Xcentrisity^{TM} BIS Tutorial

for ***extend*** ACUCOBOL-GT

Micro Focus
The Lawn, 22-30 Old Bath Road
Newbury, Berkshire, RG14 1QN, UK
www.microfocus.com

Table of Contents

Prerequisites

This tutorial has several prerequisites. Please make sure you have the following available before proceeding.

- XML Extensions reference documentation may be found in **A Guide to Interoperating with ACUCOBOL-GT**, Chapter 11, *Working with Non-Vision Data*.
- Xcentrisity Business Information Server (BIS) must be installed and operating correctly, as demonstrated by the verification and samples operating correctly.
- Xcentrisity Business Information Server reference documentation is installed in the `docs` subdirectory of the *extend* installation directory.
- The *tutorial1.zip* file unzipped as a directory named `tutorial1` in the installation's `samples` directory. The tutorial examples, as delivered, create trace information (see the BIS reference documentation for the {{Trace}} tag). The trace information is placed in a top level directory named `/tmp`. If this directory does not exist, either create the directory with permissions that allow BIS to create files in the directory, or edit the `tutorial1.srf`, `tutorial2.srf` and `tutorial3.srf` files to save the trace information in a directory of your choice.
- A web services client will be necessary for testing web services. This tutorial uses *soapUI*, which may be found at `www.soapui.org`, as well as Microsoft Visual Studio. Download and install *soapUI* as a minimum test client.

Introduction

The goal of this tutorial is to provide an introduction to the terminology of web services, to present current technology choices facing the software designer when creating web services, and to demonstrate practical design patterns for web services implemented in *extend*.

Providing web services involves the use of several technologies: HTTP servers (also known as web servers), XML, and client-server architecture. While successful use of Xcentrisity BIS does not require becoming an expert in any of these technologies, you are encouraged to become familiar with them through the use of online or printed tutorials or reference material.

What is a web service?

Web services are application services , typically combining data and procedural aspects, that are made available over a network such as the internet or an intranet. These services are described in terms of application programming interfaces (API) or web APIs that can be accessed over a network and executed on a remote system hosting the requested services.

An exhaustive description of web services is beyond the scope of this document. Indeed, often web services is a term that includes formal W3C specifications such as SOAP along with less formal, but well described, services such as REST, and even rich internet application (RIA) technologies such as AJAX. Our focus will be on the more formal web services, SOAP and REST, with a brief consideration of RIA techniques.

The role of HTTP in web services

The familiar Hypertext Transfer Protocol (HTTP) is most often associated with the World Wide Web. This is but one of the transfer protocols in use on the internet; others include File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), and the like. HTTP is by far the most used transport protocol for web services, and it plays a crucial role in REST.

Xcentrisity® Business Information Server (BIS) is an application server that 'plugs in' to the two most popular HTTP servers, Microsoft IIS and the Apache HTTP Server. BIS receives requests from and supplies responses to web-based 'user agents'. BIS will be described in more detail later.

The HTTP specification describes messages that represent requests from a client to a server and responses from a server to a client. A message from a client to a server indicates a *method* (i.e., an action) that is desired for a specific resource designated by a *URI,* or *Universal Resource Identifier.* URIs are simply formatted strings which identify by name, location or some other characteristic, a resource located on the internet.

HTTP methods include GET, PUT, POST, DELETE, HEAD, TRACE and CONNECT. For the purpose of creating web services using BIS, only the first four methods are important. The last three methods are handled by the HTTP server above (in architectural terms) the BIS service programs.

SOAP versus REST

Web Services fall into two architectural styles, SOAP and REST.

SOAP, originally defined as Simple Object Access Protocol but now simply SOAP, is a formal specification for the exchange of structured information via Web Services. It relies on XML for its message format, and uses HTTP (which is really an application layer protocol) as its transport protocol. Historically, HTTP was the mechanism used to get through firewalls (since almost all firewalls allow HTTP traffic to pass through) and remains in use today. Note also that HTTPS may also be used, since HTTP and HTTPS are identical at the application layer. The XML based message format consists of three parts: an envelope - which defines what is in the message and how to process it - a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing procedure calls and responses.

REST (which stands for REpresentational State Transfer) is not a formal specification but rather a style of software architecture for distributed hypermedia systems. (The World Wide Web is considered to be a REST system.) The term Representational State Transfer was defined by Roy Fielding in 2000 in his doctoral dissertation. REST was initially described in terms of the HTTP application layer protocol (Fielding was one of the principal authors of the HTTP specification) but is not limited to HTTP. REST uses the HTTP methods as its 'verbs' to implement a create/read/update/delete model for resources described by URIs. In particular, the HTTP GET method is a read-only access to a resource, PUT is a create request, POST is update, and DELETE is delete. There are additional constraints on a (so-called) REST-ful system, of which statelessness is the most pertinent to the programmer using BIS.

Enthusiasts of REST consider REST to be superior to SOAP for several reasons:

(1) REST makes consistent use of the HTTP methods whereas SOAP uses only POST (although a GET on a service endpoint URL is interpreted as a request for the WSDL that describes the service). SOAP's 'overloading' of the POST method obscures the nature of a request, which in turn interferes with caching and other performance related techniques used on the web.
(2) Similarly, REST uses URIs to identify the unique resource being affected by a request. SOAP uses a single URI to identify a 'service endpoint' and describes the resource being affected by a request somewhere in the body of the request. As described above, this interferes with web performance techniques.
(3) SOAP is tied to XML, and is considered by some to be 'wordy.' It should be noted, however, that modern HTTP servers have compression capability built in which would be expected to counteract this disadvantage for SOAP.
(4) REST can be associated with just about any web resource, whether XML-based or not. Remember that the World Wide Web is RESTful.

Whether to use SOAP or REST in creating BIS service programs is probably decided by requirements external to the application system that cannot be considered in this tutorial. However, it is important to remember that Xcentrisity BIS provides the flexibility to participate in HTTP-based systems of either architecture.

SOAP binding style – RPC versus Document

If SOAP web services are to be used, there are additional design considerations. SOAP, as a specification created by committee, has several variations (known as *bindings*) that are possible. The two main variations in use are called RPC/encoded and Document/literal, with the latter being extended to Document/literal wrapped.

The Remote Procedure Call (RPC) pattern is used in situations where the consumer views the web service as a single logical application or component with encapsulated data. The request and response messages map directly to the input and output parameters of the procedure call. Examples of this type the RPC pattern might include a payment service or a stock quote service.

The document-based pattern is used in situations where the consumer views the web service as a longer running business process where the request document represents a complete unit of information. In fact, this type of web service might involve human interaction. An example of this type of service would be a credit application request document with a response document containing bids from lending institutions. Because longer running business processes may not be able to return the requested document immediately, the document-based pattern is more commonly found in asynchronous communication architectures. The Document/literal variation of SOAP is used to implement the document-based web service pattern.

The RPC/encoded SOAP variation was the initial SOAP mechanism to implement the RPC design pattern. However, inefficiencies and other difficulties in large scale enterprise systems have led to RPC/encoded falling into disuse; it is most likely that eventually this variation will be deprecated by the web standardization committee responsible. A form of the Document/literal variation, called 'document/literal wrapped', has been developed and has become the *de facto* standard for RPC pattern web services.

WSDL

WSDL (which stands for Web Services Definition Language or Web Services Description Language) is an XML-based language for describing web services as well as how to locate web services. WSDL is a W3C (web standardization organization) recommendation. The term WSDL is often used to mean the description of a specific web service, as in, "The WSDL for that web service describes three web service methods," or referring generically to a document implemented in WSDL.

The WSDL document describes a web service using four major elements:

(1) <types> – a description of the data elements and type(s) used by the web service;
(2) <message> - a description of the data elements used by each operation available in the web service;
(3) <portType> - a description of the operations performed by the web service; and
(4) <binding> - a description of the message format and communication protocols used for each port described in the <portType section.

In the RPC pattern, the <portType> element describes the functions, or methods, that are available in the web service; the <message> section describes the input and output parameters; and the <types> section describes those parameters. It is not unusual for RPC pattern WSDLs to be derived from underlying functions in standard programming languages, the so-called bottom-up approach. In Xcentrisity Business Information Server, this capability is included and is described in detail in the next section.

In the document-based pattern, WSDLs are often developed by the architects of the system using a top-down approach, before any implementation of the service or the clients of the service. The documents that are exchanged are typically designed before the WSDLs that use them; these documents are described using XML Schema, another XML-based language used to describe XML documents. (The <types> section of a WSDL is actually an embedded XML Schema document. WSDL has an import capability that allows importation of externally defined XML Schema documents.)

Create a simple SOAP/RPC web service

Within legacy systems, creating web services using the bottom-up, RPC pattern methodology is often the quickest means to expose exiting functionality in the legacy system. Xcentrisity Business Information Server, using XML Extensions, provides a simple mechanism to create such web services.

While a more complex example will be created later, we will use a simple data lookup for our first example. A desired company name, which may be a fragment of a name, will be provided. The desired result is the data from the 'first' record for which the company name is greater than or equal to the desired company name. (Note that the data used for the sample programs in this tutorial is excerpted from North American Numbering Plan data. The data are not complete and contain inaccuracies to render the data unsuitable for any commercial application.)

Data naming convention for input/output parameters and methods

BIS uses a set of naming conventions to define a data area which in turn is used to define an RPC pattern web services interface. It is important to follow the naming conventions; the benefit is the ability to create a WSDL, process web service requests, and provide web service responses with a level of simplicity.

Here is the data area definition for the simple look up:

```
78 Service-URI          VALUE "{{Value(""tutorial.srf"",HTMLDECODE,MAKEABS)}}".
78 Service-Name         VALUE "tutorial".
78 SOAP-Action-URI      VALUE "http://tempuri.org/bis/samples/action/tutorial".
78 Method-Namespace-URI VALUE "http://tempuri.org/bis/samples/tutorial/".
78 HTTP-Method-POST     VALUE "POST".
78 HTTP-Method-GET      VALUE "GET".
01 SOAP-Request-Response.
   10 HTTP-Method                   VALUE HTTP-Method-POST.
      88 HTTP-Method-Is-POST        VALUE HTTP-Method-POST.
      88 HTTP-Method-Is-GET         VALUE HTTP-Method-GET.
   10 Method-Name PIC X(100)        VALUE SPACES.
      88 Method-Is-Find              VALUE "find".
   10 Fault-Area.
      20 FaultCode                  PIC X(10) VALUE SPACES.
      20 FaultString                PIC X(30) VALUE SPACES.
      20 FaultDetail                PIC X(80) VALUE SPACES.
   10 Find--Method-Parameters.
      20 Input-Parameters.
         30 desired-company-name    PIC X(50).
      20 Output-Parameters.
         30 Result                  PIC X(80).
         copy  "offcode.rec" replacing ==05== by ==30==.
```

First a series of constants (level 78) is defined, to collect many of the service-specific values into a single block.

1. The Service-URI is a string that encodes a BIS *value* tag. (See the *Xcentrisity Business Information Server user's Guide* for reference information.) This tag specifies the SRF file that will be the service endpoint; the value tag attributes HTMLDECODE and MAKEABS indicate to the BIS request handler (described below) that the SRF filename (tutorial1.srf in this case) should be synthesized into a URI by adding the additional parts of the URI to the filename. When this value tag is returned in the WSDL, it will be replaced by the actual URI of the service endpoint before the WSDL is sent to the client.
2. The Service-Name is a string that is used to identify the service to clients; it becomes the value of the *name* attribute in the <wsdl:service> tag. The actual use of the Service-name depends on the programming language of the client, but it will appear in the client's API, so a meaningful name is recommended.
3. The SOAP-Action-URI is a string that is used to uniquely identify a <soap:operation>. This string should take the form of a URI. As shown in this example, you may use the tempuri.org domain for testing purposes. However, you are encouraged to use a unique, permanent URI for published web services; you could use your company's domain as part of the URI value. Note that this URI does not point to an actual resource on the web.
4. The Method-Namespace-URI is a string that is used in the *targetNamespace* attribute of the <wsdl:definitions> tag. Like the SOAP-Action-URI, this URI should be unique, and otherwise follow the same guidelines.

In the definition of the 01-level SOAP-Request-Response, the first items (through *10 Fault-Area*) should be defined as they are here. These definitions convey values to the XSLT stylesheets that create the WSDL, import the SOAP request, and form the SOAP response, and the names of these items must remain the same. However, condition-names (level 88) may be added, as these do not affect the values. Condition-names may be convenient for enumerating the possible method name values; note that, by default, method name values are all 'folded' to lower case by the XSLT style sheets.

Method parameter definitions follow *10 Fault-Area*, with a separate level 10 group data item defined for each method (function) in the service. The naming convention for these group items is: *methodname*--method-parameters, where *methodname* is the desired name of the method, followed by **two** hyphens. In this first example, there is only one method, named *find*. (Note that the method name is 'folded' to lower case, so *Find*, *FIND* and *fInD* all result in a method named *find*.)

Within each method parameter group item, zero, one, two or three group items may exist. The names of these group items are: INPUT-PARAMETERS, OUTPUT-PARAMETERS, and INPUT-OUTPUT-PARAMETERS. Input and output parameters are defined within each of these groups as appropriate. (Note: complex structures and arrays of structures can sometimes cause difficult programming issues on clients. More about this below.)

Simple design pattern

The COBOL service program for the web service takes the form of a controller. (In the model-view-controller, MVC, web architecture for applications, the controller is the component that receives the GET or POST input and invokes domain objects – i.e. the model – that contain the business rules that perform a specific task and produce the output.)

First, the controller has to receive the input. Let's look at the code that does this.

```
       Preset Section.
       A.
           XML INITIALIZE
           If Not XML-OK Go To Z.

       Preset-Request-Data.

           CALL "C$GetEnv" USING "BIS_FILENAME",
                                 BIS-Exchange-File-Name,
                                 BIS-Exchange-File-Result.

           If not BIS-Exchange-File-Result = 0
               DISPLAY "Could not obtain the BIS Exchange filename"
               STOP RUN.

           Display "BIS Exchange File: " BIS-Exchange-File-Name.

           Perform Process-SOAP-Requests.
           Stop Run.
```

The code to this point is normal 'reference manual' initialization for a BIS service program. The XML Extensions package is initialized and the exchange document file, which contains the request from the client, is located. The dispatcher (Process-SOAP-Requests) is PERFORMed. (The term 'dispatcher' is used in the model-view-controller architecture. The dispatcher is the code that determines what is being requested, and calls the code to perform the request.)

```
       Process-SOAP-Requests Section.

       Get-Request.
           XML SET XSL-PARAMETERS
               "Method_Namespace"     Method-Namespace-URI.   *> all
           If Not XML-OK Go To Z End-If.
           Call "B$ReadRequest" Giving BIS-Status
           If Not BIS-OK Go To Z.

      *    At this point, the SOAP request payload elements
      *    are available to the application in the exchange file.

           Initialize SOAP-Request-Response.
           Move HTTP-Method-POST To HTTP-Method.
           move Service-URI to SOAP-Address.
           move SOAP-Action-URI to SOAP-Action-Prefix.
           move Method-Namespace-URI to Method-Namespace.
           move Service-Name to Interface-Name.

           XML IMPORT FILE
               SOAP-Request-Response        *> data item to import into
               BIS-Exchange-File-Name       *> import document file name
               "SOAP-Request-Response"      *> model data-name
               "soap_request_to_cobol.xsl". *> stylesheet for transform
           If Not XML-OK Go To Z.
      *    The request has been imported into SOAP-Request-Response

           If HTTP-Method-Is-GET
      *        GET is the HTTP method that is used to obtain WSDL
               Perform Write-WSDL
               Stop Run
           End-If.

       Dispatch-Request.
```

The dispatcher code first calls B$ReadRequest, which is a synchronization routine that will wait until the BIS request handler (running in the HTTP server) has actually placed the input request document in the exchange file. The request is then imported from the exchange file into the level 01 record area. At this point the dispatcher makes its first decision. If the request is a GET, the WSDL for the web service is created as the response to the client. Otherwise, the controller knows this is a POST request that is invoking a method.

WSDL creation/response

WSDL creation is almost entirely handled by a supplied XSLT style sheet. Again, let's look at the code.

```
       Write-WSDL.

           XML ENABLE ATTRIBUTES
           If Not XML-OK Go To Z.

           XML ENABLE ALL-OCCURRENCES
           If Not XML-OK Go To Z.

           XML SET XSL-PARAMETERS
               "SOAP_Address"         Service-URI              *> WSDL
               "SOAP_Action_Prefix"   SOAP-Action-URI          *> WSDL
               "Interface_Name"       Service-Name             *> WSDL
               "Method_Namespace"     Method-Namespace-URI.    *> all
           If Not XML-OK Go To Z End-If.

           XML EXPORT FILE
               SOAP-Request-Response   *> data item to export from
              BIS-Exchange-File-Name  *> exported document file name
              "SOAP-Request-Response" *> model data-name
              "cobol_to_wsdl.xsl"     *> stylesheet for transform
          If Not XML-OK Go To Z End-If
          Call "B$WriteResponse" Using
                BIS-Response-SessionComplete
                Giving BIS-Status
          If Not BIS-OK Go To Z End-If.
```

This paragraph exports the WSDL to the exchange file using the *cobol_to_wsdl.xsl* style sheet. This style sheet is somewhat special in that it uses the structure of the SOAP-Request-Response record area to derive much of the information for the WSDL. (More 'normal' exports are focused on exporting data within a structure, rather than the structure itself.) It is for that reason that attributes and all occurrences are enabled. Additional metadata values are passed to the style sheet as XSL parameters. Furthermore this style sheet depends on the previously described naming conventions properly to identify all the methods and their parameters. B$WriteResponse is then called to notify the request handler that the response is in the exchange file; the request handler will send the contents of the exchange file to the client.

SOAP request/response

When a SOAP request is detected, the dispatcher is responsible for invoking the business rules associated with the requested method.

```
       Dispatch-Request.
           If Not Method-Namespace-Is-OK
               Move "env:client" To FaultCode
               Move "bis:WrongNamespace" To FaultString
               Move "Wrong namespace for this interface"
                 To FaultDetail
               Perform Indicate-Hard-Fault
           Else
               Evaluate True
                   When Method-Is-Find
                       Perform Process-Find-Method
                   When Other
                        Move "env:client" To FaultCode
                        Move "bis:WrongMethod" To FaultString
                        Move
                             "Method invoked is unknown to this interface" To
                                   FaultDetail
                        Perform Indicate-Hard-Fault
               End-Evaluate
           End-If.

           Stop Run.

       Process-Find-Method.
           open input office-code-file.
           move spaces to output-parameters of Find--method-parameters.
           if office-code-success
               move desired-company-name of input-parameters of Find--method-parameters
                 to company-name of office-code-file
               start office-code-file key Not < company-name of office-code-file
                  invalid key move "Not Found" to result of Find--method-parameters
                  not invalid key
                      read office-code-file next
                         at end move "Not Found" to result of Find--method-parameters
                         not at end
```

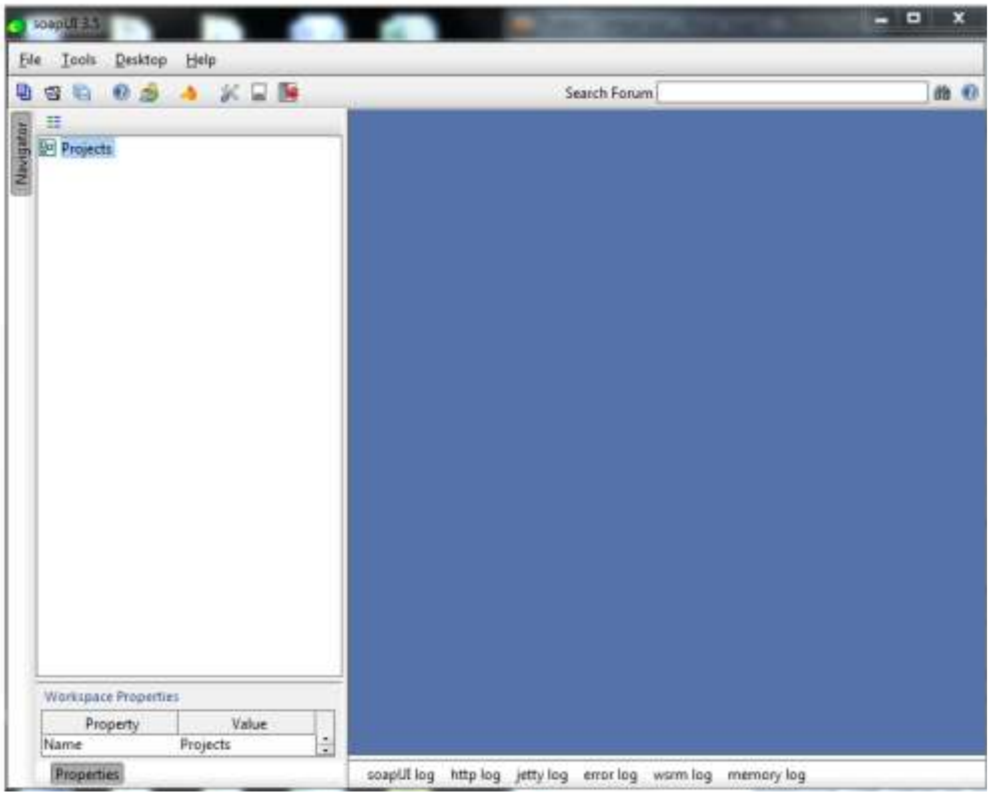
```
                          move corr office-code-record
                                 to output-parameters of Find--method-parameters
                 end-read
            end-start
      else
          move "Unrecoverable Error" to result of Find--method-parameters
      end-if.
      perform Issue-response.

   Issue-Response.
       XML EXPORT FILE
           SOAP-Request-Response         *> data item to export from
           BIS-Exchange-File-Name        *> exported document file name
           "SOAP-Request-Response"       *> model data-name
           "cobol_to_soap.xsl". *> stylesheet for transform
       If Not XML-OK Go To Z.

       Call "B$WriteResponse" Using
           BIS-Response-SessionComplete
           Giving BIS-Status
       If Not BIS-OK Go To Z.
```

After an import of a request, the method-name field contains the requested method (folded to lower case) and the input (and input-output) parameters have been stored in the appropriate --method-parameters area. (This again is the result of the style sheet using the naming conventions described earlier.) The dispatching code checks for some errors (for example, being called erroneously by a client wanting to use a different service) and then uses EVALUATE method-name to invoke business rules appropriate to the method. After the business rules execute (the paragraph Process-Find-Method in the example), the SOAP response is exported to the exchange file (once again, the style sheet uses the method-name along with the naming conventions to 'know' which output-parameters contain the desired result data) and B$WriteResponse is called to notify the request handler that the response is in the exchange file; the request handler will send the contents of the exchange file to the client.

Invoke web service using soapUI tool

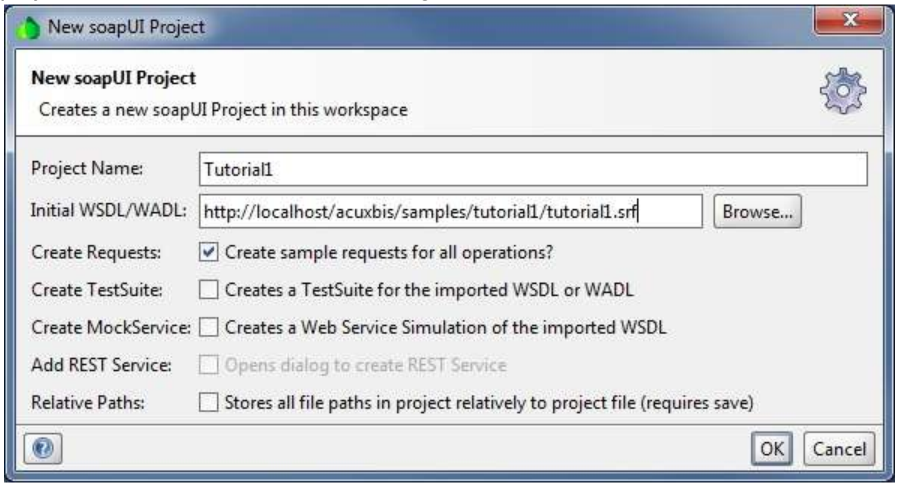
Once you have created a web service, it is time to take it out for a 'test drive.' Probably the best known web services test tools is *soapUI*, which is available in both open source and 'Pro' forms, the former being free to download. soapUI is available for most modern operating systems. (soapUI is implemented in Java, so you will be immediately testing in a cross-language environment.) Install soapUI according to its instructions. (Please note that soapUI has many capabilities that will not be exploited in this tutorial.)

When you start soapUI and dismiss its start up screen, you are presented with a work area as shown below:



The left pane provides an area to define ‘projects’ and the right pane contains windows that are associated with a selected project. Follow these steps to create a project to test the first example.

From the File menu, select New soapUI Project. A dialog appears which allows you to provide a name for your project, as well as the location for finding the WSDL.



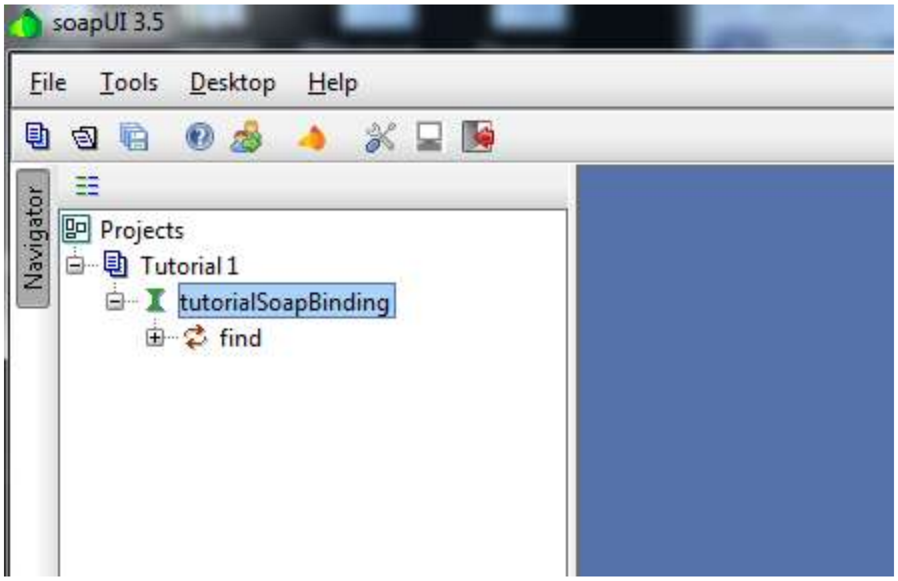
New soapUI Project

New soapUI Project

Creates a new soapUI Project in this workspace

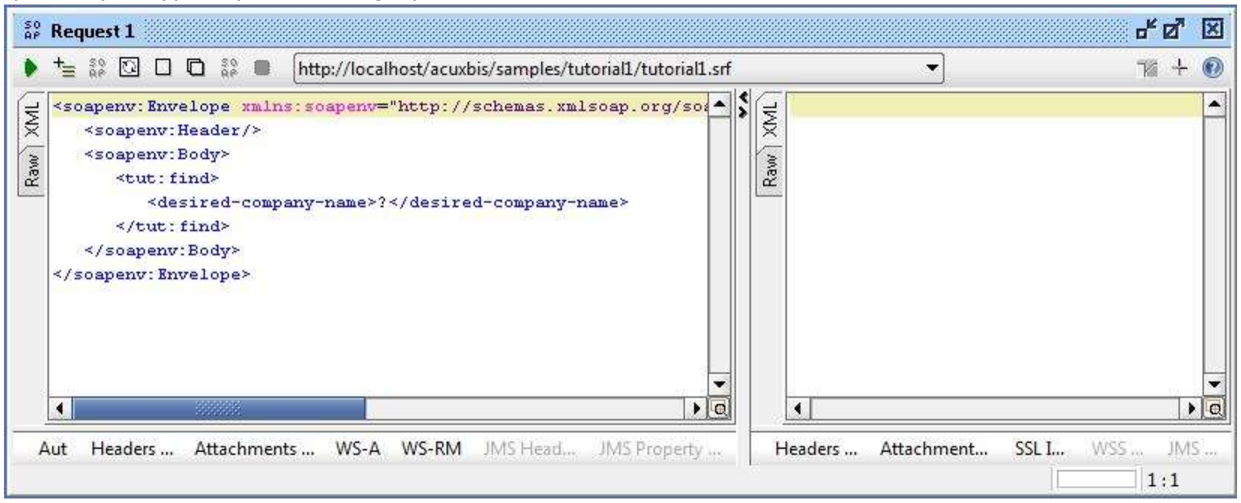
Project Name:	Tutorial1
Initial WSDL/WADL:	http://localhost/acuxbis/samples/tutorial1/tutorial1.srf
Create Requests:	[✓] Create sample requests for all operations?
Create TestSuite:	[ ] Creates a TestSuite for the imported WSDL or WADL
Create MockService:	[ ] Creates a Web Service Simulation of the imported WSDL
Add REST Service:	[ ] Opens dialog to create REST Service
Relative Paths:	[ ] Stores all file paths in project relatively to project file (requires save)

Since our web service provides a WSDL as the result of a GET on the endpoint, we simply enter a name for the project and the URL of the web service end point. Make sure the *Create Requests:* option is checked so that soapUI creates a prototype request document for each method. After fetching and processing the WSDL, the result is displayed in the left pane:



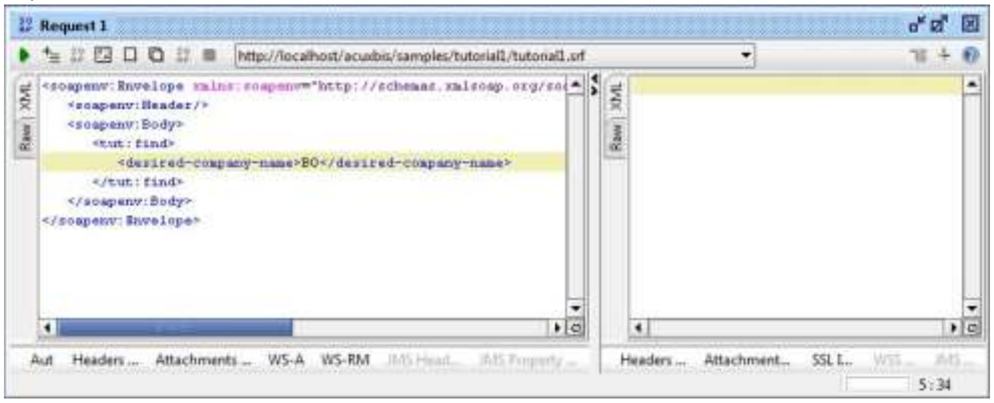
Note that the *tutorialSoapBinding* has a single method named *find*.

Expand the selection for the *find* method, exposing a prototype request named *Request 1*. Double click *Request 1* to open the prototype request in the right pane:


Request 1
http://localhost/acuxbis/samples/tutorial1/tutorial1.srf
XML
Raw
Aut Headers ... Attachments ... WS-A WS-RM JMS Head... JMS Property ...
Headers ... Attachment... SSL I... WSS ... JMS ...
1 : 1

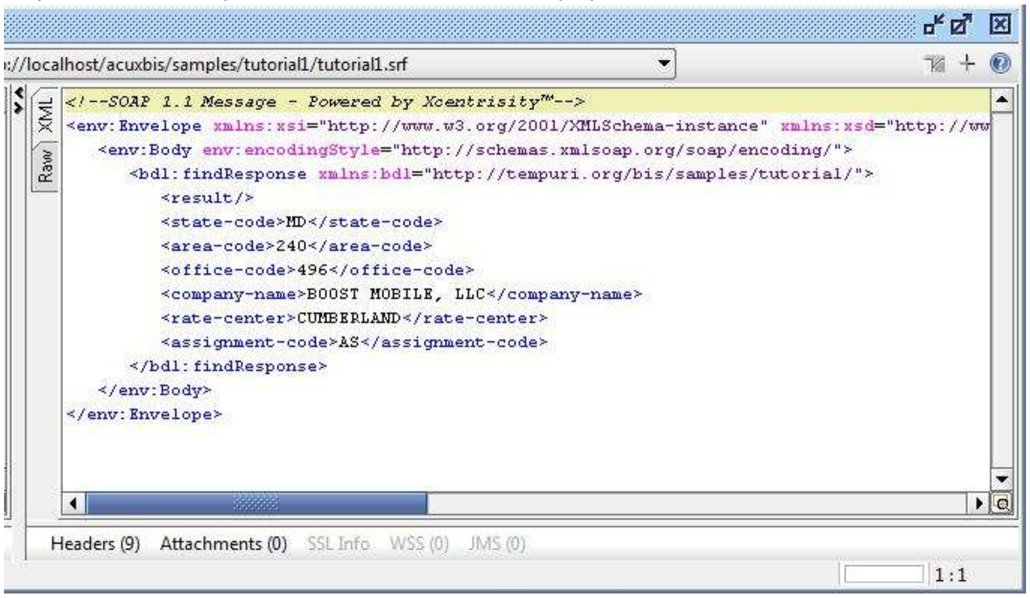
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/so
   <soapenv:Header/>
   <soapenv:Body>
      <tut:find>
         <desired-company-name>?</desired-company-name>
      </tut:find>
   </soapenv:Body>
</soapenv:Envelope>
```

The prototype request has the question mark (?) character in those areas that need input values to create a valid request. In the case of this tutorial example, we need to supply an alphanumeric value which the web service will use to determine which company name to return. Let's enter BO and press the green arrowhead at the upper left of the request window.


Request 1
http://localhost/acuxbis/samples/tutorial1/tutorial1.srf
XML
Raw
Aut Headers ... Attachments ... WS-A WS-RM JMS Head... JMS Property ...
Headers ... Attachment... SSL I... WSS ... JMS ...
5 : 34

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/so
   <soapenv:Header/>
   <soapenv:Body>
      <tut:find>
         <desired-company-name>BO</desired-company-name>
      </tut:find>
   </soapenv:Body>
</soapenv:Envelope>
```

soapUI sends the request to the web service and displays the result:



```
<!--SOAP 1.1 Message - Powered by Xcentrisity™-->
<env:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://ww
   <env:Body env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <bdl:findResponse xmlns:bdl="http://tempuri.org/bis/samples/tutorial/">
         <result/>
         <state-code>MD</state-code>
         <area-code>240</area-code>
         <office-code>496</office-code>
         <company-name>BOOST MOBILE, LLC</company-name>
         <rate-center>CUMBERLAND</rate-center>
         <assignment-code>AS</assignment-code>
      </bdl:findResponse>
   </env:Body>
</env:Envelope>
```

This indicates that the web service has created a WSDL and successfully processed a web service request.

Create clients in PHP, Perl, Java, Python

Using a web service involves creation of a client that will marshal input parameters, create and transmit the SOAP request, receive the SOAP response and de-marshal the output parameters into a form usable by the client language. This client is often called a *proxy* for the web service, presenting the web service as a function call in the client's programming language.

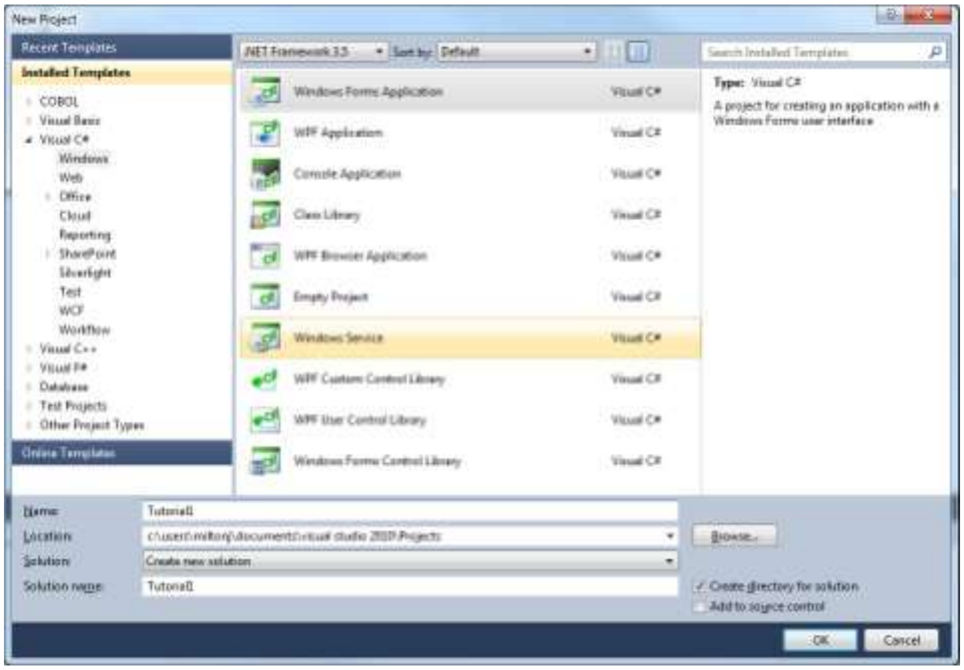
Because web services are so pervasive, each language has one or more tools available for creating web service clients. Some of these are:
PHP: native xml-soap extension, SoapClient , NuSOAP
Perl: SOAP::Lite, SOAP::WSDL
Python: SUDS
Java: Many exist including those from Sun and Apache

Using these various packages is beyond the scope of this tutorial. However, there are some cautions about using tools to create web service clients. First, unless you expect the contract represented by the WSDL to change frequently, be sure that the proxy either fetches the WSDL at design time or that the WSDL is cached between multiple invocations of a web service. Second, if you wish to pass typical COBOL hierarchical structures (known as complex types), investigate first to see if the tool you wish to use supports such structures. You may have to simplify your input and output parameters to be able to interoperate with some less capable tools.

Add a web service to a Visual Studio 2010 project

Visual Studio 2010 provides a sophisticated ability to include web services in projects. Let's take a closer look at this popular programming IDE.

First, create a new project:



Type: Visual C#

A project for creating an application with a Windows Forms user interface

Name: Tutorial1

Location: c:\users\milton\documents\visual studio 2010\Projects

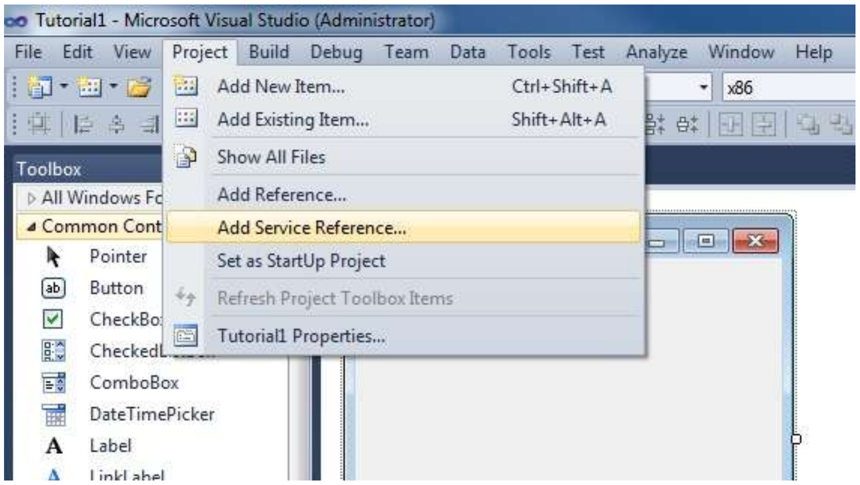
Solution: Create new solution

Solution name: Tutorial1

✓ Create directory for solution

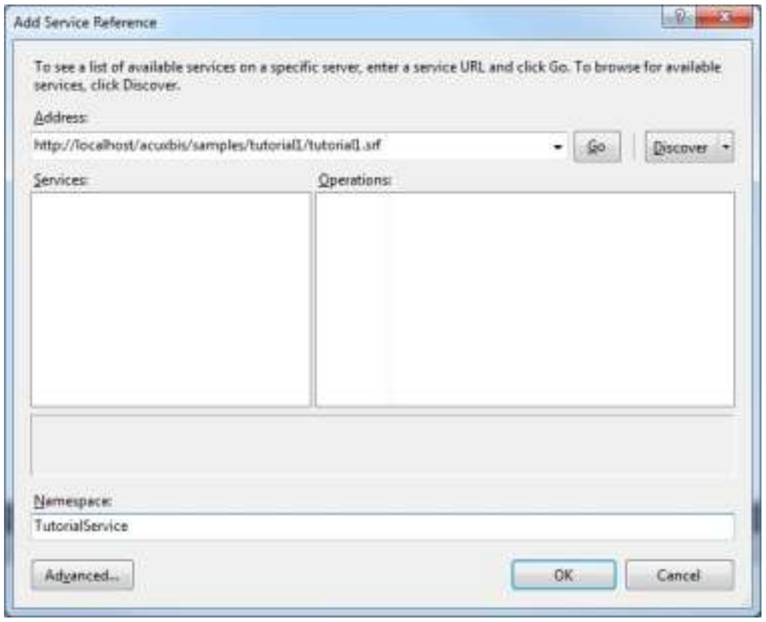
Add to source control

After the project is created, you add a service reference to the project.

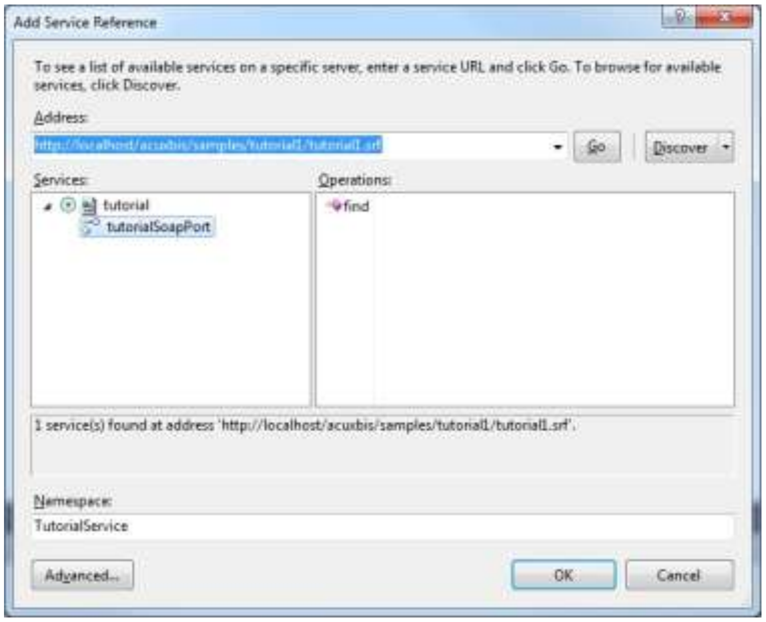


Tutorial1 - Microsoft Visual Studio (Administrator)

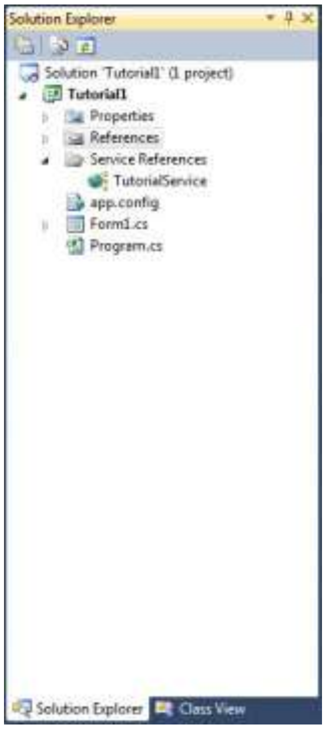
When the Add Service Reference dialog appears, enter the URL of the SRF file that is the endpoint of the service.


Add Service Reference
To see a list of available services on a specific server, enter a service URL and click Go. To browse for available services, click Discover.
Address:
http://localhost/acuxbis/samples/tutorial1/tutorial1.srf
Go
Discover
Services:
Operations:
Namespace:
TutorialService
Advanced...
OK
Cancel

The WSDL will be fetched, at which time the information about the web service is added to the dialog:


Add Service Reference
To see a list of available services on a specific server, enter a service URL and click Go. To browse for available services, click Discover.
Address:
http://localhost/acuxbis/samples/tutorial1/tutorial1.srf
Go
Discover
Services:
tutorial
tutorialSoapPort
Operations:
find
1 service(s) found at address 'http://localhost/acuxbis/samples/tutorial1/tutorial1.srf'.
Namespace:
TutorialService
Advanced...
OK
Cancel

The methods available in the service are listed in the 'Operations:' column. Click OK to complete adding the service to the project. Proxy functions are created for the operations (methods) and the service appears in the Solution Explorer.



The generated proxy functions allow you to use the web service's methods in a manner identical to any other function.

Intro to XML Extensions

XML Extensions is a facility that allows *extend* applications to interoperate freely and easily with other applications that use the XML standard. XML Extensions provide the ability to import and export XML documents to and from COBOL working storage in a natural and intuitive way to the COBOL programmer.

Concept of XSLT

XSLT, which stands for Extensible Stylesheet Language Transformations, is a declarative, XML-based language used for the transformation of XML documents into other XML documents. The input document is used to create a new output document using the data in the input document along with the rules described in the XSLT document. The style of programming is that of a functional language with string matching, rather than a procedural language. Learning to create XSLT 'programs' is not difficult, but changing to the functional programming paradigm for those accustomed to procedural languages does require a different approach to problem solving.

NOTE: The term XSL is often used to mean XSLT. XSL is actually a family of standards that includes XSLT. For most practical purposes, this subtle distinction can be ignored.

XML Extensions creates (exports) or consumes (imports) only documents which 'match' the hierarchical structure of a COBOL record area. When the external requirements for an XML document do not achieve this match, XSLT is used. For example, SOAP web services require the creation of a very complicated XML document, the WSDL, to describe the web service. Xcentrisity BIS uses XSLT to create the WSDL from a COBOL data description – matching the relatively simple data layout to the complex WSDL XML document.

NOTE: The following discussion uses terminology that presumes an understanding of XSLT. A thorough explanation of XSLT is beyond the scope of this tutorial. Several excellent books and online tutorials are available that teach XSLT.

How XSLT creates a BIS WSDL (high level only)

As described earlier, a BIS service program describes its API using a 01-level group item named *SOAP-Request-Response*. This data item has specific naming conventions that describe input and output parameters for each method within the web service. The XSLT that describes the transformation from the basic XML document that is exported from SOAP-Request-Response to the WSDL XML document that is returned to the web client is named *cobol_to_wsdl.xsl*.

cobol_to_wsdl.xsl differs somewhat from other 'normal' XSLT used with XML Extensions. While most XSLT are focused on rearranging data being exported from or imported to the COBOL program, *cobol_to_wsdl.xsl* instead uses the metadata in the XML document being exported from the COBOL web service program to create another metadata document, the WSDL. The metadata used consists of the values of some XSL parameters, the element names (resulting from the naming conventions used), and information exported due to the use of XML ENABLE ATTRIBUTES.

Let’s have a brief review of the naming conventions found in SOAP-Request-Response. Each method in the web service is defined by a group item immediately subordinate to SOAP-Request-Response with the name `methodname--method-parameters` (note the two hyphens after *methodname*), where *methodname* is the name of the method being defined. Within each of these groups, one to three group items, named `input-parameters`, `output-parameters` and/or `input-output-parameters`, may be defined. The first use of this naming convention can be seen at the beginning of the first <xsl:template>, where a variable named `$methods` is created and contains all the elements whose names end in `--method-parameters`. The `$methods` variable is then used throughout the rest of the template to iterate over all methods.

Let’s look at the major sections of *cobol_to_wsdl.xsl*, which correspond to the four sections of a WSDL.

The internal schema, which contains most of the complexity of a WSDL, is created inside the <wsdl:types> element. While there is a fairly large amount of code involved, this essentially involves iterating over all the methods and creating XML Schema descriptions of each of the request and response parameters. Special attention is paid to arrays (OCCURS) and to structures (COBOL group items). In particular, arrays must be named in a manner such that the arrays can be identified by the naming convention in an incoming SOAP request where the metadata of the COBOL structure is not available to assist interpretation. ‘Wrapper’ request and response elements are also created which conform to the requirements of *document/literal wrapped* SOAP requests.

Next, the <wsdl:message> elements are created for each method, followed by the <wsdl:operation> elements (within the <wsdl:portType> element). These elements are relatively simple in *document/literal wrapped* SOAP requests.

Finally, the <wsdl:operation> elements for each method are generated within the <wsdl:binding> element. The WSDL is then completed with the <wsdl:service> element creation.

How XSLT processes a SOAP request (high level only)

When importing a SOAP request, three things are necessary:

1. The BIS service program must know the HTTP method being used,
2. The SOAP method being requested, and
3. The input parameter values for the SOAP request.
   The *soap_to_cobol.xsl* is the XSLT document that describes this transformation.

The BIS request document consists of four major parts.

The actual payload of the request is wrapped inside a <bis:content> element. ,

1. The cookie values passed by the client are wrapped inside a <bis:cookies> element,
2. The server variable values are wrapped inside a <bis:server-variables> element, and
3. The query parameters from the URL are wrapped inside a <bis:query-params> element.

The latter three sets of values may be obtained by the service program by including the following group items (preserving the names) in the SOAP-Request-Response group item:

```
05  s--cookies.
     07  s--cookie occurs 20.
```

```
            09  s--name   pic x(40).
            09  s--value  pic x(100).
    05  s--query-parameters.
         07  s--query-parameter occurs 20.
            09  s--name   pic x(40).
            09  s--value  pic x(100).
    05  s--variables.
         07  s--variable occurs 100.
            09  s--name   pic x(40).
            09  s--value  pic x(100).
```

Cookies, query parameters and server variables are typically presented as name-value pairs. The name-value pairs are stored in the arrays described above.

As part of the retrieval of the server variables, the value of the server variable named REQUEST_METHOD is placed in the http-method.

The remainder of the XSLT is devoted to retrieving the payload information from the <bis:content> element. The payload is a SOAP envelope, which in turn contains a <SOAP:body> element that contains the actual method request..

A *document/literal wrapped* SOAP request has a single element (the so-called wrapper element) inside the SOAP:body. The name of this element is the name of the requested method. By concatenating the element (method) name with the string xx the subsequent parameter values may be directed (imported) into the appropriate method's input parameter group.

After the method name has been determined, the elements subordinate to the wrapper element are processed to obtain the input parameter values. This processing is straightforward with the exception of discovering array elements which use a naming convention (see the description of WSDL processing above).

How XSLT processes a SOAP response (high level only)

Exporting a SOAP response involves forming a correct SOAP envelope which conforms to the response defined by the WSDL. The *cobol_to_soap.xsl* XSLT is the document that describes this transformation.

The method name for the response is exported, along with all the other data in SOAP-Request-Response. The method name is used to determine which `output-parameters` contains actual output parameters of interest. And, as in the import of a SOAP request, item names for array elements are adjusted to conform to the naming convention described in the WSDL.

Data flow in BIS

While an in-depth understanding of the inner workings of BIS is not necessary for the successful implementation and use of web services, it is useful to understand the steps the request and response messages go through during the processing of a single web service request. More information about particulars can be found in the reference manual for Xcentrisity Business Information Server.

A BIS server is composed of two cooperating processes: a request handler running under the control of an HTTP server (IIS or Apache), and a service program, programmed in COBOL, running under the control of the Xcentrisity service engine, which is very similar to the AcuCOBOL-GT runtime. A third process is also involved: the web client agent.

Request handling

A web service request begins when a web client agent (for example, JavaScript in a browser) sends an HTTP request for a URL designating an SRF file (also known as a BIS stencil file). When the HTTP server receives this request, it determines that the appropriate 'request handler extension' for such a request is the BIS Request Handler.

The BIS request handler interprets the tags in the SRF file in the order in which the tags appear. A tag is composed of text surrounded by **{{** and **}}** sequences, and tags may be interpreted as processing instructions or placeholders that are replaced by plain text, HTML or XML that is generated by the BIS service engine or by the BIS request handler. In the case of SOAP web services the SRF file is very succinct as shown below.

One of the tags, the XMLExchange tag, causes the request handler to format the request information, including the state of the HTTP server variables, cookies, query parameters and the request payload., into a standard BIS request. The BIS request is transmitted to the service program via the 'exchange file'. Then, the request handler suspends its processing of the request until the service program informs it that a response is available.

SRF file

Let's look at an SRF file for a SOAP web service.

```
{{ Handler * }}{{//}}
{{ RunPath(bin,../common) }}{{//}}
{{ StartService(tutorial2 -v) }}{{//}}
{{ XMLExchange }}
```

First, let's understand the rather curious looking tag, {{//}}. This tag has the job of consuming whitespace (that is, preventing the {{//}} tag and any surrounding whitespace from being rendered back into the response payload). Whitespace includes spaces, tabs, form feeds and the like, between the other tags. The use of this tag allows us to write an SRF file that presents each tag on a separate line even though the rules of SOAP responses preclude the inclusion of such whitespace to make the response more human-readable.

The {{Handler}} tag is basically a signature tag that must appear within the first few hundred characters of the SRF file. It is replaced with a zero-length string in the response and is used internally by the HTTP server and request handler.

The {{RunPath}} tag is used to set environment variables for the service program so that the service program can find its object and data files. This is similar to shell script commands that are used to set these environment variables in the normal runtime environment. The {{RunPath}} tag is replaced by a zero-length string in the response.

The {{StartService}} tag is used to request that a service program be started by the service engine. (Note that in stateful applications, the StartService tag is used to determine whether the correct service program is running in the saved state of the session. See the reference documentation for more information.) The {{RunPath}} tag is replaced by a zero-length string in the response.

Finally, the {{XMLExchange}} tag functions as the synchronization point between the request handler and the service program. The request handler suspends processing the request until signaled by the service program.

Service program

The service program designated in the {{StartService}} tag is activated when that tag is processed. However, the contents of the request may not yet be available in the exchange file. It is for this reason that the service program calls B$ReadRequest to wait for the request to be made available.

When the request becomes available, B$ReadRequest returns control to the service program. At that point, the request XML document that is in the exchange file may be imported, using XML Extensions, into the service program's working-storage data area. A provided XSLT style sheet transformation is applied when the only expected request is a SOAP web service request. (Note that a more flexible service program might import the request XML document more than once, in order to determine what kind of request was being presented.)

After the request document is imported, the service program acts upon the request and produces a response, which is placed in the exchange file. Note that this response does not have to be an XML document (for some examples see below) but in the case of a SOAP response, it is the entire SOAP response envelope.

After the response is in the exchange file, the service program signals this fact by calling B$WriteResponse. In a stateless web service environment, the call should include the option that indicates that the session should be terminated. The service program then terminates in a normal, orderly manner.

Response handling

When the service program signals that a response is available, the request handler replaces the XMLExchange tag with the entire contents response contained in the exchange file. During this process, the request handler also examines the contents of the exchange file for certain request handler tags, and processes those tags before sending the response back to the web client. These tags are useful in controlling information such as the MIME type of the response and other information that may only be determined by the service program. Likewise, the service program may wish to send information in the response that is known only by the request handler; replacement tags are used for this purpose.

When the request handler encounters the end of the SRF file, the response that has been accumulated is sent back to the web client.

BIS Session management

The Xcentrisity BIS has session management capabilities that may be exploited. Session management involves the reservation of certain web server resources – memory, processes, etc. – for the use by a specific web client. In the case of BIS, the state of a service program, including data values, open files, current record pointers, etc., may be maintained between requests from a specific web client.

Session management is useful for interacting with services that must return large amounts of data under controlled circumstances. However, session management carries some inefficiency as well; the web client's request must be routed to a specific server, and significant memory and process resources may be held waiting for a misbehaving client that fails to return to use them. Use caution if you are considering a stateful BIS implementation.

BIS uses a cookie hold a session identification key. When a service program indicates that a session is to be maintained, and there are not SRF tags indicating otherwise, a cookie is sent to the web client with the response. The web client then sends this cookie back with its next request; the request handler uses the cookie value to reconnect the request with the correct session, thereby routing the request to a service program already waiting for it.

Complex design pattern

While there may be situations where a simple web service implementing a single method is appropriate, a somewhat more complex design pattern may be illustrative of the controller concept.

Using a simple indexed file, create a web service that implements CRUD

One popular organizing design pattern is the Create, Read, Add, Delete (CRUD) function group for persistent storage in an application. The acronym CRUD is often used to describe this function group. By adding the capability to browse on one or more fields (as provided by *tutorial1*), and by a careful renaming of the functions, we can create a design pattern known as BREAD (Browse, Read, Edit, Add, Delete).

The BREAD design pattern applied to an indexed file, or a multiple related indexed files, is useful in exposing data in an existing application. *tutorial2* provides an example of a web service implementing the BREAD function set on a single indexed file.

tutorial2 illustrates a few more naming conventions that are useful in communicating information between the COBOL service program and the XSLT style sheet. First, if you wish to control the case of the characters that spell a method name, you may override the default behavior of folding the method name to lower case. This is shown on each of the methods. In particular, you place a *method*--METHOD-NAME data item at the same level as the *method*--METHOD-PARAMETERS group item. The desired spelling (differing in case only) is entered as the value of the *method*--METHOD-NAME data item. Note that the values in the *method*--METHOD-NAME items must be maintained for both WSDL and response export.

In a similar manner, one can control the externally visible naming of parameters. This is illustrated in the Browse method output-parameter definition, where the data item found-item--name is used to rename the very COBOL-like name found-record to FoundRecord. Note that, unlike method names, the desired spelling can be different for data item names that are output parameters; however, spellings for input parameters may differ only in case.

Finally, also in the Browse method output-parameters, the specially named data item found-record--count is used to communicate the number of occurrences that should be exported. A --count data item may also be used on an imported data item array so that the actual number of array items imported may be known.

Optimistic concurrency

Many, if not most, multiuser applications that use indexed files use *pessimistic concurrency* (record locks) to avoid having two users attempt to update a record at once, thereby losing one of the updates. The name pessimistic concurrency is used because record locks are often obtained and held while a user is making changes.

If pessimistic concurrency (i.e. normal record locks) is used in a web services environment, the service program must be stateful. That is, the service program must continue to exist while the web client is acting on the data, waiting for the web client to release the lock, or to time out. In reality, though, it is often the case that the web client may never return (consider JavaScript in a browser when the user closes the browser window).

The answer to the operation difficulties in a web service environment of pessimistic concurrency is the use of *optimistic concurrency*. In optimistic concurrency, the contents of a record are recorded or remembered at the time a record is read. Then, when a user desires to change or delete the record, the original contents of the record are compared to the current contents of the record. If the contents have not changed, the update of delete operation is completed. If the contents have changed, the user is notified and the change is not made.

tutorial2 demonstrates a tactic for detecting a change in record contents without storing a copy of the original contents of a record. A *message digest* (also known as MD5) is computed when the record is originally read, and is sent to the client along with the record contents. (One of the characteristics of a message digest is the fact that changing a single bit in the message – in this case, the record contents – will cause a change in the message digest.) When the client desires to update or delete the record, the original message digest is sent as one of the input parameters of the request. As part of processing the update or delete request, the server first reads the record with lock, recomputes the message digest and compares the computed digest to the digest sent by the client. If the digests are equal, the record has not changed and the request is completed. If the record has changed, it is unlocked and the client is notified that the record contents have changed.   (Note that the –Cr option must be used to compile the program that computes the message digest.)

Not quite web service

The architecture of Xcentrisity Business Information Server does not restrict its use to SOAP web services. As noted above, BIS can support web applications that use the REST architecture. In addition, BIS may be used to implement other XML- and HTTP-based web technologies. These might include Atom, RSS, Ajax, SVG, and JSON.

AJAX/JSON

As an example of the versatility of the combination of BIS and XML extensions, consider *tutorial3,* which is a BIS service program similar to the *tutorial2* browse method that can produce either an XML response or, if the ACTION=JSON query parameter is included on the URL, the service program uses a different XSLT style sheet to export identical data in JSON (JavaScript Object Notation) format.

When invoked with a URL ending in `tutorial3.srf?STARTSWITH=C` the following XML is returned:

```
<?xml version="1.0" encoding="utf-8"?>
<companies>
	<company>
		<statecode>IA</statecode>
		<areacode>712</areacode>
		<officecode>723</officecode>
		<companyname>C-M-L TELEPHONE COOP. ASSN. OF MERIDEN, 
IOWA</companyname>
		<ratecenter>ARCHER</ratecenter>
		<assignmentcode>AS</assignmentcode>
	</company>
	<company>
		<statecode>TX</statecode>
		<areacode>512</areacode>
		<officecode>316</officecode>
		<companyname>C3 COMMUNICATIONS, INC. - TX</companyname>
		<ratecenter>AUSTIN</ratecenter>
		<assignmentcode>AS</assignmentcode>
	</company>
	<company>
		<statecode>MT</statecode>
		<areacode>406</areacode>
		<officecode>935</officecode>
		<companyname>CABLE &amp; COMMUNICATIONS 
CORPORATION</companyname>
		<ratecenter>BROADUS</ratecenter>
		<assignmentcode>AS</assignmentcode>
	</company>
```
etc

When invoked with a URL ending in tutorial3.srf?STARTSWITH=C&ACTION=JSON the following text is returned:

```
{ "companies" : [

      {
       "statecode" : "IA",
       "areacode" : 712,
       "officecode" : 723,
       "companyname" : "C-M-L TELEPHONE COOP. ASSN. OF MERIDEN, IOWA",
       "ratecenter" : "ARCHER",
       "assignmentcode" : "AS"
      },
      {
       "statecode" : "TX",
       "areacode" : 512,
       "officecode" : 316,
       "companyname" : "C3 COMMUNICATIONS, INC. - TX",
       "ratecenter" : "AUSTIN",
       "assignmentcode" : "AS"
      },
      {
       "statecode" : "MT",
       "areacode" : 406,
       "officecode" : 935,
       "companyname" : "CABLE & COMMUNICATIONS CORPORATION",
       "ratecenter" : "BROADUS",
       "assignmentcode" : "AS"
      },
      etc.
```

Additional usage examples may be found on the Micro Focus Supportline Knowledge Base.