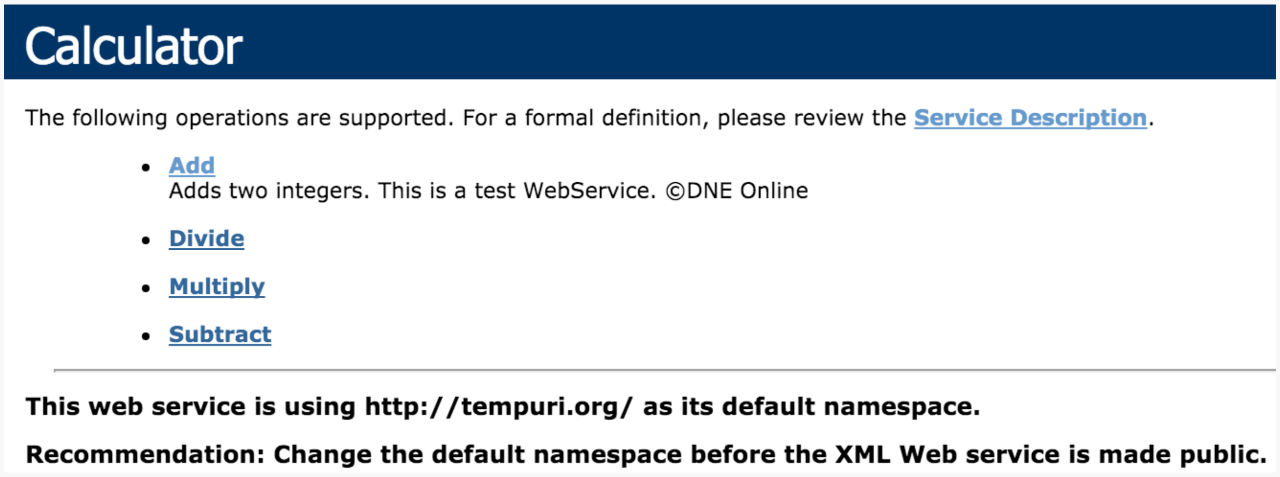
Creating a SOAP Service using an Ultra Pipeline

OVERVIEW

Recently, I implemented a SOAP service within SnapLogic which works like this:
`http://www.dneonline.com/calculator.asmx`



Calculator

The following operations are supported. For a formal definition, please review the Service Description.

- Add
  Adds two integers. This is a test WebService. ©DNE Online
- Divide
- Multiply
- Subtract

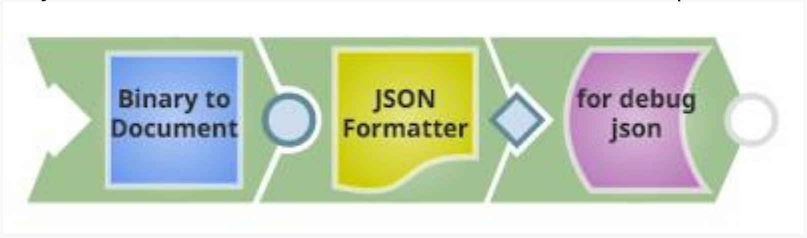
This web service is using http://tempuri.org/ as its default namespace.

Recommendation: Change the default namespace before the XML Web service is made public.

It's a SOAP Service which will Add, Divide, Multiply and Subtract any two numbers.

INITIAL APPROACH - CAPTURE A SOAP REQUEST

The first thing I did was to create an Ultra pipeline to record incoming requests.
This pipeline eventually became the SOAP Service, but I needed to verify the incoming headers since they contain one we care about -- SOAPAction -- which specifies which action is being called.



- Binary to Document (Setting: None)
- JSON Formatter (*remember to select Format Each Document*)
- File Writer to write out the request to a JSON file (*remember to add the output view*)

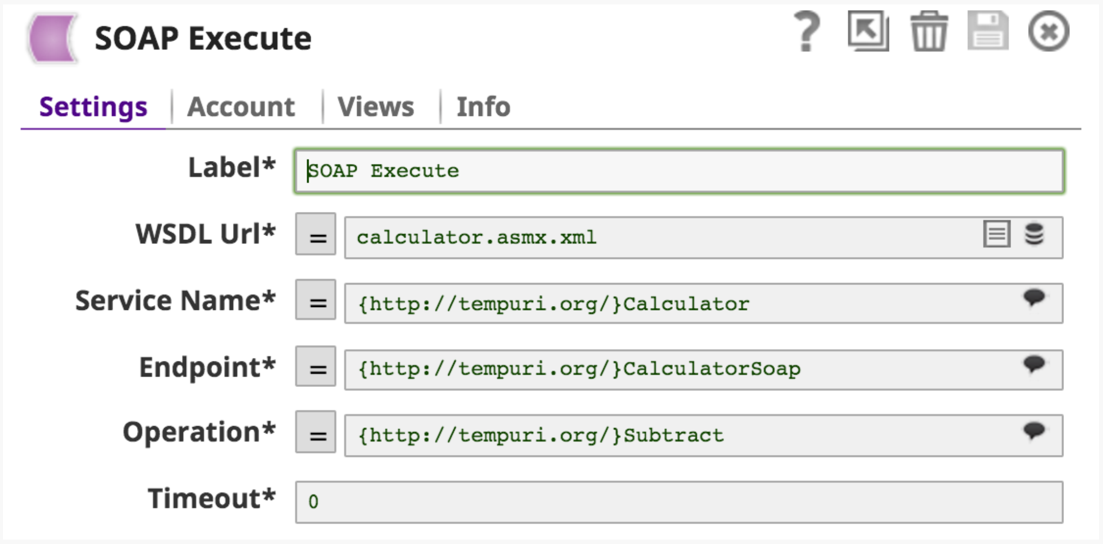
I enabled the above pipeline and noted down its URL and Authentication info.

Next, I grabbed and saved the WSDL file:
`http://www.dneonline.com/calculator.asmx?wsdl`

Before using it, I had to edit the URI's of the SOAP Ports (at the bottom of the file) to point to the Ultra task's URL, since it originally pointed to URI's at www.dneonline.com.

```
<wsdl:service name="Calculator">
    <wsdl:port name="CalculatorSoap" binding="tns:CalculatorSoap">
      <soap:address location="https://prodxl-
jcc236.fullsail.snaplogic.com:8084/api/1/rest/feed-
master/queue/<client>/projects/UC5%20-%20API%20Calls/xml-ultra-task" />
    </wsdl:port>
    <wsdl:port name="CalculatorSoap12" binding="tns:CalculatorSoap12">
      <soap12:address location="https://prodxl-
jcc236.fullsail.snaplogic.com:8084/api/1/rest/feed-
master/queue/<client>/projects/UC5%20-%20API%20Calls/xml-ultra-task" />
    </wsdl:port>
  </wsdl:service>
```

Next, I created another pipeline to use for calling the SOAP Ultra. I added a SOAP snap, uploaded the modified WSDL, and added the Authentication header from the Ultra task. Then I could select the service name, endpoint and operation in the normal fashion.


SOAP Execute

Settings | Account | Views | Info

Label* SOAP Execute

WSDL Url* calculator.asmx.xml

Service Name* {http://tempuri.org/}Calculator

Endpoint* {http://tempuri.org/}CalculatorSoap

Operation* {http://tempuri.org/}Subtract

Timeout* 0

After clicking on the "Customize Envelope" button, I ended up with the following SOAP envelope, with two parameters, intA and intB.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns0="http://tempuri.org/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns0:Subtract>
            <ns0:intA>$intA</ns0:intA>
            <ns0:intB>$intB</ns0:intB>
        </ns0:Subtract>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Next, I added a mapper in front of the SOAP Execute snap, removed its input view, and mapped values to intA and intB. The SOAP Execute will fail at this point because we aren't returning a valid SOAP response yet. At this point, all we care about is that the Ultra pipeline we're calling is recording what the SOAP Execute is sending to it. The message received by the Ultra pipeline ended up looking like the following:

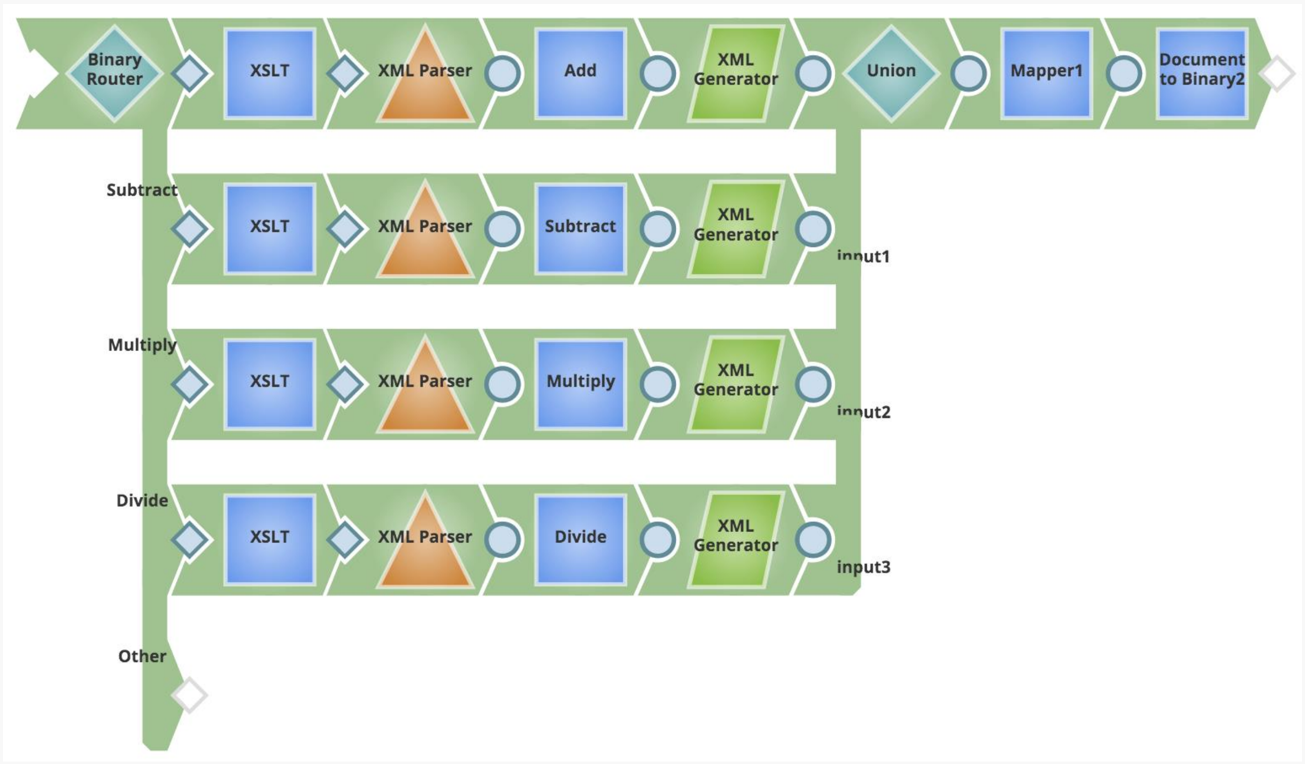
```
{
  "task_name": "<client>/projects/UC5 - API Calls/xml-ultra-task",
  "content-length": "321",
  "soapaction": "\"http://tempuri.org/Multiply\"",
  "method": "POST",
  "query": {},
  "message_id": "cf108605e2e75edee3cda74958d0aef193823e3b-27953@prodxl-
jcc236.fullsail.snaplogic.com",
  "uri": "https://prodxl-jcc236.fullsail.snaplogic.com:8084/api/1/rest/feed-
master/queue/<client>/projects/UC5%20-%20API%20Calls/xml-ultra-task",
  "content": "<soap:Envelope xmlns:ns1=\"http://tempuri.org/\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\"><SOAP-ENV:Header
xmlns:SOAP-
ENV=\"http://schemas.xmlsoap.org/soap/envelope/\"/><soap:Body><ns1:Multiply><
ns1:intA>123\n            </ns1:intA><ns1:intB>456\n            </ns1:intB></
ns1:Multiply></soap:Body></soap:Envelope>",
  "accept": "*/*",
  "client_port": 53800,
  "path_info": "",
  "host": "prodxl-jcc236.fullsail.snaplogic.com:8084",
  "server_ip": "10.153.235.168",
  "content-type": "text/xml; charset=UTF-8",
  "client_ip": "10.71.177.96",
  "server_port": 8084,
  "user-agent": "Apache CXF 2.7.16"
}
```

IMPLEMENTATION

Within the payload above, we care about two things within this captured request:

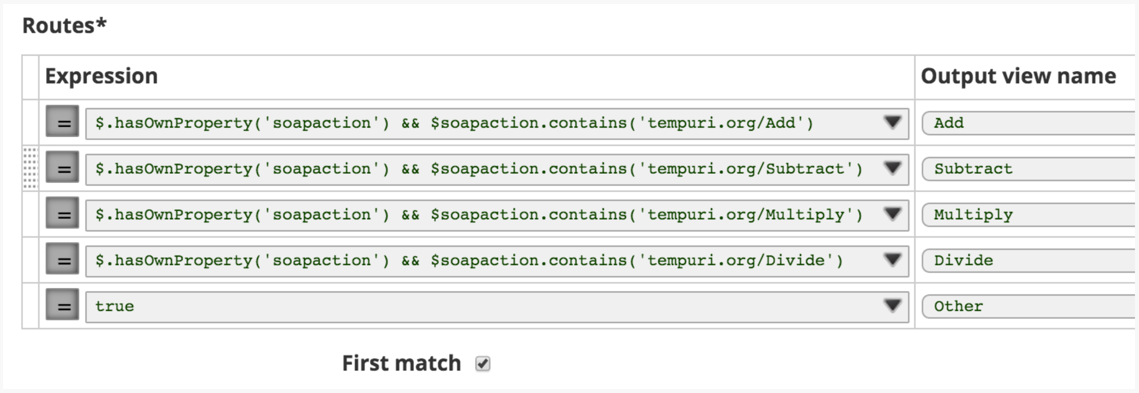
- "**soapaction**", which is a header being sent by the SOAP Execute snap
- "**content**", which is an XML string of the SOAP message being sent

So next, I built out the pipeline in place of the original Ultra depicted/described above.



Now let's step through each of the snaps and discuss what each does & why:

1) Binary Router, based on content of "soapaction" -- Normally SOAP clients add a header named "SOAPAction", but note that Ultra converts the header's key name to lowercase. I added an "Other" route at the bottom because Ultra pipelines need to return "something" when called, no matter what, or the caller can hang forever/for a long time if nothing gets returned. This extra route ensures that sometime will get returned if the value of the SOAPAction header isn't handled.


Routes*

Expression	Output view name
`$.hasOwnProperty('soapaction') && $soapaction.contains('tempuri.org/Add')`	Add
`$.hasOwnProperty('soapaction') && $soapaction.contains('tempuri.org/Subtract')`	Subtract
`$.hasOwnProperty('soapaction') && $soapaction.contains('tempuri.org/Multiply')`	Multiply
`$.hasOwnProperty('soapaction') && $soapaction.contains('tempuri.org/Divide')`	Divide
`true`	Other

First match ✓

2) XSLT to remove namespaces from XML elements. <u>This is a bonus step, but a very good one to consider.</u> Downstream, if you have logic that looks for the two numbers being operated upon, they will be named something like "ns1:intA" and "ns1:intB". Everything element will have a namespace, and we don't want our downstream mappers to be dependent on any particular/hardcoded namespace. For instance, when calling the pipeline via SOAPUI, you'll see the namespaces "soapenv" and "tem". To get the data, you'd have to use something like:
$['soapenv:Envelope'].['soapenv:Body'].['tem:Add'].['tem:intA']

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tem="http://tempuri.org/">
   <soapenv:Header/>
   <soapenv:Body>
      <tem:Add>
         <tem:intA>123</tem:intA>
         <tem:intB>123</tem:intB>
      </tem:Add>
   </soapenv:Body>
</soapenv:Envelope>
```

vs with SnapLogic's SOAP Execute, you'll see namespaces "SOAP-ENV" and "ns0". To get the data, you'd have to use something like: $['SOAP-ENV:Envelope'].['SOAP-ENV:Body'].['ns0:Subtract'].['ns0:intA']

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ns0="http://tempuri.org/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <ns0:Subtract>
            <ns0:intA>$intA</ns0:intA>
            <ns0:intB>$intB</ns0:intB>
        </ns0:Subtract>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If we leave namespaces intact, building mappings against a particular namespace can mean that your pipeline will work for one SOAP client, but may fail against another.

The following XSLT can be applied to the XML to remove the namespaces.

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="*">
    <xsl:element name="{local-name()}">
      <xsl:apply-templates select="node()|@*" />
    </xsl:element>
  </xsl:template>

  <xsl:template match="@*">
    <xsl:attribute name="{local-name()}">
      <xsl:apply-templates select="node()|@*" />
    </xsl:attribute>
  </xsl:template>

</xsl:stylesheet>
```

By stripping off the namespaces, we end up with something like this:

```
  {
    "Envelope": {
      "Header": {
        "SOAP-ENV": "http://schemas.xmlsoap.org/soap/envelope/"
      },
      "Body": {
        "Multiply": {
          "intA": "123",
          "intB": "456"
        }
      }
    }
  }
```

which allows us to grab up values via $Envelope.Body.Multiply.intA and $Envelope.Body.Multiply.intB

3) XML Parser - standard settings

4) Mappers to do the arithmetic and map the result to a field

For example, here's the mapping logic for the Add route. You can see how removing the namespaces not only *simplifies* our task, but also makes for a *much more robust* pipeline!

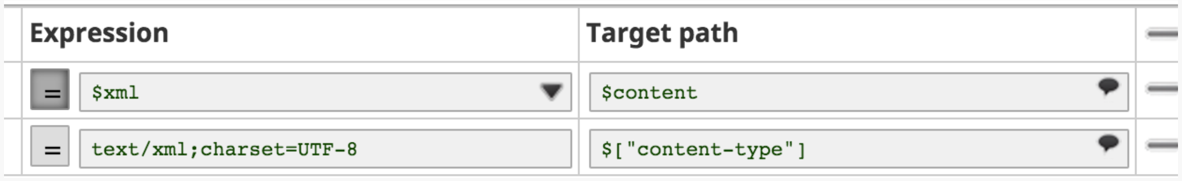
parseInt($Envelope.Body.Add.intA) + parseInt($Envelope.Body.Add.intB) --> $answer

5) XML Generator to generate what's going back to the SOAP client. Each of these will be different because of the unique elements like AddResponse and AddResult, SubtractResponse and SubtractResult, etc.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
        <AddResponse
            xmlns="http://tempuri.org/">
            <AddResult>$answer</AddResult>
        </AddResponse>
    </soap:Body>
</soap:Envelope>
```

6) Union - Join the flows together

7) Mapper into Document to Binary - map the XML generated above, which ends up in a field called $xml to $content, and add the static text/xml stuff as content-type.

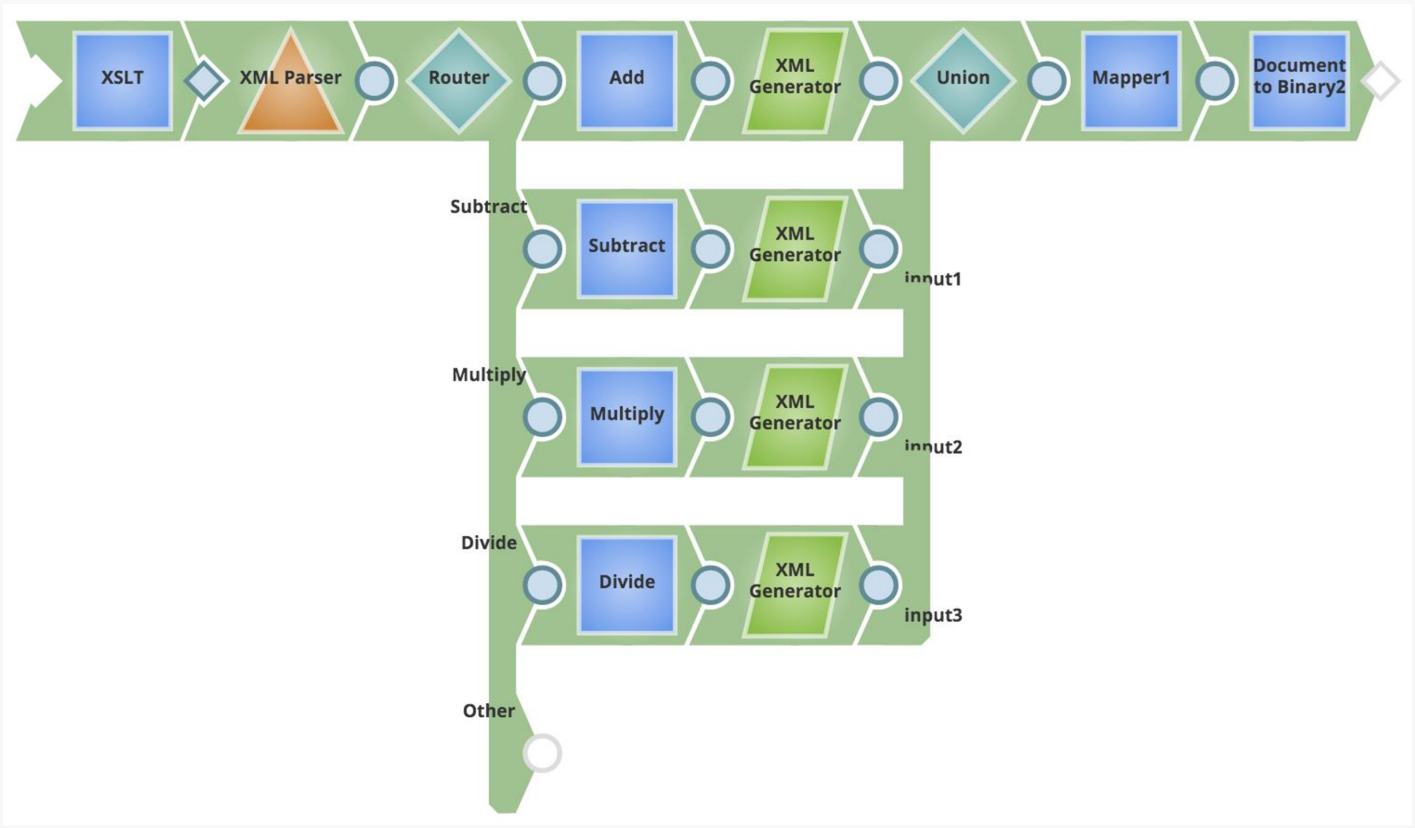


Expression	Target path
$xml	$content
text/xml;charset=UTF-8	$["content-type"]

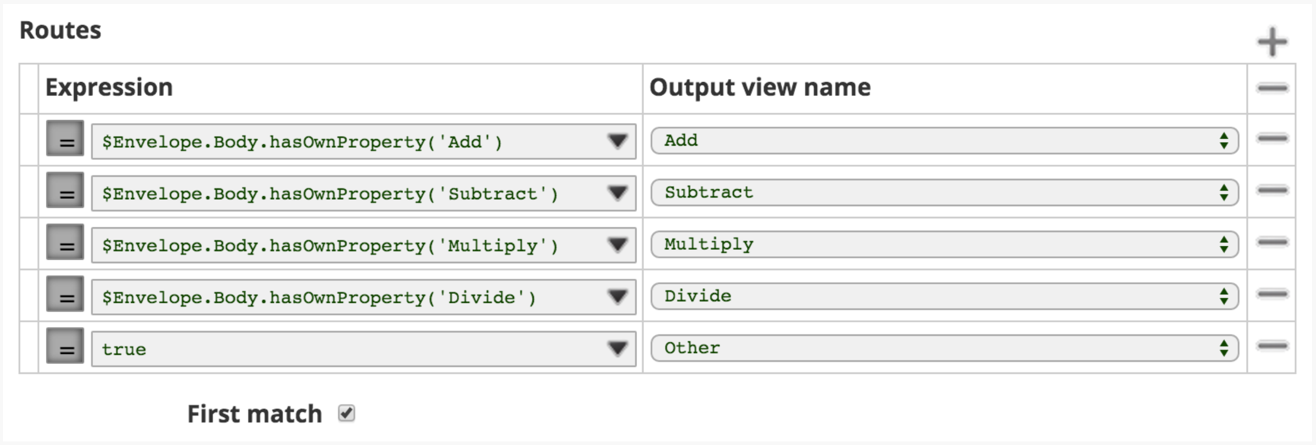
8) Document to Binary

SIMPLIFICATION

This pipeline could be simplified quite a bit (25 snaps reduced to 14) by removing the initial snaps (which we needed for routing based on the SOAPAction header). We can route our logic using the SOAP envelope's contents, rather than the SOAP Action header. This is a non-standard approach, and the assumptions within make it potentially a bit more dangerous. Note that we still have an "Other" route at the bottom of the Router.



In this simplified approach, the Router would look like this:



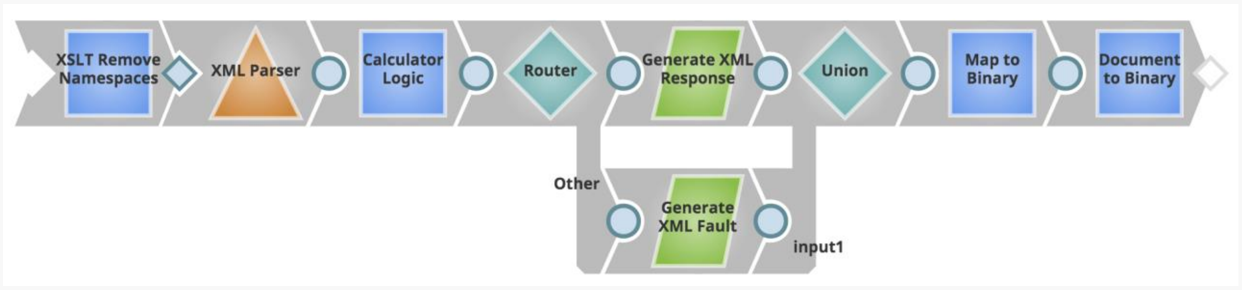
Routes

Expression	Output view name
`$Envelope.Body.hasOwnProperty('Add')`	Add
`$Envelope.Body.hasOwnProperty('Subtract')`	Subtract
`$Envelope.Body.hasOwnProperty('Multiply')`	Multiply
`$Envelope.Body.hasOwnProperty('Divide')`	Divide
`true`	Other

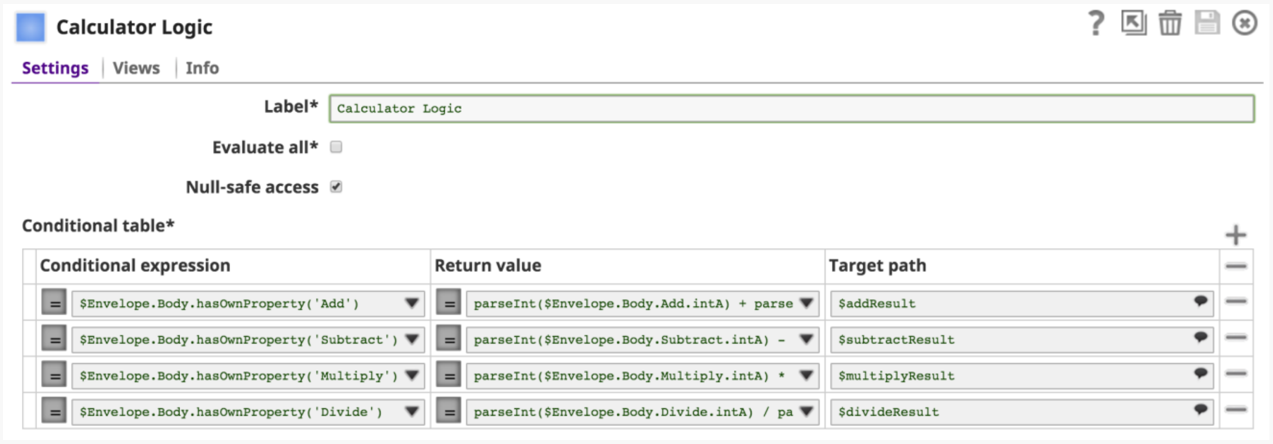
First match ✓

OVER-SIMPLIFICATION?

I considered trying to further simplify using a Conditional snap with these expressions, and mapping an compound object (AddResponse, SubtractResponse, etc) into a single XML Generator (rather than mapping just the $answer). This may be an oversimplification, because the pipeline is getting less robust as we go, but having fewer snaps can make our pipeline simpler and faster at processing requests. It ends up looking like this:



We could use a Conditional snap ("Calculator Logic" above) to perform the logic, based on the incoming envelope:



Calculator Logic

Label*: Calculator Logic

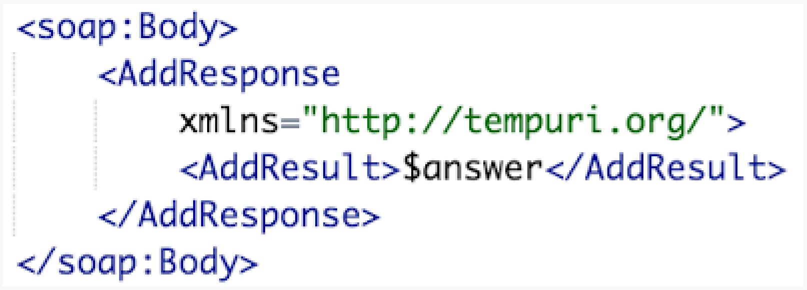
Evaluate all*: ☐

Null-safe access: ☑

Conditional table*

Conditional expression	Return value	Target path
$Envelope.Body.hasOwnProperty('Add')	parseInt($Envelope.Body.Add.intA) + parse	$addResult
$Envelope.Body.hasOwnProperty('Subtract')	parseInt($Envelope.Body.Subtract.intA) -	$subtractResult
$Envelope.Body.hasOwnProperty('Multiply')	parseInt($Envelope.Body.Multiply.intA) *	$multiplyResult
$Envelope.Body.hasOwnProperty('Divide')	parseInt($Envelope.Body.Divide.intA) / pa	$divideResult

One problem we've got to solve when generating the XML is that the response element includes a namespace. The namespace (xmlns="http://tempuri.org", see below) can't be added within our Return Value expressions in the Conditional snap.



```
<soap:Body>
    <AddResponse
        xmlns="http://tempuri.org/">
        <AddResult>$answer</AddResult>
    </AddResponse>
</soap:Body>
```

To solve this problem, the Velocity Template within the XML Generator can be applied to use the existence (or not) of different elements ($addResult, $subtractResult, etc) to generate content which corresponds to whichever operation is being handled.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>

        #if ($addResult)
        <AddResponse
            xmlns="http://tempuri.org/">
            <AddResult>$addResult</AddResult>
        </AddResponse>
        #end

        #if ($subtractResult)
        <SubtractResponse
            xmlns="http://tempuri.org/">
            <SubtractResult>$subtractResult</SubtractResult>
        </SubtractResponse>
        #end

        #if ($multiplyResult)
        <MultiplyResponse
            xmlns="http://tempuri.org/">
            <MultiplyResult>$multiplyResult</MultiplyResult>
        </MultiplyResponse>
        #end

        #if ($divideResult)
        <DivideResponse
            xmlns="http://tempuri.org/">
            <DivideResult>$divideResult</DivideResult>
        </DivideResponse>
        #end

    </soap:Body>
</soap:Envelope>
```

The Router is there to make sure that one of the Conditional cases got applied -- if not, the Ultra pipeline will still return "something" and not hang the caller.

So now we're down to 9 snaps, from the original 20. We could have reduced it to just 7 snaps, but we added some robustness back -- handing for a SOAP Fault response if the caller's request wasn't one of those we handle. Maybe that's going too far with simplification, but just goes to show how much you can leverage the snaps when you get creative!

On a final note, a properly, robustly built Ultra pipeline needs to be "fail proof". Error Views have to be added wherever a failure might occur, because the SnapLogic platform will disable an Ultra task if it fails more than the task's failure threshold (default is 10). We also want to remember to always return something to the caller, and the best way is to be sure that we aren't silently eating up errors within the pipeline.

Max Failures 10

DO I NEED TO BUILD A SOAP SERVICE USING ULTRA?

Not necessarily:

Yes, if you intend to use the SOAPAction header to route the request (see initial version of the pipeline at the beginning of this article).

No, if you intend to use the SOAP body (see simplified approach above).