FREE eBook

LEARNING

soap

Free unaffiliated eBook created from
Stack Overflow contributors.

#soap

Table of Contents

About ... 1

Chapter 1: Getting started with soap ... 2

Remarks ... 2

Versions ... 2

Examples ... 2

General Information ... 2

SOAP ... 3

Differences between SOAP 1.1 and 1.2 ... 4

Web Service Interoperability ... 5

WSDL ... 5

WSDL 1.1 ... 5

WSDL 2.0 ... 8

Differences between WSDL 1.1 and 2.0 ... 9

Which style to prefer ... 11

RPC / encoded ... 11

RPC / literal ... 12

Document / encoded ... 13

Document / literal ... 14

Document / literal (wrapped) ... 15

UDDI ... 16

Further notes: ... 16

Java Client for Weather service open source webservice avaliable on http://www.webserviceX ... 17

Creating a Simple Web Service and Clients with JAX-WS (Document / literal) ... 18

Chapter 2: Consuming SOAP Web Service ... 22

Introduction ... 22

Parameters ... 22

Examples ... 22

Without creating Stub or Java files ... 22

Credits ... 24

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: soap

It is an unofficial and free soap ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official soap.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with soap

Remarks

This section provides an overview of what soap is, and why a developer might want to use it.

It should also mention any large subjects within soap, and link out to the related topics. Since the Documentation for soap is new, you may need to create initial versions of those related topics.

Versions

Version	Release Date
1.1	2000-05-08
1.2	2003-06-24

Examples

General Information

SOAP is an acronym for *Simple Object Access Protocol* which defines a protocol that is used to exchange data via a Remote Procedure Call (RPC) with other SOAP services or clients. It is available in two version:

- SOAP 1.1 [IETF]
- SOAP 1.2 [IETF]

SOAP 1.2 obsoletes SOAP 1.1 it is therefore recommended to use SOAP 1.2 if possible.

It is often build on top of HTTP/S and rarely on SMTP or FTP, though it would support it according to the protocol. Although HTTP is often used as underlying transportation protocol, SOAP uses only a limited subset of it. For sending requests it relies almost completely on HTTP's `POST` operation. `GET` invocations are theoretically possible since 1.2, though the document has to be passed as URI parameter and thus may exceed a roughly 3000 character boundary which is rejected by most frameworks. Also, security related settings are usually defined within a special SOAP header.

Although SOAP and REST are called web-services, they are very different by nature. Some frameworks distinguish between WS (for SOAP based services) and RS (for REST based services).

The following table gives a brief overview on the differences between both web service types.

Aspect	SOAP	REST
Standard	SOAP, WSDL	No standard, just an architectural style
Resource addressing	Indirect via SOAP operations	via unique resource identifiers (URIs)
Error handling	SOAP fault message	HTTP error response codes and optionally response body
Data representation	XML	all available encodings in HTTP
HTTP usage	As transport protocol	Actions on resources (CRUD) mapped on HTTP methods (GET, POST, PUT, DELETE, ...)
Transactional support	via SOAP header	by modeling a transaction as a resource
State	Stateful (SOAP action is part of the application)	Stateless (self-contained requests)
Service discovery	UDDI / WSDL	None actually; Start-URI of the API should return a list of sub APIs though
Method	Inside SOAP body	HTTP method
Method arguments	Defined by XML schema in the WSDL	Either via HTTP Headers or Path/Query or Matrix parameters within the URI
State transition	Difficult to determine as not directly based on data	Next URI invocation
Caching support	Caching often not desired,	Simple as defined by HTTP

SOAP

A SOAP requests consists of a SOAP envelop which has to contain a body element and may contain an optional header element. The header element is used to pass certain configurations to the service like i.e. WS-Security may defined that the message is encrypted or WS-Coordination/WS-Transaction may define that the message has to be executed within a transaction.

A simple SOAP 1.2 requests via HTTP which adds two values may look like this:

```
POST /calculator HTTP/1.1
Host: http://example.org
```

```
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 224

<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Body>
  <m:AddValues xmlns:m="http://example.org/calculator">
   <m:FirstValue>1</m:FirstValue>
   <m:SecondValue>2</m:SecondValue>
  </m:AddValues>
 </env:Body>
</env:Envelope>
```

A response to the above sample request may look like this

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 329

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
 <soap:Body xmlns:m="http://www.example.org/calculator">
  <m:AddValuesResponse>
   <m:Result>3</m:Result>
  </m:AddValuesResponse>
 </soap:Body>
</soap:Envelope>
```

The example above defined a request which invoked `AddValues` method with two arguments, `FirstValue` set to `1` and `SecondValue` set to `2`. The request resulted in an execution of this method on the remote SOAP server which calculated a value of `3` as a result which is encapsulated in a separate response element, which by convention is often the invoked method name plus a trailing `Response` string so anyone who is inspecting the response can conclude that this is the response of a previous `AddValue` method invocation.

Differences between SOAP 1.1 and 1.2

SOAP 1.2 allows for other transport protocols then HTTP as long as the binding framework is supported by the protocol.

SOAP 1.1 is based on XML 1.0, while 1.2 is based on XML Infoset which allows to serialize the SOAP messages with other serializers then the default XML 1.0 serializer used by SOAP 1.1. This allows i.e. to serialize messages as binary messages and therefore prevent some overhead of the XML nature of the message. In addition to that, the serialization mechanism of the underlying protocol used can be determined via the data binding.

The interoperability aspect was also fostered with SOAP 1.2 by defining a more specific processing model then its predecessor which eliminated many possibilities for interpretation. SOAP with Attachment API (SAAJ), which allows to operate on SOAP 1.1 and 1.2 messages, helped many framework implementors to process and create messages.

W3C has released a short overview on the main changes between SOAP 1.1 and 1.2

Web Service Interoperability

Web Service Interoperability (also known as WS-I) is an interoperability guideline governed by some well known enterprises such as IBM, Microsoft, Oracle and HP to name just a few. These guidelines among others recommend to use only one single root element within the SOAP body even though the SOAP does allow to contain multiple elements within the body.

WS-I consists of

- WS-I Basic Profile a.k.a WSI-BP
- WS-I Basic Security Profile
- Simple Soap Binding Profile

WSI-BP is available in 4 different versions v1.0 (2004), v1.1 (2006), v1.2 (2010), v2.0 (2010) and defines interoperability guidelines for core web service specifications such as SOAP, WSDL and UDDI. Through the use of Web Services Description Language (WSDL), SOAP services can describe their supported operations and methods within a cohesive set to other endpoints. WSI-BP makes use of WSDL to define a narrower set then the full WSDL or SOAP schema would define and thus eliminates some of the ambiguity within the specification itself and thus improve the interoperability between endpoints.

WSDL

In order to advertise the available SOAP operations, their parameters as well as the respective endpoints to invoke to clients, a further XML based document is used called `Web Services Description Language` or WSDL for short.

WSDL describes the service endpoint, the binding of SOAP messages to operations, the interface of operations as well as their types to clients. WSDL is, like SOAP, available in 2 versions which differ in their syntax slightly though express almost the same semantics to the client.

WSDL 1.1

A WSDL 1.1 description contains of a `service`, a `binding`, a `portType` and a `message` section. It can further import or define schemas within the WSDL file as can be seen from a sample WSDL file which corresponds to the calculator sample shown above:

```
<wsdl:definitions xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
                  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
                  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
                  xmlns:calc="http://example.org/calculator"
                  xmlns:tns="http://example.org/calculatorService"
                  targetNamespace="http://example.org/calculatorService">
```

```
    <!--
      Abstract type definitions
    -->

    <wsdl:types>
        <!--
        <xs:schema>
            <xs:import namespace="http://example.org/calculator"
schemaLocation="calc/calculator.xsd" />
        </xs:schema>
        -->
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:tns="http://example.org/calculator"
                   targetNamespace="http://example.org/calculator"
                   elementFormDefault="qualified"
                   attributeFormDefault="qualified">

            <xs:element name="AddValuesRequest" type="tns:AddValuesType" />
            <xs:element name="AddValuesResponse" type="tns:AddValuesResponseType" />

            <xs:complexType name="AddValuesType">
                <xs:sequence>
                    <xs:element name="FirstValue" type="xs:int" minOccurs="1" maxOccurs="1" />
                    <xs:element name="SecondValue" type="xs:int" minOccurs="1" maxOccurs="1"
/>
                </xs:sequence>
            </xs:complexType>

            <xs:complexType name="AddValuesResponseType">
                <xs:sequence minOccurs="1" maxOccurs="1">
                    <xs:element name="Result" type="xs:int" />
                </xs:sequence>
            </xs:complexType>

            <xs:attribute name="Timestamp" type="xs:dateTime" />
            <xs:element name="CalculationFailure">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ErrorCode" type="xs:int" />
                        <xs:element name="Reason" type="xs:string" />
                    </xs:sequence>
                    <xs:attribute ref="tns:Timestamp" use="required" />
                </xs:complexType>
            </xs:element>

        </xs:schema>
    </wsdl:types>

    <!--
        Abstract message definitions
    -->

    <wsdl:message name="AddValuesRequest">
        <wsdl:part name="in" element="calc:AddValuesRequest" />
    </wsdl:message>
    <wsdl:message name="AddValuesResponse">
        <wsdl:part name="out" element="calc:AddValuesResponse" />
    </wsdl:message>
    <wsdl:message name="CalculationFault">
        <wsdl:part name="fault" element="calc:CalculationFailure" />
    </wsdl:message>
```

```
    <!--
        Abstract portType / interface definition
    -->

    <wsdl:portType name="CalculatorEndpoint">
        <wsdl:operation name="AddValues">
            <wsdl:documentation>Adds up passed values and returns the
result</wsdl:documentation>
            <wsdl:input message="tns:AddValuesRequest" />
            <wsdl:output message="tns:AddValuesResponse" />
            <wsdl:fault name="CalculationFault" message="tns:CalculationFault" />
        </wsdl:operation>
    </wsdl:portType>

    <!--
      Concrete binding definition
    -->

    <wsdl:binding name="CalculatorBinding" type="tns:CalculatorEndpoint">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="AddValues">
            <soap:operation soapAction="http://example.org/calculator/AddValuesMessage" />
            <wsdl:input>
                <soap:body parts="in" use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body parts="out" use="literal" />
            </wsdl:output>
            <wsdl:fault name="CalculationFault">
                <soap:fault name="CalculationFault" use="literal" />
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>

    <!--
        Concrete service definition
    -->

    <wsdl:service name="CalculatorService">
        <wsdl:port name="CalculatorServicePort" binding="tns:CalculatorBinding">
            <soap:address location="http://localhost:8080/services/calculator" />
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

A `service` section defines the concrete endpoints the service will listen for incoming requests. The `binding` section binds an operation to a concrete style and defines which message formats the server expects or the client can expect.

The abstract section is composed of a `portType` block which defines the operations offered by the service and which messages are exchanged. The messages are specified in their on block and linked to the schema types the arguments and return values are instances of. Messages can declare parameters or return values to be `in`, `out` or `inout`. While the first two are quite simple to grasp the latter mimics the behavior of arguments passed by reference. As pass-by-ref is not supported in some languages this effect is often simulated via certain handlers.

WSDL 2.0

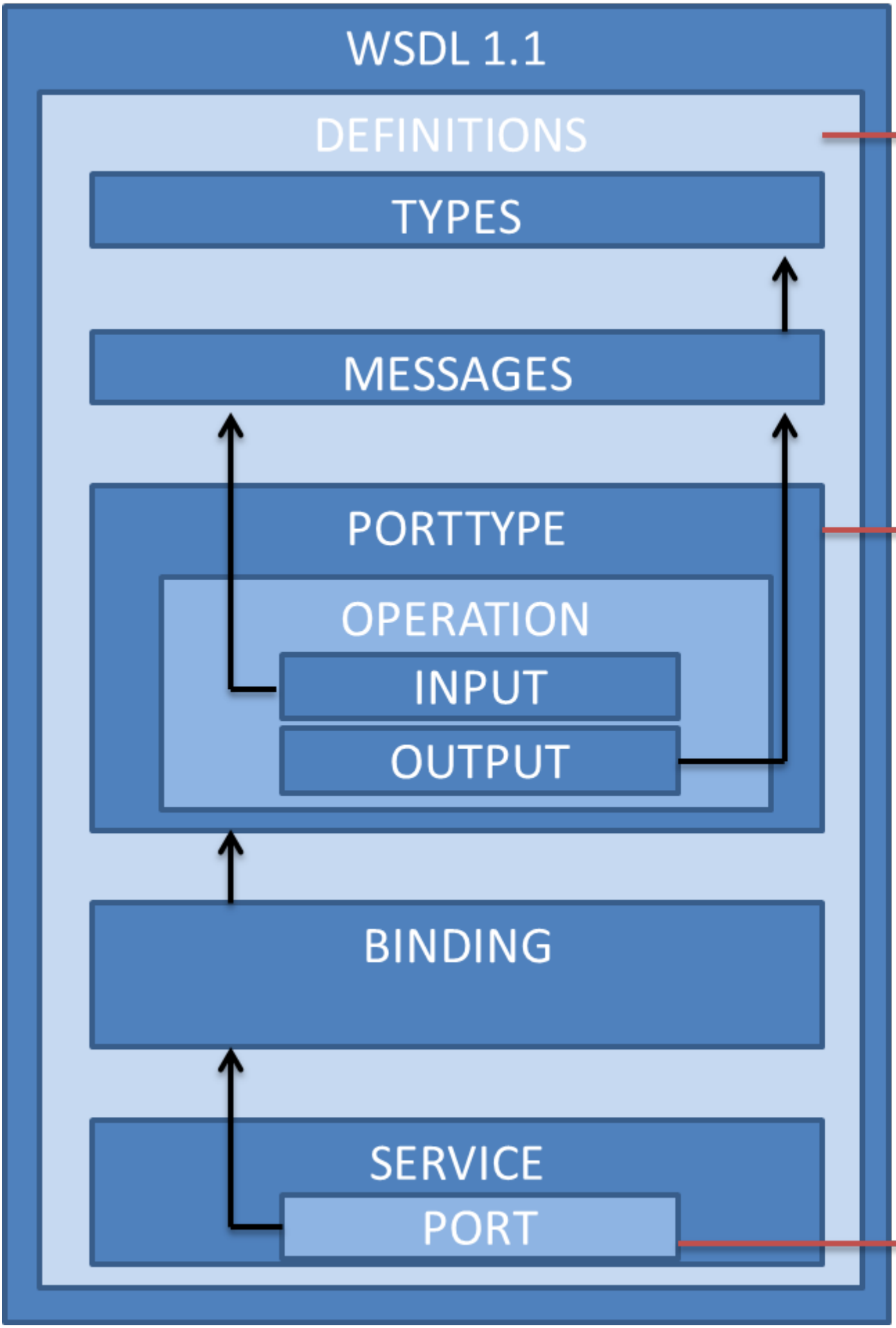
The same calculator can be described in WSDL 2.0 like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<wsdl:description xmlns:xs="http://www.w3.org/2001/XMLSchema"
                  xmlns:wsdl="http://www.w3.org/ns/wsdl"
                  xmlns:soap="http://www.w3.org/ns/wsdl/soap"
                  xmlns:calc="http://example.org/calculator"
                  xmlns:tns="http://example.org/calculatorService"
                  targetNamespace="http://example.org/calculatorService">

    <!--
      Abstract type definitions
    -->

    <wsdl:types>
        <!--
        <xs:schema>
            <xs:import namespace="http://example.org/calculator"
schemaLocation="calc/calculator.xsd" />
        </xs:schema>
        -->
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                   xmlns:tns="http://example.org/calculator"
                   targetNamespace="http://example.org/calculator"
                   elementFormDefault="qualified"
                   attributeFormDefault="qualified">

            <xs:element name="AddValuesRequest" type="tns:AddValuesType" />
            <xs:element name="AddValuesResponse" type="tns:AddValuesResponseType" />

            <xs:complexType name="AddValuesType">
                <xs:sequence>
                    <xs:element name="FirstValue" type="xs:int" minOccurs="1" maxOccurs="1" />
                    <xs:element name="SecondValue" type="xs:int" minOccurs="1" maxOccurs="1"
/>
                </xs:sequence>
            </xs:complexType>

            <xs:complexType name="AddValuesResponseType">
                <xs:sequence minOccurs="1" maxOccurs="1">
                    <xs:element name="Result" type="xs:int" />
                </xs:sequence>
            </xs:complexType>

            <xs:attribute name="Timestamp" type="xs:dateTime" />
            <xs:element name="CalculationFault">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ErrorCode" type="xs:int" />
                        <xs:element name="Reason" type="xs:string" />
                    </xs:sequence>
                    <xs:attribute ref="tns:Timestamp" use="required" />
                </xs:complexType>
            </xs:element>

        </xs:schema>
    </wsdl:types>
```

```
    <!--
        Abstract interface
    -->

    <wsdl:interface name="CalculatorInterface">
        <wsdl:fault name="fault" element="calc:CalculationFault" />
        <wsdl:operation name="AddValues" pattern="http://www.w3.org/ns/wsdl/in-out"
style="http://www.w3.org/ns/wsdl/style/iri" wsdl:safe="true">
            <wsdl:documentation>Adds up passed values and returns the
result</wsdl:documentation>
            <wsdl:input messageLabel="in" element="calc:AddValuesRequest" />
            <wsdl:output messageLabel="out" element="calc:AddValuesResponse" />
            <wsdl:outfault messageLabel="fault" ref="tns:fault" />
        </wsdl:operation>
    </wsdl:interface>

    <!--
        Concrete binding definition
    -->

    <wsdl:binding name="CalculatorBinding" interface="tns:CalculatorInterface"
type="http://www.w3.org/ns/wsdl/soap"
soap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
        <wsdl:operation ref="tns:AddValues" soap:mep="http://www.w3.org/2003/05/soap/mep/soap-
response" />
        <wsdl:fault ref="tns:fault" soap:code="soap:Sender" />
    </wsdl:binding>

    <!--
        Concrete service definition
    -->

    <wsdl:service name="CalculatorService" interface="tns:CalculatorInterface">
        <wsdl:endpoint name="CalculatorEndpoint" binding="tns:CalculatorBinding"
address="http://localhost:8080/services/calculator" />
    </wsdl:service>

</wsdl:description>
```

Differences between WSDL 1.1 and 2.0

A graphical overview on the differences between both versions can be seen in the picture below.


WSDL 1.1
DEFINITIONS
TYPES
MESSAGES
PORTTYPE
OPERATION
INPUT
OUTPUT
BINDING
SERVICE
PORT

(Source)

As can be seen from the image the `message` section got removed which is now contained in the `interface` section. Also, some of the elements got renamed, others have a different syntax but in general both WSDL version do basically the same with version 2.0 requiring a bit less writing overhead compared to 1.1.

Besides the smaller footprint on defining SOAP based services via WSDL 2.0, the newer version also provides capabilities to define REST services though WSDL 2.0 or even WADL are NOT recommended for RESTful services as they contradict the actual idea behind it.

Which style to prefer

The WSDL binding section describes how the service is bound to the SOAP messaging protocol. The sample above used `document` as the binding style, which allows to structure the SOAP body the way we want as long as the resulting output is a valid XML instance. This is the default binding style and often referred to as `Message-Oriented style`.

In contrast to `document` style, `RPC` style request bodies have to contain both the operation name and the set of method parameters. The structure of the XML instance is therefore predefined and can not be changed.

In addition to the binding style the binding section also defines a translation model for bindings to SOAP messages in the name of `literal` or `encoded`. The difference between the two is, that `literal` model has to conform to a user-defined XSD structure, which can be used to validate the requests and responses, while the `encoded` model has to use XSD datatypes like `xs:integer` or `xs:string` but in exchange therefore has not to conform to any user-defined schema. This however makes it harder to validate the message body or transform the message via XSLT to an other format.

The combination of the binding style with the use-model allows for actually 4 different message outcomes. A 5th entry is added to the list which is commonly used (though not really part of the standard).

- RPC / encoded
- RPC / literal
- Document / encoded
- Document / literal
- Document /literal (wrapped)

  In document/literal style of messaging, there exists a pattern which is known as wrapped-document/literal. This is just a pattern, and is not a part of WSDL specification. This pattern has a mention in JSR 224 (JAX-WS: Java API for XML based web services). (Source)

The section below gives an overview on the differences regarding WSDL or schema declaration and their impact on the resulting SOAP message format on changing either binding style or use model definitions.

RPC / encoded

WSDL:

```
...
<wsdl:message name="AddValues">
  <wsdl:part name="FirstValue" type="xsd:int" />
  <wsdl:part name="SecondValue" type="xsd:int" />
</wsdl:message>
<wsdl:message name="AddValuesResponse">
  <wsdl:part name="Result" type="xsd:int" />
</wsdl:message>

<wsdl:portType name="CalculatorEndpoint">
  <wsdl:operation="AddValues">
    <wsdl:input message="AddValues" />
    <wsdl:output message="AddValuesResponse" />
  </wsdl:operation>
</wsdl:portType>

<!-- binding style set to 'RPC' and use to 'encoded' -->
...
```

SOAP Request

```
<soap:envelope>
  <soap:body>
    <AddValues>
      <FirstValue xsi:type="xsd:int">1</FirstValue>
      <SecondValue xsi:type="xsd:int">2</SecondValue>
    </AddValues>
  </soap:body>
</soap:envelope>
```

SOAP Response

```
<soap:envelope>
  <soap:body>
    <AddValuesResponse>
      <Result xsi:type="xsd:int">3</Result>
    </AddValuesResponse>
  </soap:body>
</soap:envelope>
```

Pros

- straightforward WSDL
- Name of operation and elements available in request and response

Cons

- Explicit declaration of XSI types
- Hard to validate
- Not WS-I compliant

RPC / literal

WSDL:

```
...
<wsdl:message name="AddValues">
  <wsdl:part name="FirstValue" type="xsd:int" />
  <wsdl:part name="SecondValue" type="xsd:int" />
</wsdl:message>
<wsdl:message name="AddValuesResponse">
  <wsdl:part name="Result" type="xsd:int" />
</wsdl:message>

<wsdl:portType name="CalculatorEndpoint">
  <wsdl:operation="AddValues">
    <wsdl:input message="AddValues" />
    <wsdl:output message="AddValuesResponse" />
  </wsdl:operation>
</wsdl:portType>

<!-- binding style set to 'RPC' and use to 'literal' -->
...
```

SOAP Request

```
<soap:envelope>
  <soap:body>
    <AddValues>
      <FirstValue>1</FirstValue>
      <SecondValue>2</SecondValue>
    </AddValues>
  </soap:body>
</soap:envelope>
```

SOAP Response

```
<soap:envelope>
  <soap:body>
    <AddValuesResult>
      <Result>3</Result>
    </AddValuesResult>
  </soap:body>
</soap:envelope>
```

Pros

- straightforward WSDL
- Name of operation and elements available in request and response
- No XSI type specification needed
- WS-I compliant

Cons

- Hard to validate

Document / encoded

Does not make any sense therefore omitted.

Document / literal

WSDL:

```
...
<types>
  <schema>
    <element name="FirstValueElement" type="xsd:int" />
    <element name="SecondValueElement" type="xsd:int" />
    <element name="ResultValueElement" type="xsd:int" />
  </schema>
</types>

<wsdl:message name="AddValues">
  <wsdl:part name="FirstValue" element="FirstValueElement" />
  <wsdl:part name="SecondValue" element="SecondValueElement" />
</wsdl:message>
<wsdl:message name="AddValuesResponse">
  <wsdl:part name="Result" element="ResultValueElement" />
</wsdl:message>

<wsdl:portType name="CalculatorEndpoint">
  <wsdl:operation="AddValues">
    <wsdl:input message="AddValues" />
    <wsdl:output message="AddValuesResponse" />
  </wsdl:operation>
</wsdl:portType>

<!-- binding style set to 'Document' and use to 'literal' -->
...
```

SOAP Request

```
<soap:envelope>
  <soap:body>
    <FirstValueElement>1</FirstValueElement>
    <SecondValueElement>2</SecondValueElement>
  </soap:body>
</soap:envelope>
```

SOAP Response

```
<soap:envelope>
  <soap:body>
    <ResultElement>3</ResultElement>
  </soap:body>
</soap:envelope>
```

Pros

- No XSI type encoding
- Able to validate body
- WS-I compliant with restrictions

Cons

- WSDL is more complicated due to the additional XSD definition
- Operation name is lost
- WS-I only allows one child in SOAP body

Document / literal (wrapped)

WSDL:

```
...
<types>
  <schema>
    <element name="AddValues">
      <complexType>
        <sequence>
          <element name="FirstValue" type="xsd:int" />
          <element name="SecondValue" type="xsd:int" />
        </sequence>
      </complexType>
    </element>
    <element name="AddValuesResponse">
      <complexType>
        <sequence>
          <element name="ResultValue" type="xsd:int" />
        </sequence>
      </complexType>
    </element>
  </schema>
</types>

<wsdl:message name="AddValues">
  <wsdl:part name="in" element="AddValues" />
</wsdl:message>
<wsdl:message name="AddValuesResponse">
  <wsdl:part name="out" element="AddValuesResponse" />
</wsdl:message>

<wsdl:portType name="CalculatorEndpoint">
  <wsdl:operation="AddValues">
    <wsdl:input message="AddValues" />
    <wsdl:output message="AddValuesResponse" />
  </wsdl:operation>
</wsdl:portType>

<!-- binding style set to 'Document' and use to 'literal' -->
...
```

SOAP Request

```
<soap:envelope>
  <soap:body>
    <AddValues>
      <FirstValue>1</FirstValue>
      <SecondValue>2</SecondValue>
    </AddValues>
  </soap:body>
```

```
</soap:envelope>
```

SOAP Response

```
<soap:envelope>
  <soap:body>
    <AddValuesResponse>
      <Result>3</Result>
    </AddValuesResponse>
  </soap:body>
</soap:envelope>
```

Pros

- No XSI type encoding
- Able to validate body
- Name of operation and elements available in request and response
- WS-I compliant

Cons

- WSDL is more complicated due to the additional XSD definition

UDDI

`Universal Description, Discovery and Integration (UDDI)` is an open industry initiative crated in 2000 which acts as XML based yellow-pages registry for web services which helps finding services that solve specific tasks. In order to find an appropriate service, a service needs to be registered first with a Web Service Registry such as the UDDI.

UDDI works on SOAP message exchange and provides access to WSDL documents which can be used to invoke the actual web service.

The UDDI provides lookup criteria like

- business identifier
- business name
- business location
- business category
- service type by name
- discovery URLs

However, a big disadvantage of current UDDI is that it only allows to use one single criteria within a search statement. Certain implementors therefore modularized their UDDI implementations to allow for queries spawning multiple UDDIs simultaneously and then aggregate the returned results.

In practice, however, UDDI is not used that often. Some are even saying that UDDI is dead since IBM, Microsoft and SAP shut down their UDDI services in 2005.

Further notes:

SOAP/WSDL provide a wide range of tooling support and also allow to dynamically generate stub-classes for both clients and servers as the type of messages and data exchanged is well defined through the embedded or linked XSD schemata.

While WSDL 2.0 has less overhead of defining web services, certain languages still have not adopted the new standard yet. I.e. in Java popular tools like `wsimport` (from Oracle/Sun) or `wsdl2java` (from Apache CXF) are not able to handle WSDL 2.0 descriptions properly. Therefore, for compatibility reasons it is still recommended to use WSDL 1.1. If you need to develop a WSDL 2.0 based SOAP service in Java have a look at `wsdl2java`from Apache Axis2 project.

More popular nowadays, however, are either HTTP based API services, which mix HTTP operation invocations with clean human-understandable URIs and certain customizations to the protocol in order to get their job done, REST based services, which fully comply to the actual recommendations, or own byte-level protocols, like i.e. OFTP2.

SOAP is still useful nowadays if you can't map your task directly to resources, like HTTP/REST base services do, as the task to fulfill represents naturally an action or has to define certain transaction semantics. Also if you do not have the resources to define or implement your own protocol you are probably better of using SOAP. SOAP is especially useful if you have to deal with orchestration as the WSDL description in combination with UDDI allows to combine services dynamically.

Java Client for Weather service open source webservice avaliable on http://www.webserviceX.NET

```
 package com.test.ws.example;

import javax.xml.soap.MessageFactory;
import javax.xml.soap.MimeHeaders;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPConnection;
import javax.xml.soap.SOAPConnectionFactory;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;

/*
 * WSDL url :  http://www.webservicex.com/globalweather.asmx?WSDL
 * Endpoint URL: http://www.webservicex.com/globalweather.asmx   */

   public class WSClient {

  public static void main(String args[]) {
```

```
    try {
        SOAPConnectionFactory soapConnectionFactory = SOAPConnectionFactory.newInstance();
        SOAPConnection soapConnection = soapConnectionFactory.createConnection();

        // Generate SOAP request XML

        MessageFactory messageFactory = MessageFactory.newInstance();
        SOAPMessage soapMessage = messageFactory.createMessage();
        MimeHeaders header = soapMessage.getMimeHeaders();
        header.setHeader("SOAPAction", "http://www.webserviceX.NET/GetCitiesByCountry");
        SOAPPart soapPart = soapMessage.getSOAPPart();
        SOAPEnvelope envelope = soapPart.getEnvelope();
        envelope.addNamespaceDeclaration("web", "http://www.webserviceX.NET");
        SOAPBody soapBody = envelope.getBody();
        SOAPElement soapBodyElem = soapBody.addChildElement("GetCitiesByCountry", "web");
        SOAPElement soapBodyElem1 = soapBodyElem.addChildElement("CountryName", "web");
        soapBodyElem1.addTextNode("INDIA");
        soapMessage.saveChanges();
        soapMessage.writeTo(System.out);


        // Call webservice endpint
        String url = "http://www.webservicex.com/globalweather.asmx";
        SOAPMessage soapResponse = soapConnection.call(soapMessage, url);
        Source sourceContent = soapResponse.getSOAPPart().getContent();

        // Print SOAP response
        TransformerFactory transformerFactory = TransformerFactory.newInstance();
        Transformer transformer = transformerFactory.newTransformer();
        System.out.println("Response SOAP Message \n");
        StreamResult result = new StreamResult(System.out);
        transformer.transform(sourceContent, result);
        soapConnection.close();

    } catch (Exception e) {

        e.printStackTrace();
    }
}
```

}

Creating a Simple Web Service and Clients with JAX-WS (Document / literal)

This is project directory.

1. **A service endpoint interface**

First we will create a service endpoint interface. The javax.jws.WebService `@WebService` annotation defines the class as a web service endpoint.

```
import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.Style;
import javax.jws.soap.SOAPBinding.Use;

// Service Interface with customize targetNamespace
@WebService(targetNamespace = "http://hello-soap/ws")
@SOAPBinding(style = Style.DOCUMENT, use=Use.LITERAL) //optional
public interface HelloSoap {

    @WebMethod String getHelloSoap(String name);

}
```

2. **Service endpoint implementation (SEI)**

Next we will create service endpoint implementation. We will create an explicit interface by adding the `endpointInterface` element to the `@WebService` annotation in the implementation class. Here are some set of rules 28.1.1 Requirements of a JAX-WS Endpoint that JAX-WS endpoints must follow. The getHelloSoap method returns a greeting to the client with the name passed to it.

```
import javax.jws.WebService;

// Customized Service Implementation (portName,serviceName,targetNamespace are optional)

@WebService(portName = "HelloSoapPort", serviceName = "HelloSoapService",
endpointInterface = "com.wonderland.hellosoap.HelloSoap", targetNamespace = "http://hello-
soap/ws")
public class HelloSoapImpl implements HelloSoap {

    @Override
    public String getHelloSoap(String name) {
        return "[JAX-WS] Hello : " + name;
    }

}
```

3. **Web service endpoint publisher**

```
import javax.xml.ws.Endpoint;

public class HelloSoapPublisher {

    public static void main(String[] args) {
        // creating web service endpoint publisher
        Endpoint.publish("http://localhost:9000/ws/hello-soap", new HelloSoapImpl());
    }
```

```
}
```

4. Next steps, we will run `HelloSoapPublisher.java` as java application. Then we will view the WSDL file by requesting the URL `http://localhost:9000/ws/hello-soap?wsdl` in a web browser.

   http://localhost:9000/ws/hello-soap?wsdl

If XML data format is display at the web browser, then we are ready to go next step.

Note:
If you get some kind of error message, maybe you need to use `wsgen` tool to generate necessary JAX-WS portable artifacts. We are not covered about `wsgen` tool here.

5. **Web Service Client**

Final step, we will create a client that accesses our published service.

```
import java.net.URL;

import javax.xml.namespace.QName;
import javax.xml.ws.Service;

public class HelloSoapClient {

    public static void main(String[] args) throws Exception {

        // create wsdl url
```

```
        URL wsdlDocumentUrl = new URL("http://localhost:8000/ws/hello-soap?wsdl");
        QName helloSoapService = new QName("http://hello-soap/ws", "HelloSoapService");
        // create web service
        Service service = Service.create(wsdlDocumentUrl, helloSoapService);
        // get object of pointed service port
        HelloSoap helloSoap = service.getPort(HelloSoap.class);
        // testing request
        System.out.println(helloSoap.getHelloSoap("Soap "));

    }

}
```

Output: `[JAX-WS] Hello : Soap`

Note: Port number changed to `8000` in our web service client. The reason here is, I used Eclipse IDE, build-in `TCP/IP monitor` tool to trace messages (More Info: How to trace SOAP message in Eclipse IDE). For functional testing purpose try SoapUI | Functional Testing for SOAP and REST APIs.

Read Getting started with soap online: https://riptutorial.com/soap/topic/5375/getting-started-with-soap

Chapter 2: Consuming SOAP Web Service

Introduction

This section should provide details of all the possible ways to consume a SOAP web service.

Parameters

Parameter	Details
CountryName	String such as UK

Examples

Without creating Stub or Java files

```
public String getCitiesByCountry(String countryName) throws MalformedURLException, IOException
{

    //Code to make a webservice HTTP request
    String responseString = "";
    String outputString = "";
    String wsURL = "http://www.webservicex.com/globalweather.asmx";// Endpoint of the
webservice to be consumed
    URL url = new URL(wsURL);
    URLConnection connection = url.openConnection();
    HttpURLConnection httpConn = (HttpURLConnection)connection;
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    String xmlInput =
        "<soap:Envelope xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
xmlns:xsd=\"http://www.w3.org/2001/XMLSchema\"
xmlns:soap=\"http://schemas.xmlsoap.org/soap/envelope/\">
          <soap:Body>
            <GetCitiesByCountry xmlns=\"http://www.webserviceX.NET\">
              <CountryName>" + countryName + "</CountryName>
            </GetCitiesByCountry>
          </soap:Body>
        </soap:Envelope>"; //entire SOAP Request

    byte[] buffer = new byte[xmlInput.length()];
    buffer = xmlInput.getBytes();
    bout.write(buffer);
    byte[] b = bout.toByteArray();
    String SOAPAction = "http://www.webserviceX.NET/GetCitiesByCountry"; // SOAP action of the
webservice to be consumed
    // Set the appropriate HTTP parameters.
    httpConn.setRequestProperty("Content-Length",
    String.valueOf(b.length));
    httpConn.setRequestProperty("Content-Type", "text/xml; charset=utf-8");
    httpConn.setRequestProperty("SOAPAction", SOAPAction);
    httpConn.setRequestMethod("POST");
```

```
    httpConn.setDoOutput(true);
    httpConn.setDoInput(true);
    OutputStream out = httpConn.getOutputStream();
    //Write the content of the request to the outputstream of the HTTP Connection.
    out.write(b);
    out.close();
    //Ready with sending the request.

    //Read the response.
    InputStreamReader isr = null;
    if (httpConn.getResponseCode() == 200) {
      isr = new InputStreamReader(httpConn.getInputStream());
    } else {
      isr = new InputStreamReader(httpConn.getErrorStream());
    }

    BufferedReader in = new BufferedReader(isr);

    //Write the SOAP message response to a String.
    while ((responseString = in.readLine()) != null) {
        outputString = outputString + responseString;
    }
    //Parse the String output to a org.w3c.dom.Document and be able to reach every node with
the org.w3c.dom API.
    Document document = parseXmlFile(outputString);
    NodeList nodeLst = document.getElementsByTagName("GetCitiesByCountryResult"); // TagName
of the element to be retrieved
    String elementValue = nodeLst.item(0).getTextContent();
    System.out.println(elementValue);

    return elementValue;
}

public Document parseXmlFile(String in) {
    try {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        InputSource is = new InputSource(new StringReader(in));
        return db.parse(is);
    } catch (ParserConfigurationException e) {
        throw new RuntimeException(e);
    } catch (SAXException e) {
        throw new RuntimeException(e);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Read Consuming SOAP Web Service online: https://riptutorial.com/soap/topic/8292/consuming-soap-web-service

Credits

S. No	Chapters	Contributors
1	Getting started with soap	Alice, Community, Ray, Roman Vottner, ssanrao
2	Consuming SOAP Web Service	RAS