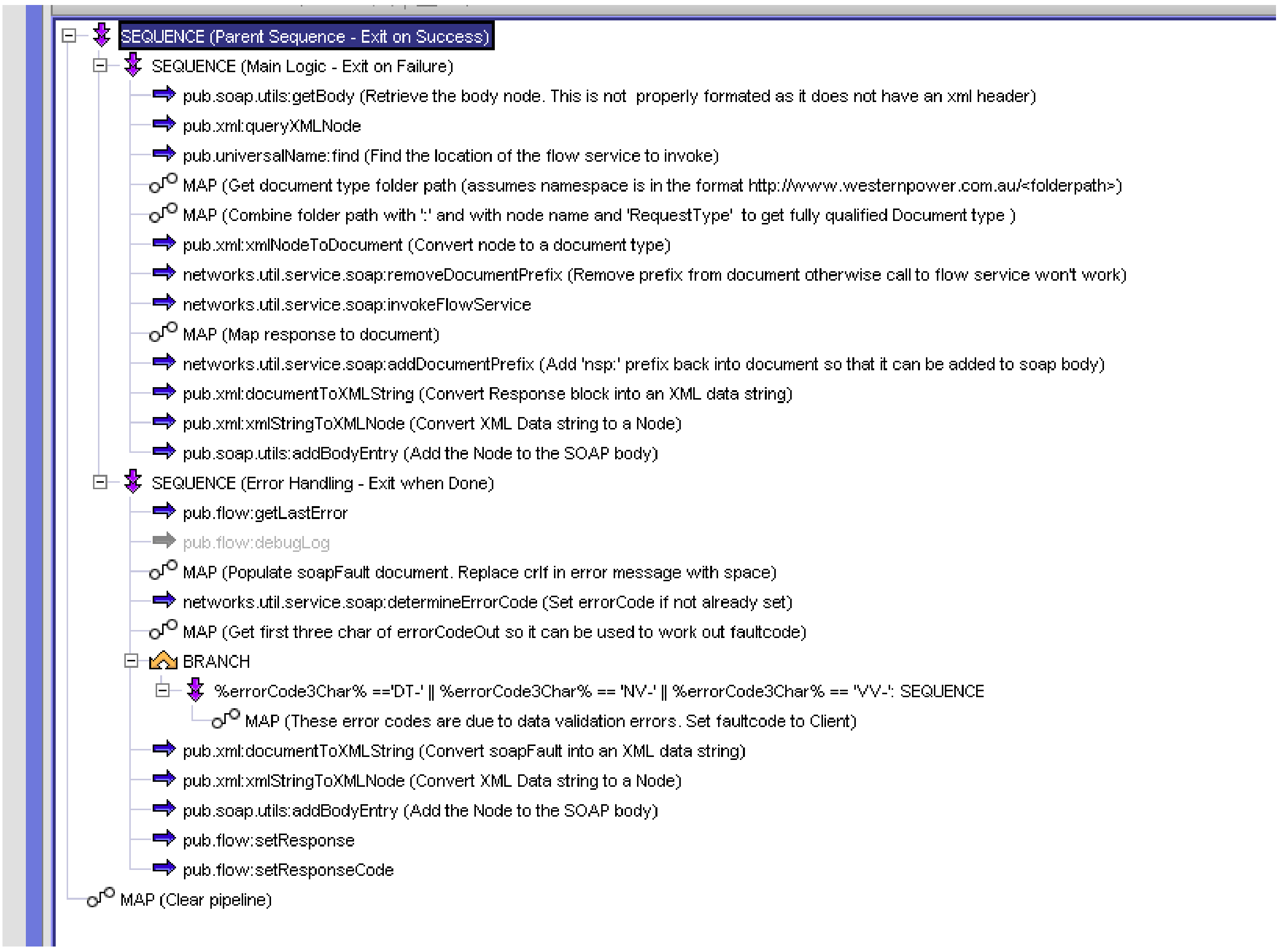
How to build a basic Custom Document Literal SOAP Processor

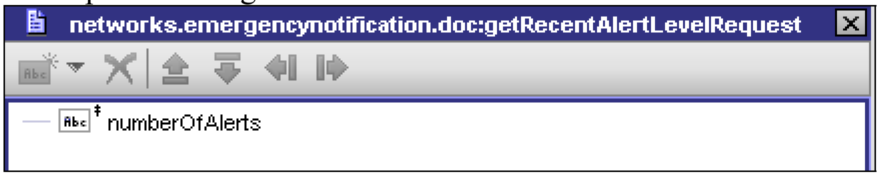
The following is the source code for a Custom Document Literal SOAP Processor that is used by Western Power – an energy utility based in Perth, Western Australia. This processor is relatively simplistic as it does not handle headers nor access a service repository. The assistance of Mark Carlson from the webMethods Users Group (www.wmusers.com) is gratefully acknowledged in helping to build this processor.



```
SEQUENCE (Parent Sequence - Exit on Success)
    SEQUENCE (Main Logic - Exit on Failure)
        pub.soap.utils:getBody (Retrieve the body node. This is not  properly formated as it does not have an xml header)
        pub.xml:queryXMLNode
        pub.universalName:find (Find the location of the flow service to invoke)
        MAP (Get document type folder path (assumes namespace is in the format http://www.westernpower.com.au/<folderpath>)
        MAP (Combine folder path with ':' and with node name and 'RequestType'  to get fully qualified Document type )
        pub.xml:xmlNodeToDocument (Convert node to a document type)
        networks.util.service.soap:removeDocumentPrefix (Remove prefix from document otherwise call to flow service won't work)
        networks.util.service.soap:invokeFlowService
        MAP (Map response to document)
        networks.util.service.soap:addDocumentPrefix (Add 'nsp:' prefix back into document so that it can be added to soap body)
        pub.xml:documentToXMLString (Convert Response block into an XML data string)
        pub.xml:xmlStringToXMLNode (Convert XML Data string to a Node)
        pub.soap.utils:addBodyEntry (Add the Node to the SOAP body)
    SEQUENCE (Error Handling - Exit when Done)
        pub.flow:getLastError
        pub.flow:debugLog
        MAP (Populate soapFault document. Replace crlf in error message with space)
        networks.util.service.soap:determineErrorCode (Set errorCode if not already set)
        MAP (Get first three char of errorCodeOut so it can be used to work out faultcode)
        BRANCH
            %errorCode3Char% =='DT-' || %errorCode3Char% == 'NV-' || %errorCode3Char% == 'VV-': SEQUENCE
                MAP (These error codes are due to data validation errors. Set faultcode to Client)
        pub.xml:documentToXMLString (Convert soapFault into an XML data string)
        pub.xml:xmlStringToXMLNode (Convert XML Data string to a Node)
        pub.soap.utils:addBodyEntry (Add the Node to the SOAP body)
        pub.flow:setResponse
        pub.flow:setResponseCode
    MAP (Clear pipeline)
```

A number of conventions (shown in red text below) are required to define the inputs and outputs for the called service so that the relevant WSDL can be generated correctly and the subsequent soap requests processed successfully. A service named **getRecentAlertLevel** is used to illustrate the approach used. This service retrieves electricity emergency notification alert details. The number of alerts to retrieve is passed as a parameter (numberOfAlerts) which must be an integer greater than or equal to 1 and less than or equal to 100. The alert details retrieved are passes back in a canonical array. Also passed back is a Response canonical to indicate the success of the process. If a serious error occurs in the service or one of the services it calls, an exception is thrown by the relevant service (via the **exit $flow failure** statement) with a suitable error message. This is trapped by the soap processor and converted into a SOAP Fault message that is passed back to the calling client.

The inputs for the getRecentAlertLevel service are defined in a document as illustrated below:

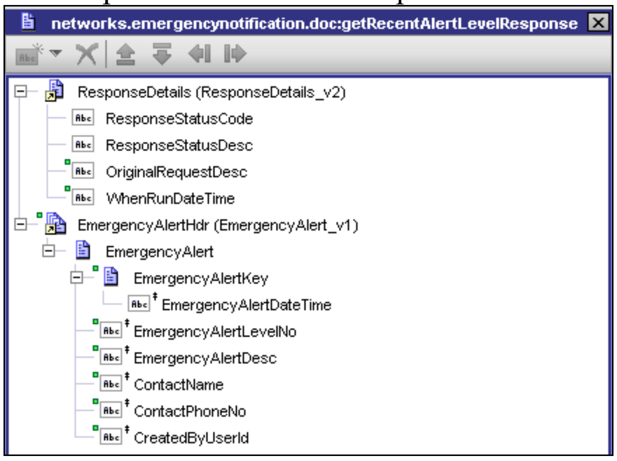


networks.emergencynotification.doc:getRecentAlertLevelRequest

numberOfAlerts

The name of this document must be the same as the name as the public service with the word Request appended to it, ie getRecentAlertLevel**Request**. The numberOfAlerts parameter is created as a string, but has its content type property set to Integer with minInclusive of 1 and maxInclusive of 100.

The outputs are defined in a separate document as illustrated below:



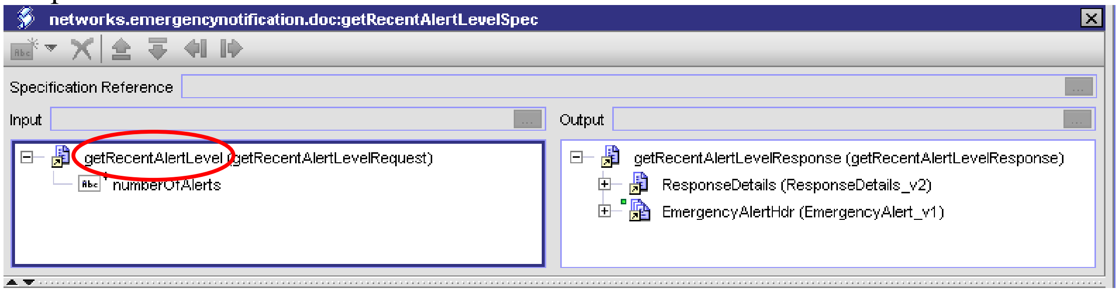
networks.emergencynotification.doc:getRecentAlertLevelResponse

```
ResponseDetails (ResponseDetails_v2)
    ResponseStatusCode
    ResponseStatusDesc
    OriginalRequestDesc
    WhenRunDateTime
EmergencyAlertHdr (EmergencyAlert_v1)
    EmergencyAlert
        EmergencyAlertKey
            EmergencyAlertDateTime
        EmergencyAlertLevelNo
        EmergencyAlertDesc
        ContactName
        ContactPhoneNo
        CreatedByUserId
```

The name of this document must be the same as the name as the public service with the word Response appended to it, ie getRecentAlertLevel**Response**. This document is made up of document references to two canonicals. In the first case a single document reference has been made to the ResponseDetails_v2 canonical to store the return status code details. In the second case a document reference list (ie an array of documents) is made to the EmergencyAlert_v1 canonical so that multiple alert level details can be passed back. Note: Because document references are used, you cannot modify the content type property of individual fields – this would have to be done in the canonical document itself.

Western Power has a policy of setting up public services that pretty much do very little other than to call a relevant worker service to do all the processing. The reason for this is that we often find the need to create public services that are a variation on a theme. In this case, the public service getRecentAlertLevel calls a worker service named getRecentAlertLevelProcess passing in the numberOfAlerts parameter to get back the desired number of alerts. We also have another public service named getCurrentAlertLevel that has no parameters. It calls the same worker service but sets the numberOfAlerts to 1 in order to only get back the latest alert level.

This means that at least one public service and the worker service share the same inputs and outputs. We therefore create a **specification document** that contains references to the Request and Response documents as illustrated below:



networks.emergencynotification.doc:getRecentAlertLevelSpec

Specification Reference

Input

```
getRecentAlertLevel (getRecentAlertLevelRequest)
    numberOfAlerts
```

Output

```
getRecentAlertLevelResponse (getRecentAlertLevelResponse)
    ResponseDetails (ResponseDetails_v2)
    EmergencyAlertHdr (EmergencyAlert_v1)
```

The name of this document must be the same as the name as the public service with the word Spec appended to it, ie getRecentAlertLevel**Spec**. **The name given to the document <u>reference</u> to the Request document must be exactly the same as the name of the public service.** In this case the reference name is getRecentAlertLevel. **The name given to the document <u>reference</u> to the Response document is the same as the name of the Response document.** In this case the reference name is getRecentAlertLevelResponse.

How to build a basic Custom Document Literal SOAP Processor

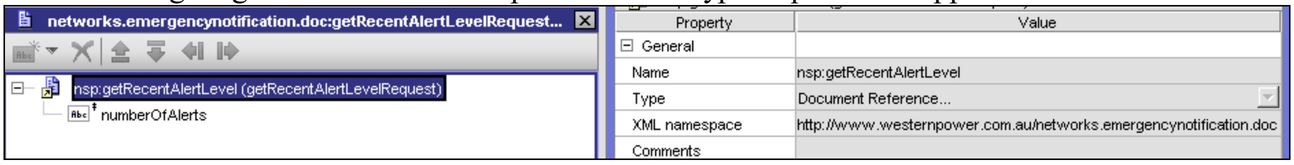
There are two reasons for creating a specification document:

1. The comments tab enables the inputs/outputs to be documented in one place.
2. This specification is easily included for both the public service and the worker service, hence you only need to create it once and use it twice.

Note: this is a Western Power coding standard … you don't have to do this, but we think it is worthwhile.

In order for a public flow service to be invoked using the document literal (SOAP-MSG) format the request and response must have a namespace. For RPC Encoded (SOAP-RPC) format a namespace is not required. The above request and response documents do not have a namespace so are fine for generating an RPC Encoded WSDL. However, in order to be able generate a doc literal WSDL, we need to set up two more documents which include namespace details.

The following diagram illustrates the input document type required to support doc literal.



networks.emergencynotification.doc:getRecentAlertLevelRequest...

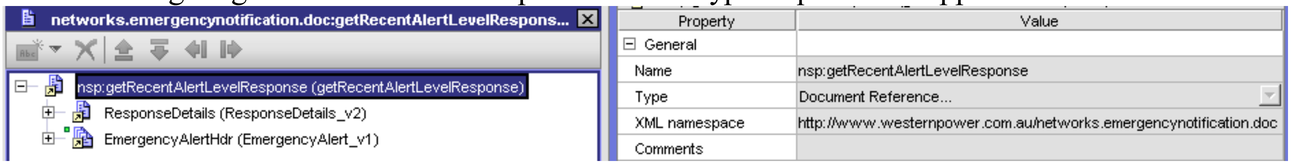
nsp:getRecentAlertLevel (getRecentAlertLevelRequest)
  numberOfAlerts

Property	Value
General	
Name	nsp:getRecentAlertLevel
Type	Document Reference...
XML namespace	http://www.westernpower.com.au/networks.emergencynotification.doc
Comments	

The name of this document must be the same as the name as the public service with the words RequestType appended to it, ie getRecentAlertLevel**RequestType**. All it contains is a document reference to the Request document. **The name given to the document reference to the request document must be exactly the same as the name of the public service and must have a prefix of nsp: prepended to it**, ie nsp:getRecentAlertLevel. **Do not use any other prefix name as the custom soap processor relies on this prefix being used.**

The XML namespace property of the document reference must be in the format:
<business internet site home address>/<folder path where request document is stored>
In this case, the business is western power and the documents are stored in networks.emergencynotification.doc therefore the namespace is:
http://www.westernpower.com.au/networks.emergencynotification.doc

The following diagram illustrates the ouput document type required to support doc literal.



networks.emergencynotification.doc:getRecentAlertLevelRespons...

nsp:getRecentAlertLevelResponse (getRecentAlertLevelResponse)
  ResponseDetails (ResponseDetails_v2)
  EmergencyAlertHdr (EmergencyAlert_v1)

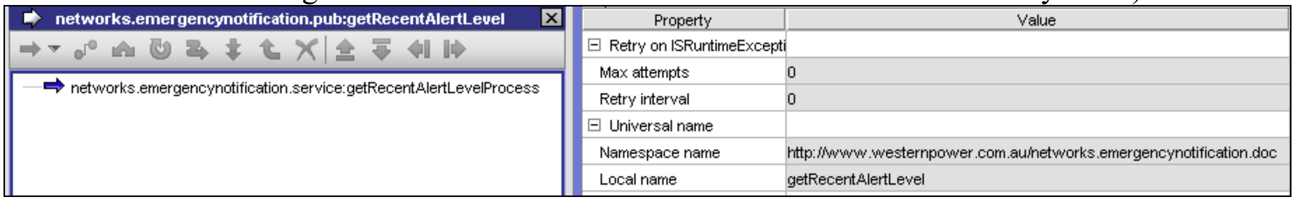
Property	Value
General	
Name	nsp:getRecentAlertLevelResponse
Type	Document Reference...
XML namespace	http://www.westernpower.com.au/networks.emergencynotification.doc
Comments	

The name of this document must be the same as the name as the public service with the words ResponseType appended to it, ie getRecentAlertLevel**ResponseType**. All it contains is a document reference to the Response document. **The name given to the document reference is the same as the Response document name with a prefix of nsp: prepended to it**, ie nsp:getRecentAlertLevelResponse. **The XML namespace property of the document reference must be the same as for the RequestType document above.**

When generating a WSDL for document literal (SOAP-MSG), you must provide the names of the documents that specify the input and outputs. In this case the documents to provide are the RequestType and ResponseType documents as these contain the XML namespace details. If you try to use the Request and Response documents to generate the WSDL, webMethods will display a message indicating that the top level fields must have XML namespace values.

How to build a basic Custom Document Literal SOAP Processor

The following diagram illustrates the getRecentAlertLevel public service (note that all it does is to call the worker service getRecentAlertLevelProcess and that it does **not** have a try/catch):



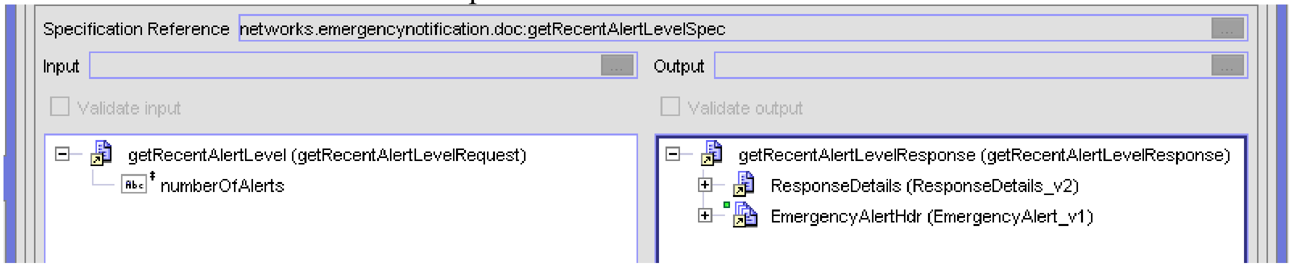
networks.emergencynotification.pub:getRecentAlertLevel

networks.emergencynotification.service:getRecentAlertLevelProcess

Property	Value
⊟ Retry on ISRuntimeExcepti	
Max attempts	0
Retry interval	0
⊟ Universal name	
Namespace name	http://www.westernpower.com.au/networks.emergencynotification.doc
Local name	getRecentAlertLevel

The universal name must have a namespace name that exactly matches the namespace used in the XML namespace property of the RequestType and ResponseType document. The Local name must be exactly the same as the public service name. webMethods stores internally a table that relates the Namespace name and Local name to the fully qualified name of the service.

The following diagram illustrates the inputs/outputs for the getRecentAletLevel public service. In this case a reference is made to the specification document.



Specification Reference: networks.emergencynotification.doc:getRecentAlertLevelSpec

Input | Output

☐ Validate input | ☐ Validate output

```
getRecentAlertLevel (getRecentAlertLevelRequest)
    numberOfAlerts
```

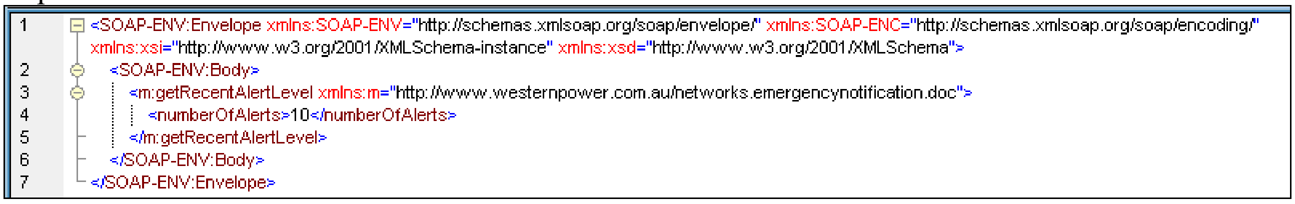
```
getRecentAlertLevelResponse (getRecentAlertLevelResponse)
    ResponseDetails (ResponseDetails_v2)
    EmergencyAlertHdr (EmergencyAlert_v1)
```

If you choose **not** to set up a specification document then set up document references to the Request and Response documents. **The name given to the document <u>reference</u> to the Request document must be exactly the same as the name of the public service** (ie getRecentAlertLevel).

<u>Why is all this necessary?</u>
When a document literal WSDL is generated, it contains far less information about a service than does RPC. This can be a good thing (less compatibility issues) but can also be a bad thing (don't know as much about the details of the service, so some extra processing is required). When a consumer client uses the WSDL to format a doc literal soap message and sends it to webMethods the custom soap processor performs the following tasks:

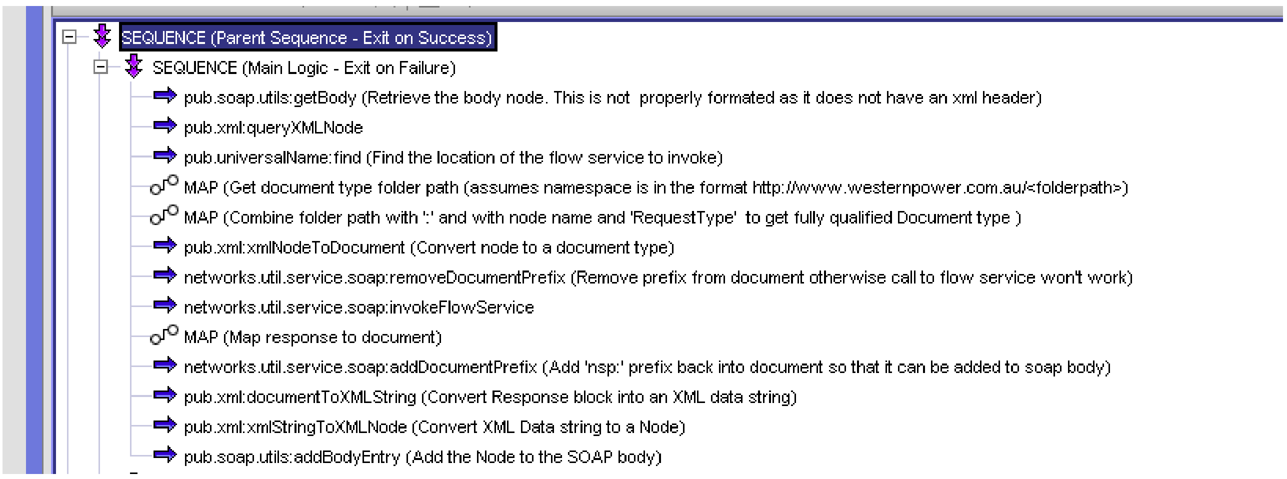
1. Find out the fully qualified name of the service to invoke.
2. Find out the fully qualified name of the Request document
3. Invoke the service using the Request document
4. Pass the results back to the caller in a soap message.

The following is a sample of a doc literal soap request for the getRecentAlertLevel service which requests the latest 10 alerts.



```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP-ENV:Body>
    <m:getRecentAlertLevel xmlns:m="http://www.westernpower.com.au/networks.emergencynotification.doc">
      <numberOfAlerts>10</numberOfAlerts>
    </m:getRecentAlertLevel>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

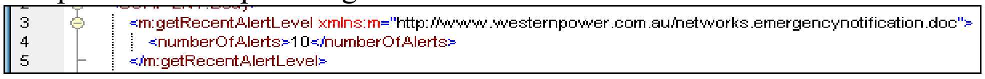
Here again is the source code for the main logic sequence of the custom soap processor.



```
SEQUENCE (Parent Sequence - Exit on Success)
  SEQUENCE (Main Logic - Exit on Failure)
    pub.soap.utils:getBody (Retrieve the body node. This is not  properly formated as it does not have an xml header)
    pub.xml:queryXMLNode
    pub.universalName:find (Find the location of the flow service to invoke)
    MAP (Get document type folder path (assumes namespace is in the format http://www.westernpower.com.au/<folderpath>)
    MAP (Combine folder path with ':' and with node name and 'RequestType'  to get fully qualified Document type )
    pub.xml:xmlNodeToDocument (Convert node to a document type)
    networks.util.service.soap:removeDocumentPrefix (Remove prefix from document otherwise call to flow service won't work)
    networks.util.service.soap:invokeFlowService
    MAP (Map response to document)
    networks.util.service.soap:addDocumentPrefix (Add 'nsp:' prefix back into document so that it can be added to soap body)
    pub.xml:documentToXMLString (Convert Response block into an XML data string)
    pub.xml:xmlStringToXMLNode (Convert XML Data string to a Node)
    pub.soap.utils:addBodyEntry (Add the Node to the SOAP body)
```

Line 1: pub.soap.utils:getBody

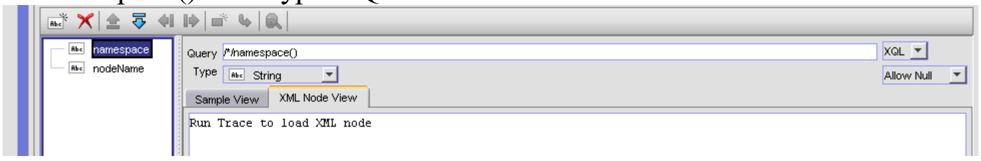
Get the body of the soap message as an XML node. In this example, the body is essentially this part of the soap message:



```
<m:getRecentAlertLevel xmlns:m="http://www.westernpower.com.au/networks.emergencynotification.doc">
  <numberOfAlerts>10</numberOfAlerts>
</m:getRecentAlertLevel>
```

Line 2: pub.xml:queryXMLNode

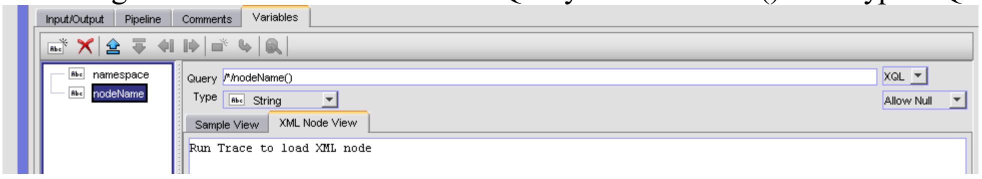
This function enables you to interrogate an XML node without having to know anything about the nodes structure or namespace. For string variable namespace execute Query /*/namespace() with type XQL



```
namespace
nodeName
Query /*/namespace()          XQL
Type  String                  Allow Null
Sample View | XML Node View
Run Trace to load XML node
```

This retrieves the namespace of the root node in the soap body. In this case: http://www.westernpower.com.au/networks.emergencynotification.doc

For string variable nodeName execute Query /*/nodeName() with type XQL



```
Input/Output | Pipeline | Comments | Variables
namespace
nodeName
Query /*/nodeName()           XQL
Type  String                  Allow Null
Sample View | XML Node View
Run Trace to load XML node
```

This retrieves the name of the root node in the soap body. In this case: getRecentAlertLevel

Line 3: pub.universalName:find

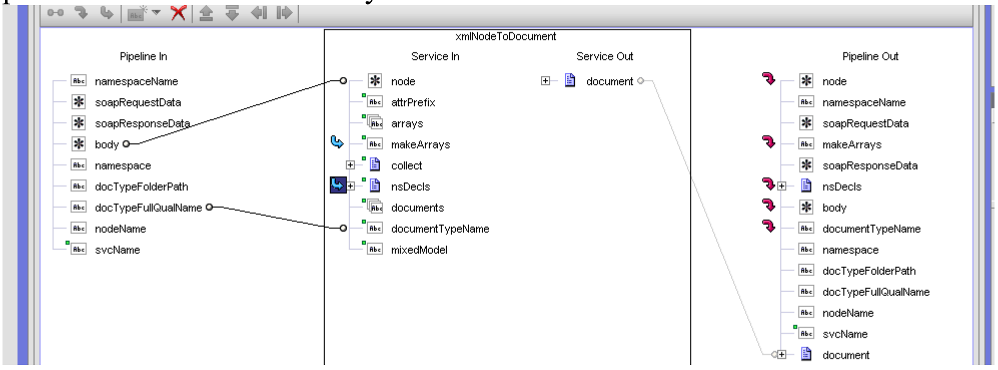
Use the namespace and nodeName pipeline variables from line 2 to find the fully qualified name of the flow service to invoke. This assumes the flow service in question has been set up with a universal name (ie namespace and local name) that matches this data.

Lines 4 & 5: Map statements

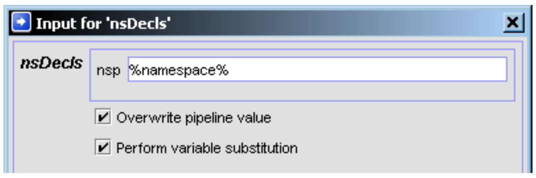
Take the last part of the namespace, add a colon (' :') , and the node name and the text 'RequestType' to come up with the fully qualified name of the request document type (ie networks.emergencynotification.doc:getRecentAlertLevelRequestType) and store this in a pipeline variable (docTypeFullQualName)

Line 6: pub.xml:xmlNodeToDocument

This function converts an XML node to a document type. The first trick is to know what type of document to map it to. By following the above conventions, we know exactly what that document type is. Map the docTypeFullQualName to the documentTypeName parameter and set makeArrays to false.



The second trick is to convert the prefix in the request soap message root node (which could be anything – in this case it is m:) into something we know (ie nsp:). To do this set the nsDecls as follows:


Input for 'nsDecls'

nsDecls	nsp %namespace%

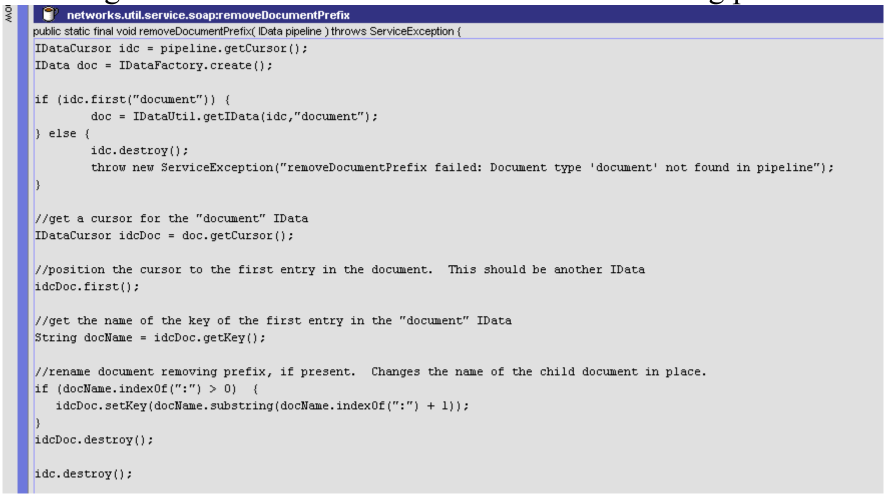
☑ Overwrite pipeline value

☑ Perform variable substitution

The result is a document that exactly matches the structure of the RequestType document, even down to the prefix of nsp:

Line 7: removeDocumentPrefix

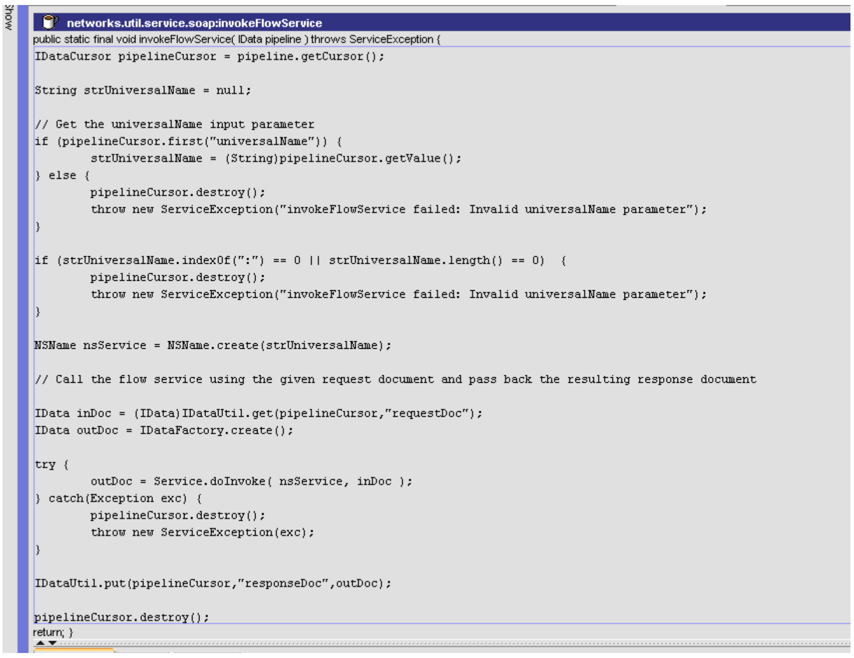
However, the Request document of the target service does not have a prefix of nsp: so we have to get rid of this from the document. The following piece of Java code does this.


networks.util.service.soap:removeDocumentPrefix

```
public static final void removeDocumentPrefix( IData pipeline ) throws ServiceException {
IDataCursor idc = pipeline.getCursor();
IData doc = IDataFactory.create();

if (idc.first("document")) {
        doc = IDataUtil.getIData(idc,"document");
} else {
        idc.destroy();
        throw new ServiceException("removeDocumentPrefix failed: Document type 'document' not found in pipeline");
}

//get a cursor for the "document" IData
IDataCursor idcDoc = doc.getCursor();

//position the cursor to the first entry in the document.  This should be another IData
idcDoc.first();

//get the name of the key of the first entry in the "document" IData
String docName = idcDoc.getKey();

//rename document removing prefix, if present.  Changes the name of the child document in place.
if (docName.indexOf(":") > 0)  {
   idcDoc.setKey(docName.substring(docName.indexOf(":") + 1));
}
idcDoc.destroy();

idc.destroy();
```

Line 8: invokeFlowService

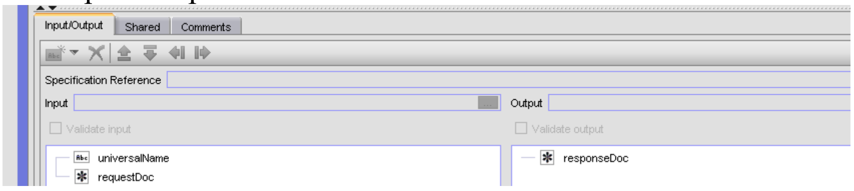
So we now have the fully qualified name of the service to invoke and the input parameters in a document that exactly matches the Request document of the service. All we have to do is invoke the service and pass in the document. The following java code does this. Note that it traps any exceptions thrown by the called service and re-throws a ServiceException.



networks.util.service.soap:invokeFlowService

```
public static final void invokeFlowService( IData pipeline ) throws ServiceException {
IDataCursor pipelineCursor = pipeline.getCursor();

String strUniversalName = null;

// Get the universalName input parameter
if (pipelineCursor.first("universalName")) {
        strUniversalName = (String)pipelineCursor.getValue();
} else {
        pipelineCursor.destroy();
        throw new ServiceException("invokeFlowService failed: Invalid universalName parameter");
}

if (strUniversalName.indexOf(":") == 0 || strUniversalName.length() == 0)  {
        pipelineCursor.destroy();
        throw new ServiceException("invokeFlowService failed: Invalid universalName parameter");
}

NSName nsService = NSName.create(strUniversalName);

// Call the flow service using the given request document and pass back the resulting response document

IData inDoc = (IData)IDataUtil.get(pipelineCursor,"requestDoc");
IData outDoc = IDataFactory.create();

try {
        outDoc = Service.doInvoke( nsService, inDoc );
} catch(Exception exc) {
        pipelineCursor.destroy();
        throw new ServiceException(exc);
}

IDataUtil.put(pipelineCursor,"responseDoc",outDoc);

pipelineCursor.destroy();
return; }
```

The inputs/outputs for this service are:



Specification Reference

Input

Validate input

universalName
requestDoc

Output

Validate output

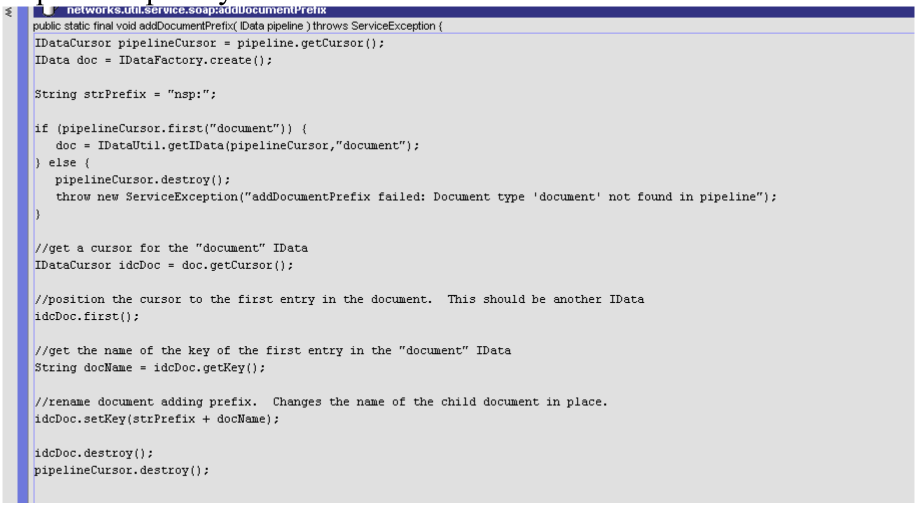
responseDoc

Line 9: Map Response Document

This line maps the responseDoc object to a document type called document. This document has the Response structure underneath it (ie in this case getRecentAlertLevelResponse).

Line 10: addDocumentPrefix

The root node of soap body must have a prefix and a namespace defined. Therefore, before we can convert the document into a soap body, we have to add the prefix back in to the document. The prefix can be anything, but we make it 'nsp:' to be consistent with the request soap body code.

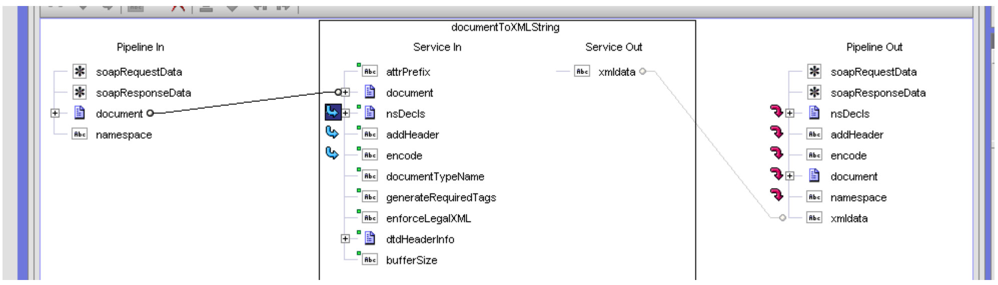


networks.util.service.soap:addDocumentPrefix

```
public static final void addDocumentPrefix( IData pipeline ) throws ServiceException {
IDataCursor pipelineCursor = pipeline.getCursor();
IData doc = IDataFactory.create();

String strPrefix = "nsp:";

if (pipelineCursor.first("document")) {
   doc = IDataUtil.getIData(pipelineCursor,"document");
} else {
   pipelineCursor.destroy();
   throw new ServiceException("addDocumentPrefix failed: Document type 'document' not found in pipeline");
}

//get a cursor for the "document" IData
IDataCursor idcDoc = doc.getCursor();

//position the cursor to the first entry in the document.  This should be another IData
idcDoc.first();

//get the name of the key of the first entry in the "document" IData
String docName = idcDoc.getKey();

//rename document adding prefix.  Changes the name of the child document in place.
idcDoc.setKey(strPrefix + docName);

idcDoc.destroy();
pipelineCursor.destroy();
```

Line 11: pub.xml:documentToXMLString

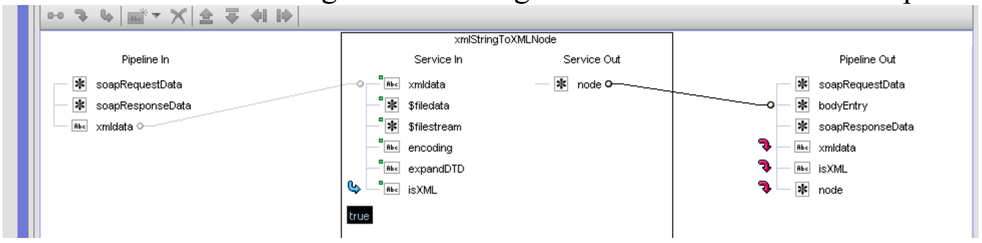
This function converts the document into an XML string. The addHeader parameter is false. The encode parameter is true (to convert any nasties such as '>' to the encoded format so as not to mess up the XML). The nsDecls parameter is the same as in Line 6.



The result is xmldata which is a string in the right format for adding into the body of the response soap message.

Line 12: pub.xml:xmlStringToXMLNode

This function converts a given xml string to an xml node. The isXML parameter is true.

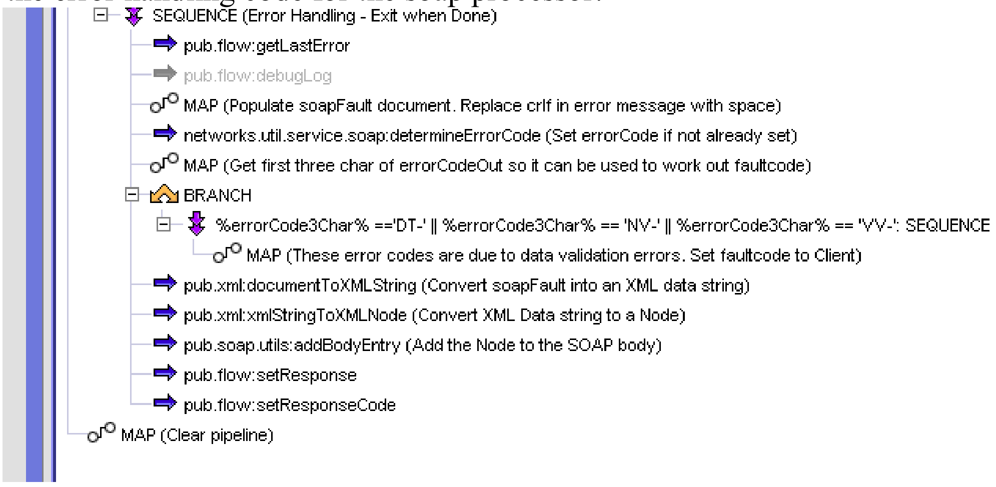


Line 13: pub.soap.utils:addBodyEntry

This function takes the xml node (bodyEntry) that contains the resonse details and adds it into the soapResponseData object. This is then passed back to the calling client.

Error Handling

Here is the error handling code for the soap processor:



```
SEQUENCE (Error Handling - Exit when Done)
    pub.flow:getLastError
    pub.flow:debugLog
    MAP (Populate soapFault document. Replace crlf in error message with space)
    networks.util.service.soap:determineErrorCode (Set errorCode if not already set)
    MAP (Get first three char of errorCodeOut so it can be used to work out faultcode)
    BRANCH
        %errorCode3Char% =='DT-' || %errorCode3Char% == 'NV-' || %errorCode3Char% == 'VV-': SEQUENCE
            MAP (These error codes are due to data validation errors. Set faultcode to Client)
    pub.xml:documentToXMLString (Convert soapFault into an XML data string)
    pub.xml:xmlStringToXMLNode (Convert XML Data string to a Node)
    pub.soap.utils:addBodyEntry (Add the Node to the SOAP body)
    pub.flow:setResponse
    pub.flow:setResponseCode
MAP (Clear pipeline)
```

Error Line 1: pub.flow:getLastError

This function gets the details of the last error thrown.

Error Line 2: Map soapFault document

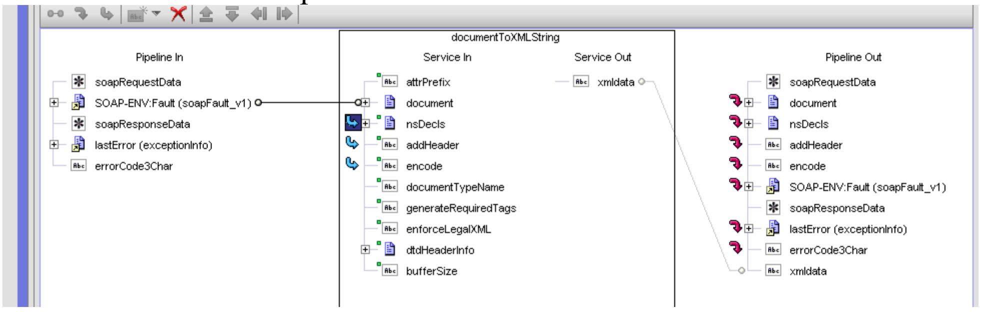
Instantiate a soapFault document type canonical and populate it with details from the getLastError call. In the process, remove carriage return linefeed characters from the error message. There is a standard format for a soap fault (ie faultcode, faultstring, faultactor and a details section that can be customised). Western Power have set up a standard structure for the details section which includes an errorCode, errorMessage, errorDateTime and call stack amongst other things.

Error Lines 3-5: Set up soapFault details

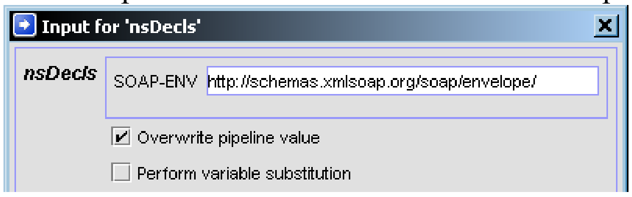
Code specific to Western Power to populate the details section of the soapFault document.

Error Line 6: pub.xml:documentToXMLString

This function converts the soapFault document to an XML string. The addHeader parameter is false and the encode parameter is true.



Note the prefix is SOAP-ENV: so the nsDecls parameter is:



Input for 'nsDecls'

nsDecls

SOAP-ENV http://schemas.xmlsoap.org/soap/envelope/

[✓] Overwrite pipeline value

[ ] Perform variable substitution

Error Line 7: pub.xml:xmlStringToXMLNode

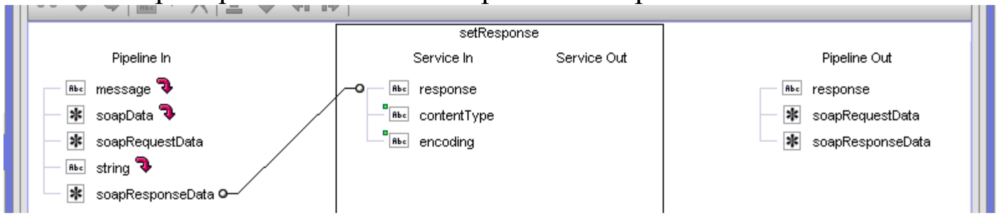
This function converts a given xml string to an xml node. The isXML parameter is set to true.

Error Line 8: pub.soap.utils:addBodyEntry

This function takes the xml node (bodyEntry) that contains the soapFault details and adds it into the soapResponseData object.

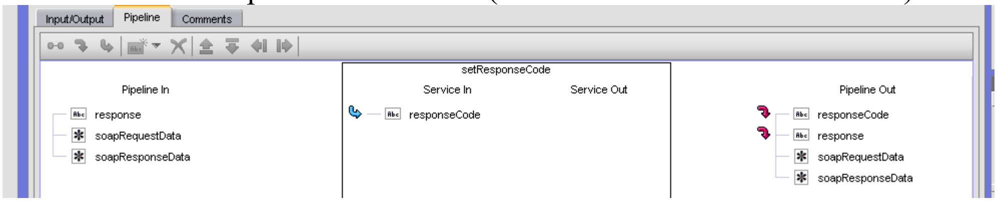
Error Line 9: pub.flow:setResponse

Sets the soapResponseData as the response to be passed back.



Error Line 10: pub.flow:setResponseCode

Sets the HTML response code to 500 (standard HTML error condition).



Security

It is recommended that you apply an ACL rule to the custom soap processor. This allows you to control access to the processor but also enables a challenge – response model, that is, if an external client calls the soap processor without any basic authentication details (user id/password) then an

HTML 401 will be passed back. This indicates that the client should call again but this time pass in the relevant authentication details. Clients built using VB .net (an also XMLSpy) expect to work this way.

Gotchas
A known issue is that when a **doc literal** WSDL for a web service which has **no input parameters** is imported into XML Spy a 'model is non-deterministic' error is reported. The solution is to manually remove the following entries from the WSDL.

```
<xsd:any processContents="lax"/>
```

About the Author
Craig Harper is the Applications Architect at Western Power.
Email: craig.harper@westernpower.com.au

This document is intended as a guide only. The code given is not 'industrial strength' and is unlikely to work in all scenarios. However, it should help most people climb up the rather steep learning curve involved in building a soap processor.

If you find any errors/improvements/changes can you please pass these back to the author.

July 2006.