

# Web Services and Implications in Software Development

Alisson Sol & Ivan Santa Maria Filho

Microsoft Corporation

One Microsoft Way, Redmond, WA 98052

{asol, ivansmf}@microsoft.com

**Abstract:** Web services are processing functionality made available on web servers. Web services established a simple and scalable paradigm to both local and distributed processing, with comprehensive implications in software development.

Unlike many application programming interfaces of the past, which came with an implementation that users could like or not, the web services paradigm just establish a definition language for services that can be used locally or remotely, on any available implementation. Acceptance of the paradigm has been huge on the software industry, due to the easy interoperability of web services.

This tutorial introduces the audience to the concepts of web services, and demonstrates how to go from an idea to the definition of a service on a WSDL file with subsequent implementation on a web server, and use from any possible client. Besides a survey of currently available technology, implementation and design considerations are presented, along with design guidelines and research themes related to Software Engineering.

## 1 Introduction

The next section describes concepts related to XML, a technology for definition of structured and self-describing documents, providing a convenient and extensible information representation mechanism for heterogeneous environments. Following, a brief presentation of SOAP is provided, defining the key concepts of this XML-based protocol for both local and remote calls. After that, the basic concepts of WSDL schemas and UDDI are presented, being followed by a description of sample web service applications. Guidelines on web services design are then discussed, followed by research topics and considerations on the importance of standards.

## 2 XML (eXtensible Markup Language)

XML is an extensible markup language with origins in text management systems. While still represented using text files, it evolved from decorating texts to a serializing format capable of representing complex type systems and structured data types [1]. The documents are graph structured with a tree like representation and contain a prolog – identifying the text encoding, XML version and external or internal entities – and a single document element that contains all serialized data.

XML is organized around tags, used to identify and delimit the information. XML is considered extensible primarily because it allows the definition of new tags, like in the following example:

```
<myOwnTag>element</myOwnTag>
```

Tags are surrounded by the delimiters ‘<’ and ‘>’ and the closing tag is preceded by a ‘/’.

The tags plus the information contained between them are called an element. Elements may contain other elements, creating the mentioned tree-structured format. Contained elements are called child elements and are also elements on their own. The following example shows an element color and two child nodes, green and blue:

```
<color><green></green><blue></blue></color>
```

Both “green” and “blue” elements are empty. They do not contain child nodes or any other information. Empty nodes have a shortcut representation collapsing tags, like <blue/>.

Elements may have attributes that are always a pair formed by a name and a value. Attribute values are represented between quotes and may be single values or a white space delimited list of values, like the following example:

```
<myOwnTag attribute="value1 value2 value3">element</myOwnTag>
```

A document that follows the syntax described until now is a **well-formed** document. But to store or exchange information, it is also necessary to define how it is represented. The basic mechanism to define structure and type for XML documents is the DTD. Well-formed documents that adhere to a DTD or schema - as explained later - are considered **valid** documents.

## 2.1 DTD (Document Type Definition)

DTDs are like a glossary of all valid tags for a particular document type plus structure information. A DTD can be defined as a separate document from the XML instance it describes and linked to it, or embedded in it. They are written in a specific language with hierarchical organization, starting with the document type declaration and then the document element and child nodes declaration. The language used to define DTDs is closer to an extended BNF than to XML [2].

An element declaration follows the format <!ELEMENT elementName childNodes>. Here “elementName” is the name of the element and “childNodes” is a list of nodes that can be child of this element like, for example, the following declaration:

```
<!ELEMENT vehicles (car|bike)*>
<!ELEMENT car (model, color, year)>
<!ELEMENT bike (#PCDATA, model, color, year)>
```

It defines a “vehicles” element that will contain zero or more “car” or “bike” elements. That is expressed using a grouping operator (“()”), a cardinality wildcard (“\*”), and a logical OR operator (“|”). The valid cardinality wildcards are “?” (element is optional), “+” (repeat one or more times), and “\*” (repeat zero or more times). The logical “or” operator (“|”) indicates that either “car” or “bike” should appear, but not both. It is possible to combine grouping and cardinality masks to define complex sequences of elements.

It also introduces a pre-defined type: #PCDATA or parsed character data. When #PCDATA is present it must be the first child node in the declaration. It indicates that free text can appear mixed with elements in what is called mixed-mode content. Modern XML documents will avoid mixed mode content. It is also possible to replace the list of child nodes by the keywords EMPTY, meaning no text or child nodes, or ANY, meaning that any other element can appear as child nodes.

Attribute declarations follow the form `<!ATTLIST elementName attributeName Type defaultValue>`. The “elementName” is the name of the element that contains the attribute(s). DTD defines only a few “Type” values, like CDATA (string), ID (unique name), IDREF (reference to another element ID), enumeration of values and some more [2]. The allowed “defaultValue” values are #REQUIRED (always appears), #IMPLIED (optionally appears), #FIXED plus a value (has always the same value, appearing or not), #DEFAULT plus a value or just a value (if doesn’t appear the default value is implied). Not all combinations are valid and, for instance, defining an attribute as ID with #FIXED default value is not allowed. The following fragment declares a list of attributes from a book element:

```
<!ATTLIST book
  ISBN ID #REQUIRED
  pubDate CDATA #IMPLIED
  pageCount CDATA #IMPLIED
  authors IDREFS #IMPLIED>
```

In this example the ISBN is used to uniquely identify the book and the authors attribute is a list of references to author data.

A DTD ENTITY declares information that may be reused in many places, but always after its declaration. It may represent text, a part of type declaration, or a reference to an external file containing either text or binary data. Entities are internal, external or parameter entities. Internal entities define shortcuts for frequently typed text or that is expected to change, like &lt; (‘<’) and &amp; (‘&’). Custom internal entities are declared using the format `<!ENTITY myEntity "some value">`. In this case every occurrence of “&myEntity;” is replaced by “some value” (without quotes), just like a macro substitution.

External entities refer to another file that may contain either text or binary data. If the file contains text then the file contents are inserted at the point where the entity is referenced. To insert the contents of the file “notice.xml” wherever the entity “&myEntity;” is found declare it as follows: `<!ENTITY myEntity SYSTEM "notice.xml">`. When the entity points to external binary data it is possible to use the NDATA keyword after the location name plus a type identifier. The type identifier has no meaning for DTDs; it is designed to hint the receiving application about the entity format. The third type of entities - parameter entities [2,3] - is beyond the scope of this brief introduction.

While instrumental to the early adoption of XML the DTDs have its shortcomings. The data type and cardinality information is limited. DTDs are closed constructs and do not allow sharing of structures across documents. It is defined using its own language, different from the document instances. There is no easy way to extend its syntax. XML parsers have no easy access to DTD declarations, reducing its utility to document validation and human programmer information. There is also no mechanism to dynamically define DTDs.

## 2.2 Namespaces

The widespread adoption of XML generated another problem. With more and more DTDs defined by different vendors and applications it is increasingly common to find something useful already defined somewhere else. Unfortunately there is no simple way to share or extend definitions using DTDs. Copying contents is not always possible, especially when multiple elements with the same name are declared.

A key concept to avoid naming collisions is the namespace, used to qualify elements and attributes when mixing vocabularies. Their declaration follows the `xmlns:namespaceId="URI"` format, like for example:

```
xmlns:ns1="http://www.microsoft.com/samples/sample.dtd"
xmlns:ns2="urn:microsoft-urn-sample"
```

Elements and attributes are associated with namespaces using its identifier or through the namespace scope resolution. To use the identifier just prefix the element name with it like, for example, `<ns1:someElement someAttribute="someValue"/>`. The element “someElement” belongs to the namespace “ns1”. Association by scope resolution is slightly more complicated. Look at the following XML fragment:

```
<el1 xmlns="http://www.microsoft.com/samples/sample.dtd">
  <el2 at1="val1">some text</el2>
  <el3>
    <el4 xmlns="http://www.w3.org/TR/REC/REC-html40">
      <tr>
        <td>text</td><td>more text</td>
      </tr>
    </el4>
  </el3>
  <el5>This is some other text</el5>
</el1>
```

The elements “el1”, “el2”, “el3” and “el5” belong to the first namespace while the elements “el4”, “tr” and “td” belong to the second namespace. It is possible to mix elements and attributes from multiple namespaces [2,3]. Besides the mentioned limitations, DTDs also do not support namespaces. It is necessary thus to find another mechanism to define type and structure.

## 2.3 XML Schemas

There are multiple efforts to define a XML schema language, like the XML-Data (XDR) [4], Document Content Description (DCD) [5], Schema for Object-Oriented XML (SOX) [6], Document Definition Markup Language (DDML or XSchema) [7] and the more ambitious Resource Description Framework (RDF) [8]. This tutorial focus on the W3C work on a language for XML Schemas Definition (XSD).

The XML Schema Working Group published two working drafts, one about structures [9] and another about data types [10,11]. This tutorial will present a brief overview of both. The first document defines how to independently create data types, element types and how to associate them. The second document defines a type system and an extension mechanism. Type definitions are independent of their lexical representation, allowing multiple data notations to

represent the same information like, for example, “3”, “3.0” and “03.00” may represent the same thing.

XML Schemas are defined in XML removing the need to learn another language. It uses a set of reserved tags to declare its types and structures. A schema consists of a preamble plus zero or more type declarations. The preamble has the following syntax:

```
<?xml version="1.0"?>
<schema targetNS="http://myServer/mySchema.xsd" version="1.0"
xmlns="http://www.w3.org/1999/XMLSchema">
. . .
</schema>
```

The version number and the “xmlns” value will change for every new version of the XML Schema specification. The sample schema preamble shows that the schema resides in the server “myServer” and is called “mySchema.xsd”. The default namespace declaration (xmlns=“http://www.w3.org/1999/XMLSchema”) points to the location of the “XML Schemas: Structures” specification [9], which defines a closed model schema. It means that documents conforming to it will be completely defined by the schema and must not have any other content. Schemas may also describe open models, wherein other elements and attributes can exist within an element without being declared, given certain conditions are met [11].

### 2.3.1 Type definitions

Simple types are declared using the “datatype” element. It contains a name and a source attributes. The source attribute value is the name of a base type from which the new type is derived. The “datatype” element may also have a series of child nodes called facets. Facets are restrictions over the data type value domain like, for example, numerical bounds. Each type defined by the XML Schemas has a specific set of facets that include their lexical representation. A set of primitive data types including string, boolean, numerical and time related types and available facets is defined at [10]. The new type facets must be adequate to the base type from which it derives. Typically, constraining facets are specified for a new type by providing specific values for the constraining facets of the base type, like in the following example:

```
<datatype name="smallValue" source="integer">
  <minExclusive value="0"/>
  <maxInclusive value="1000"/>
</datatype>
```

The “integer” type has constraining facets denoting both bounds named “minInclusive”, “minExclusive”, “maxInclusive” and “maxExclusive” that can be added to the new derived type “smallValue” and refine its value space.

### 2.3.2 Complex Types

Complex types are declared using the type element. It has child nodes that declare attributes and child elements, or references to named model groups. Model group is a “group” element containing element and attribute declarations as child nodes. If the group element has a “name” attribute (an ID) then it is called named model group and can be referenced elsewhere in the schema.

Elements are declared using the “element” keyword. An element contains “name” and “content” attributes plus an optional set of child nodes. The “content” attribute regulates whether undeclared child elements can appear under this element in a XML document instance. The allowed values are “unconstrained” (any element, declared or not), “empty” (none) and “mixed” (declared elements and character data).

A “group” element allows grouping of definitions as child nodes and the cardinality is expressed through the “minOccurs” and “maxOccurs” facets. The logical “or” is represented by an attribute named “order” whose allowed values are “seq” and “choice”. The first means that the exact sequence of child elements must appear and the later means that exactly one of the elements must appear.

It is possible not only to change the type value domain using facets, but also to restrict the structure. Suppressing an element or attribute is just a matter of adding the “maxOccurs” facet with - or changing it to - value zero.

A rewrite of the “vehicles” example as a XML schema would be:

```
<datatype name="modelYear" source="integer">
  <minInclusive value="1998" />
  <maxInclusive value="2002" />
</datatype>
<group name="modelDetail" order="seq">
  <element type="model" minOccurs="1" maxOccurs="1" />
  <element type="color" minOccurs="1" maxOccurs="1" />
  <element type="year" minOccurs="1" maxOccurs="1" />
</group>
<element name="car">
  <group ref="modelDetail" />
</element>
<element name="bike">
  <group ref="modelDetail" />
</element>
<element name="vehicles">
  <type>
    <group order="seq" minOccurs="0" maxOccurs="*">
      <group order="choice">
        <element type="car" />
        <element type="bike" />
      </group>
    </group>
  </type>
</element>
```

This rewrite enforces type to all descriptions, including a valid range for the “year” element. Note that both “car” and “bike” reference the “modelDetail” named group.

Attributes are declared using the “attribute” element. Minimally it will contain the “name” of the attribute and “type” information. It may also contain the “default” and “fixed” attributes that are similar to the DTD’s keywords IMPLIED and FIXED. A group of attributes is declared using the “attributeGroup” element. This element will contain a “name” attribute and zero or more attribute declarations as child nodes.

### 2.3.3 Extensibility

XML Schemas have the concept of derivation and composition for types. Deriving a type can extend or restrict a base type. Extension will add new information or change previous information. Restriction will impose limits to the previous information. For example, to enumerate all valid “car” colors it is possible to declare the following schema fragment (including the namespace from the XML Schemas specification [10]):

```
<element name="color">
  <xs:simpleType>
    <xs:restriction base="string">
      <xs:enumeration value="green" />
      <xs:enumeration value="white" />
      <xs:enumeration value="silver" />
    </xs:restriction>
  </xs:simpleType>
</element>
```

Types may control derivation from themselves as well as their appearance in instance documents through the use of three attributes: “abstract”, “exact” and “final”. If “abstract” has the value “true”, no instance of the declared type can appear in an instance document. If “exact” has the value “true”, no derived type may appear in an instance document in its place. Only the type declared may be used. If “final” is given the value “true”, then no further derivation of the type is permitted.

The composition mechanism allows combining schemas and namespaces provided that external namespaces are referenced by an “import” element. This element has a “namespace” attribute whose value is the URI of the imported schema. An optional “schemaLocation” attribute can point to the actual schema file. Once the namespace is imported, it is possible to use types from it within the importing schema. An imported schema still considered an external resource. Importing a schema is equivalent to link parts of it rather than copying them to the importing schema. A XML parser will have to retrieve the imported schema to validate a document instance.

It is also possible to include schemas using the “include” element. This element has a required attribute “schemaLocation” whose value contains the included schema URI. All declared types and structures from the included schema will become part of the including schema. That will happen only if the “schemaLocation” value actually resolves to a schema, and that the included schema has a “targetNamespace” attribute whose value is identical to the including schema’s “targetNamespace” value.

## 3 SOAP – Simple Object Access Protocol

The influence of XML shouldn’t and hasn’t been only in the representation of documents for information exchange. As soon as any technology succeeds in solving problems for data representation, it soon gets applied to process definition and execution (and vice-versa). A perennial problem for remote procedure calls (and consequently distributed processing) has been the reusable definition of the remote functions and the transmission of parameters at runtime in heterogeneous environments. Applying XML in the solution of such problems was a natural idea, resulting in the creation of protocols like XML-RPC [12] and SOAP [13].

### 3.1 SOAP Message

A SOAP message is a XML document containing a mandatory envelope node, an optional header, and a mandatory body. The following would be an example of SOAP message:

```
<?xml version="1.0" encoding="utf-8"?>
<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>
    <AuthenticateUser xmlns="http://tempuri.org/">
      <user>string</user>
      <password>string</password>
    </AuthenticateUser>
  </soap:Body>
</soap:Envelope>
```

This SOAP message is requesting the execution of function “AuthenticateUser”, which has two string parameters: “user” and “password”.

The previous example didn’t include a header, which is an optional element that may appear as the first child of the SOAP envelope. Quoting the SOAP specification, a header element “provides a flexible mechanism for extending a message in a decentralized and modular way without prior knowledge between the communicating parties”. For the previous example, suppose an execution hint may be provided, which would specify the server where the authentication function should look for credentials. One possible way to achieve such functionality would be to include a header element with the additional information.

```
<soap:Header>
  <t:Server xmlns:t="http://something/" soap:mustUnderstand="0">
    string
  </t:Server>
</soap:Header>
```

One of the standard attributes defined in the SOAP specification is used in the above example. The “mustUnderstand” attribute may be used to specify that those receiving the SOAP request must consider the change in semantics due to the presence of the optional element; otherwise the request should fail (which is not the case in this example, since the attribute is set to “0”).

### 3.2 SOAP Bindings

While the previous section briefly presented the syntax and semantics of the SOAP message, one basic task still remain to be performed when using this message for a remote procedure call: transmitting the message to an execution server. The SOAP specification defines such “bindings”, the most used being HTTP. The main reason is that the HTTP protocol, already used for the web pages requests, is usually enabled in most firewalls, which are applications filtering requests from outside the domain of a network (be it a small company local area network, a large University of even an entire country connection to the Internet).

An entire HTTP request using the SOAP protocol, for the previously presented AuthenticateUser(user,password) function, would be:

```

POST /WebService/sbes2002/sbes2002.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/AuthenticateUser"

<?xml version="1.0" encoding="utf-8"?>
<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>
    <AuthenticateUser xmlns="http://tempuri.org/">
      <user>string</user>
      <password>string</password>
    </AuthenticateUser>
  </soap:Body>
</soap:Envelope>

```

And a successful response to the request would be similar to:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<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>
    <AuthenticateUserResponse xmlns="http://tempuri.org/">
      <AuthenticateUserResult>boolean</AuthenticateUserResult>
    </AuthenticateUserResponse>
  </soap:Body>
</soapEnvelope>

```

The SOAP protocol also defines how the server should answer in case of an error, by sending back “fault messages” containing the error information. A particularly useful feature of using the HTTP binding is that an answer will most likely be always available. Even if the server has no SOAP implementation installed, some HTTP error code will be returned to the caller.

### **3.3 SOAP Considerations**

Most of the criticism SOAP receives is performance related, by comparing the load of the message exchange for SOAP calls with the binary data transfer of other “external data representations”. However, as almost any widely used protocol, the main problem for SOAP today is that it already has several versions and it is now being augmented using several proposed “extensions”, which may compromise the interoperability of implementations.

The SOAP protocol also doesn’t address several topics. Left to the discretion of each SOAP implementation are details of real object activation and destruction (if the implementation is object-oriented at all), security, authentication of calls, error processing, etc.

## 4 Web Services Concepts

While XML provides a mechanism for definition of abstract and portable data types, and SOAP defines a remote procedure call protocol, the definition of an API (Application Programming Interface) for “Web Services” needs a language to allow the description of the functionality available in a particular server. That is the role of WSDL (Web Services Description Language) [14].

### 4.1 WSDL Schema

WSDL files are XML files following the WSDL schema. Each file contains a series of definitions, describing both logical and concrete information about the service. The following is a simplified WSDL file describing a Web Service able to accept one simple operation, which is the same previously used `AuthenticateUser(user, password)` of the SOAP examples.

```
<definitions xmlns:prefix="http://namespace-uri"...>
  <types>
    <s:schema elementFormDefault="qualified" targetNamespace="urn:">
      <s:element name="AuthenticateUser">
        <s:complexType>
          <s:sequence>
            <s:element name="user" type="s:string" .../>
            ...
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </types>
  <message name="AuthenticateUserSoapIn">
    <part name="parameters" element="s0:AuthenticateUser" />
  </message>
  <portType name="sbes2002Soap">
    <operation name="AuthenticateUser">
      <input message="s0:AuthenticateUserSoapIn" />
      <output message="s0:AuthenticateUserSoapOut" />
    </operation>
  </portType>
  <binding name="sbes2002Soap" type="s0:sbes2002Soap">
    <soap:binding transport=".../soap/http"...>
    <operation name="AuthenticateUser">
      <soap:operation soapAction="http://.../AuthenticateUser"...>
    </operation>
  </binding>
  <service name="sbes2002">
    <port name="sbes2002Soap" binding="s0:sbes2002Soap">
      <soap:address location="http://.../sbes2002/sbes2002.asmx" />
    </port>
  </service>
</definitions>
```

The sections defining types, messages and portTypes are usually referred as containing the “abstract definitions” for a Web Service. Meanwhile, the sections for binding, service and port elements are referred as “concrete definitions”. To ease reuse and better separate concerns, the information for the abstract sections may be in a file by itself, imported in the main WSDL document.

Each WSDL file may contain more than one service definition. Each service is a set of ports. A port contains the specification of an address for real invocation of a service operation, and refers to the binding section for information about operation details. Each binding element contains the information about how to physically transfer parameters for operation execution, including protocol information, encoding, etc.

The operations described by binding elements have already been logically defined in portType elements. Each portType is a logical group of operations. Those operations have as parameters some named messages, which are definitions that logically group several “message parts” (some authors refer to messages as complex structures of the XML types previously defined). The message parts are elements of a certain type, which may be basic XML types or complex types composed from those, and previously defined in the same file or imported (as any of the other “abstract sections”).

## **4.2 UDDI Specification**

Any new software technology demand tools and methodologies (notation and processes) for successful adoption. Object-oriented development didn't succeed until there were many options for languages, design notations, and an entire generation of tools supporting from the analysis to debugging of object-oriented applications. The same is needed for Web Services.

The basic need for those implementing or using Web Services is a mechanism to publish the availability of a service in a directory that allows searches from web service users. Addressing this need was the reason for the UDDI (Universal Description, Discovery, and Integration) specification [15]. UDDI itself is a series of web services, described by WSDL files that can be found in the specification section of the UDDI.org site.

While a complete explanation of UDDI standard is beyond the scope of this paper, some basic concepts should be understood by all those seeking the publication and use of web services:

- **Technical Models (tModels):** these are abstract definitions of services (an example of tModel is a WSDL file, but there are others). The tModels concept allows a particular sector, like the bank industry, to create an abstract service definition that any bank may implement. Later, entries in a web services directory may be better organized if directly linked to the tModel being implemented.
- **Business Entity:** an institution that is really implementing a Web Service and making it available in the directory (also referred as “provider”).
- **Service:** the available implementation of a defined web service.

As in any catalog, entries in the UDDI databases link tModels to business entities providing the services. Such entries may be organized in several ways (by tModel, business entity, hierarchies of categories, service “providers”, etc.).

## **5 Web Service Applications**

One problem that application developers face when adopting a new API is the long time needed to learn the newly available functionality. That is the case with every new generation of operating systems, languages, tools, etc. Also, each new generation of APIs has an increasingly difficult task of convincing decision makers of the advantages of expending resources to migrate from legacy technology to usually still unproven implementations. Use of

web services in small projects, mostly for investigation purposes, or as an integration technology among heterogeneous systems is preceding widespread adoption in larger and critical applications. While this in part derives from risk management, it also reflects the lack of established artifacts (methodologies, tools, etc.) supporting web services development.

## **5.1 Web Service Use**

Several “providers” have made available services for simple and atomic operations. Examples are:

- Web search, like the Google Web APIs [16];
- Catalog search, like the Amazon.com Web Services [17];
- Map rendering, driving directions and distance calculation, like the MapPoint.Net Web Services [18].

The sort of functionality implemented in services like those listed above has been the driving factor toward the interest on web services. Implementing such services may not be particularly complex, but requires computational power, data gathering, classification and storage beyond the resources of most organizations.

On the other side, users of such web services easily achieve a level of functionality that would be impossible to implement with their own resources. As an example, using a web service to get the current temperature based on ZIP code requires only the design and implementation of an user interface to get the parameter, and then creating a message request following the proper schema [19]. The segment defining the input parameter would be:

```
<message name="getTempRequest">  
  <part name="zipcode" type="xsd:string" />  
</message>
```

After embedding a valid request in the binding envelope and posting to some service port, the application would wait for the XML response; parse the document and present results to the user (or an error explanation).

## **5.2 Web Service Implementation**

For those implementing web services, given an already designed WSDL file, there are several options regarding environment (operating system, HTTP server, SOAP implementation, programming language, etc.), and then the development tools. Going from WSDL definitions to source code may be tedious and time consuming if done manually. “Web Services”-aware development tools provide assistance in the generation of code skeletons from WSDL definition for most major programming languages.

Coding the classes implementing the operations corresponding to the definitions in a WSDL file may be a complex task. However, designing the “interfaces” in the WSDL file that completely and efficiently implement the functionality required in the user scenarios, and doing so in a reusable and extensible way is what may decide if web services will remain relevant. Such “Web Service Patterns” should achieve widespread use if WSDL, in any of its versions or some future evolution, is to become to software design something similar to what hardware description languages achieved years ago [20].

### **5.3 Complete Use Cases**

The previous sections considered “application development” as the simple use of a web service operation in an atomic way by client-side code or the implementation of such operation by developers in the “service providers”. However, the most interesting and useful “web services”-based applications will deal with far more complex scenarios.

Some of such scenarios will be focused in using web services as a mechanism for EAI (Enterprise Application Integration). As an example, one may find a University integrating the payroll system with the academic personnel timekeeping system. Each system may allow access to its functionality through web services, allowing data entry automation, or the query of information for the consumption of the other system.

It is expected however that most web services will be available in the World Wide Web. That would maximize economical return for the service providers while, at the same time, create complex challenges for those designing the service descriptions. Scenarios demanding computational power and specialized technology, like optimal (or semi-optimal) cargo ship loading, are natural candidates to a “pay per use” business model.

The most interesting use cases require the “Orchestration” of web services, whereby a sequence of operations in distinct services should be properly coordinated. This is a major challenge for business systems designers intending to use web services. Integrating a heterogeneous banking system require the proper combination of operations in WSDL definitions to guarantee that money is never taken from one account but never arrives at a destination. WSDL itself isn’t enough for the definition of such scenarios, and several process definition languages are being proposed [21].

## **6 Web Services Design**

As with any other software technology, the challenges in Web Service engineering include performance, reliability, availability, usability, scalability, interoperability, maintainability, privacy and security. The life cycle includes requirement gathering, analysis, design and implementation, through integration, testing and evaluation, deployment and maintenance.

Despite its similarities with traditional software development, web services have a set of challenges of their own. We claim that four factors will greatly determine how successful a web service will be: data storage, trust relationship (security), performance and ontology.

### **6.1 Data Storage**

When defining the data storage model, one can divide application in three models: online only, client only and mixed storage. In the first case all information is stored in a server somewhere in the Internet and no processing happens in the client when it is offline. The second model stores all the data locally, transferring information when some processing is needed. The third model allows offline processing that can be enriched when the client gets online. The great success factor is to define the right balance between online and offline storage and, of course, a synchronization mechanism.

## **6.2 Security**

The information storage also touches the trust problem. When information is stored online the application trusts a remote server with frequently sensitive information like credit card numbers or balance books. Unfortunately trust is not a transitive property and the online service may or may not propagate the information to third parties. Controlling how the information is propagated and how it is protected during transit is not necessarily an easy problem. Authentication services help transferring information to and from secure sources, but can do little - if anything – to control trust propagation.

## **6.3 Performance**

When analyzing performance its obvious two dimensions – response time and throughput – come to mind. Sometimes performance is also related with resource allocation – traditionally processor time and memory consumption and, if much, storage space. Under the web service perspective the critical resource is bandwidth. Defining the most revolutionary service and requiring huge transfers of data will very likely have poor to unbearable performance.

## **6.4 Ontology**

The last but not least important factor is the definition of ontology [22]. While the term ontology may be defined in many different ways, here it refers to what is sometimes called a "structural" ontology: a machine-readable set of definitions to create a taxonomy of classes and subclasses, and relationships between them. Every class and sub-class must have a clear semantic meaning for both client application and web service. When web services become widespread more semantic ontology will be required.

Combining all factors into a single critical path, we claim that web services should be defined based on the desired communication. The first step to define a web service is to define what information is interchanged between client and server. By defining the communication protocol, one defines where the information is stored, how much of it can be trusted to the service, how responsive the service will be and how the throughput will be affected and also define all necessary structure ontology.

## **7 Research topics**

The conception of XML and Web Services technologies could have happened several years ago (and, in some form, that has really happened). However, the implementation of such ideas is only possible today by the privilege of powerful and widespread hardware resources, which continue to follow Moore's law, making available today cheap mobile devices with more memory capacity and processing power than most mainframes of a few decades ago. On top of that, the connectivity provided by local networks and the Internet is a condition without which most of the concepts and applications based on the presented technologies would be unrealizable.

Several topics are still under research to enhance current functionality or provide unavailable features to web services applications, and their development cycle. Topics like notations, design patterns, methodologies, efficient binary data transfer, security, events, orchestration

languages and others still demand research to successfully define the artifacts that will provide the basis for a new generation of web service applications, developed economically and efficiently, accordingly to the principles of Software Engineering.

## 7.1 Standards

The web services paradigm is heavily dependent on interoperability. That may be achieved without protocols being adopted by standard organizations, and having “standard protocols” cannot by itself guarantee interoperability. However, standard protocols provide a basis for discussion without which unproductive discussions based on vendor feature lists take over relevant subjects. Nowadays, there is little debate about the use of the TCP/IP protocol in most networks (what was a hot topic only few years ago), allowing the focus to move toward higher levels of abstraction in systems development.

## 8 Conclusions and Acknowledgements

This tutorial presented a survey of currently available technology related to Web Services, along with implementation and design considerations, and research themes.

The design guidance is that web services should be defined based on the desired communication. The first step to define a web service should be the definition of what information should be interchanged between client and server. By defining the communication protocol, one defines where the information is stored, how much of it can be trusted to the service, how responsive the service will be and how the throughput will be affected and also define all necessary structure ontology. Whenever possible, and in the best interest of interoperability, application architectures should rely in standard protocols.

The authors would like to thank Microsoft Corporation for the support in preparing this paper.

## 9 References

---

1 Essential XML – Beyond Markup. Don Box, Aaron Skonnard, John Lam. Addison-Wesley developer series, 2000.

2 Extensible Markup Language (XML) 1.0 (Second Edition).

<http://www.w3.org/TR/2000/REC-xml-20001006>

3 Professional XML. Jonathan Pinnock, Stephen Mohr, Steve Livingstone, Nik Ozu, Didier Martin, Michael Kay, Brian Loesgen, Peter Stark, Kevin Williams, Bruce Peat. Wrox, 2000. ISBN 1861003110.

4 XML-Data (XDR). <http://www.w3.org/TR/1998/NOTE-XML-data-0105>

5 Document Content Description (DCD). <http://www.w3.org/TR/NOTE-dcd>

6 Schema for Object-Oriented XML 2.0. <http://www.w3.org/TR/NOTE-SOX/>

7 Document Definition Markup Language (DDML) Specification.

<http://www.w3.org/TR/NOTE-ddml>

8 Resource Description Framework (RDF). <http://www.w3.org/RDF/>

- 
- 9 XML Schema Part 1: Structures. <http://www.w3.org/TR/xmlschema-1/>
  - 10 XML Schema Part 2: Datatypes. <http://www.w3.org/TR/xmlschema-2/>
  - 11 XML Schemas Tutorial. Roger L. Costello. <http://www.xfront.com/xml-schema.html>
  - 12 XML-RPC – <http://www.xmlrpc.com/>
  - 13 Simple Object Access Protocol (SOAP) – <http://www.w3.org/TR/SOAP/>
  - 14 Web Services Description Language (WSDL) – <http://www.w3.org/TR/wsdl/>
  - 15 Universal Description, Discovery and Integration (UDDI) – <http://www.uddi.org/>
  - 16 Google Web APIs – <http://www.google.com/apis/>
  - 17 Amazon.com Web Services – <http://www.amazon.com/webservices/>
  - 18 MapPoint.NET – <http://www.microsoft.com/mappoint/net/>
  - 19 XMethods, Inc., Weather – <http://www.xmethods.net/sd/2001/TemperatureService.wsdl>
  - 20 Verilog HDL: A Guide to Digital Design and Synthesis, S, Palnitkar, SunSoft Press, 1996
  - 21 Process Markup Languages – <http://www.ebpml.org/status.htm>
  - 22 Semantic Web. <http://www.w3.org/2001/sw/>