Search documentation Search

Advanced Web Service Configuration

Appian 7.7

NOTE: This site does not include documentation on the latest release of Appian. Please upgrade to benefit from our newest features. For more information on the latest release of Appian, please see the Appian 7.8 documentation

The information on this page is provided by the Appian Center of Excellence

Toggle Na

This page is *not* supported by Appian Technical Support. For additional assistance, please contact your Appian Account Executive to engage with Appian Professional Services.

There may be circumstances where the out-of-the-box functionality for consuming web services (Call Web Service and Web Service Functions) may no be possible due to how the service is defined.

Rather than resorting to creating a Custom Smart Service or Custom Function to consume the service, first consider if the web service definition can be modified to allow it to be consumed natively in Appian. Failing that the HTTP Connector (a!httpQuery) should be used before finally considering customisation.

Modifying the web service definition involves changing the services WSDL and hosting the modified version where Appian can consume it. This docume details the benefits, considerations and how to modify the WSDL.

Benefits and Considerations

A modified WSDL does not incur any extra risk that isn't already inherent to dealing with web services in general. Modifying a WSDL is the equivalent to using the HTTP Connector (a!httpQuery) or creating a Custom Smart Service or Custom Function. All require a similar level of analysis and design to identify the problematic elements blocking out-of-the-box consumption and capturing the essential aspects of the interface. However each has their ow level of effort and maintenance.

WSDL modification does not require Java expertise or extensive knowledge about how HTTP works. The modifications can be made quickly, reducing t time to test and minimizes the risk associated with custom development (support, maintenance, local development costs, etc).

Known issues with a WSDL that can be addressed by modifying the WSDL:

- No WSDL provided
- Multi-part WSDL
- WSDLs missing SOAP header declaration
- WSDLs missing data type definitions or other required elements
- WSDLs missing policy definitions that define WS-Security requirements
- Data types that use the extension tag

Prerequisites

Before attempting to perform WSDL modification the following prerequisites must be met:

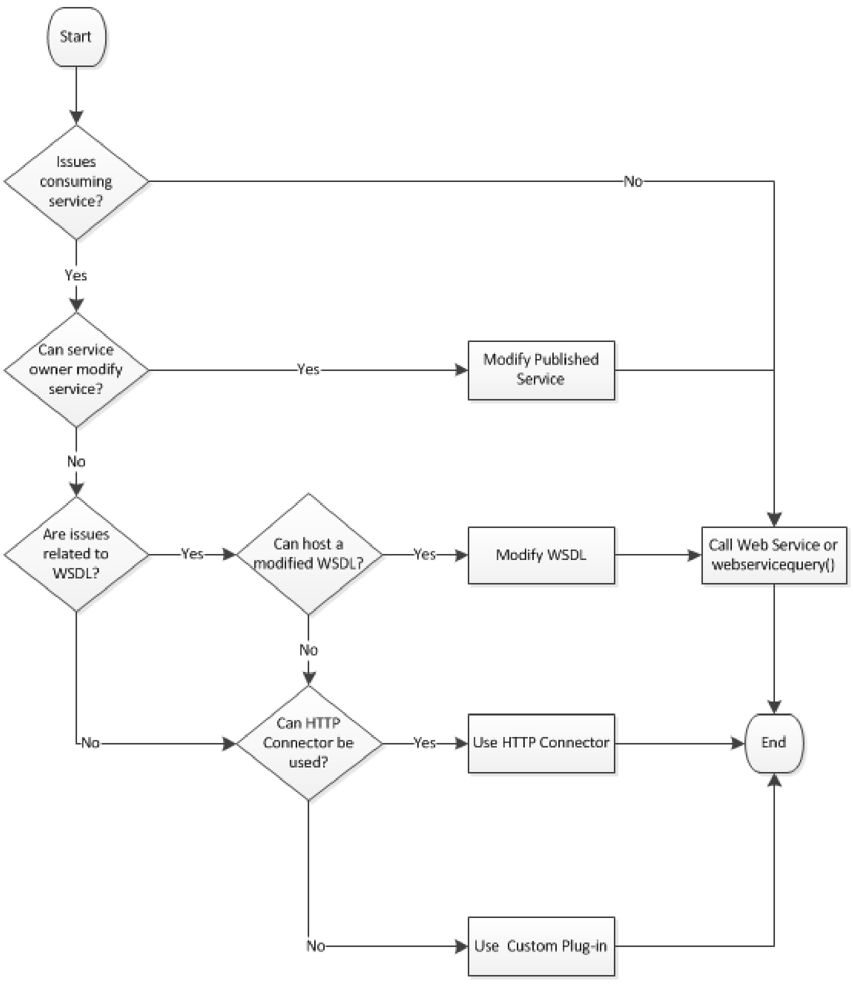
- The service cannot be modified to support consumption.
  - For example, if the service owner will not modify the service or create a support version for Appian.
- No WSDL is provided and the only known information comes from sample SOAP requests and responses in XML format.
  - If the examples are not SOAP based and simply XML payloads use the HTTP Connector instead of creating a WSDL.
- The WSDL uses import tags to reference other files that contain pieces of the definition and the Call Web Service node is unable to piece togethe all the parts.
- The official WSDL is not supported as-is by the "Call Web Service" smart service due to incompatible or missing elements (e.g. SOAP Header definition).
- Hosting capabilities are available to host the modified WSDL.

Best Practices

- Always use the out-of-the-box functionality and avoid customizations.
- Work with the web service owner to modify the published service before attempting WSDL modification.
- Use WSDL modification before using HTTP Connector functions or customisations.
- Custom smart services and functions for the most part should only be used when issues unrelated to the WSDL definition prevent use of base product functionality.

Decision Matrix

Follow the flow diagram below to identify the most appropriate solution:


Start
Issues consuming service?
No
Yes
Can service owner modify service?
Yes
Modify Published Service
No
Are issues related to WSDL?
Yes
Can host a modified WSDL?
Yes
Modify WSDL
Call Web Service or webservicequery()
No
No
Can HTTP Connector be used?
Yes
Use HTTP Connector
End
No
Use Custom Plug-in

WSDL Modification

Creating a New WSDL

To create a new WSDL from scratch, it is first necessary to understand the basic structure of a WSDL. Since there are plenty of tutorials on this only th
basics will be covered. See tutorialspoint WSDL tutorial for more details.

There are 5 main sections for a WSDL to keep in mind:

1. Types
2. Message
3. PortType
4. Binding
5. Service

Here's an example SOAP request and response. Highlighted in red are the standard SOAP elements and in blue the key pieces of information specific to this example:

Request

```
<soapenv:Envelope
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:urn="urn:examples:helloservice">
 <soapenv:Body>
   <urn:sayHello
     soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <firstName xsi:type="xsd:string">John</firstName>
   </urn:sayHello>
 </soapenv:Body>
</soapenv:Envelope>
```

Response

```
<soapenv:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:urn="urn:examples:helloservice">
  <soapenv:Body>
    <urn:sayHelloResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <greeting xsi:type="xsd:string">Hello John</greeting>
    </urn:sayHelloResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

To create a new WSDL, we need a basic WSDL structure that we can fill out with the details identified in the request and response. Here's an example below. Highlighted in red are all the elements that need to be filled out:

Basic Blank WSDL Structure

```
<?xml version="1.0"?>
<definitions name="?"
  targetNamespace="?"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="?"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
  </types>
  <message name="?">
  </message>
  <portType name="?">
    <operation name="?">
      <input message="?"/>
      <output message="?"/>
    </operation>
  </portType>
  <binding name="?" type="?">
    <soap:binding style="?"
    transport="?"/>
    <operation name="?">
     <soap:operation soapAction="?"/>
      <input>
        <soap:body
          encodingStyle="?"
          namespace="?"
          use="?"/>
      </input>
      <output>
        <soap:body
          encodingStyle="?"
          namespace="?"
          use="?"/>
      </output>
    </operation>
  </binding>
  <service name="?">
    <documentation>WSDL Base Structure Example</documentation>
    <port binding="?" name="?">
      <soap:address
        location="?"/>
    </port>
  </service>
</definitions>
```

Here's the WSDL filled out with the information that's available in the request and response. Highlighted in red are the elements that are still missing and blue the elements just filled out:

WSDL Filled with Request and Response Data

```
<?xml version="1.0"?>
<definitions name="?"
  targetNamespace="?"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="?"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <!-- No special type definitions needed. -->
    <!-- Only the basic xsd:string type is used in this example. -->
  </types>
  <message name="?">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="?">
    <part name="greeting" type="xsd:string"/>
  </message>
  <portType name="?">
    <operation name="sayHello">
      <input message="?"/>
      <output message="?"/>
    </operation>
  </portType>
  <binding name="?" type="?">
    <soap:binding style="?"
    transport="?"/>
    <operation name="sayHello">
      <soap:operation soapAction="?"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="?">
    <documentation>WSDL File for Hello Service - Manual</documentation>
    <port binding="?" name="?">
      <soap:address
        location="?"/>
    </port>
  </service>
</definitions>
```

The missing elements are of 2 types:

- **Generic** - These are names and namespaces for the different elements which don't affect the resulting requests and response. They are just needed to define the different elements and to be able to reference them throughout the WSDL.
- **Web Service Configuration** - These are configuration properties to connect to the target Web Service. That information isn't available in the request and response. This information should be provided by the owner of the Web Service.

Next, we are going to fill out the generic elements. We will just choose random values different from the original tutorialspoint example to illustrate how i doesn't change the end result. Highlighted in red are the elements that are still missing and in blue the elements just filled out:

WSDL Filled With Generic Data

```
<?xml version="1.0"?>
<definitions name="SayHelloWSDL"
  targetNamespace="urn:examples:helloservice:defaultNamespace"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="urn:examples:helloservice:defaultNamespace"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <!-- No special type definitions needed. -->
    <!-- Only the basic xsd:string type is used in this example. -->
  </types>
  <message name="firstNameMessage">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="greetingMessage">
    <part name="greeting" type="xsd:string"/>
  </message>
  <portType name="sayHelloPortType">
    <operation name="sayHello">
```

```
      <input message="tns:firstNameMessage"/>
      <output message="tns:greetingMessage"/>
    </operation>
  </portType>
  <binding name="sayHelloBinding" type="tns:sayHelloPortType">
    <soap:binding style="?"
    transport="?"/>
    <operation name="sayHello">
      <soap:operation soapAction="?"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="?">
    <documentation>WSDL File for Hello Service - Manual</documentation>
    <port binding="tns:sayHelloBinding" name="SayHelloPort">
      <soap:address
        location="?"/>
    </port>
  </service>
</definitions>
```

Finally, we fill out the Web Service Configuration elements. For this example, we will just use the sample values from the tutorialspoint original example. I practice this information would be requested from the Web Service owner. Highlighted in blue are the last elements filled out.

WSDL Filled With Web Service Configuration Data

```
<?xml version="1.0"?>
<definitions name="SayHelloWSDL"
  targetNamespace="urn:examples:helloservice:defaultNamespace"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="urn:examples:helloservice:defaultNamespace"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <types>
    <!-- No special type definitions needed. -->
    <!-- Only the basic xsd:string type is used in this example. -->
  </types>
  <message name="firstNameMessage">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="greetingMessage">
    <part name="greeting" type="xsd:string"/>
  </message>
  <portType name="sayHelloPortType">
    <operation name="sayHello">
      <input message="tns:firstNameMessage"/>
      <output message="tns:greetingMessage"/>
    </operation>
  </portType>
  <binding name="sayHelloBinding" type="tns:sayHelloPortType">
    <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
        use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="Hello_Service">
    <documentation>WSDL File for Hello Service - Manual</documentation>
    <port binding="tns:sayHelloBinding" name="SayHelloPort">
      <soap:address
        location="http://www.examples.com/SayHello/"/>
    </port>
  </service>
</definitions>
```

New WSDL Creation Checklist

1. Identify sample requests and responses that cover all the possible elements that can be included.
2. Create a base generic WSDL structure.
3. Fill the WSDL with the data available in the sample requests and responses.
4. Fill in the generic missing data following your projects conventions.
5. Fill in the Web Service configuration data.

Merging XSD Files Into A Single WSDL

Sometimes a WSDL might reference other files, especially when it contains a complex definition. Any piece of the WSDL can be defined in its own separate file and referenced on the main WSDL via the "import" tag.

These files would be one of 2 possible types:

1. **wsdl** - Contains one or more of the 5 basic root elements of the WSDL.
2. **xsd** - Contains exclusively type definitions.
   - Note: The "types" element should NOT be defined here. The "types" element itself can be defined in its own separate wsdl file though.

To ensure the Appian Web Service node reads the full WSDL definition properly when it fails to do so for multipart WSDLs, you can merge all the referenced files into a single WSDL by following these steps:

1. Identify the main WSDL.
   - Note: Look for the only WSDL file that is not referenced by imports from any of the other WSDL files.
2. Substitute all "import" tags with the respective content from the file referenced in the "location" or "schemaLocation" attributes.
3. Update the "definitions" tag (this is the main tag that contains all the WSDL elements) to include all the namespace definitions from the "definition tags from all the separate WSDL files.
   - Note: If there are any conflicts with repeated naming conventions between files (e.g. they all use xmlns:tns to reference their own namespace that's different from each of the files), manually refactor the namespaces in each file to make them unique among all the WSDI files before merging. This can be done with a simple find and replace of the namespace prefix (e.g. xmlns:tns).

Merging XSD Files Checklist

1. Identify or create the main WSDL.
2. Substitute all the "import" tags into the main WSDL.
3. Update the attributes of the "definitions" tag.

Transforming Extension Tags

The extension tag works in the same way as Java inheritance via class extension. The extension tag is used as a placeholder to indicate that the referenced data type’s properties should be included as well on the data type using the extension tag. See w3schools extension element tutorial for mo details.

Using the example from the w3schools tutorial, here's an illustration on how to transform extension tags to generate an equivalent data type definition that doesn't rely on the extension tag.

Highlighted in red are the elements that need to be removed from the original data type with extension, and in blue the elements that have to be moved/added to the data type without extension.

Data Type With Extension

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="employee" type="fullpersoninfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:schema>
```

Data Type Without Extension

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="employee" type="fullpersoninfo"/>
<xs:complexType name="fullpersoninfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="address" type="xs:string"/>
   <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

Transform Extension Tags Checklist

1. Identify the extension tag within a data type definition.
2. Search for the data type definition of the type referenced in the extension tag.
   - Note: To quickly find the data type and skip all other extension tags referencing it, search for the following string: name=".
3. Replace the extension tag with the elements of the data type referenced following the examples listed above.
4. Repeat until all extension tags have been replaced.

Adding a SOAP Header

To add a SOAP header to a WSDL definition, there are 3 sections that need to be updated:

1. Data type definition
2. Message definition
3. Operation input/output definitions

The SOAP header itself is defined in the operation input/output definitions. It points to a message definition, which contains the different parts of the header. The header parts are mapped to elements, which have a defined data type associated it to them, be it simple or complex.

Here's a minimalist example of a WSDL definition that contains a SOAP header to capture user credentials. Highlighted in blue are the elements involved defining the SOAP header.

```
<?xml version="1.0"?>
<definitions name="HelloService"
  targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:types="urn:examples:helloservice:types">
  <types>
    <xsd:schema
      targetNamespace="urn:examples:helloservice:types">
      <xsd:element name="SecurityElement" type="types:SecurityType"/>
      <xsd:complexType name="SecurityType">
        <xsd:sequence>
          <xsd:element name="username" type="xsd:string"/>
          <xsd:element name="password" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </types>
  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>
  <message name="SecurityMessage">
    <part name="SecurityHeader" element="types:SecurityElement"/>
  </message>
  <portType name="Hello_PortType">
```

```
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
  <binding name="Hello_Binding" type="tns:Hello_PortType">
    <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
        <soap:header
          message="tns:SecurityMessage"
          part="SecurityHeader"
          use="literal"/>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
        location="http://www.examples.com/SayHello/"/>
    </port>
  </service>
</definitions>
```

Adding a SOAP Header Checklist

1. Create the type definition that describes the SOAP header content.
2. Update the attributes of the "definitions" tag.
3. Create the message element that references the SOAP Header data type.
4. Add/Update the "soap:header" element on the "input" tag of the respective operations inside the bindings tag.

Adding WS-Security UsernameToken Policy

A service may be secured using WS-Security UsernameToken, but not define it in the published WSDL. The WSDL can be updated to include this policy

This is a minimalist example of a WSDL definition that contains a WS-Security UsernameToken policy definition. Highlighted in red is the policy identifier and in blue the key pieces of information added to this example:

```
<?xml version="1.0"?>
<definitions name="HelloService"
  targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:sp="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsp:UsingPolicy wsdl:required="true"/>
  <wsp:Policy wsu:Id="UsernameTokenPolicy">
    <wsp:ExactlyOne xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
      <wsp:All>
        <wsp:Policy xmlns:wsp="http://www.w3.org/ns/ws-policy"
          xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
          <wsp:ExactlyOne>
            <wsp:All>
              <sp:SupportingTokens xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
                <wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
                  <sp:UsernameToken/>
                </wsp:Policy>
              </sp:SupportingTokens>
            </wsp:All>
          </wsp:ExactlyOne>
        </wsp:Policy>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
  <types>
    <!-- No special type definitions needed. -->
    <!-- Only the basic xsd:string type is used in this example. -->
  </types>
  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>
  <portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
  <binding name="Hello_Binding" type="tns:Hello_PortType">
    <wsp:Policy>
      <wsp:PolicyReference URI="UsernameTokenPolicy"/>
    </wsp:Policy>
    <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHello">
      <soap:operation soapAction="sayHello"/>
      <input>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </input>
      <output>
        <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:examples:helloservice"
          use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="Hello_Service">
    <documentation>WSDL File for HelloService</documentation>
    <port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address
        location="http://www.examples.com/SayHello/"/>
    </port>
  </service>
</definitions>
```

Adding WS-Security UsernameToken Policy Checklist

1. Add the WS-Security related namespaces to the definitions tag.
2. Define the policy as required.
3. Add a policy definition that defines the WS-Security UsernameToken method.
4. Reference the new policy is the service bindings.

Testing

SoapUI can be used to test the different examples illustrated above.

- Records
- Interfaces
- Process

Release Information

- Release Notes
- Installation
- Migration
- System Requirements
- Hotfixes
- Release History

Other

- STAR Methodology
- Best Practices
- Glossary
- APIs