SOAP.WebService class implements methods that respond to HTTP events to do the following:

- Publish the WSDL document for the web service as an XML document.
- Publish a human-readable catalog page (using HTML) describing the web service and its methods. The descriptions on this page show the comments included in the class definition.

2.2 Basic Requirements

To create and publish a web service in Caché, create and compile a Caché class that meets the following basic requirements:

- The class must extend %SOAP.WebService.
- The class must define the *SERVICENAME* parameter. Caché 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 chapters) 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*. Caché 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 the chapter "Methods" in *Using Caché Objects*.)

- 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:

  ```
  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 Caché Web Services," later in this chapter).

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 Caché 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:

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

    Or, in the case of an input argument:

    ```
    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 Caché 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 Caché 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 Cache or one is based on .NET, you can use the specialized result set class, %XML.DataSet, which is discussed in the chapter "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; there is an example in SOAP.Demo in the SAMPLES namespace.

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:

```
/// 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 = "http://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 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>
    <Forecast xmlns="http://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 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>
    <ForecastResponse xmlns="http://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 chapter
- By using the Web Service Wizard
- By using the SOAP Wizard with an existing WSDL
- By subclassing one or more Caché web services

2.4.1 Using the Web Service Wizard

The Web Service Wizard generates a simple stub.

1. Click **File > New**.

   This displays the **New** dialog box.

2. Click the **General** tab.

3. Click **New Web Service** and then click **OK**.

   This displays a wizard.

4. Enter values for the package name, class name, and web service name. These are required.
5. Optionally edit the namespace URI (or change this initial value later). This is the XML namespace, not the Caché namespace.
6. Optionally type a list of method names, on separate lines.
7. Click **OK**.

Now, you have a new web service class that contains stubs for the web methods. For example:

```
/// 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 = "http://tempuri.org";

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

/// TODO: add arguments and implementation.
/// Forecast
Method Forecast() As %String [ WebMethod ]
{
   ;Quit "Forecast"
}

}
```

2.4.2 Using the SOAP Wizard with an Existing WSDL

In some cases, the WSDL has been designed already and it is necessary to create a web service that matches the WSDL; this is known as "WSDL-first development." In Caché, there are three steps to this development:

1. Use the SOAP Wizard to read the WSDL and to generate the web service and all supporting classes.

   This wizard can also generate web client classes (which is more common).

   For information on using this wizard, see "Using the SOAP Wizard," later in this book. Follow the steps described in that section and also select the **Create Web Service** option within the wizard.

   Or use the %SOAP.WSDL.Reader class as described in "Using the %SOAP.WSDL.Reader Class."

2. Examine the generated classes to see if you need to change any %String values in the method signatures.

   When the wizard reads a WSDL, it assumes that any string-type input or output can be represented in Caché as %String, which is not always true. Some strings might exceed the Caché 32 KB limit for strings.

   See "Adjusting the Generated Classes for Long Strings," later in this book.

3. Edit the methods in the generated web service so that they perform the desired actions.

   Each method is initially a stub like the following example:

   ```
   Method Add(a As Test.ns2.ComplexNumber, b As Test.ns2.ComplexNumber) As Test.ns2.ComplexNumber
   [ Final, SoapAction = "http://www.mynamespace.org/GSOAP.AddComplexWS.Add",
   SoapBindingStyle = document, SoapBodyUse = literal, WebMethod ]
   {
    // Web Service Method Implementation Goes Here.
   }
   ```

   The wizard includes compiler keywords such as Final and SoapBindingStyle. You should not change the values of these keywords.

If the WSDL includes WS-Policy elements, the wizard also generates a configuration class for the web service. The default configuration class name is the web service name, with `Config` appended to it. For information on WS-Policy, see *Securing Caché Web Services*.

2.4.3 Subclassing an Existing Caché Web Service

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

```
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 use the SOAP wizard to 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.

Caché 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 `"http://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 `"http://tempuri.org"`.

For a Caché web service, there is no way to put request messages in different namespaces. A Caché web client, however, does not have this limitation; see "Namespaces for the Messages," later in this book.

RESPONSENAMESPACE

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

For a Caché web service, there is no way to put response messages in different namespaces. A Caché web client, however, does not have this limitation; see "Namespaces for the Messages," later in this book.

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 `"http://tempuri.org"`).

For a Caché web service, there is no way to put the request message types in different namespaces. A Caché web client does not have this limitation; see "Namespaces for Types," later in this book.

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 a Caché web service or a Caché 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," later in this book.

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

2.6 About the Catalog and Test Pages

When you compile a web service class, the class compiler creates a convenient catalog page that you can use to examine the web service. This catalog page provides a link to a simple test page.

To see these CSP pages:

1. In Studio, display the web service class.
2. Click **View > Web Page**.

The catalog page is immediately displayed. Its URL is constructed as follows:

```
base/csp/app/web_serv.cls
```

Here *base* is the base URL for your web server (including port if necessary), */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:

```
http://localhost:57772/csp/samples/MyApp.StockService.cls
```

2.6.1 Access to These Pages

These CSP pages are part of a Caché web application, as noted in the previous chapter. If there is no web application for the namespace you are using, you cannot access these pages. 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/"`. This access is enabled by default for the /csp/samples web application.

Also, 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 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 WSDL; for information, see the section "Viewing the WSDL," later in this chapter. 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 that is discussed later in this chapter.
- It accepts only simple, literal inputs, so you cannot use it to call methods whose arguments are objects, collections, or datasets.

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

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 Caché" in the first chapter.

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 Caché may have a slightly different form depending on the version of Caché. 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

To view the WSDL for the web service, use the following URL:

```
base/csp/app/web_serv.cls?WSDL
```

Here *base* is the base URL for your web server (including port if necessary), */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:

```
http://localhost:57772/csp/samples/MyApp.StockService.cls?WSDL
```

The browser displays the WSDL document, for example:

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 chapter discusses the primary variations for SOAP messages and how they are specified in Caché web services and clients.

For a Caché 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 by using the SOAP Wizard, the system sets the values as required by the WSDL. On some occasions, however, you may find it necessary to choose a specific message variation.

This chapter discusses the following:

- Overview of message variations
- How the message variation is determined for a given method
- Examples of the message variations

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 Caché 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 Caché) 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 the book *Projecting Objects to XML*.

3.2 How Message Variation Is Determined

For a Caché 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 a Caché 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 use the SOAP wizard to generate a web service or client class, the wizard sets the values for these keywords and parameter as appropriate for the WSDL from which you started.

Important: For a web service that you create manually, the default values are usually suitable.

When you use the SOAP Wizard to create a web client or service from a WSDL, Caché sets these keywords as appropriate for that WSDL. If you modify the values, your web client or service may no longer work.

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 the section "<message>" in the appendix "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 Caché web services).

```
<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>
    <MyMethod xmlns="http://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.

```
<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>
      <A xmlns="http://www.demoservice.org">stringA</A>
      <B xmlns="http://www.demoservice.org">stringB</B>
      <C xmlns="http://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:

```
<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'
                   xmlns:SOAP-ENC='http://schemas.xmlsoap.org/soap/encoding/'
                   xmlns:tns='http://www.demoservice.org'
                   xmlns:types='http://www.demoservice.org'>
  <SOAP-ENV:Body SOAP-ENV:encodingStyle='http://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:

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

```
<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'
                   xmlns:tns='http://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 chapter describes how to create and use web clients in Caché:

- An overview of the SOAP Wizard
- How to use the SOAP Wizard
- How to generate the client classes programmatically
- When and how to modify the generated classes
- How to create a wrapper for the generated web client
- How to adjust basic properties of the web client instance
- How to access the HTTP response that the Caché web client receives

For information on logging SOAP calls to your Caché web clients, see "Caché SOAP Log," later in this book.

Note: For a Caché 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 the chapter "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 Caché 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 the same book. 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 Overview of the SOAP Wizard

To create a Caché web client, you can use the SOAP Wizard in Studio or the corresponding class method provided by Caché. In either case, the input is a WSDL document. The tools generate a web client class and all needed supporting classes.

You can use the tool with nearly any WSDL document; see "WSDL Support in Caché" in the first chapter.

You can provide either the URL or the file path for the WSDL.

Note: If the WSDL indicates support for both SOAP 1.1 and SOAP 1.2, then the SOAP Wizard generates two sets of classes, if needed.

4.2 Using the SOAP Wizard

Studio provides the SOAP Wizard, which enables you to generate a client for a given web service, given the WSDL of that service.

Note: When using templates such as the SOAP Wizard, Studio will use a proxy server, if it is enabled. For information on specifying the proxy server and port, see "Using a Proxy Server" in *Using Caché Internet Utilities*.

To use the SOAP wizard:

1. Examine the WSDL and check for the following items:
   - Do the `<message>` elements contain multiple parts?
   - Do the types used by the response messages belong to multiple namespaces?

   In either case, if the answer is yes, then you will need to select the option **Use unwrapped message format for document style web methods**, which is described in a later step.
2. Click **Tools > Add-ins > SOAP Wizard**.
3. On the first screen:
   a. Click either **URL** or **FILE**, depending on the location of the WSDL.
   b. Enter the location of the WSDL. Type either the entire URL or the complete path and filename for the WSDL.
   c. If the WSDL is at a location that uses SSL (that is, if the URL starts with `https`), then select a suitable SSL configuration from the **SSL Configuration** dropdown list. For information on creating and managing SSL/TLS configurations, see the chapter "Using SSL/TLS with Caché" in the *Caché Security Administration Guide*.

      Important: This wizard specifies an SSL configuration to use when accessing the WSDL. This field is not used in any other way.

   d. If the WSDL is at a location that uses SSL, the wizard 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, clear the check box **When making an SSL connection check that the server identity in the server certificate matches the name of the system being connected to**.

4. Click **Next**.

   The wizard then attempts to access the WSDL and display it so that you can verify that you have chosen the correct one.

   If the wizard cannot access the WSDL, it displays a screen that displays the error and provides options that you can use to provide a username and password. (Many WSDLs are at URLs that are protected by a username and password, and this is a common reason for failing to access the WSDL.) In this scenario, first do one of the following:

   - To provide a Caché username and password, select **CACHÉ user name and password**
   - To provide a username and password for HTTP Basic authentication, select **HTTP authentication user name and password**

   Then enter values in the **Username** and **Username** fields and press **Try Again**. Note that the wizard does not save your entries for these items.

5. On the screen that displays the WSDL, specify options as follows:

   - **Create Client for Web Service** — Select this if you want the wizard to generate web client classes.
   - **Create Web Service** — Select this if you want the wizard to generate web service classes.
   - **Compile generated classes** — Select this if you want the wizard to compile the classes after generating them.
   - **Compile flags** — Specify flags to control how the compiler behaves. To get information about the compiler flags, execute the following command:

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

   - **Class Type** — Choose a type to use for the generated type classes:
     - **Persistent** — The type classes should inherit from %Persistent. Any collections should be defined as lists.
     - **Persistent using one-many relationships** — The type classes should inherit from %Persistent and any collection properties should be defined as one-to-many relationships.
     - **Persistent using indexed one-many relationships** — The same as the preceding item except that Caché also defines an index for the relationship.
     - **Persistent using parent-child for relationships** — The type classes should inherit from %Persistent and any collection properties should be defined as parent-child relationships.
     - **Serial** — The type classes should inherit from %SerialObject.
     - **Registered** — The type classes should inherit from %RegisteredObject.
   - **Add %OnDelete method to classes in order to cascade deletes** — This option is displayed if you choose any persistent class type. If you select this option, when the wizard generates class definitions, it includes an implementation of the **%OnDelete()** callback method. The generated **%OnDelete()** method deletes all persistent objects that are referenced by the class.

     Note: Do not use this option if you select **Persistent using parent-child for relationships**.

     If you modify the generated classes, be sure to modify the **%OnDelete()** callback method as needed.
   - **Proxy Class Package** — Type a package name for the web client. This is also used as the base package for any generated type classes. The default package name is the service name.

     If this package is the same as an existing package, by default the tool overwrites any existing classes that have the same name.

6. If you have an Ensemble installation, optionally provide values for the following additional fields:

- **Create Business Operation** — Select this to generate an Ensemble business operation and related request and response message classes.
- **Business Operation Package** — Specify the package for the business operation class.
- **Request Package** — Specify the package for the request message class.
- **Response Package** — Specify the package for the response message class.

For information on Ensemble, see the Ensemble documentation.

7. Click **Next**. The wizard then displays a screen like the following:
8. Specify the following options:
   - **Add NAMESPACE class parameter**, which affects how generated type classes are assigned to namespaces:
     - If the WSDL explicitly indicates the namespace to which a given type belongs, **Add NAMESPACE Class Parameter** is selected and grayed out. In this case, the generated type class will include the *NAMESPACE* class parameter set equal to that namespace.
     - If the WSDL does not indicate the namespace for a given type, you can select or clear **Add NAMESPACE Class Parameter**.

       If you select this option, the generated type class will include the *NAMESPACE* class parameter set equal to the namespace of the web service.

       If you clear this option, the generated type class will not include the *NAMESPACE* class parameter.
   - **Use unwrapped message format for document style web methods**, which affects the signature of the methods in the generated web client. See the appendix "Details of the Generated Classes" and see "Using Caché Web Client Classes," later in this chapter.

     Important: If the WSDL that you are using contains `<message>` elements with multiple parts, be sure to select this option. If you do not, the wizard fails and displays a message like 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.
     ```

     Similarly, if the types used by the response messages are in different namespaces than each other, be sure to select this option.

     This option affects only methods that have SoapBindingStyle equal to `"document"`.
   - **Do not create array properties**, which controls whether the wizard generates array properties. If you select this option, the wizard does not generate array properties but instead generates another form. See "Creation of Array Properties" in the appendix "Details of the Generated Classes."
   - **Generate XMLNIL property parameter for nillable elements**, which controls whether the wizard specifies 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 you select this option, the wizard 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" in *Projecting Objects to XML*.
   - **Generate XMLNILNOOBJECT property parameter for nillable elements**, which controls whether the wizard specifies 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 you select this option, the wizard 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" in *Projecting Objects to XML*.

- **Set XMLSEQUENCE parameter to 0**, which controls how the wizard sets the *XMLSEQUENCE* class parameter in the generated classes.

  By default, the wizard 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" in *Projecting Objects to XML*.

- **Generate XMLIGNORENULL parameter set to 1**, which controls whether the wizard specifies the *XMLIGNORENULL* class parameter in the generated classes. If you select this option, the wizard 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" in *Projecting Objects to XML*.

- **Use Streams for Binary**, which controls how the wizard handles any element of type `xsd:base64Binary`. If you select this option, the corresponding property is of type %Stream.GlobalBinary. If this option is clear, the property is of type %xsd.base64Binary instead.

  Note that the wizard ignores any *attributes* of type `xsd:base64Binary`.

- **Specify SECURITYIN class parameter**, which controls the class parameter *SECURITYIN* 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" in *Securing Caché Web Services*.

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

9. Optionally edit the package names. The wizard uses the following rules for package names:

   - The packages specified in **Web Client Package** or **Web Service Package** contain the main generated web client or web service class. These values are initialized by your entry in the previous page of the wizard.

   - If the wizard generates a policy class as well (because the WSDL contains WS-Policy information), that class is in the same package by default. To specify a different package, specify a value for **Configuration Sub-Package**.

     By default, the class name is the web client or service name, with `Config` appended to it. If you specify a value for **Configuration Sub-Package**, then this generated class has the same name as the web client or service (but is in the given subpackage instead).

     For information on WS-Policy, see *Securing Caché Web Services*.

   - For any supporting classes that are generated, the wizard detects all the namespaces used in the WSDL. By default, it organizes the generated classes into packages, with one package for each namespace.

     Optionally edit these package names.

     Not all namespaces necessarily correspond to generated classes. For example, the WSDL used in this example uses the namespaces `http://schemas.xmlsoap.org/wsdl`, `http://schemas.xmlsoap.org/wsdl/mime`, and `http://schemas.xmlsoap.org/wsdl/soap`. No generated classes are in these namespaces in this case, and the corresponding packages are not generated.

10. Click **Next**. The wizard generates and compiles the classes and displays a list of these classes.

11. Click **Finish**.

Tip: If you use a WSDL URL and there are problems with the wizard, save the WSDL as a file and try again, using that file as input.

If the WSDL contains references to externally defined entities, the wizard attempts to resolve those; for this task, the timeout period is 10 seconds.

For properties of these classes, if the corresponding element in the schema has a name that starts with an underscore (_), the name of the property starts with a percent sign (%).

4.3 Generating the Client Classes Programmatically

You can instead generate the client classes programmatically by using the %SOAP.WSDL.Reader class.

The resulting generated classes and their organization may not be the same as if you used the SOAP wizard, which provides greater control over the packages for the generated classes.

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

   Note that if the WSDL is at a location that uses SSL, %SOAP.WSDL.Reader (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 the chapter "Sending HTTP Requests" in *Using Caché Internet Utilities*.
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, Caché 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:

```
Parameter XMLKEEPCLASS = 1;
```

The following shows an example Terminal session:

```
set r=##class(%SOAP.WSDL.Reader).%New()
GSOAP>set url="http://localhost:57772/csp/gsop/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 Caché web clients or third-party web clients, see "Using a Password-Protected WSDL URL," later in this book.

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.

4.4 Modifying the Generated Client Classes

After you generate a Caché 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 the section "Additional Notes on Web Methods in the Generated Class" and the chapter "Fine-Tuning a Caché Web Client."

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.

4.4.1 Adjusting the Generated Classes for Long Strings

In some cases, you might need to edit the generated client class to accommodate long strings or long binary values.

When the SOAP Wizard reads a WSDL, it assumes that any string-type input or output can be represented in Caché as %String, which is not always true. Some strings might exceed the Caché 32 KB limit for strings, and there is no information in the WSDL to inform the SOAP Wizard of this.

Similarly, it assumes that any input or output with XML type base64Binary can be represented in Caché as %xsd.base64Binary), which is not always true, due to the same long string limitation. There is no information in the WSDL to inform the SOAP Wizard that this input or output could exceed the long string limit.

In either case, if you have enabled long string operations in Caché, your web client works.

If you have not enabled long string operations in Caché, however, 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. If you expect long values for these inputs and outputs, 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, greater than 32 KB. If you use the SOAP Wizard to generate the web client class, the web client class originally looks something like this:

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

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

}
```

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

```
Method WriteIt() As %GlobalCharacterStream
[Final,SoapBindingStyle=document,SoapBodyUse=literal,WebMethod]
{
 Quit ..WebMethod("WriteIt").Invoke(##this,"http://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 Caché 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 32 KB in the `<ContainerPart>` element, and if you have not enabled long string operations, your web client throws an error. To avoid this error, change the type of the `ContainerPart` property to %GlobalCharacterStream.

4.4.2 Other Adjustments

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

```
Parameter LOCATION = "http://localhost:57772/csp/gsop/GSOP.AddComplexWS.cls";
```

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

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

4.5 Using the Generated Web Client Classes

As noted in the previous section, after you generate a Caché 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 in this chapter as well as the chapter "Fine-Tuning a Caché Web Client."
3. Invoke the methods of the web client as needed.
4. Perform client-side error handling. See the chapter "SOAP Fault Handling."
5. Optionally examine the HTTP response received by the web client, as described later in this chapter.

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
```

4.5.1 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 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="http://localhost:8080/csp/gsop/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.

4.5.2 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 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.

4.6 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.

4.6.1 Changing the Endpoint for the Web Client

The SOAP Wizard automatically sets the endpoint for the web client by setting the *LOCATION* parameter of the web client. By default, it sets this parameter equal to 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:

```
Parameter LOCATION = "http://localhost:57772/csp/gsop/GSOP.AddComplexWS.cls";
```

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

```
   Set client.Location="http://localhost:8080/csp/gsop/GSOP.AddComplexWS.cls"
```

Note: If the WSDL does not specify the location of the web service, the SOAP Wizard does not specify the *LOCATION* parameter of the web client. 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.

4.6.2 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 the chapter "Using SSL/TLS with Caché" in the *Caché Security Administration 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 a Caché web client to use a proxy server, see the chapter "Fine-Tuning the Web Client."

4.6.3 Specifying the SOAP Version

The SOAP Wizard 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.

4.6.4 Other Adjustments

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

4.7 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 a Caché multidimensional array that contains any data in the HTTP response.
- StatusCode, StatusLine, and ReasonPhrase provide status information.

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

5

SOAP Fault Handling

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

- How Caché web services handle errors by default
- How to return custom SOAP faults in a Caché web service
- How to create fault objects manually
- How to add WS-Addressing header elements when faults occur
- How to add custom header elements when faults occur
- How to handle errors and SOAP faults in a web client

For a detailed discussion of error processing, see the chapter on errors in the book *Using Caché ObjectScript*.

Note that the *SOAPPREFIX* parameter affects the prefix used in any SOAP faults; see "Specifying the SOAP Envelope Prefix" in the chapter "Fine-Tuning a Caché Web Service."

5.1 Default Fault Handling in a Web Service

By default, when your Caché 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='http://www.myapp.org' >
       <text>ERROR #5002: Cache error: <DIVIDE>zDivide^FaultEx.Service.1</text>
     </error>
  </detail>
</SOAP-ENV:Fault>
</SOAP-ENV:Body>
```

5.2 Returning Custom SOAP Faults in a Caché 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 chapter.

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:

```
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," later in this chapter.
- *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. Caché 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 book 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 Caché handles the SOAP versions of request messages, see "Specifying Parameters of the Web Service," earlier in this book.

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.

Macro	SOAP Version(s)	When to Use This Macro
$$$FAULTSender	1.2	When the sender made an incomplete, incorrect, or unsupported request.
$$$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:

```
<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'
xmlns:flt="http://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

Caché 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. Caché 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," earlier in this chapter.

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 book 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:

```
set fault.faultPrefixDefinition = "xmlns:FLT=""http://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="http://www.w3.org/2003/05/soap-envelope"
xmlns:flt="http://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. Caché 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 book 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:

```
 set fault.faultPrefixDefinition = "xmlns:FLT=""http://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," earlier in this chapter.

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 Caché 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 Caché 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 the chapter "Adding and Using Custom Header Elements."

2. Within the fault handling of your web service (as described earlier in this chapter), 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 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:

```
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='http://schemas.xmlsoap.org/soap/envelope/'
xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xmlns:s='http://www.w3.org/2001/XMLSchema' xmlns:flt="http://myfault.org" >
  <SOAP-ENV:Header>
<CustomHeaderElement xmlns:hdr="http://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 a Caché Web Client

In a Caché web client, you can use the **TRY-CATCH** mechanism or the older **$ZTRAP** mechanism.

In either case, when a Caché web client receives an error, Caché 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**:

```
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=$$$ERROR($$$CacheError,$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 **$ZTRAP** mechanism. In this case, when a Caché web client receives an error, control is transferred to the label indicated by the **$ZTRAP** special variable (if that label is defined).

```
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 a Caché 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 Caché support for MTOM (Message Transmission Optimization Mechanism).

This chapter discusses the details:

- An overview of attachments and SOAP message packaging
- Default behavior of Caché web services and web clients (with regards to binary data)
- How to force a Caché web service to send responses as MTOM packages
- How to force a Caché web client to send requests as MTOM packages
- How to control the details of the MTOM package
- An example that sends MTOM messages between a web service and a web client

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

You can also configure Caché web services and web clients to use gzip to compress their messages after performing any packaging; see the chapters "Fine-Tuning a Caché Web Service" and "Fine-Tuning a Caché Web Client."

6.1 Attachments and SOAP Message Packaging

Attachments are generally used to carry binary data. Caché 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 Caché 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 Caché 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 "Standards Supported in Caché," in the first chapter of this book.

- 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 the next chapter, "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/gsop/;
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='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>
  <DownloadResponse xmlns="http://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/gsop/;
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='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>
<DownloadResponse xmlns="http://www.filetransfer.org">
<DownloadResult>
   <Filename>sample.pdf</Filename>
   <IsBinary>true</IsBinary>
   <BinaryContents>
      <xop:Include href="cid:1.B1150656.EC8A.4B5A.8835.A932E318190B"
               xmlns:xop="http://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/gsop/;
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='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><DownloadBinaryResponse xmlns="http://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 Caché Web Services and Web Clients

By default, a Caché 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, a Caché 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 a Caché web service to send every response as an MTOM package. To do, do any of the following:

- In your Caché web service class, set the *MTOMREQUIRED* parameter to 1.
- In your Caché 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," later in this book.
- 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 Caché 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 a Caché 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 a Caché web client to send every request as an MTOM package. To do, do either of the following:

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

- In your Caché 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 Caché 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 Caché 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 a Caché 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:

```
/// 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 = "http://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:

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

 .attachment)
}
```

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

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

```
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="http://localhost:8080/csp/gsop/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 Caché web clients and web services, you can add and use attachments to SOAP messages by using the Caché support for SOAP with Attachments, instead of using the Caché MTOM support, as described in the previous chapter.

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

- How to send attachments from a web service or web client via SOAP with Attachments
- How to access sent attachments within a web service or web client
- An example that shows attachments sent in both directions

For a link to the specifications for the SOAP with Attachments standard, see "Standards Supported in Caché," in the first chapter of this book.

7.1 Sending Attachments

When you use the Caché 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 a Caché 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 GSOAP.FileTransferWS Extends %SOAP.WebService
{

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

///  SOAP namespace for the web service
Parameter NAMESPACE = "http://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:

```
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 chapter describes how to add and use custom SOAP header elements.

This chapter discusses the following:

- Introduction to SOAP headers in Caché
- How to define a custom header element
- How to add a header element to a SOAP message
- How to specify the supported header elements in general
- How to specify the supported header elements in an XData block
- How to specify the supported header elements in the SOAPHEADERS parameter
- How to use a header element received in a SOAP message

For information on adding header elements when faults occur, see the chapter "SOAP Fault Handling," earlier in this book.

A later chapter discusses WS-Addressing header elements. For information on WS-Security header elements, see *Securing Caché Web Services*.

8.1 Introduction to SOAP Header Elements in Caché

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.

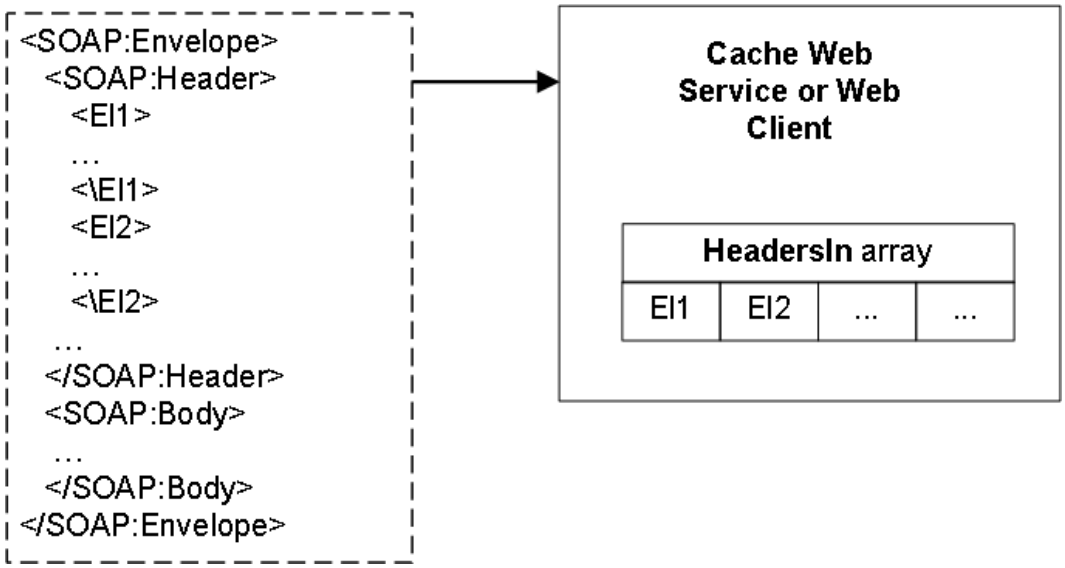
8.1.1 How Caché Represents SOAP Headers

Caché 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).

Caché 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.

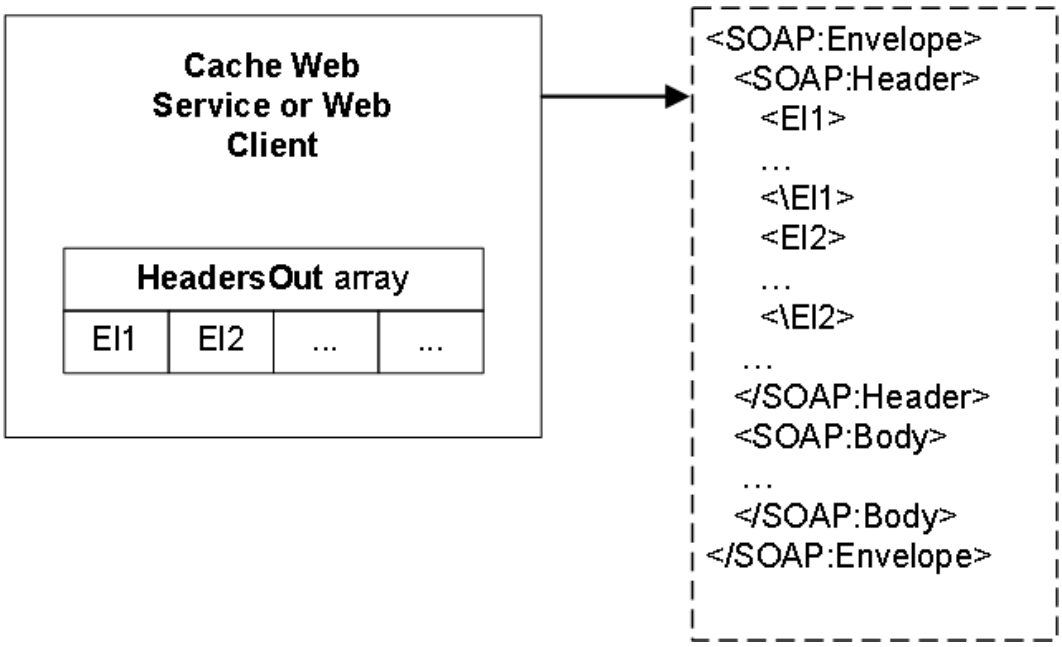
When a Caché web service or client receives a SOAP message, it imports and processes the message. During this step, if the message contains a header with custom header elements, Caché compares the header elements to the list of supported header elements (discussed in the next subsection).

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 Caché 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 Caché 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 the chapter "Adding and Using WS-Addressing Header Elements" later in this book.
- For WS-Security header elements, use the WS-Policy features, described in *Securing Caché Web Services*.

  Or directly use the SecurityIn and SecurityOut properties, discussed in the same book. 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.)
- Caché SOAP session support uses the HeadersIn and HeadersOut properties. See the chapter "SOAP Session Management," later in this book.

8.1.2 Supported Header Elements

Caché 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 a Caché 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 a Caché 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," later in this chapter. 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 Caché 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 the same book. 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, Caché rejects SOAP messages that contain required but unsupported header elements. Specifically, if Caché 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 use the SOAP Wizard to create a Caché 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 Scenario1.MyHeaderElement Extends %SOAP.Header
{

Parameter NAMESPACE = "http://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="http://www.myheaders.org"
                    xmlns:hdr="http://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 the book *Projecting Objects to XML*.

For information on the SOAP Wizard, see "Using the SOAP Wizard."

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:

```
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:

```
/// 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

Caché 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 a Caché 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 use the SOAP Wizard to create a Caché web service or web client based on a given WSDL, the system generates an XData block in that class to represent any header elements supported in its SOAP messages. (For information on the SOAP Wizard, see "Using the SOAP Wizard.")

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="http://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 SOAP Wizard uses the name `parameters` when it creates this block.

- The top level element must be <parameters>
- The <parameters> element and all its child elements (and their children) must be in the namespace "http://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 Caché class that should be used to represent it. This element includes the following attributes:

Attribute	Purpose
name	Name of the header element.
class	Caché 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:

```
XData service
{
<parameters xmlns="http://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>http://soapinterop.org/Round2Base.Service.echoBase64Request</Action>
    </request>
    <response>
      <header name="Header2" class="User.Header2" alias="Header2"/>
      <header name="IposTransportHeader" class="ipos.IposTransportHeader"/>
      <Action>http://soapinterop.org/Round2Base.Service.echoBase64Result</Action>
    </response>
  </method>
  <method name="echoString">
    <request>
      <Action>http://soapinterop.org/Round2Base.Service.echoStringRequest</Action>
    </request>
    <response>
      <Action>http://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.
- The SOAP Wizard no longer generates the *SOAPHEADERS* parameter in your generated web service and client classes.
- 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 chapter describes how to add and use WS-Addressing header elements.

For a link to details about this standard, see the section "Standards Supported by Caché" in the first chapter.

Also see the section "Adding WS-Addressing Header Elements When Faults Occur," earlier in this book.

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 Caché Web Services*.

  If you attach such a policy, Caché 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 a Caché 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 that Caché uses.

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 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'
                   xmlns:wsa='http://www.w3.org/2005/08/addressing'>
   <SOAP-ENV:Header>
      <wsa:Action>http://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">http://www.w3.org/2005/08/addressing/anonymous</wsa:Address>

      </wsa:ReplyTo>
      <wsa:To>http://localhost:8080/csp/gsop/GSOP.DivideAddressingWS.cls</wsa:To>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <Divide xmlns="http://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 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'
                   xmlns:wsa='http://www.w3.org/2005/08/addressing'>
   <SOAP-ENV:Header>
      <wsa:Action>http://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>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <DivideResponse xmlns="http://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 a Caché 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 Caché SOAP support provides a way for you to do this. This chapter includes the following topics:

- Overview of how Caché SOAP sessions work
- How to enable session support
- How to use session support

Also see the section "Specifying Custom HTTP Requests" in the chapter "Fine-Tuning a Caché Web Client."

And see "WSDL Differences for Caché Sessions" in the appendix "Details of the Generated WSDLs."

10.1 Overview of SOAP Sessions

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

- CSP session management, which is described in the book *Using Caché Server Pages (CSP)*.
- The Caché 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 Caché SOAP session header.
2. The web service receives the message and starts a new CSP session.
3. When the web service sends its reply, it adds the Caché 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 a Caché 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 CSP session, and includes the SOAP session header again when it responds.

It is not necessary to include a method to log out. The CSP session times out after a brief interval (the timeout period for the web application). See the book *Using Caché Server Pages (CSP)*.

10.2 Enabling Sessions

In order to use Caché support for SOAP sessions, you must be using a Caché web service.

- If the web client is based on Caché, 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 Caché 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:

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

10.3 Using Session Information

The process of using session information is the same as it is with other CSP pages. Namely, the web service can use a variable named *%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 — Caché 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 and see the book *Using Caché Server Pages (CSP)*.

11

Using the Caché Binary SOAP Format

Caché 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.

As of this release, any Caché web service can receive SOAP requests either in Caché binary SOAP format or in the usual SOAP format. No parameter is needed to enable this behavior. A Caché web client uses binary SOAP format only if it is configured to do so.

This chapter discusses the following:

- An introduction to Caché binary SOAP
- How to extend the WSDL for a Caché web service to indicate that it supports this format
- How to redefine an existing Caché web client so that it uses Caché binary SOAP format
- How to specify the character set used
- Additional details on the API for this format

Also see "WSDL Differences for Caché Binary SOAP Format" in the appendix "Details of the Generated WSDLs."

Note: If a Caché 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 Caché Web Services*.

11.1 Introduction

Caché 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 CSP 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/gsop/GSOP.WebServiceBinary.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; Cache;)
Host: localhost:8080
Connection: Close
ISCSoap: http://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 a Caché Web Service

Any Caché web service can receive SOAP requests either in Caché binary SOAP format or in the usual SOAP format. If a Caché 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 Caché binary SOAP format in addition to the usual SOAP format.
2. The WSDL includes information on using the Caché binary SOAP format.

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

To extend the WSDL of a Caché web service in this way, set the *SOAPBINARY* parameter to 1 for the web service.

For details on the changes, see "WSDL Differences for Caché Binary SOAP Format" in the appendix "Details of the Generated WSDLs."

11.3 Redefining a Caché Web Client to Use Binary SOAP

You can redefine an existing Caché web client so that it uses Caché 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 Caché Binary SOAP Format" in the appendix "Details of the Generated WSDLs."

11.4 Specifying the Character Set

The SoapBinaryCharset property of the web client specifies the Caché 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 Caché Binary SOAP Format

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

- For CSP 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 CacheUserName and CachePassword 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 chapter 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 Caché or when one side uses .NET. Other SOAP vendors do not support datasets.

This chapter provides the details on the following:

- General introduction
- How to define a typed dataset
- How to control the format of datasets
- Utility methods that let you see datasets as XML
- Effect on the WSDL

12.1 About Datasets

A *dataset* is an XML-format result set defined by Microsoft (and used for .NET) and also supported in Caché. If both the web service and client are based on Caché 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 Caché, you use %XML.DataSet (or a custom subclass) to represent a dataset. %XML.DataSet is an XML-enabled subclass of the standard %ResultSet class, and you use it in the same way you use that class. See "Using Dynamic SQL" in Using Caché SQL.

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 Caché. 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 Visual Studio.

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 the chapter "Controlling the Projection to XML Types" in *Projecting Objects to XML*

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="http://www.myapp.org">
    <Get0Result>
    <s:schema id="DefaultDataSet" xmlns=""
    attributeFormDefault="qualified"
    elementFormDefault="qualified"
    xmlns:s="http://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="http://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 Caché XML Tools*.

12.5 Effect on the WSDL

If a Caché web service uses %XML.DataSet as input or output to a web method, that affects the WSDL so that clients other than Caché 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="http://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 Caché 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

Fine-Tuning a Caché Web Service

This chapter discusses various ways to fine-tune a Caché web service. It discusses the following:

- How to disable access to the online WSDL
- How to require a username and password
- How to control the types used by a web service
- How to control the namespace of the schema and types used by a web service
- How to include the class documentation for the types as annotations within the WSDL
- How to add namespace declarations to the SOAP envelope
- How to check for required elements and attributes
- How to control the form of null string arguments in messages sent by a web service
- How to control the message name of the SOAP response
- How to override the SOAP action and SOAP request message name
- How to specify whether elements are qualified
- How to control whether the SOAP message parts use elements or types
- How to control the use of the xsi:type attribute in the SOAP messages
- How to control use of inline references in encoded format
- How to control the prefix used for the SOAP message envelope
- How to restrict the SOAP versions handled by a web service
- How to send responses compressed by gzip
- How to define a one-way web method
- How to add automatic line breaks to binary data
- How to add a UTF-8 BOM to the start of each message sent by a web service
- How to customize the timeout period for a web method
- How to force the use of process-private globals to support very large messages
- How to customize callbacks of a web service
- How to use custom transport to communicate with a web service
- How to define custom processing in a web service

For more basic information, see the section "Basic Settings of the Web Service" in the chapter "Basics of Creating Web Services."

Also see the information about the *ALLOWREDUNDANTARRAYNAME* web service class parameter in "Projection of Collection Properties to XML Schemas" in the chapter "Controlling the Projection to XML Schemas" in *Projecting Objects to XML*.

13.1 Disabling Access to the Online WSDL

By default, it is possible to view the WSDL for a Caché web service via a URL of the following form:

```
base/csp/app/web_serv.cls?WSDL
```

Here *base* is the base URL for your web server (including port if necessary), */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.

To disable the ability to access the WSDL in this way, specify the *SOAPDISABLEWSDL* parameter of the web service as 1. Note that even with *SOAPDISABLEWSDL* equal to 1, it is possible to use the **FileWSDL()** method to generate the WSDL as a static file.

13.2 Requiring a Username and Password

To configure a Caché web service to require a password, you configure its parent web application to use password authentication, and to disallow unauthenticated access. For information, see the chapter "CSP Architecture" in *Using Caché Server Pages (CSP)*.

13.3 Controlling the XML Types

The WSDL defines the XML types for the arguments and return values of all methods of the web service. For a Caché web service, the types are determined as follows:

- If the Caché type corresponds to a simple type (such as %String), an appropriate corresponding XML type is used.
- If the Caché type corresponds to an XML-enabled class, the *XMLTYPE* parameter of that class specifies the name of the XML type. If that parameter is not specified, the class name (without the package) is used as the XML type name.

  Also, the WSDL defines this type, by using the information in the corresponding class definition.
- If the Caché type corresponds to some other class, the class name (without the package) is used as the XML type name. Also, the WSDL does not define this type.

For further details, see the book *Projecting Objects to XML*.

Also see the section "WSDL Support in Caché" in the first chapter.

13.4 Controlling the Namespaces of the Schema and Types

This section describes how to control the namespace for the schema of the WSDL, as well as the namespaces for any types defined within it.

13.4.1 Controlling the Namespace of the Schema

The *TYPENAMESPACE* parameter (of your web service) controls the target namespace for the schema of your web service.

If *TYPENAMESPACE* is null, the schema is in the namespace given by the *NAMESPACE* parameter of the web service. The WSDL might look as follows:

```
<?xml version='1.0' encoding='UTF-8' ?>
...
<types>
<s:schema elementFormDefault='qualified'
targetNamespace = 'http://www.myapp.org'>
...
```

If you set *TYPENAMESPACE* to a URI, that URI is used as the namespace for the types. In this case, the WSDL might look as follows:

```
<?xml version='1.0' encoding='UTF-8' ?>
...
<types>
<s:schema elementFormDefault='qualified'
targetNamespace = 'http://www.mytypes.org'>
...
```

13.4.2 Controlling the Namespace of the Types

For any types referenced within the schema, the following rules govern how they are assigned to namespaces:

- If the *USECLASSNAMESPACES* parameter of the web service is 0 (the default), then the types are in the same namespace as the schema; see the previous section.
- If the *USECLASSNAMESPACES* parameter of the web service is 1 (and if the web service uses the document binding style), then each type is in the namespace given by the *NAMESPACE* parameter of the corresponding type class.

  For a given type, if the *NAMESPACE* parameter is null for the type class, then the type is in the same namespace as the schema; see the previous section.

  For information on binding styles, see "Specifying the Binding Style for the SOAP Messages," earlier in this book.

13.5 Including Documentation for the Types

By default, the WSDL for a web service does not include documentation for the types used by the web service.

To include the class documentation for the types within `<annotation>` elements in the schema of the WSDL, specify the *INCLUDEDOCUMENTATION* parameter of the web service as 1.

This parameter does not cause the WSDL to include comments for the web service and its web methods; there is no option to automatically include these comments in the WSDL.

13.6 Adding Namespace Declarations to the SOAP Envelope

To add a namespace declaration to the SOAP envelope (<SOAP-ENV:Envelope> element) of a SOAP message sent by a given web service, modify each web method of that web service so that it invokes the **%AddEnvelopeNamespace()** method of the web service. This method has the following signature:

```
Method %AddEnvelopeNamespace(namespace As %String,
                             prefix As %String,
                             schemaLocation As %String,
                             allowMultiplePrefixes As %Boolean) As %Status
```

Where:

- *namespace* is the namespace to add.
- *prefix* is the optional prefix to use for this namespace. If you omit this argument, a prefix is generated.
- *schemaLocation* is the optional schema location for this namespace.
- *allowMultiplePrefixes* controls whether a given namespace can be declared multiple times with different prefixes. If this argument is 1, then a given namespace can be declared multiple times with different prefixes. If this argument is 0, then if you add multiple declarations for the same namespace with different prefixes, only the last supplied prefix is used.

13.7 Checking for Required Elements and Attributes

By default, a Caché web service does not check for the existence of elements and attributes that correspond to properties that are marked as Required. To cause a web service to check for the existence of such elements and attributes, set the *SOAPCHECKREQUIRED* parameter of the web service to 1. The default value for this parameter is 0, for compatibility reasons.

13.8 Controlling the Form of Null String Arguments

Normally, if an argument is omitted, a Caché web service omits the corresponding element in the SOAP message that it sends. To change this, set the *XMLIGNORENULL* parameter to 1 in the web service class; in this case, the SOAP message includes an empty element.

Note: This parameter affects only web method arguments of type %String.

13.9 Controlling the Message Name of the SOAP Response

You can control the message name used in the response received from a web method. By default, this message name is the name of the web method with `Response` appended to the end. The following example shows a response from a web method called `Divide`; the response message name is `DivideResponse`.

```
<SOAP-ENV:Body>
   <DivideResponse xmlns="http://www.myapp.org">
       <DivideResult>.5</DivideResult>
   </DivideResponse>
</SOAP-ENV:Body>
```

To specify a different response message name, set the SoapMessageName keyword within the web method definition.

Note that you cannot change the name of the SOAP message that invokes a given web method; this name of this message is the name of the method. You can, however, override the SOAP action as given in the HTTP request; see the later section "Overriding the Default HTTP SOAP Action."

13.10 Overriding the HTTP SOAP Action and Request Message Name

When you invoke a web method via HTTP, 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 indicates the intent of the SOAP HTTP request. The value is a URI identifying the intent; it 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.

For a web method created in Caché, the `SOAPAction` HTTP header has the following form by default (for SOAP 1.1):

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

Where *NAMESPACE* is the value of the *NAMESPACE* parameter for the web service, and *Package.Class.Method* is the name of the method that you are using as a web method. For example:

```
SOAPAction: http://www.myapp.org/GSOAP.WebService.GetPerson
```

To override this, specify a value for the SoapAction method keyword, within the definition of the web method. Specify a quoted string that indicates that identifies the intent of the SOAP request. In the typical scenario, each web method in the web service specifies a unique value (if any) for SoapAction.

If SoapAction is not unique within this web service, each method must have a unique value of the SoapRequestMessage method keyword. This keyword specifies the name of the top element in the SOAP body of the request message. Note that SoapRequestMessage has an effect only for wrapped document/literal messages.

13.11 Specifying Whether Elements Are Qualified

The *ELEMENTQUALIFIED* parameter (of your web service) controls the value of the `elementFormDefault` attribute in the schema of the WSDL. Specifically:

- If *ELEMENTQUALIFIED* is 1, then `elementFormDefault` is `"qualified"`.
- If *ELEMENTQUALIFIED* is 0, then `elementFormDefault` is `"unqualified"`.

The default value for this parameter depends on the value of the SoapBodyUse class keyword. See the *Caché Class Definition Reference*. Normally SoapBodyUse is `"literal"`, which means that *ELEMENTQUALIFIED* is 1.

For information on the differences between qualified and unqualified elements, as well as examples, see the book *Projecting Objects to XML*.

13.12 Controlling Whether Message Parts Use Elements or Types

Your web service has another parameter (*XMLELEMENT*) that controls the precise form of the message parts of the SOAP messages. Specifically:

- If *XMLELEMENT* is 1, then the `<part>` element has attributes called `name` and `element`. In this case, the WSDL contains a sample `<message>` element as follows:

```
<message name="GetPersonSoapOut">
  <part name="GetPersonResult" element="s0:Person" />
</message>
```

- If *XMLELEMENT* is 0, then the `<part>` element has attributes called `name` and `type`. In this case, the WSDL contains a sample `<message>` element as follows:

```
<message name="GetPersonSoapOut">
  <part name="GetPersonResult" type="s0:Person" />
</message>
```

The default value for this parameter depends on the value of the SoapBodyUse class keyword. See the *Caché Class Definition Reference*. Normally SoapBodyUse is `"literal"`, which means that *XMLELEMENT* is 1.

13.13 Controlling Use of the xsi:type Attribute

By default, Caché SOAP messages include the `xsi:type` attribute only for the top-level types. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
...
<types:GetPersonResponse>
<GetPersonResult href="#id1" />
</types:GetPersonResponse>
<types:Person id="id1" xsi:type="types:Person">
<Name>Yeats,Clint C.</Name>
<DOB>1944-12-04</DOB>
</types:Person>
...
```

In these examples, line breaks have been added for readability. To use this attribute for *all* types in the SOAP messages, set the *OUTPUTTYPEATTRIBUTE* parameter or the OutputTypeAttribute property to 1. The same output would look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
...
<types:GetPersonResponse>
<GetPersonResult href="#id1" />
</types:GetPersonResponse>
<types:Person id="id1" xsi:type="types:Person">
<Name xsi:type="s:string">Yeats,Clint C.</Name>
<DOB xsi:type="s:date">1944-12-04</DOB>
</types:Person>
...
```

This parameter has no effect on the WSDL of the web service.

13.14 Controlling Use of Inline References in Encoded Format

With encoded format, any object-valued property is included as a reference, and the referenced object is written as a separate element in the SOAP message.

To instead write the encoded objects inline, specify the *REFERENCESINLINE* parameter or the ReferencesInline property as 1.

The property takes precedence over the parameter.

13.15 Specifying the SOAP Envelope Prefix

By default, a Caché web service uses the prefix SOAP-ENV in the envelope of the SOAP messages it sends. You can specify a different prefix. To do so, set the *SOAPPREFIX* parameter of the web service. For example, if you set this parameter equal to MYENV, the web service includes this prefix in its messages, as follows:

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

The *SOAPPREFIX* parameter also affects the prefix used in any SOAP faults generated by the web service.

This parameter has no effect on the WSDL of the web service.

13.16 Restricting the SOAP Versions Handled by a Web Service

By default, a Caché web service can handle SOAP requests that use SOAP version 1.1 or 1.2. To modify the web service so that it can handle only SOAP requests for a specific SOAP version, set the *REQUESTVERSION* parameter. This parameter can equal `"1.1"`, `"1.2"`, or `""`. If this parameter is `""`, the web service has the default behavior.

Note that the *SOAPVERSION* parameter does not affect the versions supported by the web service; it only controls which versions are advertised in the WSDL.

13.17 Sending Responses Compressed by gzip

A Caché web service can compress its response messages with gzip, a free compression program that is widely available on the Internet. This compression occurs after any other message packaging (such as creating MTOM packages). To cause a web service to do so, set the *GZIPOUTPUT* parameter equal to 1.

This parameter has no effect on the WSDL of the web service.

If you make this change, be sure that the web client can automatically decompress the message with gunzip, the corresponding decompression program.

If the web client is a Caché web client, note that the CSP Gateway automatically decompresses inbound messages before sending them to the web client.

13.18 Defining a One-Way Web Method

Normally, when you execute a web method, a SOAP message is returned, even if the method has no return type and returns nothing when executed in Caché. This SOAP response message has the following general form:

```
<?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>
   <MethodNameResponse xmlns="http://www.myapp.org"></MethodNameResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In rare cases, you might need to define a web method as being "one-way." Such a method must return no value, and no SOAP response is expected to the request message. To define a one-way web method, define the return type of the method as %SOAP.OneWay. In this case:

- The WSDL does not define output defined for this web method.
- The web service does not return a SOAP message (unless the service adds a header element; see the subsection). That is, the HTTP response message does not include any XML content.

Note: One-way methods should normally not be used. A request-response pair is much more common, supported, and expected — even for a method that has no return type.

See "WSDL Differences for One-Way Web Methods" in the appendix "Details of the Generated WSDLs."

13.18.1 One-Way Web Methods and SOAP Headers

If the web method adds a header element, then the HTTP response does include XML content as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/' ...
  <SOAP-ENV:Header>
     header elements as set by the web service
 </SOAP-ENV:Header>
  <SOAP-ENV:Body></SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

13.18.2 Dynamically Making a Web Method One Way

You can also dynamically redefine a web method to be one way. To do so, invoke the **ReturnOneWay()** of the web service within the definition of the web method. For example:

```
Method HelloWorldDynamic(oneway as %Boolean = 0) As %String [ WebMethod ]
{
  If oneway {Do ..ReturnOneWay() }
  Quit "Hello world "
}
```

If the argument is 0, this web method returns a SOAP response whose body contains `Hello world`. If the argument is 1, this method does not return a SOAP response.

13.19 Adding Line Breaks to Binary Data

You can cause the Caché web service to include automatic line breaks for properties of type %Binary or %xsd.base64Binary. To do so, do either of the following:

- Set the *BASE64LINEBREAKS* parameter to 1 in the web service class.
- Set the Base64LineBreaks property to 1, for the web service class instance. The value of this property takes precedence over the value set by the *BASE64LINEBREAKS* parameter.

For the parameter and the property, the default value is 0; by default, a Caché web service does not include automatic line breaks for properties of type %Binary or %xsd.base64Binary.

13.20 Adding a Byte-Order Mark to the SOAP Messages

By default, a message sent by a Caché web service does not start with a BOM (byte-order mark).

The BOM is usually not needed because the message is encoded as UTF-8, which does not have byte order issues. However, in some cases, it is necessary or desirable to include a BOM in a SOAP message; this BOM merely indicates that the message is UTF-8.

To add a BOM to the messages sent by a Caché web service, set the RequestMessageStart property of the service. This property must equal a comma-separated list of the parts to include at the start of a message. These parts are as follows:

- `DCL` is the XML declaration:

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

- `BOM` is the UTF-8 BOM.

The default is `"DCL"`.

In practice, RequestMessageStart can equal any of the following values:

- `"DCL"`
- `"BOM"`
- `"BOM,DCL"`

13.21 Customizing the Timeout Period

The CSP Gateway waits for a fixed length of time (specified in the CSP Gateway) for a Caché web service to send a response message. For information on setting the timeout period, see the *CSP Gateway Configuration Guide*.

In some cases, you might know that a given web method requires a longer period before it can complete. If so, you can specify the timeout period for that method. To do so, near the start of the definition of that web method, add a line to set the *Timeout* property of the web service. Specify a timeout period in seconds. For example, if the default timeout period is three minutes and you need the timeout period to be five minutes, you might do the following:

```
Method LongRunningMethod(Input) as %Status [ WebMethod ]
{
   set ..Timeout=300; this method will not time out until 5 minutes
   //method implementation here
}
```

13.22 Using Process-Private Globals to Support Very Large Messages

By default, a Caché web service usually uses local array memory when it parses requests or responses. You can force it to use process-private globals instead; this enables the web service to process very large messages.

To do so, specify the *USEPPGHANDLER* parameter of the web service class as follows:

```
Parameter USEPPGHANDLER = 1;
```

If this parameter is 1, then the web service always uses process-private globals when it parses requests or responses. If this parameter is 0, then the web service always uses local array memory for these purposes. If this parameter is not set, then the web service uses the default, which is usually local array memory.

13.23 Customizing Callbacks of a Web Service

You can customize the behavior of a Caché web service by overriding its callback methods:

OnRequestMessage()

Called when the web service receives a request message, if there is no security error; this callback is not invoked in the case of a security error. The system invokes this callback after performing security processing, after checking the envelope for errors, and after processing the actions specified in the WS-Addressing header (if any). This callback is useful for tasks such as logging raw SOAP requests.

This method has the following signature:

```
Method OnRequestMessage(mode As %String, action As %String, request As %Stream.Object)
```

Where:

- *mode* specifies the type of SOAP request. This is either `"SOAP"` or `"binary"`.
- *action* contains the value of SOAPAction header.
- *request* contains the SOAP request message in a stream.

This method can use the CSP *%request* object. In this object:

- The Content property contains the raw request message.
- The **NextMimeData()** instance method enbles you to retrieve individual MIME parts (if this is a MIME SOAP request).

This method can also use properties of the web service instance. The following properties are set during initialization:

- The ImportHandler property contains the DOM for parsed SOAP request message.
- The SecurityIn property contains the WS-Security header element. For details, see *Securing Caché Web Services*.
- The SecurityNamespace property contains the namespace for the WS-Security header element.
- The SoapFault property is set if SOAP fault has been generated.

To return a fault within **OnRequestMessage()**, set the SoapFault property. Do not call the **ReturnFault()** method.

OnPreWebMethod()

Called just before a web method is executed; does nothing by default. This method takes no arguments and cannot return a value. This method therefore cannot change the execution of the web service except by returning a SOAP fault in the same way that a web method would do.

This method can use *%request*, *%session*, and the web service properties. Note that the MsgClass property of the web service is the message descriptor class that contains the web method arguments.

OnPostWebMethod()

Called just after a web method is executed; does nothing by default. This method takes no arguments and cannot return a value. This method therefore cannot change the execution or return value of the web method. You customize this method primarily to clean up required structures created by **OnPreWebMethod()**.

13.24 Specifying Custom Transport for a Web Service

By default, a Caché web service responds to transport in a specific way, described here. You can customize this behavior.

13.24.1 Background

When a Caché web service receives a SOAP message, it executes its **OnSOAPRequest()** class method. By default, this method does the following:

1. Initializes the web service instance by calling its **Initialize()** method. This method parses the inbound SOAP message, returns several pieces of information by reference, and processes the security header. See the documentation for the %SOAP.WebService class.
2. Sets properties of the web service instance, such as SoapFault and others.
3. Initializes the response stream.
4. Invokes the **Process()** method of the web service, passing to it the SOAP action and the method to invoke.
5. Resets the web service instance by calling its **Reset()** method.
6. Copies the result into the response stream.

13.24.2 Defining Custom Transport for a Web Service

To implement a web service using your own transport, get the SOAP message as a stream using your transport, instantiate the web service class and call its **OnSOAPRequest()** class method.

The **OnSOAPRequest()** method must transport the request to the web service and obtain the response. To indicate an error, it must return a SOAP fault in the response stream. The signature of this method must be as follows:

```
Method OnSOAPRequest(action,requestStream, responseStream)
```

Here:

1. *action* is a %String that specifies the SOAP action. The piece of the action string after the last "." is used as the method name for using the correct descriptor class. If action is null, then the element name from the first element (the wrapping element) in the SOAP body is used as the method name.
2. *requestStream* is a stream that contains the SOAP request message encoded according to the encoding attribute of the XML directive.
3. *responseStream* is a character stream produced as the SOAP response that contains the response SOAP message encoded in UTF-8. You can create this argument before calling **OnSOAPRequest()** and passed it in with the method call. Or this argument can be a variable passed by reference. In this case, **OnSOAPRequest()** must set it equal to an instance of %FileCharacterStream that contains the response.

13.25 Defining Custom Processing in a Web Service

In rare scenarios, it may be useful to define a Caché web service that uses custom processing to handle inbound messages and to build response messages. In these scenarios, you implement either the **ProcessBodyNode()** method or the **ProcessBody()** method in the web service. This section provides the details.

13.25.1 Overview

In custom processing, you parse the inbound message and construct the response manually. The requirements are as follows:

- In the web service, you define web methods that have the desired signatures. You do this to establish the WSDL of the web service. These web methods (or some of them) can be stubs. A method is executed only if **ProcessBodyNode()** or **ProcessBody()** returns 0.
- Also in the web service, you implement one of the following methods:
  - **ProcessBodyNode()** — This method receives the SOAP body as an instance of %XML.Node. You can use Caché XML tools to work with this instance and build the response message. The SOAP envelope is available in the Document property of this instance of %XML.Node.
  - **ProcessBody()** — This method receives the SOAP Body as a stream. Because the SOAP body is an XML fragment rather than an XML document, you cannot use the Caché XML tools to read it. Instead, you parse the stream with ObjectScript functions and extract the needed parts.

  If you define both of these methods, the **ProcessBodyNode()** method is ignored.

In either case, the response message that you construct must be consistent with the WSDL of the web service.

13.25.2 Implementing ProcessBodyNode()

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

```
method ProcessBodyNode(action As %String, body As %XML.Node,
       ByRef responseBody As %CharacterStream) as %Boolean
```

Where:

- *action* is the SOAP action specified in the inbound message.
- *body* is an instance of %XML.Node that contains the SOAP <Body>.
- *responseBody* is the response body serialized as an instance of %Library.CharacterStream. This stream is passed by reference and is initially empty.

If you implement this method in a web service, the method should do the following:

1. Examine the *action* and branch accordingly. For example:

   ```
   if action["action1" {
    //details
   }
   ```

2. If you need to access the SOAP <Envelope> (for example, to access its namespace declarations), use the Document property of *body*. This equals an instance of %XML.Document, which represents the SOAP envelope as a DOM (Document Object Model).

   Otherwise, use *body* directly.

3. Now you have the following options:

   - Use %XML.Writer to write the body as a string, which you can then manipulate. For example:

     ```
     set writer=##class(%XML.Writer).%New()
     do writer.OutputToString()
     do writer.DocumentNode(body)
     set request=writer.GetXMLString(.sc)
     // check returned status and continue
     ```

   - Use methods of %XML.Document or %XML.Node, as appropriate, to navigate through the document. Similarly, use properties of %XML.Document or %XML.Node to access information about the current part of the document.
   - Use XPath expressions to extract data.
   - Perform XSLT transformations.

For details, see *Using Caché XML Tools*. Be sure to check the status returned by methods in these classes, to simplify troubleshooting in the case of an error.

4. If an error occurs during the processing of the request, return a fault in the usual way using the **ReturnFault()** method.
5. Use the **Write()** method of the response stream to write the XML fragment which will become the child element of <Body>.
6. If a response stream is created, return 1. Otherwise, return 0, which causes Caché to run the web method associated with the given action.

   For example:

```
if action["action1" {
  //no custom processing for this branch
  quit 0
 } elseif action["action2" {
    //details
    //quit 1
 }
```

13.25.3 Implementing ProcessBody()

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

```
method ProcessBody(action As %String, requestBody As %CharacterStream,
                   ByRef responseBody As %CharacterStream) as %Boolean
```

Where:

- *action* is the SOAP action specified in the inbound message.
- *requestBody* is an instance of %Library.CharacterStream that contains the SOAP <Body> element. The stream contains an XML fragment, not a complete XML document.
- *responseBody*, is the response body serialized as an instance of %Library.CharacterStream. This stream is passed by reference and is initially empty.

If you implement this method in a web service, the method should do the following:

1. Examine the *action* and branch accordingly. For example:

```
if action["action1" {
 //details
}
```

2. Use the **Read()** method of *requestBody* to obtain the SOAP <Body>. For example:

```
set request=requestBody.Read()
```

3. Parse this stream by using tools such as **$EXTRACT**. For example:

```
set in1="<echoString xmlns=""http://soapinterop.org/xsd""><inputString>"
set in2="</inputString></echoString>"
set contents=$extract(request,$length(in1)+1,*-$length(in2))
```

4. If an error occurs during the processing of the request, return a fault in the usual way using the **ReturnFault()** method.
5. Use the **Write()** method of the response stream to write the XML fragment that will become the child element of <Body>. For example:

```
set in1="<echoString xmlns=""http://soapinterop.org/xsd""><inputString>"
set in2="</inputString></echoString>"
set request=requestBody.Read()
if ($extract(request,1,$length(in1))'=in1) || ($extract(request,*-$length(in2)+1,*)'=in2) {
  do responseBody.Write("Bad Request: "_request)
  quit 1
}

set out1="<echoStringResponse xmlns=""http://soapinterop.org/xsd""><echoStringResult>"
set out2="</echoStringResult></echoStringResponse>"
do responseBody.Write(out1)
do responseBody.Write($extract(request,$length(in1)+1,*-$length(in2)))
do responseBody.Write(out2)
```

6. If a response stream is created, return 1. Otherwise, return 0, which causes Caché to run the web method associated with the given action.

14

Fine-Tuning a Caché Web Client

After you generate a Caché web client class, you do not usually edit the class. Instead you write code that creates an instance of that class and that provides provide client-side error handling. This chapter discusses various ways to fine-tune the Caché web client, either by modifying the web client instance or (less commonly) by modifying the generated class. It discusses the following:

- How to disable keep-alive for a web client
- How to control the form of null string arguments in messages sent by a client
- How to control when a Caché web client times out
- How to enable your Caché web client to communicate via a proxy server
- How to set HTTP headers
- How to specify the HTTP version to use
- How to disable SSL server name checking
- How to control the use of the xsi:type attribute in the SOAP messages
- How to control use of inline references in encoded format
- How to control the prefix used for the SOAP message envelope
- How to add namespace declarations to the SOAP envelope
- How to send responses compressed by gzip
- How to quote the SOAP action for SOAP 1.1 requests
- How to treat the HTTP status 202 in the same way as HTTP status 200
- How to define one-way web methods
- How to add automatic line breaks to binary data
- How to add a UTF-8 BOM to the start of each message sent by a web client
- How to force the use of process-private globals to support very large messages
- How to create custom SOAP messages
- How to specify custom HTTP requests to carry SOAP messages from a web client
- How to customize callbacks of a web client
- How to use transport other than HTTP to carry the SOAP message from a web client
- How to specify flags for the SAX parser to use when a web client invokes a web service

- How to use the WS-Security login feature
- How to use HTTP authentication, as an alternative to WS-Security

Also see the section "Basic Settings of a Caché Web Client" in the chapter "Creating Web Clients." This section discusses parameters that control the namespace, the type namespace, and other basics of the web client.

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.

14.1 Disabling Keep-Alive for a Web Client

By default, if you reuse a Caché web client instance to send multiple request messages, Caché sends all the messages in a single HTTP transmission (using a HTTP 1.1 keep-alive connection). Specifically, Caché keeps the TCP/IP socket open so that Caché does not need to close and reopen it. To disable this keep-alive behavior, do any of the following:

- Kill the web client instance and create and use a new one.
- Set the client's HttpRequest.SocketTimeout property to 0 after you send the first message. For example:

  ```
  Set client.HttpRequest.SocketTimeout=0
  ```

- Add the `Connection: close` HTTP header in the second request from the client. To do so, add code like the following after you send the first message:

  ```
  Set sc=client.HttpRequest.SetHeader("Connection","close")
  ```

Note: If you are using WS-ReliableMessaging and you are using SSL/TLS to communicate with the web service, do not disable keep-alive. For information on WS-ReliableMessaging, see *Securing Caché Web Services*.

14.2 Controlling the Form of Null String Arguments

Normally, if an argument is omitted, a Caché web client omits the corresponding element in the SOAP message that it sends. To change this, set the *XMLIGNORENULL* parameter to 1 in the web client class; in this case, the SOAP message includes an empty element.

Note: This parameter affects only web method arguments of type %String.

14.3 Controlling the Client Timeout

You can control two separate timeout periods for a Caché web client:

- The Timeout property of the web client is the read timeout. This specifies how long, in seconds, the web client waits for a response.

  If this property is not specified, the web client uses the default value specified for the Timeout property of the %Net.HttpRequest class. This default is 30 seconds.

If you are using a proxy server, this property controls how long the client waits for a response from the proxy.

- The OpenTimeout property specifies the open timeout, which is the number of seconds to wait for the TCP/IP connection to open. If this property is not specified, the value specified by Timeout is used instead.

14.4 Using a Proxy Server

Your Caché web client can communicate with a web service via a proxy server. In order to set this up, specify properties of your web client instance to indicate the proxy server to use. These properties are as follows:

HttpProxyServer

Specifies the host name of the proxy server to use. If this property is not null, the HTTP request is directed to this machine.

For information on specifying the default proxy server, see "Using a Proxy Server" in *Using Caché Internet Utilities*.

HttpProxyPort

Specifies the port to connect to, on the proxy server.

For information on specifying the default proxy port, see "Using a Proxy Server" in *Using Caché Internet Utilities*.

HttpProxyHTTPS

Specify this as true if you are using a proxy server and if that proxy server supports HTTPS.

Note that if you are using HTTPS, you must also set the SSLConfiguration property of the client equal to the name of the SSL/TLS configuration; for more details on security, see the section "Configuring the Client to Use SSL" earlier in this book.

HttpProxyAuthorization

If the web client should authenticate itself with the proxy server, specify this as the required `Proxy-Authorization` header field.

HttpProxyTunnel

Specify this as true if the web client should establish a tunnel through the proxy to the target HTTP server. If true, the request uses the HTTP CONNECT command to establish a tunnel. The address of the proxy server is taken from the HttpProxyServer and HttpProxyPort properties. If the endpoint URL has the https: protocol, then once the tunnel is established, Caché negotiates the SSL connection. In this case, the HttpProxyHTTPS property is ignored because the tunnel establishes a direct connection with the target system.

14.5 Setting HTTP Headers

If you need further control over the HTTP headers sent by a web client, you can use the following methods of %SOAP.WebClient:

SetHttpHeader()

Adds a header to the HTTP request. Note that the `Content-Type`, `Content-Encoding`, and `Content-Length` headers are part of the entity body rather than the HTTP main headers. You cannot set the `Content-Length` header, which is read-only. Nor can you set the `Connection` header, because this class does not support persistent connections.

ResetHttpHeaders()

Clear all HTTP headers.

Also see the section "Using HTTP User Authentication," earlier in this chapter.

14.6 Specifying the HTTP Version to Use

By default, a Caché web client uses HTTP/1.1. You can instead use HTTP/1.0. To do so, set the HttpVersion property of the client to `"1.0"`.

14.7 Disabling SSL Server Name Checking

By default, when a Caché web client connects to a server via SSL, it checks that the certificate server name matches the DNS name used to connect to the server. (This checking is described in RFC 2818 section 3.1. Wildcard support is described in RFC 2595, but browsers generally do check the server name, and InterSystems has chosen to do the same.)

To disable this checking, set SSLCheckServerIdentity property of the client to `0`.

14.8 Controlling Use of the xsi:type Attribute

By default, Caché SOAP messages include the xsi:type attribute only for the top-level types. For example:

```
<?xml version="1.0" encoding="UTF-8" ?>
...
<types:GetPersonResponse>
<GetPersonResult href="#id1" />
</types:GetPersonResponse>
<types:Person id="id1" xsi:type="types:Person">
<Name>Yeats,Clint C.</Name>
<DOB>1944-12-04</DOB>
</types:Person>
...
```

In these examples, line breaks have been added for readability. To use this attribute for *all* types in the SOAP messages, do either of the following:

- Set the OutputTypeAttribute property equal to 1 in the web client instance.
- Set the *OUTPUTTYPEATTRIBUTE* parameter equal to 1 in the web client class.

The same output would look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
...
<types:GetPersonResponse>
<GetPersonResult href="#id1" />
</types:GetPersonResponse>
<types:Person id="id1" xsi:type="types:Person">
<Name xsi:type="s:string">Yeats,Clint C.</Name>
<DOB xsi:type="s:date">1944-12-04</DOB>
</types:Person>
...
```

The property takes precedence over the parameter.

14.9 Controlling Use of Inline References for Encoded Format

With encoded format, any object-valued property is included as a reference, and the referenced object is written as a separate element in the SOAP message.

To instead write the encoded objects inline, specify the *REFERENCESINLINE* parameter or the ReferencesInline property as 1. The property takes precedence over the parameter.

14.10 Specifying the Envelope Prefix

By default, a Caché web client uses the prefix `SOAP-ENV` in the envelope of the SOAP messages it sends. You can specify a different prefix. To do so, set the *SOAPPREFIX* parameter of the web client class. For example, if you set this parameter equal to `MYENV`, the web client includes this prefix in its messages instead of `SOAP-ENV`.

14.11 Adding Namespace Declarations to the SOAP Envelope

To add a namespace declaration to the SOAP envelope (`<SOAP-ENV:Envelope>` element) of the SOAP responses returned by a given web client, call the **%AddEnvelopeNamespace()** method of the web client before calling the web method. This method has the following signature:

```
Method %AddEnvelopeNamespace(namespace As %String,
                             prefix As %String,
                             schemaLocation As %String,
                             allowMultiplePrefixes As %Boolean) As %Status
```

Where:

- *namespace* is the namespace to add.
- *prefix* is the optional prefix to use for this namespace. If you omit this argument, a prefix is generated.
- *schemaLocation* is the optional schema location for this namespace.
- *allowMultiplePrefixes* controls whether a given namespace can be declared multiple times with different prefixes. If this argument is 1, then a given namespace can be declared multiple times with different prefixes. If this argument is

0, then if you add multiple declarations for the same namespace with different prefixes, only the last supplied prefix is used.

14.12 Sending Responses Compressed by gzip

A Caché web client can compress its response messages with gzip, a free compression program that is widely available on the Internet. This compression occurs after any other message packaging (such as creating MTOM packages). To cause a web client to do so, do either of t he following:

- Set the GzipOutput property equal to 1 in the web client instance.
- Set the *GZIPOUTPUT* parameter equal to 1 in the web client class.

If you do so, be sure that the web service can automatically decompress the message with gunzip, the corresponding decompression program. (If the web service is a Caché web service, note that the CSP Gateway automatically decompresses inbound messages before sending them to the web service.)

14.13 Quoting the SOAP Action (SOAP 1.1 Only)

In SOAP 1.1 request messages, the HTTP header includes a `SOAPAction` line as follows:

```
POST /csp/gsop/GSOP.DivideWS.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; Cache;)
Host: localhost:8080
Connection: Close
Accept-Encoding: gzip
SOAPAction: http://www.mynamespace.org/GSOAP.DivideWS.Divide
Content-Length: 397
Content-Type: text/xml; charset=UTF-8

...
```

By default, the value for `SOAPAction` is not quoted. To place this value in quotes, specify *SOAPACTIONQUOTED* as 1 in the web client class. Then the HTTP header of the request message would be as follows:

```
POST /csp/gsop/GSOP.DivideWS.cls HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; Cache;)
Host: localhost:8080
Connection: Close
Accept-Encoding: gzip
SOAPAction: "http://www.mynamespace.org/GSOAP.DivideWS.Divide"
Content-Length: 397
Content-Type: text/xml; charset=UTF-8

...
```

In SOAP 1.2, the *SOAPACTIONQUOTED* parameter has no effect. This is because the request messages do not have a `SOAPAction` line. Instead the SOAP action is always quoted and is included within the `Content-Type` line as follows:

```
Content-Type: application/soap+xml;
   charset=UTF-8; action="http://www.mynamespace.org/GSOAP.DivideWS.Divide"
```

Note: This example shows an artificial line break, to ensure that the line fits on the page when this book is formatted as PDF.

14.14 Treating HTTP Status 202 Like Status 200

By default, a Caché web client follows the standard WS-I Basic Profile, which uses the HTTP response status 202 only if the HTTP response does not contain a SOAP envelope.

If you want to treat HTTP status 202 in the same way as HTTP status 200, set the HttpAccept202 property of the client to 1. To see the actual return status, check the HttpResponse.StatusCode property of the client.

The WS-I Basic Profile supports but does not encourage this practice: "The Profile accepts both status codes because some SOAP implementations have little control over the HTTP protocol implementation and cannot control which of these response status codes is sent."

14.15 Defining a One-Way Web Method

Normally, when a web client calls a web service, a SOAP message is returned, even if the method has no return type and returns nothing when executed in Caché.

In rare cases, you might need to define a web method as being "one-way." Such a method must return no value, and no SOAP response is expected to the message.

Note: One-way methods should normally not be used. A request-response pair is much more common, supported, and expected — even for a method that has no return type.

To define a one-way web method, define the return type of the method as %SOAP.OneWay. The WSDL does not define output defined for this web method, and the web service does not return a SOAP message.

14.16 Adding Line Breaks to Binary Data

You can cause the Caché web service to include automatic line breaks for properties of type %Binary or %xsd.base64Binary. To do so, do either of the following:

- Set the *BASE64LINEBREAKS* parameter to 1 in the web service class.
- Set the Base64LineBreaks property to 1, for the web service class instance. The value of this property takes precedence over the value set by the *BASE64LINEBREAKS* parameter.

For the parameter and the property, the default value is 0; by default, a Caché web service does not include automatic line breaks for properties of type %Binary or %xsd.base64Binary.

14.17 Adding a Byte-Order Mark to the SOAP Messages

By default, a message sent by a Caché web client does not start with a BOM (byte-order mark).

The BOM is usually not needed because the message is encoded as UTF-8, which does not have byte order issues. However, in some cases, it is necessary or desirable to include a BOM in a SOAP message; this BOM merely indicates that the message is UTF-8.

To add a BOM to the messages sent by a Caché web client, set the RequestMessageStart property of the client. This property must equal a comma-separated list of the parts to include at the start of a message. These parts are as follows:

- DCL is the XML declaration:

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

- BOM is the UTF-8 BOM.

The default is "DCL".

In practice, RequestMessageStart can equal any of the following values:

- "DCL"
- "BOM"
- "BOM,DCL"

14.18 Using Process-Private Globals When Parsing

By default, a Caché web client usually uses local array memory when it parses requests or responses. You can force it to use process-private globals instead; this enables the web client to process very large messages.

To do so, specify the *USEPPGHANDLER* parameter of the web service class as follows:

```
Parameter USEPPGHANDLER = 1;
```

If this parameter is 1, then the web client always uses process-private globals when it parses requests or responses. If this parameter is 0, then the web client always uses local array memory for these purposes. If this parameter is not set, then the web client uses the default, which is usually local array memory.

You can override this parameter at runtime. To do so, set the *UsePPGHandler* property of the web client instance.

14.19 Creating Custom SOAP Messages

In special cases, you may want a web client to send a custom SOAP message. The essential requirements are as follows:

1. Create a subclass of %SOAP.WebRequest and set its *LOCATION* parameter or Location property.
2. In this subclass, create a method to send a SOAP message. This method must create an instance of %Library.CharacterStream and place into it the SOAP message you want to send. It is your responsibility to ensure that the message is correctly formed.
3. The method must next invoke the **SendSOAPBody()** method:

   ```
   method SendSOAPBody(Action As %String,
                       OneWay As %Boolean = 0,
                       Request As %CharacterStream,
                       ByRef Response) as %Status
   ```

   - *Action* is a string that gives the name of SOAP action to perform.
   - *OneWay* is a true/false flag that controls whether the message is one way.
   - *Request* is an instance of %Library.CharacterStream that contains the body of the SOAP request in the character set of the current locale.

- *Response* is the response, returned by reference either as a character stream or an instance of %XML.Node.

  If *Response* is null when you invoke **SendSOAPBody()**, then the method sets *Response* equal to an instance of %Library.CharacterStream. This stream contains the body of the SOAP response in the character set of the current locale.

  If *Response* is an instance of %Library.CharacterStream when you invoke **SendSOAPBody()**, then the method updates *Response* to contain the body of the SOAP response in the character set of the current locale.

  If *Response* is an instance of %XML.Node when you invoke **SendSOAPBody()**, then the method updates *Response* to point to the body DOM.

%SOAP.WebRequest is a subclass of %SOAP.WebClient, so you may want to set other parameters and properties. You can also add SOAP headers as described earlier in this book. See the class documentation for %SOAP.WebRequest for further notes.

14.20 Specifying Custom HTTP Requests

By default, if you use a Caché web client, the web client uses HTTP to transport the SOAP message to the web service and to receive the response. The web client automatically creates and sends an HTTP request, but you can create a custom HTTP request. To do, you use the following procedure:

1. Create an instance of %Net.HttpRequest and set properties as needed. For information on this class, see the book *Using Caché Internet Utilities* or see the class documentation for %Net.HttpRequest.
2. Set the HttpRequest property of your web client equal to this instance.

This is useful in particular if you want to support multiple calls to a SOAP service within the same session. By default, the Caché web client does not support multiple calls to a SOAP service using the same session. To work around this, create a new instance of %Net.HttpRequest and use it as the HttpRequest property of your web client. This change forces the same HTTP request to be reused for all calls, which returns all cookies in a response to the next request.

14.21 Customizing Callbacks of a Web Client

You can customize the behavior of a Caché web client by overriding its callback methods:

%OnSOAPRequest()

```
Method %OnSOAPRequest(mode As %String,
                      client As %SOAP.WebClient,
                      action As %String,
                      oneWay As %Boolean,
                      method As %String,
                      requestStream As %BinaryStream)
```

Called just before the web client invokes the **DoSOAPRequest()** method of the transport class (which makes the actual SOAP request). The default DoSOAPRequest() method is included in %SOAP.WebClient and uses HTTP for request/response.

- *mode* specifies the type of SOAP request (`"SOAP"` or `"binary"`).
- *client* is the OREF of the web client instance.
- *action* contains the value of the SOAPAction header.

- *oneWay* is true if no body is to be sent.
- *method* argument is the name of the web method that is being invoked.
- *requestStream* argument contains the SOAP request message in a stream.

%OnSOAPResponse()

```
Method %OnSOAPResponse(mode As %String,
                       client As %SOAP.WebClient,
                       action As %String,
                       oneWay As %Boolean,
                       method As %String,
                       requestStream As %BinaryStream,
                       responseStream As %BinaryStream,
                       sc As %Status)
```

Called after the web client has invoked the **DoSOAPRequest()** method of the transport class. The *sc* argument is the status returned by the **DoSOAPRequest()** method of the transport class. The other arguments are the same as for **%OnSOAPRequest()**.

%OnSOAPFinished()

```
Method %OnSOAPFinished(mode As %String, client As %SOAP.WebClient, method As %String, sc As
%Status)
```

Called after the web client has performed all its processing. The *sc* argument is the status returned by the web method that was invoked. The *mode*, *client*, and *method* arguments are the same as for the other callback methods.

14.22 Specifying Custom Transport from a Web Client

By default, if you use a Caché web client, the web client uses HTTP to transport the SOAP message to the web service and to receive the response. You can define and use your own transport class.

14.22.1 Background

To communicate with the web service that it uses, a Caché web client requires a transport class. The transport class contains parameters, properties, and methods related to communication. The overall communication works as follows:

1. When a web client proxy method is run, the web client instance checks the value of its Transport property.

   If this property is null, the web client instance uses itself as the transport class instance. You can instead set the Transport property equal to an instance of some other suitable class, if you have defined such a class.

2. The web client instance executes the **DoSOAPRequest()** method of the transport class, passing the following arguments:

   a. The OREF of the web client class.

   b. A string that specifies the SOAP action.

   c. A stream containing the request encoded in UTF-8.

   d. (By reference) A stream containing the response.

3. The web client instance checks the status of the result and acts accordingly.

For HTTP transport, the **DoSOAPRequest()** method includes the following logic:

1. Create a request object (an instance of %Net.HttpRequest) and set its properties. Here, the method uses values of the properties of the web client instance, in particular HttpRequestHeaderCharset and other HTTP-related properties.
2. Go through the headers in the SOAP request and initialize the headers in the request object.
3. Execute the **Post()** method of the request object, which is a suitable action for HTTP transport.
4. Get the response and return that.

Important: Do not directly use the **DoSOAPRequest()** method of %SOAP.WebClient. No guarantee is made about its behavior or future operation. The preceding summary is provided only as a general tip.

14.22.2 Defining Custom Transport for a Caché Web Client

To enable a Caché web client to use custom transport, define a custom transport class. Then after you create an instance of the web client, set its Transport property equal to an instance of the transport class.

The requirements for the transport class are as follows:

- The class must be instantiable (that is, non-abstract).
- The class must implement the **DoSOAPRequest()** method as described below.

The **DoSOAPRequest()** method should transport the request to the web service and obtain the response. The signature of this method must be as follows:

```
Method DoSOAPRequest(webClient,action,requestStream, responseStream) As %Status
```

- *webClient* is the OREF of the web client class.
- *action* is a %String that specifies the SOAP action.
- *requestStream* is a stream containing the request encoded in UTF-8.
- *responseStream* is a %FileBinaryStream argument that **DoSOAPRequest()** uses to write the response. This stream must contain data in the character set specified by the encoding attribute of the `?xml` directive. UTF-8 is recommended.

14.23 Specifying Flags for the SAX Parser

When a Caché web client invokes a web service, it internally uses the SAX parser, a third-party product that is shipped with Caché. You can set the SAXFlags property of the web client in order to set the flags for the parser to use.

For information on the parser flags themselves, see the book *Using Caché XML Tools*.

14.24 Using the WS-Security Login Feature

If your Caché web client is using a web service that requires authentication, and if do not want to use the newer WS-Security features, you can use the older and simpler WS-Security login feature.

To use the WS-Security login feature:

1. Ensure that you are using SSL between the web client and the web server that hosts the web service. The WS-Security header is sent in clear text, so this technique is not secure unless SSL is used. See the section "Configuring the Client to Use SSL," earlier in this book.
2. Invoke the **WSSecurityLogin()** method of the web client. This method accepts the username and password, generates a WS-Security username token with clear text password, and adds a WS-Security header to the web request.
3. Invoke the web method.

This technique adds the security token only to the next SOAP message.

For information on the newer WS-Security features, see *Securing Caché Web Services*.

14.25 Using HTTP Authentication

Some web services require HTTP authentication instead of using WS-Security (which is described in *Securing Web Services*). For these web services, IRIS supports the following HTTP authentication schemes:

1. Negotiate (SPNEGO and Kerberos, per RFC 4559 and RFC 4178)
2. NTLM (NT LAN Manager Authentication Protocol)
3. Basic (Basic Access Authentication as described in RFC 2617)

Note that on HTTP 1.0, only Basic authentication is used; the other authentication schemes require multiple round trips within a single connection, which is not permitted in HTTP 1.0.

To use HTTP authentication:

- Set the HttpUsername and HttpPassword properties of the web client before invoking the web method.
- If you want the client to send an initial header indicating the scheme to use (and you know that the server permits the scheme), set the HttpInitiateAuthentication property before invoking the web method. For the value of this property, specify an authentication scheme name, as given in "Providing Login Credentials" in the chapter "Sending HTTP Requests" in *Using Internet Utilities*.
- If you want to customize the list of schemes that the client tries, set the HttpInitiateAuthentication property before invoking the web method. For the value of the property, use a comma-separated list of names, as given in "Providing Login Credentials" in the chapter "Sending HTTP Requests" in *Using Internet Utilities*.

Important: If there is a chance that Basic authentication will be used, ensure that you are using SSL between the web client and the web server that hosts the web service. In Basic authentication, the credentials are sent in base-64 encoded form and thus can be easily read. See the section "Configuring the Client to Use SSL," earlier in this book.

15

Troubleshooting Caché SOAP Problems

This chapter provides information to help you identify causes of SOAP problems in Caché. It discusses the following topics:

- Information needed for troubleshooting
- Problems consuming WSDLs (problems using the SOAP Wizard)
- Problems sending SOAP messages

For information on problems that are obviously related to security, see "Troubleshooting Security Problems" in *Securing Caché Web Services*. 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 the chapter "Sending HTTP Requests" in *Using Caché Internet Utilities*.

15.1 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
Caché SOAP log	Yes	No	For security errors, this log shows more detail than is contained in the SOAP fault.
CSP 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 the chapter "SOAP Fault Handling."

15.1.1 Caché SOAP Log

To log the SOAP calls made to or from a Caché namespace, set the following nodes of the *^ISCSOAP* global in that namespace:

Node	Purpose
^ISCSOAP(`"Log"`)	Specifies kind of logging. Use one of the following values (case-sensitive): • `"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"` — Headers only (no SOAP body). If you use `"h"` with `"i"` and/or `"o"`, then the log includes only the SOAP Envelope element and SOAP headers (if any). You can also use a string that contains any combination of these values, for example: `"iosh"`
^ISCSOAP(`"LogFile"`)	Specifies the complete path and filename of the log file to create.

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:

```
01/05/2010 13:27:02 ********************
Output from web client with SOAP action = http://www.mysecureapp.org/GSOAP.AddComplexSecureWS.Add
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
...
  <SOAP-ENV:Header>
     <Security xmlns="http://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>


01/05/2010 13:27:33 ********************
Input to web client with SOAP action = http://www.mysecureapp.org/GSOAP.AddComplexSecureWS.Add

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

Note the following points:

- With Caché 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 Caché XML Tools*.
- The Caché 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 console log instead. For information, see "Monitoring Log Files" in the *Caché Monitoring Guide*.

15.1.2 HTTP Trace in the CSP Gateway

The CSP Web Gateway Management page enables you to trace HTTP requests sent to CSP pages (such as web services) and responses sent in return. See "Viewing HTTP Trace" in the *CSP Gateway Configuration Guide*.

15.1.3 Third-Party Tracing Tools

To test your web service, you can use tracing tools such as Wireshark, ProxyTrace, tcpTrace, XMLSpy, soapUI, or Web Service Studio Express. Some of these tools are free and others are licensed. Note that InterSystems does not make any specific recommendations about these tools; they are listed here for your general information.

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 Cache web service at `http://localhost:57772/csp/gsop/GSOP.Divide.CLS`

And suppose you have a Cache web client that you created to talk to that service. The web client has a *LOCATION* parameter equal to `"http://localhost:57772/csp/gsop/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 57772.
- In the web client, edit the *LOCATION* parameter to use port 8080 instead of 57772. Then recompile.

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

```
//reset location to port 8080 to enable tracing
set client.Location="http://localhost:8080/csp/gsop/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; Cache;)
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.

15.2 Problems Consuming WSDLs

If you have a problem using the SOAP Wizard (or the %SOAP.WSDL.Reader class, which the wizard uses), the problem is likely to be one of the following:

- The WSDL is protected by SSL. In this case, the wizard issues the following error:

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

  You can specify an SSL configuration when using the wizard, but if you have done so, and you receive the preceding error, your SSL configuration is incorrect. You can try accessing the WSDL in another way, downloading it to a file, and using the file WSDL instead.

- The WSDL is password protected. In this case, the wizard issues an error like the following (notice that the line and offset vary but are different from the previous scenario):

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

If you receive this error, check whether a password is required; see the comments in "Using the SOAP Wizard," earlier in this book.

- The WSDL uses elements that are currently inaccessible. In this case, the wizard issues an error like the following:

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

The key word is *unrecognized*; this indicates that the WSDL refers to an element that is currently inaccessible. Carefully check the WSDL for <import> and <include> directives.

An import directive might look like this:

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

In this case, the workaround is as follows:

– Download the primary WSDL to a file.

– Download the referenced WSDL to a file.

– Edit the primary WSDL to refer to the new location of the referenced WSDL.

Similarly, it is possible for a WSDL to refer to other documents via a relative URL. For example:

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

If you have downloaded the WSDL to a file, this relative reference cannot work. In this case, you must also download the referenced document and edit the WSDL to use its new location.

- The WSDL contains <message> elements with multiple parts and uses document-style binding. In this case, the wizard issues the following error:

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

In this case, select the **Use unwrapped message format for document style web methods** option when you use the wizard.

- The WSDL is invalid. In such cases, the SOAP wizard issues an error to indicate the problem. The following shows one example:

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

This error message indicates a problem with an <operation> element. The following shows a partial example of the invalid WSDL that produced this error:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://acme.acmecorp.biz:9999/widget/services"
                  xmlns="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:wsdl="http://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="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getWidgetInfo">
      <wsdlsoap:operation soapAction=""/>
      <wsdl:input name="getWidgetInfoRequest">
        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://acmesubsidiary.com"
                       use="encoded"/>
```

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

                       namespace="http://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 `"http://acmesubsidiary.com"`, but the earlier part of the WSDL shows that this message is the target namespace of the web service: `"http://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, and you can also examine the WSDL directly, using information returned by the SOAP wizard.

- The WSDL contains features that are not supported in Caché. See the section "Consuming WSDLs" in the first chapter.

15.3 Problems Sending Messages

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

- The SOAP message might include a long string or long binary value, and you have not enabled long string operations in Caché. In this case, Caché 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 the wizard reads a WSDL, it assumes that any string-type input or output can be represented in Caché as %String. Similarly, it assumes that any input or output with XML type base64Binary can be represented in Caché as %xsd.base64Binary). There is no information in a WSDL to inform the SOAP Wizard that this input or output could exceed the Caché long string limit.

  See "Adjusting the Generated Classes for Long Strings," earlier in this book; 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:

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

  Also, if Caché generated a security policy (in a configuration class), you might need to edit that policy to provide missing details; see the section "Editing the Generated Policy" in *Securing Caché Web Services*. 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 Caché.) 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. Caché 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 a Caché 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 Caché 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 Caché. It is useful to examine the WSDL in use and make sure it is supported in Caché; see the section "Consuming WSDLs" in the first chapter. Note that these details have changed in Caché over time.

A

Summary of Web Service URLs

This appendix summarizes the URLs related to a Caché web service.

A.1 Web Service URLs

The URLs related to a Caché web service are as follows:

end point for the web service

```
base/csp/namespace/web_serv.cls
```

Where:

- *base* is the base URL for your web server (including port if necessary).
- */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:

```
http://localhost:57772/csp/samples/MyApp.StockService.cls
```

WSDL

```
base/csp/app/web_serv.cls&WSDL
```

For example:

```
http://localhost:57772/csp/samples/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

When you use the WSDL URL to generate a web client, if the underlying web application is protected by password authentication, it is necessary to append a string like the following (using the username `_SYSTEM` and password `SYS` as an example):

```
&CacheUserName=_SYSTEM&CachePassword=SYS
```

If you use a third-party tool to create a web client, make sure that you understand how that tool handles its logins. For example, .NET performs a URL redirect after logging in. In this case, it is necessary to *also* append the following string to the URL:

```
&CacheNoRedirect=1
```

In all cases, you can also retrieve the WSDL from a browser after supplying the required username and password, save it as a file, and use the file instead. Or you could create a non-password-protected web application to serve the WSDL to consumers as needed, if it is necessary to have continuous access to the WSDL.

B

Details of the Generated WSDLs

For reference, this appendix shows the parts of a sample WSDL document for a Caché web service, along with information about how keywords and parameters affect these parts. It discusses the following topics:

- Overview of WSDL documents
- Sample web service used for the central WSDL shown in this appendix
- Namespace declarations at the start of the <definitions> element
- The <service> element
- The <binding> element
- The <portType> element
- The <message> elements
- The <types> element
- WSDL variations due to differences in method signatures
- Other WSDL variations

The signatures of your web methods also affect the WSDL, but this appendix 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 the book *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 book 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 Caché 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," earlier in this book.

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 a Caché 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 a Caché 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 Caché support for SOAP does not support all possible variations. See the section "Standards Supported by Caché," earlier in this book.

B.2 Sample Web Service

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

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

Parameter SERVICENAME = "MyServiceName";

Parameter NAMESPACE = "http://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:

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

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

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

}
```

Except where noted, the appendix shows parts of the WSDL for this web service. (In some cases, the appendix 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 appendix, these declarations are as follows:

```
<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://www.mynamespace.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://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 `"http://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="http://schemas.xmlsoap.org/wsdl/soap12/"
  ...
  ```

  For *SOAPVERSION* equal to `""`, the WSDL would instead include the following:

  ```
  <definitions
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soap12="http://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 `http://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 appendix, this element is as follows:

```
<service name="MyServiceName">
    <port name="MyServiceNameSoap" binding="s0:MyServiceNameSoap">
        <soap:address location="http://localhost:57772/csp/gsop/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 a Caché web service contains only one.

For the sample web service shown earlier in this appendix, this element is as follows:

```
<binding name="MyServiceNameSoap" type="s0:MyServiceNameSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="document"/>
    <operation name="Add">
        <soap:operation soapAction="http://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," later in this appendix.
- If the *SOAPBINARY* parameter is 1 for the web service, that affects this element as described in "WSDL Differences for Caché Binary SOAP Format," later in this appendix.
- If the *SOAPSESSION* parameter is 1 for the web service, that affects this element as described in "WSDL Differences for Caché SOAP Sessions," later in this appendix.
- The SoapBindingStyle class keyword, SoapBindingStyle method keyword, and SoapBindingStyle query keyword affect the `<binding>` element as described in the *Caché 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 *Caché 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 book 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 Caché 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 a Caché web service contains only one.

For the sample web service shown earlier in this appendix, the `<portType>` element is as follows:

```
<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 appendix, 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 of this appendix.
- The namespaces to which those elements and types belong are determined as described in the next section of this appendix.
- Also see "WSDL Differences for One-Way Web Methods," later in this appendix.
- If the *SOAPSESSION* parameter is 1 for the web service, that affects this element as described in "WSDL Differences for Caché SOAP Sessions," later in this appendix.

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 appendix, this element is as follows:

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://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 the book *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 the book *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:

```
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:

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://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 the book *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 WSDLSamples.Namespaces Extends %SOAP.WebService
{

Parameter SERVICENAME = "MyServiceName";

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

Parameter RESPONSENAMESPACE = "http://www.myresponsenamespace.org";

Parameter TYPENAMESPACE = "http://www.mytypes.org";
```

```
Parameter RESPONSETYPENAMESPACE = "http://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:

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

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

Property Real As %Float;

Property Imaginary As %Float;

}
```

For the WSDL of this web service, the `<types>` part is as follows:

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://www.mytypes.org">
        <s:import namespace="http://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="http://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="http://www.myresponsetypes.org">
        <s:import namespace="http://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:

```
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 Caché Binary SOAP Format," later in this appendix.
- If the *SOAPSESSION* parameter is 1 for the web service, that affects this element as described in "WSDL Differences for Caché SOAP Sessions," later in this appendix.
- If specified, the SoapTypeNameSpace keyword affects this part of the WSDL. See the *Caché 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 the chapter "Controlling the Projection to XML Schemas" in *Projecting Objects to XML*.

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:

```
<types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://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 the chapter "Methods" in *Using Caché Objects*.

B.10 Other WSDL Variations for Caché Web Services

This section discusses other possible variations for WSDLs for Caché web services.

B.10.1 WSDL Differences for Caché 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:

  ```
  <soap:header message="s0:CacheSessionHeader" part="CSPCHD" use="literal"/>
  ```

  For example:

  ```
  <operation name="Add">
     <soap:operation soapAction="http://www.mynamespace.org/WSDLSamples.BasicWS.Add" style="document"/>

      <input>
          <soap:body use="literal"/>
          <soap:header message="s0:CacheSessionHeader" part="CSPCHD" use="literal"/>
      </input>
      <output>
          <soap:body use="literal"/>
          <soap:header message="s0:CacheSessionHeader" part="CSPCHD" use="literal"/>
      </output>
  </operation>
  ```

- The WSDL includes the following additional <message> element:

  ```
  <message name="CacheSessionHeader">
      <part name="CSPCHD" element="chead:CSPCHD"/>
  </message>
  ```

- The <types> element includes the following additional item:

```
<s:schema elementFormDefault="qualified" targetNamespace="http://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="http://www.intersystems.com/SOAPheaders"
```

B.10.2 WSDL Differences for Caché Binary SOAP Format

For a Caché 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="Cache_charset">
  ```

  Where *Cache_charset* is the Caché character set (for example: Unicode, Latin1) of the Caché 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 Caché 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 Caché 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 %Float (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="http://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 a Caché web service or web client uses the Caché binary SOAP format, you cannot use WS-Security or WS-Policy features with this web service or client. See *Securing Caché 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 Classes

For reference, this appendix provides information on the classes generated by the SOAP Wizard. It discusses the following:

- Overview of the generated classes
- Keywords that control encoding and binding style
- Parameters and keywords that control namespace assignment
- How array properties are handled
- Additional notes on web methods in the generated client

C.1 Overview of the Generated Classes

The SOAP Wizard generates classes as follows:

- It generates the web client class, the web service class, or both, according to your selections in the SOAP Wizard. 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:

```
Method DemoMethod() As %String [ Final, SoapBindingStyle = document,
SoapBodyUse = literal, WebMethod ]
{
 Quit ..WebMethod("DemoMethod").Invoke($this,"http://tempuri.org/Demo.MyService.DemoMethod")
}
```

  For a web service, the method looks like the following:

```
Method DemoMethod() As %String [ Final,
SoapAction = "http://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, the SOAP wizard generates an XML-enabled class.
- For each complex type that is a component of the preceding types, the SOAP wizard generates an XML-enabled class.

  The SOAP wizard 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 SOAP Wizard 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

In the generated web client and web service classes, the SOAP Wizard specifies 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 *Caché Class Definition Reference*.

C.3 Parameters and Keywords That Control Namespace Assignment

In the generated classes, the SOAP wizard uses 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 *Caché Class Definition Reference*.

C.3.1 Namespaces for the Messages

The SOAP Wizard specifies the following values to 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 Wizard
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 SOAP Wizard automatically assigns the message types to namespaces as follows:

Table III–2: Namespaces for Types Used By Web Clients and Web Services

Item	Value Given by SOAP Wizard
TYPENAMESPACE (class parameter)	The SOAP Wizard sets this parameter if all methods refer to types in the same namespace.
RESPONSETYPENAMESPACE (class parameter)	The SOAP Wizard 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. The SOAP Wizard 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, the SOAP Wizard creates array-type properties in certain scenarios. You can use the **Create No Array Properties** 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, the SOAP Wizard generates the following class:

```
Class testws.Obj Extends (%RegisteredObject, %XML.Adaptor)
{

Parameter ELEMENTQUALIFIED = 1;

Parameter NAMESPACE = "http://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 select **Create No Array Properties** when you use the SOAP Wizard, the property `MyProp` in generated `Obj` class is defined as follows instead:

```
Property MyProp As list Of testws.Item(XMLITEMNAME = "Item", XMLNAME = "MyProp", XMLPROJECTION =
"COLLECTION");
```

This property refers to the following class, which the SOAP Wizard also generates:

```
Class testws.Item Extends (%SerialObject, %XML.Adaptor)
{

Parameter ELEMENTQUALIFIED = 1;

Parameter NAMESPACE = "http://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.

- Caché 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" in the *Caché Programming Orientation Guide*,) 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 `""`