

## Prerequisites

This tutorial has 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*.
- A web services client will be necessary for obtaining web services meta-information. This tutorial uses *soapUI*, which may be found at [www.soapui.org](http://www.soapui.org). Download and install *soapUI*.
- RMNet is a web client that uses the HyperText Transfer Protocol (HTTP). While it is not necessary to be an HTTP expert, this tutorial presumes familiarity with basic HTTP terminology such as the HTTP methods GET and PUT, HTTP headers, and cookies. Several web-based HTTP tutorials are available.
- This tutorial will create XSLT documents for creating SOAP requests and consuming SOAP responses. Familiarity with basic XSLT is presumed. Several web-based XSLT tutorials are available.

## Introduction

This tutorial introduces RMNet, an HTTP web client callable from *extend*. For example, you may use the RMNet Application Programming Interface (API) to emulate a browser to extract information from a web site or interact with a web service using SOAP.

RMNet is used to move data between the web client – your *extend* program – and a web server. The resource on the web server is described by a Universal Resource Locator (URL). The format of the data is dictated by the resource. For example, if you are submitting data to a web form intended to be used by a browser, the data would need to be formatted to comply with the W3C specification for application/x-www-form-urlencoded. When interacting with a SOAP web service, XML documents are exchanged.

This tutorial shows how the temperature conversion sample program (found in the *extend* sample/rmnet directory) was created. Additional information is also provided that may be used when scripting an HTTP interaction with a web server.

## Building a SOAP web service client

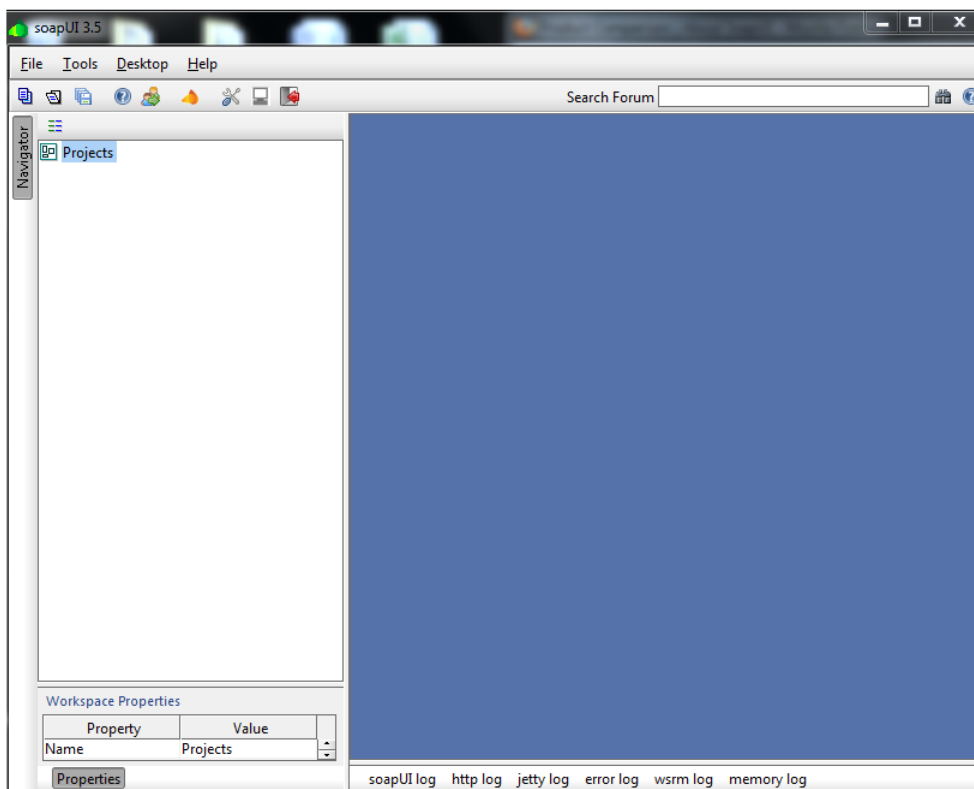
The temperature conversion web service is an example web service provided by the **w3schools.com** tutorial web site. (Note that w3schools.com provides a large offering of tutorials on many web technologies.) A web service provides a Web Services Definition Language document, also called a WSDL, that defines the location of the service, its methods (or functions) and the request and response

documents for those methods. The temperature conversion web service is located at URL <http://www.w3schools.com/webservices/tempconvert.asmx>.

The WSDL for the temperature conversion may be obtained by doing an HTTP GET on the URL <http://www.w3schools.com/webservices/tempconvert.asmx?WSDL>. (This is a fairly conventional mechanism of publishing the WSDL for a web service. Since SOAP requests always use POST to a URL, a GET can be interpreted as a request for the WSDL or, as in this case, a GET of the URL with a query parameter 'WSDL' is interpreted as a request for the WSDL.) Some web services do not make a WSDL available via the web, due to security or other concerns, but rather have some other means to deliver a file containing the WSDL. Note that if you use a browser to GET the WSDL, you may need to look at the page source to see the XML information that makes up the WSDL.

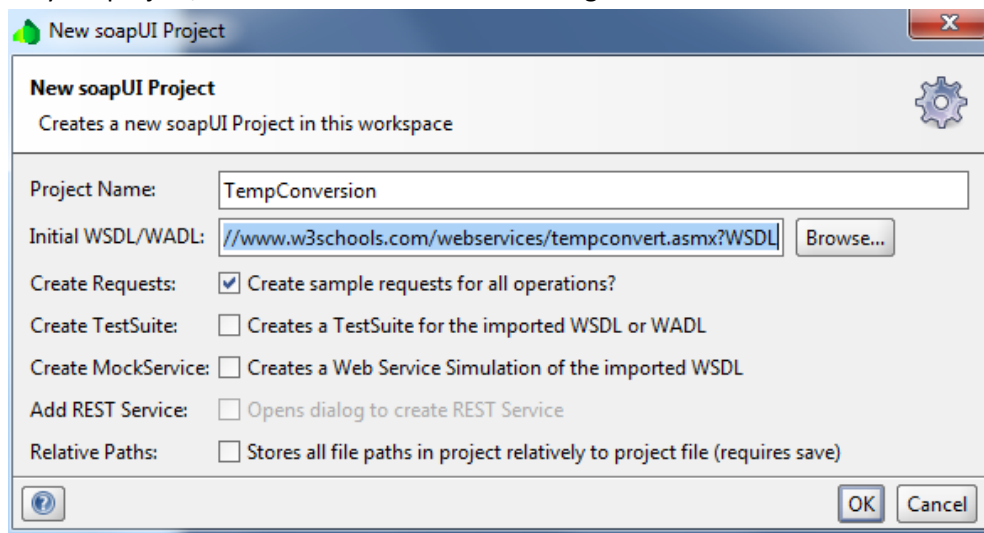
A WSDL is a very complex document and not readily decipherable. We will use *soapUI*, an open source web services testing tool (see Prerequisites), to obtain the WSDL. *soapUI* can create much more usable request and response documents which can in turn be modified to create XSLT style sheets to be used with XML Extensions.

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

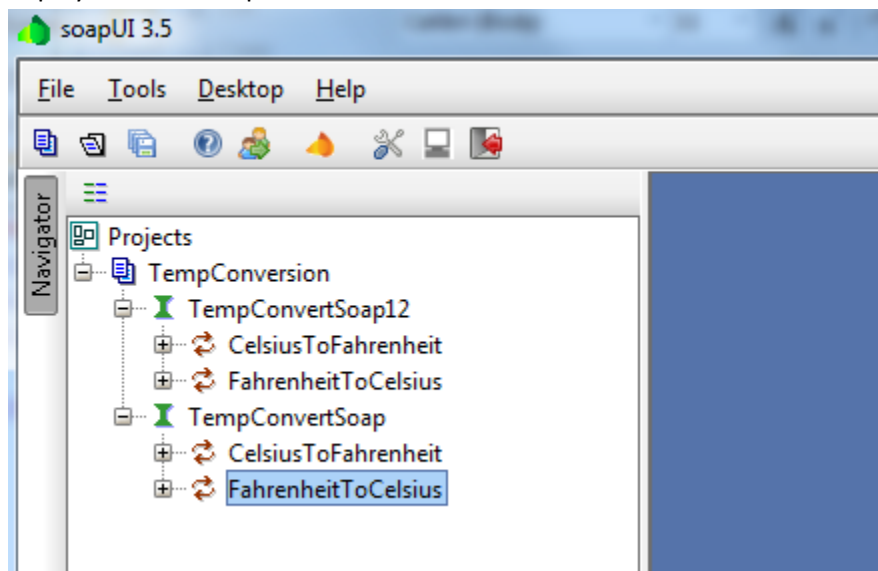


The left pane provides an area to define 'projects' and the right pane contains windows that are associated with a selected project. Follow these steps to create a project to test the first example.

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.

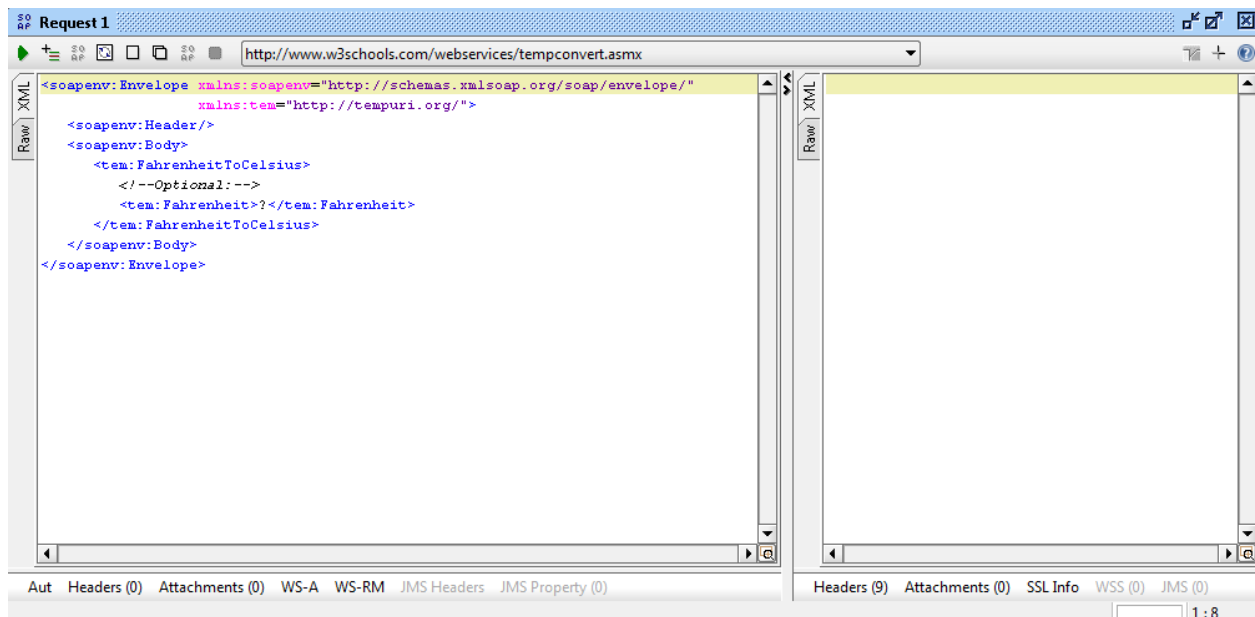


Since the temperature conversion web service provides a WSDL as the result of a GET on the endpoint URL with the 'WSDL' query parameter appended, we simply enter a name for the project and the URL of the web service end point. Make sure the *Create Requests*: option is checked so that *soapUI* creates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

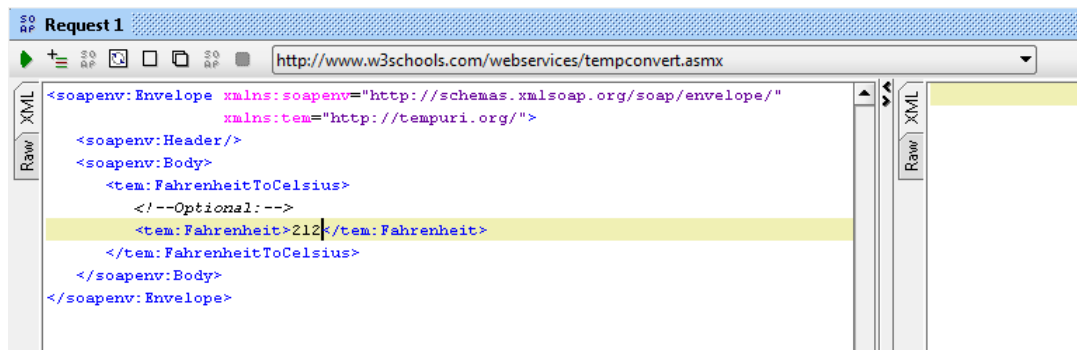


The web service has two 'bindings' named *TempConvertSoap12* and *TempConvertSoap*, each of which has two methods, *CelsiusToFahrenheit* and *FahrenheitToCelsius*. For this tutorial, we will use the *TempConvertSoap* binding. (A binding is a set of operations, or functions, that can be invoked. This web service presents the same set of functions available in each binding, and the bindings differ in the technical details between different versions of SOAP.)

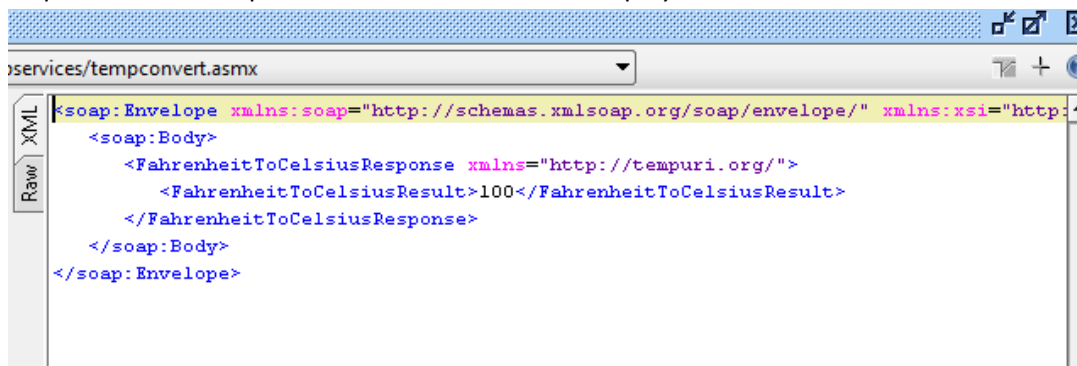
Expand the selection for the *FahrenheitToCelsius* method, exposing a prototype request named *Request 1*. Double click *Request 1* to open the prototype request in the right pane:



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 a numeric value for Fahrenheit which the web service will use to compute the Celsius to return. Let's enter 212 and press the green arrowhead at the upper left of the request window.

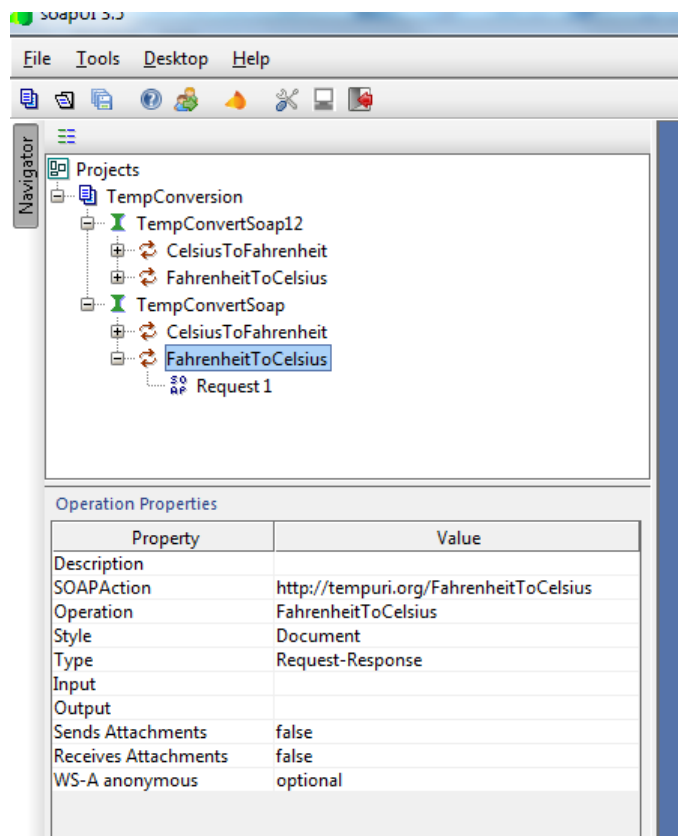


soapUI sends the request to the web service and displays the result:



We now have prototypes for both the request and response and we are ready to create XSLT style sheets for use with XML Extensions; copy and paste the lines of the request and response into two separate text editor sessions, one named TempConvertRequestF2C.xsl and the other named TempConvertResponseF2C.xsl.

Before we leave *soapUI* we must also determine whether a SOAPAction HTTP binding parameter is needed. The SOAPAction parameter is an optional part of a SOAP 1.1 and SOAP 1.2 web service definition. SOAPAction is an HTTP header that assists the web server's determination of the type of the SOAP request without investigating the XML payload. If the SOAPAction parameter is present in the WSDL, the value must be used in a SOAP request. You determine the SOAPAction parameter value by examining the Operation Properties of the method.



We can see that the FahrenheitToCelsius method requires a SOAPAction HTTP header with the value 'http://tempura.org/FahrenheitToCelsius'.

## Converting Request and Response to XSLT

XSLT is used to convert the XML output of an XML OUTPUT FILE/TEXT operation from the XML dictated by the COBOL record layout to the desired XML. The COBOL record layout for the request is as follows:

01 Fahrenheit-To-Celsius.

02 Fahrenheit           pic x(3) value zeros.

When exported, the untransformed XML document has the following structure:

```
<fahrenheit-to-celsius>
  <fahrenheit>0</fahrenheit>
</fahrenheit-to-celsius>
```

However, the desired XML document for the request has the structure:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header/>
  <soapenv:Body>
    <FahrenheitToCelsius xmlns="http://tempuri.org/">
      <Fahrenheit>?</Fahrenheit>
    </FahrenheitToCelsius>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the desired document has been simplified in two ways: (1) the XML comment has been removed, and (2) the XML namespace alias 'tem' has been removed and the namespace declaration has been moved to the <FahrenheitToCelsius> element. This simplified document is logically equivalent to the original copied from *soapUI*.

An XSLT for this transformation can be created by adding a few XSLT processing instructions to the desired XML document. Where values are required from the input document (that is, the untransformed document shown above), the processing instruction <xsl:value-of...> is used. The resulting XSLT style sheet is as follows:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
  <xsl:template match="/">
    <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
      <soapenv:Header/>
      <soapenv:Body>
        <FahrenheitToCelsius xmlns="http://tempuri.org/">
          <Fahrenheit><xsl:value-of select="fahrenheit-to-celsius/fahrenheit"/></Fahrenheit>
        </FahrenheitToCelsius>
      </soapenv:Body>
    </soapenv:Envelope>
  </xsl:template>
</xsl:stylesheet>
```

The information shown in **bold** are the added lines.

Creation of the XSLT to process the response is similar, but targeting the XML structure derived from the COBOL record layout. The COBOL layout to receive the response is:

```
01 Fahrenheit-To-Celsius-Response.  
   02 Fahrenheit-To-Celsius-Result  pic X(20).
```

The XML document derived from this record layout has the following structure:

```
<fahrenheit-to-celsius-response>  
  <fahrenheit-to-celsius-result></fahrenheit-to-celsius-result>  
</fahrenheit-to-celsius-response>
```

The web service SOAP response is (omitting unused namespace alias declarations):

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <FahrenheitToCelsiusResponse xmlns="http://tempuri.org/">  
      <FahrenheitToCelsiusResult>100</FahrenheitToCelsiusResult>  
    </FahrenheitToCelsiusResponse>  
  </soap:Body>  
</soap:Envelope>
```

As in the case of the request, we will add XSLT processing instructions into the desired XML document (that is, the one derived from the COBOL record layout), and use <xsl:value-of...> to fetch values from the input document (the SOAP response).

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
                xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"  
                xmlns:a="http://tempuri.org/">  
  <xsl:output method="html" encoding="UTF-8" indent="yes"/>  
  <xsl:template match="/">  
    <Fahrenheit-To-Celsius-Response>  
      <Fahrenheit-To-Celsius-Result>  
        <xsl:value-of  
select="soap:Envelope/soap:Body/a:FahrenheitToCelsiusResponse/a:FahrenheitToCelsiusResult"/>  
      </Fahrenheit-To-Celsius-Result>  
    </Fahrenheit-To-Celsius-Response>  
  </xsl:template>  
</xsl:stylesheet>
```

This stylesheet is only slightly more complex than the request stylesheet. A SOAP document uses XML namespaces for the SOAP envelope as well as the content of the SOAP Body. Therefore, the required namespace aliases are declared in the <xsl:stylesheet> processing instruction and used in the XPath expression in the <xsl:value-of> instruction. Also note that this stylesheet does not do any error processing as might be needed if the web service returned a SOAP fault.

## Creating the Web Service Client Program

Let us now look at the COBOL program that issues the web request, focusing on the XML manipulations as well as the CALLs that interact with RMNet.

After a Fahrenheit temperature is accepted from the operator, the following statements create the SOAP envelope for the request:

```
XML EXPORT FILE
    Fahrenheit-To-Celsius
    "TempFahrenheitRequest.xml"
    "Fahrenheit-To-Celsius"
    "TempConvertRequestF2C.xsl".
if not XML-OK go to z.

XML EXPORT TEXT
    Fahrenheit-To-Celsius
    request-payload
    request-len
    "Fahrenheit-To-Celsius"
    "TempConvertRequestF2C.xsl".
if not XML-OK go to z.
```

The XML EXPORT FILE is for illustrative purposes only; it produces a copy of the XML that will be created by the following XML EXPORT TEXT. A production program would not include the XML EXPORT FILE. At the end of these statements, request-payload and request-len point at the text string that contains the XML to be sent to the web service as the payload in the HTTP request

```
call "NetInit"
    giving
        response-status.

call "HttpPost"
    using
        Post-Address
        Content-Type
        request-payload
        request-len
        response-payload
        response-len
        Desired-SOAP-Action
    giving
        response-status.
```



NetInit is called to initialize the RMNet interface. Then, HttpPost is called to make a POST request to the URL contained in Post-Address. The Content-Type parameter describes the MIME type and character encoding of the payload; in the case of a SOAP request, the MIME type should always be `text/xml`. If you were creating a POST to a web form, then your payload would not be XML, and the content type would be `application/x-www-form-urlencoded`. The request-payload and request-len parameters are the pointer and length of the request payload (created in the XML EXPORT TEXT). The response - payload and response-len parameters will receive the pointer and length of the response payload received in the HTTP response to the POST. Finally, the last parameter, used to supply extra HTTP headers, (named Desired-SOAP-Action here) is used to add the SOAPAction header name/value pair. (The need for a SOAPAction header was discovered when we used soapUI to look at the WSDL.)

Following the CALL to HttpPost, there follows some error checking:

```
set address of http-response to response-payload.  
  
display "Response: ", response-status.  
  
if not response-status = 0  
    call "NetGetError" using response-payload response-len  
                                giving response-status-2  
    set address of http-response to response-payload  
    display "Error! ", response-status  
    display "Error message: ", http-response(1:response-len)  
    call "NetFree" using response-payload  
    go to z  
end-if.
```

This is 'normal' error processing if the response status is nonzero. Note that the reuse of response payload pointer is a convenience

Finally, the response document, a SOAP envelope containing the response value, needs to be imported.

```

XML FREE TEXT
    request-payload.

if response-payload = NULL
    display "Error:  NULL pointer returned", line 10, blink
    accept a-single-char prompt
    go to z
end-if.

XML PUT TEXT
    response-payload
    response-len
    "TempFahrenheitResponse.xml".
if not XML-OK go to z.

XML IMPORT TEXT
    Fahrenheit-To-Celsius-Response
    response-payload
    response-len
    "Fahrenheit-To-Celsius-Response"
    "TempConvertResponseF2C.xsl".
if not XML-OK go to z.

call "NetFree"
    using
        response-payload.

call "NetCleanup".

```

The request, which is memory allocated by XML Extensions, is returned; it is no longer needed. The XML PUT TEXT is for illustrative purposes, and would not be present in a production environment. The XML IMPORT TEXT imports the required data from the response SOAP envelope, using the XSLT developed above. At this point the response payload is no longer needed and the memory is returned using NetFree. Finally, NetCleanup is called to terminate RMNet processing, allowing it to release any resources that may have been acquired during its processing.

## Additional Topics

### Maintaining State – Cookies

Some web services, and many form-based web sites, rely on the web client to maintain information that, when transmitted to the web server, allows the server to reestablish the context of the request, or to reconnect with a stateful process awaiting the request. The mechanism most often used for this purpose is cookies.

Cookies come in two forms: persistent cookies that are maintained between sessions in a disk file on the client machine, and session cookies that persist only during the lifetime of a web client instance.

Cookies work behind the scenes in browsers, but you must take a special step to enable cookies in your web client, by calling HttpSetCookieFile at the beginning of your client session. HttpSetCookieFile may

be called multiple times to inform RMNet about several files used to maintain persistent cookies. If you wish to use only session cookies, call `HttpSetCookieFile` with the file name parameter equal to spaces, or by simply omitting the cookie file parameter. In RMNet terms, a session exists between the call to `NetInit` and the call to `NetCleanup`

## Secure Sockets (SSL)

Some web services will require a secure HTTP transfer. The most common protocol to implement a secure HTTP transfer over a network is known as HTTPS (or HTTP over SSL). RMNet is compatible with SSL (or TLS as the latest version of the standard is called). SSL encrypts all the data that is sent and received over a network. In an HTTPS environment, certificates can be used to validate the identity of the server and/or client. RMNet supports the use of both server-side and client-side SSL certificates. This tutorial, however, does not employ SSL/HTTPS and does not teach this subject.

## Processing Response Headers

Some web sites have used an *ad hoc* approach to providing information over the web. Although HTTP is normally treated as a transport layer, existing only to move the content or payload, some web developers have chosen to return information in the HTTP headers of the response. Note that this discussion is not about cookies, which are transmitted by HTTP headers, because cookies are normally used to associate a series of messages, which is a reasonable function for the transport layer. If you find yourself needing to obtain information from the HTTP headers of the response, you must call `HttpSetResponseHeader` prior to the call to `HttpPost` or `HttpGet`.

## Final Notes

RMNet is built upon the open source library known as libcurl which is part of the cURL (<http://curl.haxx.se/>) project. Useful information can be found in the curl documentation set, especially if you seek to use the more advanced routines available in RMNet.

For additional information about SSL, you may begin by searching the OpenSSL Project (<http://www.openssl.org/>); OpenSSL is used by libcurl, and therefore by RMNet, for SSL support.