Inside the WSDL & Web Services Proxies

Why do we need to know?

At the end of the day when the dust settles, there will be only a handful of Web Service Providers on the internet and most of us will remain as "consumers". We can better utilize Web Services only when we understand the "Contract" of the services. So understanding the Web Service contract (spelled in WSDL) is the key to leverage a Web Service.

Though there are tools available to generate a proxy (like WSDL.EXE from Microsoft's .NET Toolkit), having a better understanding of the WSDL and the proxy always helps in troubleshooting issues while consuming Web Services.

In this article we will consider a real Web Service: Scandinavian Airline (SAS) Flight Status Web Service to understand the WSDL and then we will dissect the proxy that is generated for the SAS Flight Status Web Service.

All code samples presented in this article are developed in C#.

Scandinavian Airlines "Flight Status Web Service"

The flight status Web Service for Scandinavian Flight Status went live recently and can be found in the UDDI directory by searching for "Business Name – Scandinavian".
The location of ".asmx" for this Web Service is:
http://webservices.scandinavian.net/flightstatus/flightservice.asmx.
Now, let us take a look at the Web Service Definition Language (WSDL) document for the above Web Service. A WSDL document defines the Web Service as a collection of pre-defined elements. As we all know we can get the WSDL using a web browser by adding a query "?WSDL" at the end of the above URL.
The WSDL document for the Web Services looks as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:s0="http://tempuri.org/" targetNamespace="http://tempuri.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://tempuri.org/">
      <s:element name="GetFlightStatus">
        <s:complexType>
          <s:sequence>
              <s:element minOccurs="1" maxOccurs="1"
                name="nFlightNo" type="s:int" />
              <s:element minOccurs="1" maxOccurs="1"
                name="nDayOffset" type="s:int" />
            </s:sequence>
          </s:complexType>
        </s:element>
      <s:element name="GetFlightStatusResponse">
```

```
        <s:complexType>
          <s:sequence>
              <s:element minOccurs="1" maxOccurs="1"
                name="GetFlightStatusResult"
                nillable="true" type="s:string" />
            </s:sequence>
          </s:complexType>
        </s:element>
        <s:element name="string" nillable="true" type="s:string" />
      </s:schema>
    </types>
  <message name="GetFlightStatusSoapIn">
      <part name="parameters" element="s0:GetFlightStatus" />
    </message>
```

Fig 1: WSDL document for the Scandinavian Flight Status Web Service (shown partially).

Now let us take a look at the WSDL document in detail to understand the "Web Service Contract"!

Understanding the contract:

The "collapsed" view of the full WSDL document (generated in previous section) looks as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="http://tempuri.org/" targetNamespace="http://tempuri.org/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
+ <types>
+ <message name="GetFlightStatusSoapIn">
+ <message name="GetFlightStatusSoapOut">
+ <message name="GetFlightStatusHttpGetIn">
+ <message name="GetFlightStatusHttpGetOut">
+ <message name="GetFlightStatusHttpPostIn">
+ <message name="GetFlightStatusHttpPostOut">
+ <portType name="FlightServiceSoap">
+ <portType name="FlightServiceHttpGet">
+ <portType name="FlightServiceHttpPost">
+ <binding name="FlightServiceSoap" type="s0:FlightServiceSoap">
+ <binding name="FlightServiceHttpGet" type="s0:FlightServiceHttpGet">
+ <binding name="FlightServiceHttpPost" type="s0:FlightServiceHttpPost">
+ <service name="FlightService">
</definitions>
```

Fig 2: "Collapsed" view of the full WSDL document.

By examining the above document structure, one can see that there are five unique element types under the root node "definitions" (the root node is "definitions" after the fact that WSDL is simply a set of definitions that define a Web Service). They are

?? types
?? message
?? portType
?? binding
?? service

Apart from the above elements there is another important element that is used in defining a Service, the "port". Including this sixth element *we have a total of six element types in a WSDL document*.

Let us take a look each of them to understand what they really are and what we can decipher from them.

1. types

The types element encloses data type definitions that are relevant for the exchanged messages between the proxy and the Web Service. For maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system.

The "types" element from SAS Web Service WSDL is shown in Fig 3. As we can see, there are three elements: GetFlightStatus, GetFlightStatusResponse and string.

```
<types>
    <s:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://tempuri.org/">
      <s:element name="GetFlightStatus">
        <s:complexType>
          <s:sequence>
              <s:element minOccurs="1" maxOccurs="1"
                name="nFlightNo" type="s:int" />
              <s:element minOccurs="1" maxOccurs="1"
                name="nDayOffset" type="s:int" />
            </s:sequence>
          </s:complexType>
        </s:element>
      <s:element name="GetFlightStatusResponse">
        <s:complexType>
          <s:sequence>
              <s:element minOccurs="1" maxOccurs="1"
                name="GetFlightStatusResult"
                nillable="true" type="s:string" />
            </s:sequence>
          </s:complexType>
        </s:element>
        <s:element name="string" nillable="true" type="s:string" />
      </s:schema>
    </types>
```

Fig 3: types element from SAS Web Service WSDL

As we can see, the simple type definitions (the "type" attribute in "element" node) are referring to namespace "s" which is nothing but: http://www.w3.org/2001/XMLSchema.

The structure of "GetFlightStatus" is defined as a sequence of two integers: "nFlightNo" and "nDayOffset".
The element "GetFlightStatusResponse" is defined as a "string" by name "GetFlightStatusResult".
The third and last element "string" is defined as a "string" type.

2. message

Messages are the abstract definition of the data being exchanged between the proxy and the Web Service.
The first message element from SAS Web Service WSDL is shown in Fig 4 below.

```
- <message name="GetFlightStatusSoapIn">
    <part name="parameters" element="s0:GetFlightStatus" />
  </message>
```

Fig 4: First message element from SAS Web Service WSDL document.

As we can see, the message element contains one or more logical parts. Parts are a flexible way of describing the logical abstract content of a message or to put it simply, a part may represent a parameter in a message.

In our example from fig4, we can see the element "part" contains an element type "GetFlightStatus" from namespace "s0", which in turn points to the same document with the definition (some times the same document can be referred to as "tns"): xmlns:s0="http://tempuri.org/" targetNamespace=http://tempuri.org/.

The relationship between message and types is shown below in Fig 5:

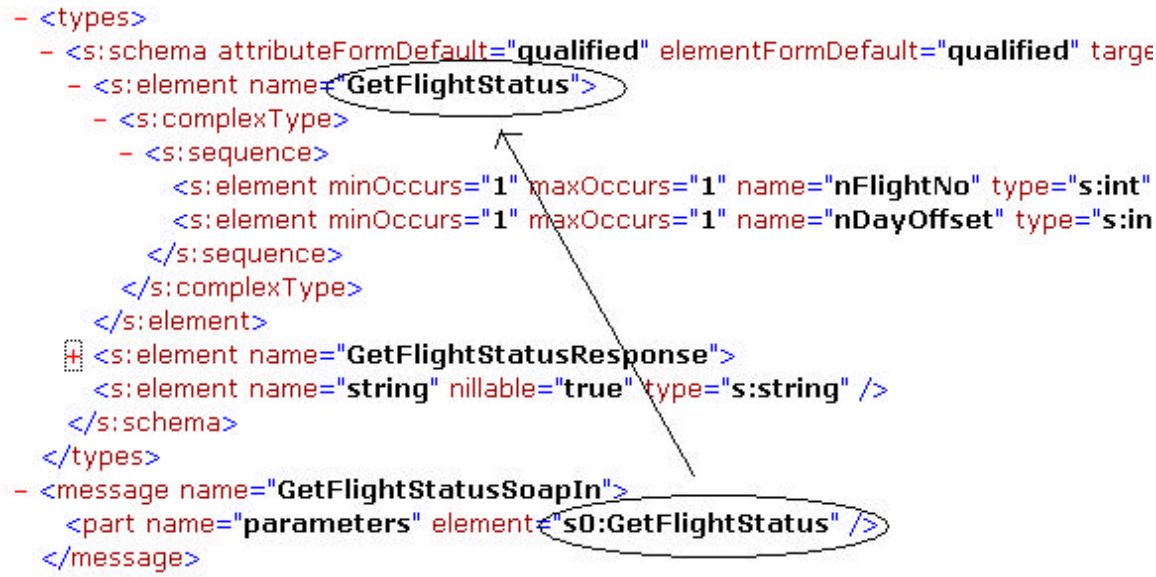

Fig 5: Relationship between "message" and "types"

So from the above relationship we can derive:

The message with name "GetFlightStatusSoapIn" contains a part called "parameters" of type "GetFlightStatus" element which is nothing but a sequence of two integers: namely "nFlightNo" and "nDayOffset".

There could be messages that contain parts with simple types like "string". In that case the part may point to element types from the XSD namespace as shown in Fig 6 below.

```
- <message name="GetFlightStatusHttpGetIn">
    <part name="nFlightNo" type="s:string" />
    <part name="nDayOffset" type="s:string" />
  </message>
```

Fig 6: message with simple types defined as parts.

Finally there could be many messages present in a service definition. In our example there are six different messages defined: a set of two messages (Request and Response), for the protocols "SOAP", "HTTP-GET" and "HTTP-POST".

3. portType

A portType is defined as a set of abstract "operations" that involves messages. The operations are nothing but the transportation primitives that a network-end point can support, such as a "Request-Response".

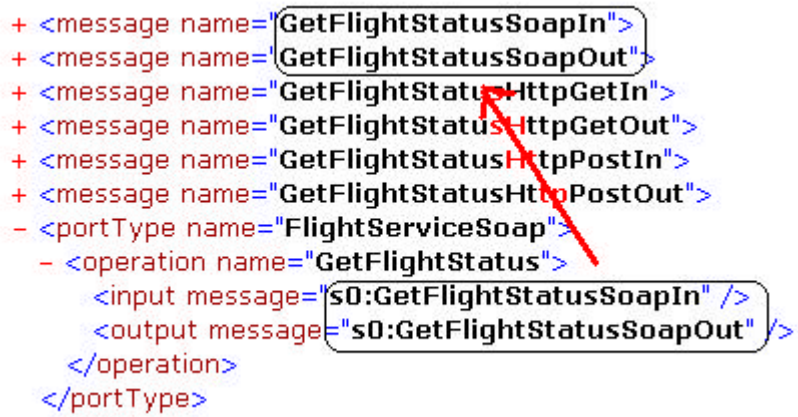

Fig 7: Relationship between portType and message.

The relationship between portType and message is shown pictorially in Fig 7. There are three different portTypes defined in the SAS Web Service WSDL document and each portType represents an operation-message combination for different protocols.

So a **portType = operation + messages**

4. binding

A binding defines message format and protocol details for operations and messages defined by a particular portType. Let us take a look at the first binding element from our WSDL to understand what it means:

```
- <binding name="FlightServiceSoap" type="s0:FlightServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  - <operation name="GetFlightStatus">
      <soap:operation soapAction="http://tempuri.org/GetFlightStatus" style="document" />
    - <input>
        <soap:body use="literal" />
      </input>
    - <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
```

Fig 8: SOAP binding definition from SAS Web Service WSDL document.

There are some important points that we need to observe in the binding element:

- ?? The binding element refers to the portType (FlightServiceSoap) using “type” attribute.
- ?? The binding element is the first element to specify a “protocol” (SOAP in Fig 8) in the WSDL document so far!

An operation element within a binding element specifies binding information for the operation specified in that particular binding’s portType. Since operation names are not required to be unique, the name attribute in the operation element might not be enough to uniquely identify an operation. In that case, providing the name attributes of the corresponding input and output elements should identify the correct operation (Method Overloading!).

So a **binding = protocol + portType**

5. service

A service element is a collection of related ports and a port defines an individual network endpoint by specifying a single address for a binding.

The first “port” element’s definition in the service element is shown below in Fig 9.

```
- <port name="FlightServiceSoap" binding="s0:FlightServiceSoap">
    <soap:address location="http://webservices.scandinavian.net/flightstatus/flightservice.asmx" />
  </port>
```

Fig 9: port element definition.

As we can see, the port contains the URL (the network endpoint) of the Web Service and also has a reference to the binding element as an attribute.

So a **port = binding + network address**

With this background information about the “port”, let us take a look at the “service” element:

```
+ <portType name="FlightServiceHttpPost">
+ <binding name="FlightServiceSoap" type="s0:FlightServiceSoap">
+ <binding name="FlightServiceHttpGet" type="s0:FlightServiceHttpGet">
+ <binding name="FlightServiceHttpPost" type="s0:FlightServiceHttpPost">
- <service name="FlightService">
  - <port name="FlightServiceSoap" binding="s0:FlightServiceSoap">
      <soap:address location="http://webservices.scandinavian.net/flightstatus/flightservice.asmx" />
    </port>
  - <port name="FlightServiceHttpGet" binding="s0:FlightServiceHttpGet">
      <http:address location="http://webservices.scandinavian.net/flightstatus/flightservice.asmx" />
    </port>
  - <port name="FlightServiceHttpPost" binding="s0:FlightServiceHttpPost">
      <http:address location="http://webservices.scandinavian.net/flightstatus/flightservice.asmx" />
    </port>
  </service>
```

Fig 10: service element definition.

As shown in the Fig 10, a service is a collection of related but mutually exclusive ports. Which means that all the three ports belong to the same service but none of them really communicate with each other. Also all these related ports provide semantically equivalent behavior by allowing Web Service consumers to choose

a particular port to communicate with the Web Service based on protocol implementation or some other criteria.

With this background, now we understand the Web Service contract spelled in WSDL. To summarize our understanding, we can define the SAS Flight Status Web Service contract in plain English as below:

SAS Flight Status Web Service
?? Accepts two input parameters (of type “int”)
?? Returns one parameter (of type “string”)
?? Address of the Web Service is: http://webservices.scandinavian.net/flightstatus/flightservice.asmx
?? Allows us to choose the protocol from SOAP, HTTP-GET and HTTP-POST

With this information, we are ready to move on to “consume” this Web Service!

Dissecting the proxy:

In this part of the article let’s dissect the proxy to understand it better. We can use the “WSDL.EXE” utility to generate a proxy, but for this one time let us hand-code the proxy to have a better understanding!

Let us keep the UI for this Web Application simple: a Web Form, which accepts “Flight Number” and “Day offset” and a text box to display the returned XML in the form of a string. The UI looks like the figure shown below:

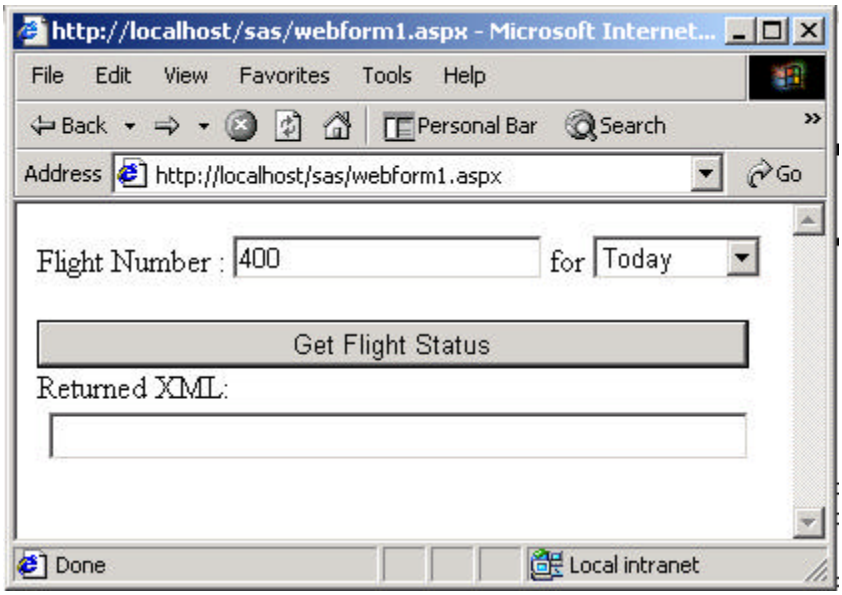

Fig 11: Web Form for SAS Web Service proxy

The “code behind” page for this web form is “WebForm1.aspx.cs”. Now, let us decide if we really want a separate “proxy” dll that contains code to invoke the Web Service, or if it is okay to add that functionality in the “code behind” file itself! Assuming that we are not re-using the code to invoke the SAS Flight Status Web Service anywhere else and also since we are hand-coding the proxy, let us incorporate the proxy code in the “code behind” file, which is “WebForm1.aspx.cs”.

Web Service Client Rules for Inheritors

Rule#1

Any Web Service client using ASP.NET needs to declare a class deriving indirectly from WebClientProtocol. I used the word “indirectly” because, there are sub classes under WebClientProtocol that are specific to the protocol that we decided to use while communicating the Web Service. They are

?? SoapHttpClientProtocol for SOAP
?? HttpGetClientProtocol for HTTP-GET
?? HttpPostClientProtocol for HTPP-POST

All the three classes mentioned above are derived directly or indirectly from WebClientProtocol class.

For example:

```
public class SoapCall : System.Web.Services.Protocols.SoapHttpClientProtocol
```

Rule# 2

While implementing SOAP protocol to consume the Web Service, we need to define 2 important attributes for that class:

WebServiceBindingAttribute:
This attribute’s “Name” property maps to a specific binding (*FlightServiceSoap* in our example) from the WSDL document (shown in Fig 12). As we have seen earlier a binding is like an “interface” and should be implemented with each protocol.

```
+ <binding name="FlightServiceSoap" type="s0:FlightServiceSoap">
+ <binding name="FlightServiceHttpGet" type="s0:FlightServiceHttpGet">
+ <binding name="FlightServiceHttpPost" type="s0:FlightServiceHttpPost">
```

Fig 12: binding mapping to the WebServiceBindingAttribute

In our example it would be:

```
[System.Web.Services.WebServiceBindingAttribute(Name="FlightServiceSoap",
Namespace="http://tempuri.org/")]
```

SoapDocumentMethodAttribute:
This attribute holds the value of the “soapAction” attribute of operation element defined in the “FlightServiceSoap” binding. Shown in the Fig 13 below.

```
- <binding name='FlightServiceSoap" type="s0:FlightServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
  - <operation name="GetFlightStatus">
      <soap:operation soapAction="http://tempuri.org/GetFlightStatus" style="document" />
    - <input>
        <soap:body use="literal" />
      </input>
    - <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
```

Fig 13: SoapDocumentMethodAttribute mapping in WSDL document

In our example it would be:

```
[System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://tempuri.org/
GetFlightStatus", Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
```

The other two properties “Use” and “ParameterStyle” are set to the default values. These parameters are used to set the format of the SOAP request or response sent to or from the Web Service. Hence this attribute is optional.

Rule# 3

If we are using HTTP-GET protocol to invoke the Web Service, ReturnFormatter property must be set to XmlReturnReader and ParameterFormatter property must be set to UrlParameterWriter in HttpMethodAttribute attribute.

This attribute is required to indicate the types that serialize parameters sent to a Web Service method and read the response from the Web Service.

In our case it would be as follows:

```
[System.Web.Services.Protocols.HttpMethodAttribute(typeof(System.Web.Services.P
rotocols.XmlReturnReader),
typeof(System.Web.Services.Protocols.UrlParameterWriter))]
```

Rule# 4

Web Service clients using HTTP-POST protocol must set ReturnFormatter to XmlReturnReader and ParameterFormatter to HtmlFormParameterWriter in HttpMethodAttribute attribute.

In our case it would look like below:

```
[System.Web.Services.Protocols.HttpMethodAttribute(typeof(System.Web.Services.P
rotocols.XmlReturnReader),
typeof(System.Web.Services.Protocols.HtmlFormParameterWriter))]
```

With the above rules in place we can now hand-code “proxy” code in code-behind file to invoke the Web Service to get the Flight Status.

I have defined the following three classes to implement each of the above-mentioned protocol.

For SOAP protocol:

```
//This implements SOAP
[System.Web.Services.WebServiceBindingAttribute(Name="FlightServiceSoap",
Namespace="http://tempuri.org/")]
public class SoapCall : System.Web.Services.Protocols.SoapHttpClientProtocol
{
       public SoapCall(string url)
       {
              this.Url = url;
       }
     [System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://tempur
i.org/GetFlightStatus",
Use=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
       public string GetFlightStatus(int nFlightNo, int nDayOffset)
```

```
        {
                object[] result = this.Invoke("GetFlightStatus", new object []
{nFlightNo, nDayOffset});
                return (string)(result[0]);
        }
}
```

For HTTP-GET:

```
//This implements HTTP-GET
public class HttpGetCall : System.Web.Services.Protocols.HttpGetClientProtocol
{
        public HttpGetCall(string url)
        {
                this.Url = url;
        }
      [System.Web.Services.Protocols.HttpMethodAttribute(typeof(System.Web.Serv
ices.Protocols.XmlReturnReader),
typeof(System.Web.Services.Protocols.UrlParameterWriter))]
                [return: System.Xml.Serialization.XmlRootAttribute("string",
Namespace="http://tempuri.org/", IsNullable=true)]
        public string GetFlightStatus(string nFlightNo, string nDayOffset)
        {
                object result = this.Invoke("GetFlightStatus", this.Url +
"/GetFlightStatus",  new object [] {nFlightNo, nDayOffset});
                return result.ToString();
        }
}
```

For HTTP-POST protocol:

```
//This implements HTTP-POST
public class HttpPostCall :
System.Web.Services.Protocols.HttpPostClientProtocol
{
        public HttpPostCall(string url)
        {
                this.Url = url;
        }

      [System.Web.Services.Protocols.HttpMethodAttribute(typeof(System.Web.Serv
ices.Protocols.XmlReturnReader),
typeof(System.Web.Services.Protocols.HtmlFormParameterWriter))]
                [return: System.Xml.Serialization.XmlRootAttribute("string",
Namespace="http://tempuri.org/", IsNullable=true)]
        public string GetFlightStatus(string nFlightNo, string nDayOffset)
        {
                return (string)this.Invoke("GetFlightStatus", this.Url +
"/GetFlightStatus",  new object [] {nFlightNo, nDayOffset});
        }
}
```

In all the cases mentioned above, you could find one similarity: the Invoke method. This call will invoke the actual web service using corresponding protocol. Also as we can see the Invoke method takes the Web Service Method name and an array of objects containing parameters to pass to the remote Web Service. One important thing to keep in mind is that the order of the values in the array must correspond to the order of the parameters in the calling method.

And also we have to make sure the following namespaces are defined in the “code behind” file:

```
using System.Web.Services.Protocols;
using System.Xml.Serialization;
using System.Web.Services;
```

Now, we are all set to consume the Web Service. The code snippet below shows how to invoke the Web Service using the SOAP protocol:

```
SoapCall objSP = new
SoapCall("http://webservices.scandinavian.net/flightstatus/flightservice.asmx")
;
flightInfo.Text =  objSP.GetFlightStatus(int.Parse(flightNo.Text),
int.Parse(offset.SelectedItem.Value));
```

(For detailed code please take a look at the support material)

I did not add any processing to parse the XML to show the results since it is out of the scope of this article. But basically we are ready to communicate with the Scandinavian Web Service for SAS Flight Status. You can also verify the results using the SAS Flight Status page.

Test Data and Result XML

To test your Web Service Consumer code, you can use the following test data:

For testing purposes I have considered the Flight No# SK400, which is a scheduled flight from STOCKHOLM, SWEDEN (ARN) to COPENHAGEN, DENMARK (CPH). You have to enter "400" for the flight number (usually they are SK400, but we need to pass in only integer for flight number).

You can find all Scandinavian flight codes at SAS Website provided in related links.

Flight Status check on the above flight will return a XML string that looks like below:

```
- <Flight FlightNo="400" Date="20011007" Cancelled="false">
  - <Segments>
    - <Segment Index="0">
        <From>CPH</From>
        <To>ARN</To>
        <STD>07:30</STD>
        <STA>08:40</STA>
        <ETD>--:--</ETD>
        <ETA>08:34</ETA>
        <ATD>07:28</ATD>
        <ATA>08:30</ATA>
      </Segment>
    </Segments>
  </Flight>
```

"TD" and "TA" represent "Time of Departure" and "Time of Arrival" respectively, and "S", "E" and "A" as prefix represents "Scheduled", "Estimated" and "Actual" respectively.

Summary

In most of the cases, we need no knowledge of the above stuff on how the WSDL is deciphered into proxy since we use the WSDL.EXE tool. However, the knowledge of how the proxy works would be helpful in some cases when we want to extend the Web Service's functionality by adding more layers of processing in the proxy to give a customized output. The best example would be deciphering the airport codes in the above SAS Flight Status XML!

In either case better understanding of WSDL and proxy's make our life easy as Web Service Consumers.

Related Links:

?? WSDL Specification
?? SOAP Specification
?? Scandinavian Air Lines Web Site
?? UDDI Organization