

# .Net Training

Xml Web Services

# Agenda

---

- XML
- XML in .Net
- XML Serialization
- Web Services Overview
- Web Services in .Net
- Web Services Standards

# What is XML?

---

- eXtensible Markup Language
- HTML describes format
- XML describes DATA
- XML follows certain rules
  - A Tag is an item enclosed in <> (e.g. <xml>)
  - There must be an end tag for each beginning tag (e.g. <xml></xml>)
  - If the item is a singleton it must have the end tag built in (e.g. <xml />)
  - There must be a single parent (root) node for each XML document

# The Parts of XML

---

- XML can be divided in the following components:
  - Uniform Resource Identifiers
  - XML Basics
  - XML Schemas
  - XML Namespaces
  - XML Attributes

# Uniform Resource Identifiers

---

- A.k.a. URI
- A way to describe a unique item over the Internet
- Described in detail by RFC1630
- A URI has the form:
  - `<scheme>:<scheme-specific-part>`
  - When the scheme-specific-part contains slashes ('/'), those slashes indicate some hierarchical structure within the path
- The best known URI is the URL (Uniform Resource Locator)
- Less known is the URN (Uniform Resource Names)

# Uniform Resource Names

---

- Unlike a URL, a URN does not resolve to a unique, physical location
- URNs serve as persistent resource identifiers
- They allow other collections of identifiers from one namespace to be mapped into URN-space
- In general, it looks like this:
  - `<URN> ::= "urn:" <NID> ":" <NSS>`
- A URN uses the string "urn:" to identify the scheme
- NID specifies the Namespace ID and NSS specifies the Namespace Specific String
- When interpreting URNs we look to the NID to tell us how to interpret the NSS
- When reading or creating a URN, the initial construct "urn:" <NID> is case-insensitive

# XML Basics

- XML is a **hierarchical** language
- But XML is also a pretty free-form language
- This piece of XML on the left could accidentally be represented as the XML on the right
- And the machine would think you have a Book object, a Bokk object and a Picture object

```
<?xml version="1.0">
<Library>
  <Book>
    <Title>Green Eggs and Ham</Title>
    <Author>Dr. Seuss</Author>
  </Book>
  <Book>
    <Title>Windows Shell Programming</Title>
    <Author>Scott Seely</Author>
  </Book>
  <Picture>
    <Title>American Gothic</Title>
    <Artist>Grant Wood</Artist>
  </Picture>
</Library>
```

```
<?xml version="1.0">
<Library>
  <Book>
    <Title>Green Eggs and Ham</Title>
    <Author>Dr. Seuss</Author>
  </Book>
  <Bokk>
    <Title>Windows Shell Programming</Title>
    <Author>Scott Seely</Author>
  </Bokk>
  <Picture>
    <Title>American Gothic</Title>
    <Artist>Grant Wood</Artist>
  </Picture>
</Library>
```

# XML Schemas

---

- To prevent this problem XML Schemas were invented
- The XML Schema defines the expected structure of the document, so the programmer or program can know what to expect and what not to expect
- If “bad” XML is received, the program/programmer can take appropriate action

# XML Schemas

---

- There are many types of XML Schemas
- The most commonly known are
  - DTD (Document Type Declaration) (older spec)
  - XDR (External Data Representation) (older Sun standard – RFC 1014)
  - XSD (Xml Schema Definition) (newer spec)
- The later is the preferred version by Microsoft and most modern vendors

- The previous XML could be described in XSD as

```
<schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace=  
  "http://www.MySite.com/LibrarySchema.xml"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  <complexType name="Book">  
    <element type="Title"></element>  
    <element type="Author"></element>  
    <element type="Copyright"></element>  
  </complexType>  
  <simpleType name="Title" xsi:type="string">  
  </simpleType>  
  <simpleType name="Author" xsi:type="string">  
  </simpleType>  
  <simpleType name="Copyright" xsi:type="integer">  
  </simpleType>  
</schema>
```

# XSD Schemas

---

- To use it, simply reference the targetNamespace in the document like so:

```
<myLibrary:Library xmlns:myLibrary="http://www.MySite.com/LibrarySchema.xml">
  <myLibrary:Book>
    <myLibrary:Title>Green Eggs and Ham</myLibrary:Title>
    <myLibrary:Author>Dr. Seuss</myLibrary:Author>
    <myLibrary:Copyright>1957</myLibrary:Copyright>
  </myLibrary:Book>
  <myLibrary:Book>
    <myLibrary:Title>Windows Shell Programming</myLibrary:Title>
    <myLibrary:Author>Scott Seely</myLibrary:Author>
    <myLibrary:Copyright>2000</myLibrary:Copyright>
  </myLibrary:Book>
</myLibrary:Library>
```

# XML Namespaces

---

- Namespaces define a set of unique names within a given context
- A namespace can use any URN as long as that URN is unique
- For example, the preceding schema defined the namespace myLibrary
- The schema contained in the file LibrarySchema.xml is in the same directory as the source page and uniquely identifies the namespace

# XML Namespaces

---

- What does a namespace *do* for us?
- It allows us to create multiple elements with the same name (such as `postOffice:address` and `memory:address`)
- Helps the computer tell apart which structure is being referenced

# XML Attributes

- Elements require begin and end tags
- Attributes are contained by the begin tag of an element
- A given element can have one or more elements of the same type
- It can only have one attribute of any given type

```
<Library>  
  <Book title="Windows Shell  
Programming">  
    <Author>Scott Seely</Author>  
  </Book>  
</Library>
```

```
<Library>  
  <Book title="Windows Shell  
Programming"  
author="Scott Seely" />  
</Library>
```

- To navigate through an XML document a query language was developed : XPATH
- To get to the Book nodes: `//Library/Book`
- To get the Author nodes: `//Library/Book/Author`

```
<?xml version="1.0">
<Library>
  <Book>
    <Title>Green Eggs and Ham</Title>
    <Author>Dr. Seuss</Author>
  </Book>
  <Book>
    <Title>Windows Shell
Programming</Title>
    <Author>Scott Seely</Author>
  </Book>
  <Picture>
    <Title>American Gothic</Title>
    <Artist>Grant Wood</Artist>
  </Picture>
</Library>
```

- .Net exposes several libraries to assist in handling XML code
- Most parts of the .Net Framework make extensive use of XML
  - ASP.Net
  - Configuration files
  - ADO.Net
  - XAML (WPF)
  - Etc.

# XML Libraries in .Net

---

- Most XML libraries can be found in the System.Xml namespace and sub-namespaces
- [System.Xml.XmlReader](#) – is an abstract class that provides non-cached, forward-only, read-only access to XML data. It conforms to the W3C Extensible Markup Language (XML) 1.0 and the Namespaces in XML recommendations. The implementation class is the XmlTextReader. The current node refers to the node on which the reader is positioned. The reader is advanced using any of the read methods and properties return the value of the current node
- [System.Xml.XmlWriter](#) – The XmlWriter class writes XML data to a stream, file, TextReader, or string. The **XmlWriter** class writes XML data to a stream, file, TextReader, or string. It provides a means of creating well-formed XML data in a forward-only, write-only, non-cached manner. The **XmlWriter** class supports the W3C XML 1.0 and Namespaces in XML recommendations. Similarly, the implementation class is the XmlTextWriter

# XML Libraries in .Net

---

- [System.Xml.XmlDocument](#) – provides an object that can navigate through an XML document via the DOM (Document Object Model) – it allows you to read in XML in a string or file (or IO Stream) and navigate through it via the Nodes collections, or write XML to a file or (IO Stream)
- [System.Xml.XPath.XPathDocument](#) – Provides a fast, read-only, in-memory representation of an XML document using the XPath data model
- [System.Xml.XPath.XPathNavigator](#) – Provides a cursor model for navigating and editing XML information items as instances of the XQuery 1.0 and XPath 2.0 Data Model. It is created from a class that implements the [IXPathNavigable](#) interface such as the [XPathDocument](#) and [XmlDocument](#) classes. **XPathNavigator** objects created by **XPathDocument** objects are read-only while objects created by **XmlDocument** objects can be edited

# XML Libraries in .Net

---

- With so many choices, which do I pick?!?!?



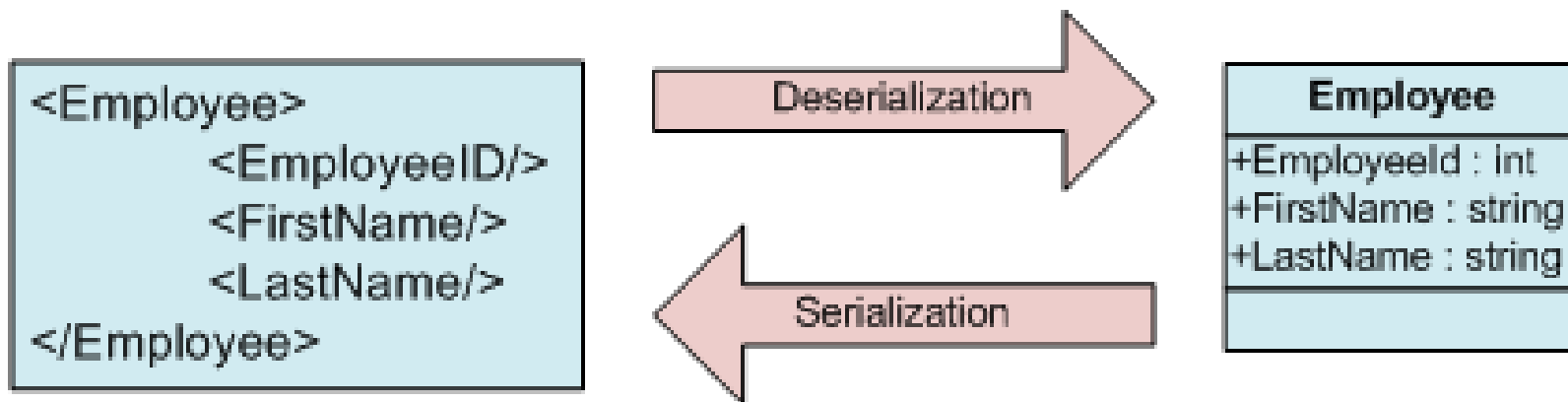
# XML Libraries in .Net

---

- You pick the one most appropriate for the situation at hand
  - The **XPathDocument** class provides a fast, read-only, in-memory representation of an XML document using the XPath data model
  - The **XmlDocument** class provides an editable in-memory representation of an XML document
    - If you just want to read, the XPathDocument is the fastest
    - If you need to edit, you want the XmlDocument
  - If you want to write an XML document from scratch, **XmlWriter** would be the fastest (but also the most painful, by hand – it is better suited for automated tools)
  - The **XmlReader** provides quick access to document elements as well, but doesn't provide easy object access – it reads at the top and stops at the bottom of the document, element by element
    - Both XmlReader and XmlWriter are non-cached (connected) – the file will block until reading/writing is complete

# XML Serialization

- .Net provides an object in the System.Xml.Serialization namespace that can convert XML documents into in-memory objects and vice-versa
- This concept is referred to as XML Serialization



# XML Serialization

---

- The object is called XmlSerializer
  - Deserialization (where “xml” is a readable Stream or XmlReader):

```
XmlSerializer ser = new XmlSerializer(typeof(Employee));  
Employee emp = ser.Deserialize(xml) as Employee;
```
  - Serialization (where “xml” is a writable Stream or XmlWriter):

```
XmlSerializer ser = new XmlSerializer(typeof(Employee));  
Employee emp = new Employee();  
ser.Serialize(xml);
```
  - In order to be serialized to XML a class must be serializable (*certain objects, like DataRow's are not serializable*)
  - The XmlSerializer takes the type of the Xml ROOT element that will need to be serialized/deserialized

# XML Serialization

---

- The XmlSerializer can only serialize/deserialize public properties or fields (variables) – no methods can be serialized/deserialized
- You can control how the XML serializes by using attributes on class to help it along
  - By default all public fields will be serialized at elements (e.g. <FirstName>)
  - To get an attribute (<Employee FirstName="...">) you can decorate the FirstName property with the attribute [XmlAttribute]

# XML Serialization

---

- Why would I want to do this?
- What is it for?
  
- XML Serialization makes it easy to work with XML data – much easier than parsing
- Less prone to mistakes
- Easier to maintain when changes are made

- Downsides of XML Serialization:
  - If the schema is very large it can take a lot of time to serialize/deserialize a document
  - Problems with Inheritance – the xml serializer has issues with inherited classes
  - Problems with Polymorphism – the xml serializer has issues with classes with the same name in different namespaces
    - It ends up tacking the parent name to the child element name (e.g. Order.OrderDetails would end up being serialized as `<Order><OrderOrderDetails/></Order>`)
    - It's possible to control this somewhat with namespaces and attributes

- XML Serialization Demo

- The XSD tool is provided for you via the Visual Studio command line
- It is your XML serialization buddy
- It can help:
  - Infer an XSD schema from an XML document
  - Create an XSD document from classes in a DLL
  - Create classes out of an XSD document
- To see syntax type `xsd /?` At VS command line

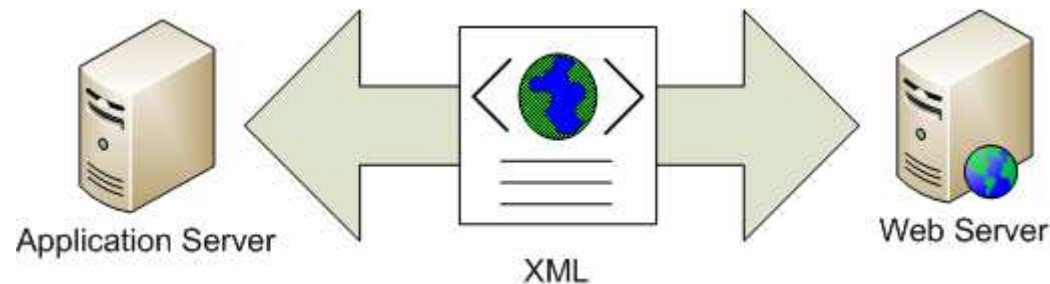
# Web Services Overview

---

- In loose terms a “Web Service” is simply a web page that returns an XML document, instead of an HTML document
- In the early days of Web Services, it became convenient to exchange data formats in XML rather than use COM/CORBA over the Internet
- The first few Web Services were very unstructured, which required extensive documentation in order for programmers to interoperate

# Web Services Overview

- The picture below illustrates the basic concept of a Web Service
- An application server (rather than a user) needs data
- It connects to the Web Server that contains the web service it wants to call
- Makes a request and receives an XML document





# Demo

---

- Raw Web Services Demo

# Web Services Overview

---

- Pretty soon it was found that this unstructured approach was very cumbersome
- It took a lot of code to handle and XML parsing was a task no real programmer wanted to be stuck doing for very long...
- Large XML Schemas made for very long documents to read and understand, in order for the developer to figure out how to parse
- So instead, it was decided to let machines do some of the work

# Web Services Overview

---

- Out of that effort a few technologies were borne that attempt to simplify the task
- Instead of unstructured XML the messages would be wrapped in SOAP envelopes
- XSD schemas would describe the objects used in the communication
- And WSDL would bind it all together and allow a developer to programmatically query the structure of the calls
- This is known as [WS-I Basic Profile 1.1](#)

# Web Services Overview

---

- This unfortunately also led to the need for developers to learn all new XML based languages...
- And then Microsoft put it all together in Visual Studio .Net and made so all that complexity is hidden from you...
  - Provided the web service in question supports XSD schemas and WSDL

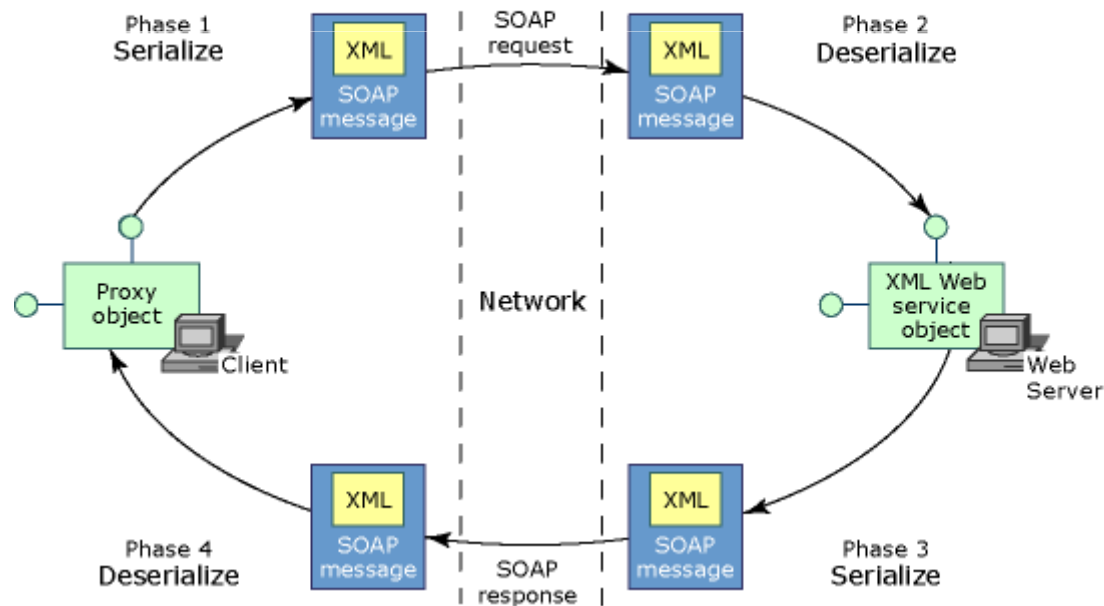
# Web Services in .Net

---

- How does Visual Studio achieve this?
- The “Add Web Reference” Wizard examines the WSDL of the web service in question
- It then generates proxy classes through XML Serialization that mimic the various SOAP calls (through the WSDL Tool)
- What you get are classes that can make the calls and send and receive the custom objects defined in the schema
  - You could do all this by hand also, but why bother?

# Web Services in .Net

- Since Visual Studio creates the proxies for you, you never actually get to touch the XML
- The XML gets created for you by the framework





# Demo

---

- .Net Web Service Demo

# Web Services in .Net

---

- What if you need more control, or to log the XML for debugging purposes?
- What you need is a SoapExtension
- Create a class that inherits from SoapExtension
- Override a few methods
- The ProcessMessage() method allows you to trap the XML stream before and after serialization and deserialization
- To get raw XML you want the BeforeDeserialize and AfterSerialize stages (see previous slide)
- There's a sample on the MSDN documentation

## Web Services in .Net

---

- What if the web service you need to contact does not support WSDL?
- You have to send raw HTTP POST's with the XML in the format supported by the manufacturer of the web service, which could be as raw as just pure XML, XML with a DTD or XDR schema, or sometimes even SOAP, but without WSDL
- In that case you need the System.Net WebRequest and WebResponse classes



# Demo

---

- Raw Web Service client Demo
- .Net Web Service client Demo

- Web Services Description Language
- An XML standard designed to specify the interfaces exposed by a web service
- WSDL encapsulates the schema (XSD) used by the objects involved in the communication
- It describes the services as collections of network endpoints, or ports
- It describes the various calls that can be made, the parameters to be supplied and the format of the results



# WSDL

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:tm="http://microsoft.com/wsdl/mime/textMatching" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:tns="http://tempuri.org/" xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" targetNamespace="http://tempuri.org/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
      <s:element name="HelloWorld">
        <s:complexType />
      </s:element>
      <s:element name="HelloWorldResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="HelloWorldResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="HelloWorldSoapIn">
    <wsdl:part name="parameters" element="tns:HelloWorld" />
  </wsdl:message>
  <wsdl:message name="HelloWorldSoapOut">
    <wsdl:part name="parameters" element="tns:HelloWorldResponse" />
  </wsdl:message>
  <wsdl:portType name="ServiceSoap">
    <wsdl:operation name="HelloWorld">
      <wsdl:input message="tns:HelloWorldSoapIn" />
      <wsdl:output message="tns:HelloWorldSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="ServiceSoap" type="tns:ServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="HelloWorld">
      <soap:operation soapAction="http://tempuri.org/HelloWorld" style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="ServiceSoap12" type="tns:ServiceSoap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="HelloWorld">
      <soap12:operation soapAction="http://tempuri.org/HelloWorld" style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="Service">
    <wsdl:port name="ServiceSoap" binding="tns:ServiceSoap">
      <soap:address location="http://localhost:6245/SampleWebService/Service.asmx" />
    </wsdl:port>
    <wsdl:port name="ServiceSoap12" binding="tns:ServiceSoap12">
      <soap12:address location="http://localhost:6245/SampleWebService/Service.asmx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

- The first part of the document defines all the namespaces and contains the top-level element `wSDL:definitions` which contains all others

```
<?xml version="1.0" encoding="utf-8"?>
<wSDL:definitions
  xmlns:soap="http://schemas.xmlsoap.org/w
  sdl/soap/"
  xmlns:tm="http://microsoft.com/wSDL/mime/te
  xtMatching/"
  xmlns:soapenc="http://schemas.xmlsoap.or
  g/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/w
  sdl/mime/" xmlns:tns="http://tempuri.org/"
  xmlns:s="http://www.w3.org/2001/XMLSche
  ma"
  xmlns:soap12="http://schemas.xmlsoap.org/
  wSDL/soap12/"
  xmlns:http="http://schemas.xmlsoap.org/wsd
  l/http/"
  targetNamespace="http://tempuri.org/"
  xmlns:wSDL="http://schemas.xmlsoap.org/wsd
  l/">
```

- Next comes the `wsdl:types` section
- This contains the XSD schema defining the objects used in the request and response

```
<wsdl:types>
  <s:schema elementFormDefault="qualified"
    targetNamespace="http://tempuri.org/">
    <s:element name="HelloWorld">
      <s:complexType />
    </s:element>
    <s:element name="HelloWorldResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0"
            maxOccurs="1" name="HelloWorldResult"
            type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</wsdl:types>
```

- Then comes the `wSDL:message` section
- This contains a list of all the messages that will can be sent – there's an In and corresponding Out object for each call supported
  - I.e. for a `HelloWorld()` call, there's a `HelloWorldSoapIn` object and a `HelloWorldSoapOut` object

```
<wSDL:message  
  name="HelloWorldSoapIn">  
  <wSDL:part name="parameters"  
    element="tns:HelloWorld" />  
</wSDL:message>  
<wSDL:message  
  name="HelloWorldSoapOut">  
  <wSDL:part name="parameters"  
    element="tns:HelloWorldResp  
onse" />  
</wSDL:message>
```

- The next section defines the “portTypes” – these contain the various operations that can be performed and the in and out object types that were defined in the message section

```
<wsdl:portType
  name="ServiceSoap">
  <wsdl:operation
    name="HelloWorld">
    <wsdl:input
      message="tns:HelloWorldSoa
pIn" />
    <wsdl:output
      message="tns:HelloWorldSoa
pOut" />
    </wsdl:operation>
  </wsdl:portType>
```

- Then come the `wSDL:binding` sections
- The web service will support binding to various flavors of SOAP
- Each one is appropriately described and you can pick which to use depending on which version you can support
  - Needless to say, support the latest version you possibly can

```
<wSDL:binding name="ServiceSoap" type="tns:ServiceSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wSDL:operation name="HelloWorld">
    <soap:operation soapAction="http://tempuri.org/HelloWorld"
      style="document" />
    <wSDL:input>
      <soap:body use="literal" />
    </wSDL:input>
    <wSDL:output>
      <soap:body use="literal" />
    </wSDL:output>
  </wSDL:operation>
</wSDL:binding>
<wSDL:binding name="ServiceSoap12" type="tns:ServiceSoap">
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wSDL:operation name="HelloWorld">
    <soap12:operation soapAction="http://tempuri.org/HelloWorld"
      style="document" />
    <wSDL:input>
      <soap12:body use="literal" />
    </wSDL:input>
    <wSDL:output>
      <soap12:body use="literal" />
    </wSDL:output>
  </wSDL:operation>
</wSDL:binding>
```

- Finally the `wsdl:service` section lists out the various supported ports that you can connect to, as well as the appropriate bindings for each port

```
<wsdl:service name="Service">
  <wsdl:port name="ServiceSoap"
    binding="tns:ServiceSoap">
    <soap:address
      location="http://localhost:80/SampleWebService/Service.asmx" />
    </wsdl:port>
  <wsdl:port name="ServiceSoap12"
    binding="tns:ServiceSoap12">
    <soap12:address
      location="http://localhost:80/SampleWebService/Service.asmx" />
    </wsdl:port>
</wsdl:service>
```

# WSDL Tool

---

- The WSDL tool is similar to the XSD tool
- It is your web services buddy
- It helps:
  - To generate WSDL from existing classes
  - To generate proxy classes from existing WSDL
- To see options run `wSDL.exe /?` from Visual Studio command line

# SOAP

---

- Originally: Simple Object Access Protocol
- Now: **Service Oriented Architecture Protocol**
  - (since the 1.2 standard)
- Is a protocol for exchanging XML-based messages over computer networks, normally using HTTP/HTTPS
- Provides a basic messaging framework that more abstract Web Services layers can build on
  - There are several different types of messaging patterns in SOAP, but by far the most common is the Remote Procedure Call (RPC) pattern, in which one network node (the client) sends a request message to another node (the server), and the server immediately sends a response message to the client

- This is a sample SOAP request message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetails xmlns="http://warehouse.example.com/ws">
      <productID>827635</productID>
    </getProductDetails>
  </soap:Body>
</soap:Envelope>
```

- This is a sample SOAP response message:

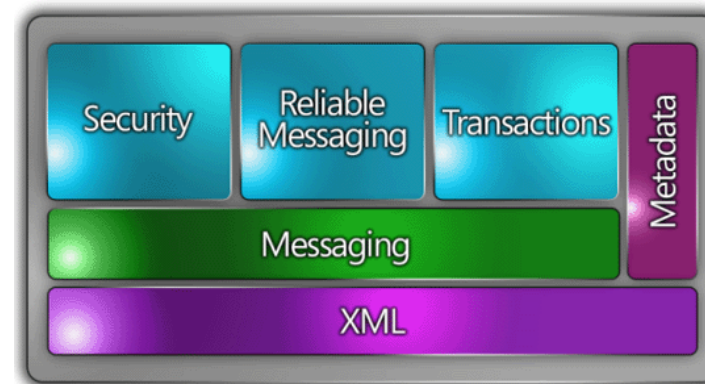
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
      <getProductDetailsResult>
        <productName>Toptimize 3-Piece Set</productName>
        <productID>827635</productID>
        <description>3-Piece luggage set. Black Polyester.</description>
        <price currency="NIS">96.50</price>
        <inStock>true</inStock>
      </getProductDetailsResult>
    </getProductDetailsResponse>
  </soap:Body>
</soap:Envelope>
```

- Advantages
  - SOAP over HTTP travels easily over proxies and firewalls
  - It is platform agnostic and is a convenient medium for data exchange between different platforms
  - SOAP is versatile enough to support different transport protocols. Besides HTTP and HTTPS it also supports SMTP, TCP and SNMP
  - These are advantages over other middleware technology such as DCOM or CORBA which use proprietary binary serialization over TCP/UDP on random ports

- Disadvantages
  - XML is verbose – slower and “fatter” than binary serialization such as DCOM/CORBA/.Net Remoting, etc.
  - Basic Profile 1.1 is limited to synchronous calls – you must rely on polling instead of notification
  - Many SOAP implementations limit the data that can be sent

# Web Services Standards

- Beyond Basic Profile 1.1 there are numerous standards that extend the capabilities of Web Services
- These are termed WS-\*Standards and are defined in the W3C (World Wide Web Consortium) website
- These standards cover different extensibility areas from enhanced security, transport of large binary objects, routing and addressing, notification, etc.



# Web Services Standards

---

- These standards are beyond the scope of this class
- But the .Net Framework adds support to these standards via the various iterations of Microsoft **Web Services Enhancements** (WSE 1.0, 1.1, 2.0 and 3.0) and the more modern **Windows Communication Foundation** (formerly codename Indigo)



# Demo

---

- Make your own .Net Web Service and Client