Best Practices in Web Service Style, Data Binding and Validation for use in Data-Centric Scientific Applications

Asif Akram, **David Meredith** and Rob Allan

e-Science Centre, CCLRC Daresbury Laboratory, UK
a.akram@dl.ac.uk, d.j.meredith@dl.ac.uk, r.j.allan@dl.ac.uk

Abstract

We provide a critical evaluation of the different Web Service styles and approaches to data-binding and validation for use in 'data-centric' scientific applications citing examples and recommendations based on our experiences. The current SOAP API's for Java are examined, including the Java API for XML-based remote procedure calls (JAX-RPC) and Document style messaging. We assess the advantages and disadvantages of 'loose' verses 'tight' data binding and outline some best practices for WSDL development. For the most part, we recommend the use of the document/ wrapped style with a 100% XML schema compliant data-model that can be separated from the WSDL definitions. We found that this encouraged collaboration between the different partners involved in the data model design process and assured interoperability. This also leverages the advanced capabilities of XML schema for precisely constraining complex scientific data when compared to RPC and SOAP encoding styles. We further recommend the use of external data binding and validation frameworks which provide greater functionality when compared to those in-built within a SOAP engine. By adhering to these best practices, we identified important variations in client and experimental data requirements across different institutions involved with a typical e-Science project.

1. Introduction

A Service Oriented Architecture is an architectural style whose goal is to achieve loose coupling among interacting software agents (services and clients). A SOA achieves loose coupling by employing two architectural constraints: 1) a small set of well-defined interfaces to all participating software agents and, 2) ensuring the interfaces are universally available to all providers and consumers. In simple terms, a service is a function that is self-contained and immune to the context or state of other services. These services can communicate with each other, either through explicit messages, or by a number of 'master' services that coordinate or aggregate activities together, typically in a workflow. In recent years, Web services have been established as a popular technology for implementing a SOA. The well-defined interface required for services is described in a WSDL (Web Service Description Language) file. Services exposed as Web services can be integrated into complex workflows which may, in a typical e-Science project, span multiple domains and organizations.

2. Binding/ Encoding Styles

The following section examines the different styles of WSDL file, and the resulting format of each SOAP message. It is important to understand that the WSDL binding style dictates the style of SOAP encoding (formatting) of the SOAP message that is transmitted 'over the wire.' This has serious implications upon Web Service interoperability. Collectively, the process of generating SOAP messages according to different styles of WSDL file is referred to as 'SOAP encoding' or 'WSDL binding,' and can either be Remote Procedure Call (RPC), or Document style. These two Web Service styles represent the RPC-centric and Message-centric view points. Most of the documentation however, focuses on the simpler RPC-centric viewpoint and often gives the misleading impression that SOAP and Web services are just another way of doing RPC. Table 1 provides a comparison of the different WSDL binding styles and resulting styles of SOAP encoding. The schema examples are referred to in the text in the following section. Table 2 provides a summary of the main advantages and disadvantages of each approach.

2.1 RPC Encoding/ Binding Style

The following summary examines the key features of the RPC WSDL binding style and the format of a resulting SOAP message. Table 1 illustrates these key points (schema examples are numbered and referred to in the text).

2.1.1 RPC (applies to encoded and literal)

- An RPC style WSDL file contains multiple <part> tags per <message> for each request/ response parameter (10b, 11b).
- Each message <part> tag defines type attributes, not element attributes (message parts are not wrapped by elements as in Document style WSDL files) (10b, 11b).
- The type attribute in each <part> tag can either; a) reference a complex or simple type defined in the <wsdl:types> section, e.g. <part name="x" type="tns:myType">, or b) define a simple type directly, e.g. <part name="x" type="xsd:int"> (10b, 11b respectively).
- An RPC SOAP request wraps the message parameters in an element named after the invoked operation (2a, 12a). This 'operational wrapper element' is defined in the WSDL target namespace. An RPC SOAP response wraps the message parameters in an element named after the invoked operation with 'Response' appended to the element name.
- The difference between RPC encoded and RPC literal styles relates to how the data in the SOAP message is serialised/ formatted when sent 'over the wire'. The abstract parts of the WSDL files are similar (i.e. the <wsdl:types>, <wsdl:message> and <wsdl:portType> elements – refer to Section 6.0). The only significant difference relates to the definition of the <wsdl:binding> element. The binding element dictates how the SOAP message is formatted and how complex data types are represented in the SOAP message.

2.1.2 RPC/ encoded

- An RPC/ encoded WSDL file specifies an encodingStyle attribute nested within the <wsdl:binding>. Although different encoding styles are legal, the most common is SOAP encoding. This encoding style is used to serialise data and complex types in the SOAP message. (http://schemas.xmlsoap.org/soap/encoding).
- The use attribute, which is nested within the <wsdl:binding> has the value "encoded".
- An RPC/ encoded SOAP message has type encoding information for each parameter element. This is overhead and degrades throughput performance (4a, 7a, 8a).
- Complex types are SOAP encoded and are referenced by "href" references using an identifier (3a). The href's refer to "multiref" elements positioned outside the operation wrapping element as direct children of the <soap:Body> (6a). This complicates the message as the <soap:Body> may contain multiple "multiref" elements.
- RPC/ encoded is not WS-I compliant [1] which recommends that the <soap:Body> should only contain a single nested sub element.

2.1.3 RPC/ literal

- RPC/ literal style improves upon the RPC/ encoded style.
- An RPC/ literal WSDL does not specify an encodingStyle attribute.
- The use attribute, which is nested within the <wsdl:binding>, has the value "literal".
- An RPC/literal encoded SOAP message has only one nested child element in the <soap:Body> (12a). This is because all parameter elements become wrapped within the operation element that is defined in the WSDL namespace.
- The type encoding information for each nested parameter element is removed (14a, 15a).
- RPC/ literal is WS-I compliant [1].

The main weakness with the RPC encoding style is its lack of support for the constraint of complex data, and lack of support for data validation. The RPC/ encoded style usually adopts the SOAP encoding specification to serialize complex objects which is far less comprehensive and functional when compared to standard XML Schema. Validation is also problematic in the RPC/ literal style; when an RPC/ literal style SOAP message is constructed, only the operational wrapper element remains fully qualified with the target namespace of the WSDL file, and all other type encoding information is removed from nested sub-elements (this is shown in Table 1). This means all parts/ elements in the SOAP message share the WSDL file namespace and lose their original Schema namespace definitions. Consequently, validation is only possible for limited scenarios, where original schema elements have the same namespace as the WSDL file. For the most part however, validation becomes difficult (if not impossible) since qualification of the operation name comes

from the WSDL definitions, not from the individual schema elements defined in the <wsdl:types> section.

2.2 Document Encoding/ Binding Style

In contrast to RPC, Document style encoding provides greater functionality for the validation of data by using standard XML schema as the encoding format for complex objects and data. The schema defined in the <wsdl:types> section can be embedded or imported (refer to Section 6.1). The following summary examines the key features of the Document WSDL binding style and the format of a resulting SOAP message. Table 1 illustrates these key points (schema examples are numbered and referred to in the text).

2.2.1 Document (applies to literal and wrapped)

- Document style Web services use standard XML schema for the serialisation of XML instance documents and complex data.
- Document style messages do not have type encoding information for any element (23a, 24a), and each element in the soap message is fully qualified by a Schema namespace by direct declaration (22a), or by inheritance from an outer element (30a).
- Document style services leverage the full capability of XML Schema for data validation.

2.2.2 Document/ literal

- Document/ literal messages send request and response parameters to and from operations as direct children of the <soap:Body> (22a, 26a).
- The <soap:Body> can therefore contain many immediate children sub elements (22a, 26a).
- A Document/literal style WSDL file may therefore contain multiple <part> tags per <message> (19b, 20b).
- Each <part> tag in a message can specify either a type or an element attribute, however, for WS-I compliance, it is recommended that only element attributes be defined in <part> tags for Document style WSDL (19b, 20b).
- This means that every simple or complex type parameter should be wrapped as an element and be defined in the <wsdl:types> section (15b, 16b).
- The main disadvantages of the Document/ literal Web Service style include: a) the operation name is removed from the <soap:Body> request which can cause interoperability problems (21a), and b) the <soap:Body> will contain multiple children (22a, 26a) if more than one message part is defined in a request/ response message (19b, 20b).
- Document/ literal is not fully WS-I compliant [1], which recommends that the <soap:Body> should only contain a single nested sub element.

2.2.3 Document/ wrapped

- An improvement on the Document/ literal style is the Document/ wrapped style.
- When writing this style of WSDL, the request and response parameters of a Web Service operation (simple types, complex types and elements) should be 'wrapped' within single all-encompassing request and response elements defined in the <wsdl:types> section (24b - akin to the RPC/ literal style).
- These 'wrapping' elements need to be added to the <wsdl:types> section of the WSDL file (24b).
- The request wrapper element (24b) must have the same name as the Web Service operation to be invoked (this ensures the operation name is always specified in the <soap:Body> request as the first nested element).
- By specifying single elements to wrap all of the request and response parameters, there is only ever a single <part> tag per <message> tag (32b).
- A Document/ literal style WSDL file is fully WS-I compliant [1] because there is only a single nested element in the <soap:Body> (29a).
- Document/ wrapped style messages are therefore very similar to RPC/ literal style messages since both styles produce a single nested element within a <soap:Body>. The only difference is that for Document/ wrapped style, each element is fully qualified with a Schema namespace.

The main advantage of the Document style over the RPC style is the abstraction/ separation of the type system into a 100% XML Schema compliant data model. In doing this, several important advantages related to data binding and validation are further realised which are discussed in the next section.

Table 1 - A Comparison of the Different WSDL Binding Styles and SOAP Encoding

Style	SOAP Request	WSDL
RPC Encoded	1a <soapenv:Body> 2a <getIndex xmlns:="urn:ehtpx-process"> 3a <admin href="#id0"/> 4a <URL xsi:type="xsd:string"> </URL> 5a </getIndex> 6a <multiRef id="id0" ……> 7a <email xsi:type=”xsd:string”> </email> 8a <PN xsi:type=”xsd:string”> </PN> 9a </multiRef> 10a </soapenv:Body>	1b <types> 2b <complexType name="AdminT"> 3b <sequence> 4b <element name="email" type="enc:string"/> 5b <element name="PN" type="enc:string"/> 6b </sequence> 7b </complexType> 8b </types> 9b <wsdl:message name="getIndexRequest"> 10b <wsdl:part name="admin" type="tns:AdminT"/> 11b <wsdl:part name="URL" type="enc:string"/> 12b </wsdl:message>
RPC Literal	11a <soapenv:Body> 12a <getIndex xmlns="urn:ehtpx-process"> 13a <admin xmlns=""> 14a <email> </email> 15a <PN> </PN> 16a </admin> 17a <URL xmlns=""> </URL> 18a </getIndex> 19a <soapenv:Body>	
Doc Literal	20a <soapenv:Body> 21a 22a <admin xmlns="urn:ehtpx-process"> 23a <email xmlns=""> </email> 24a <PN xmlns=""> </PN> 25a </admin> 26a <URL xmlns=""> </URL> 27a </soapenv:Body>	13b <types> 14b <complexType name="AdminT">… </complexType> 15b <element name=”admin” type=”tns:AdminT”> 16b <element name=”URL” type=” enc:string”> 17b </types> 18b <wsdl:message name="getIndexRequest"> 19b <wsdl:part name="in0" element="tns:admin"/> 20b <wsdl:part name="URL" element="enc:string"/> 21b </wsdl:message>
Doc Wrapped	28a <soapenv:Body> 29a <getIndex xmlns=" urn:ehtpx-process"> 30a <admin> 31a <email xmlns=""> </email> 32a <PN xmlns=""> </PN> 33a </admin> 34a <URL xmlns=""> </URL> 35a </getIndex> 36a </soapenv:Body>	22b <types> 23b <complexType name="AdminT"> ...</complexType> 24b <element name ="getIndex"> 25b <complexType> <sequence> 26b <element name ="admin" type =" tns:AdminT“/> 27b <element name =" URL " type =" xsd:string" /> 28b </sequence> </complexType > 29b </element> 30b </types> 31b <wsdl:message name="getIndexRequest"> 32b <wsdl:part name="in0" element="tns:getIndex"/> 33b </wsdl:message>

Table 2 - Advantages and Disadvantages of Each WSDL Binding Style and SOAP Encoding

Style	Advantages	Disadvantages
RPC Encoded	• Simple WSDL • Operation name wraps parameters	• Complex types are sent as multipart references meaning <soap:Body> can have multiple children • Not WS-I compliant • Not interoperable • Type encoding information generated in soap message • Messages can’t be validated • Child elements are not fully qualified
RPC Literal	• Simple WSDL • Operation name wraps parameters • <soap:Body> has only one element • No type encoding information • WS-I compliant	• Difficult to validate message • Sub elements of complex types are not fully qualified.
Doc Literal	• No type encoding information • Messages can be validated • WS-I compliant but with restrictions • Data can be modelled in separate schema	• WSDL file is more complicated • Operation name is missing in soap request which can create interoperability problems • <soap:Body> can have multiple children • WS-I recommends only one child in <soap:Body>
Doc Wrapped	• No type encoding information • Messages can be validated • <soap:Body> has only one element • Operation name wraps parameters • WS-I compliant	• WSDL file is complicated – request and response wrapper elements may have to be added to the <wsdl:types> if original schema element name is not suitable for Web Service operation name.

3. Data Abstraction, Data Binding and Validation

Abstraction of the Web Service type system into a 100% XML Schema compliant data model produces several important advantages;

- *Separation of Roles*
  The type system can be fully abstracted and developed in isolation from the network protocol and communication specific details of the WSDL file. In doing this, the focus becomes centred upon the business/ scientific requirements of the data model. In our experience, this greatly encourages collaboration between the scientists who are involved with the description of scientific data and data model design.
- *Data Model Re-usability*
  Existing XML Schema can be re-used rather than re-designing a new type system for each new Web Service. This helps reduce development efforts, cost and time.
- *Isolation of Changing Components*
  In our experience, the data model is the component that is most subject to change, often in response to changing scientific requirements. Its isolation therefore limits the impact on other Web Service components such as the concrete WSDL file implementation (see Section 6.0).
- *Avoid Dependency on SOAP Namespaces and Encoding Styles*
  Manual modeling of XML Schema may constitute extra effort but this gives the developer the most control and avoids using SOAP framework dependent namespaces and encoding styles. Most of the SOAP frameworks are traditionally RPC-centric and create WSDL based on the RPC/ encoding style which is not WS-I compliant. This also applies to languages other than Java.
- *Full XML Schema Functionality*
  The XML Schema type system leverages the more powerful features of the XML Schema language for description, constraint and validation of complex data (e.g. XSD patterns/ regular expressions, optional elements, enumerations, type restrictions etc). In our experience, this has proven invaluable for the description and constraint of complex scientific data.
- *Pluggable' Binding and Validation Frameworks*
  In most JAX-RPC implementations, XML serialization of a message's encoded parameters is hidden from the developer, who works with objects created automatically from XML data using semi-standardized mapping schemes for the generation of client and server stub/ skeleton classes. Consequently, the developer is both hidden from, and dependent upon the data binding/ validation framework of the SOAP engine. In our experience, SOAP engine data binding frameworks are usually not 100% Schema complaint, and often do not support the more advanced features of XML Schema (e.g. xsd:patterns). We believe that this is a major source of ambiguity and in our experience, this has often been a source of error that is beyond immediate control of the developer. An alternative approach is to use a dedicated, 100% Schema compliant data binding/ validation framework that is independent of the SOAP engine for the construction and validation of Web Service messages and instance documents (e.g. JAXB [2], XMLBeans [3]). Developers still manipulate XML in the familiar format of objects (courtesy of the binding framework), but there is no dependency upon the SOAP engine. In doing this, the more powerful/ functional features of an external binding framework can be levered, and the roles of the SOAP engine and data binding/validation framework become clearly separated into 'data-specific' and 'communication-specific' roles.
- *On-Demand Document Construction and Validation*
  This clear separation of roles means that XML messages/ documents can be constructed and validated at times when the SOAP engine is not required, for example, when constructing messages over an extended period of time (e.g. graphically through a GUI) and especially for the purposes of persistence (e.g. saving validated XML to a database). The separation of the data binding from the SOAP engine is gaining popularity in the next generation of SOAP engines that are now beginning to implement 'pluggable' data bindings (e.g. Axis2 [4] and Xfire [5]).

4. Loose Versus Strong Data Typing

A 'loosely typed' Web Service means the WSDL file does not contain an XML schema in

the type system to define the format of the messages, instead it uses generic data types to express communication messages. Loosely typed services are flexible, allowing Web Service components to be replaced in a 'plug-and-play' fashion. Conversely, a 'strongly typed' Web Service means the WSDL type system strictly dictates the format of the message data. Strongly typed Web services are less flexible, but more robust. Each style influences the chosen approach to data binding and each has its own advantages and disadvantages which are summarized in Table 3.

4.1 Loosely Typed Web services

A loosely typed WSDL interface specifies generic data types for an operation's input and output messages (either "String", "Base64-Encoded", "xsd:any", "xsd:anyType" or "Attachment" types). This approach requires extra negotiation between providers and consumers in order to agree on the format of the data that is expressed by the generic type. Consequently, an understanding of the WSDL alone is usually not sufficient to invoke the service.

- *"String" loose Data Type*

A String variable can be used to encode the actual content of a messages complex data. Consequently, the WSDL file may define simple String input/ output parameters for operations. The String input could be an XML fragment or multiple name value pairs (similar to Query string). In doing this, the implementation has to parse and extract the data from the String before processing. An XML document formatted as a String requires extra coding and decoding to escape XML special characters in the SOAP message which can drastically increase the message size.

- *"any"/ " anyType" loose Data Type*

The WSDL types section may define <xsd:any> or <xsd:anyType> elements. Consequently, any arbitrary XML can be embedded directly into the message which maps to a standard "javax.xml.soap.SOAPElement". Partners receive the actual XML but no contract is specified regarding what the XML data describes. Extraction of information requires an understanding of raw XML manipulation, for example, using the Java SAAJ API. Limited support for "any"/ "anyType" data type from various SOAP Frameworks may result in portability and interoperability issues.

- *"Base64 encoding" loose Data Type*

An XML document can be transmitted as a Base64 encoded string or as raw bytes in the body of a SOAP message. These are standard data types and thus every SOAP engine handles this data in compatible fashion. Indeed, embedding Base64 encoded data and raw bytes in the SOAP body is WS-I compliant.

- *"SOAP Attachment" loose Data Type*

SOAP attachments can be used to send data of any format that cannot be embedded within the SOAP body, such as raw binary files. Sending data in an attachment is efficient because the size of the SOAP body is minimized which enables faster processing (the SOAP message contains only a reference to the data and not the data itself). Additional advantages over other techniques include the ability to handle large documents, multiple attachments can be sent in a single Web Service invocation, and attachments can be compressed-decompressed for efficient network transport.

4.2 Strongly typed Web services

A purely strongly typed WSDL interface defines a complete definition of an operation's input and output messages with XML Schema, with additional constraints on the actual permitted values (i.e. Document style with value constraints). It is important to understand however, that Document style services do not have to be solely strongly typed, as they may combine both strongly typed fields with loose/ generic types where necessary. Strong typing is especially relevant for scientific applications which often require a tight control on message values, such as length of string values, range of numerical values, permitted sequences of values (e.g. organic compounds must have Carbon and Hydrogen in the chemical formula and rotation angle should be between 0 – 360 degrees).

From our experiences related to e-HTPX [6], the best approach involved mixing of the different styles where necessary. For mature Web services, where the required data is established and stable, the use of strong data typing was preferable. For immature Web services where the required data is subject to negotiation/ revision, loose typing was preferable. We often used the loose typing approach during initial developments and prototyping.

Table 3 – Advantages and Disadvantages of Loose versus Strong Data Typing in Web services

Modeling Approach	Advantages	Disadvantages
Loose Type	• Easy to develop • Easy to implement • Minimum changes in WSDL interface • Stable WSDL interface • Flexibility in implementation • Single Web Service implementation may handle multiple types of message • Can be used as Gateway Service routing to actual services based on contents of message	• Requires additional/ manual negotiation between client and service provider to establish the format of data wrapped or expressed in a generic type • This may cause problems regarding maintaining consistent implementations and for client/ service relations • No control on the messages • Prone to message related exceptions due to inconsistencies between the format of sent data, and accepted data format (requires Web Service code to be liberal in what it accepts – this adds extra coding complexity). • Encoding of XML as a string increases the message size due to escaped characters
Strong Type	• Properly defined interfaces • Tight control on the data with value constraints • Message validation • Different possibilities for data validation (pluggable data binding/validation) • Robust (only highly constrained data enters Web Service) • Minimized network overhead • Benefits from richness of XML	• Difficult to develop (requires a working knowledge of XML and WSDL) • Resistive to change in data model

5. Code First or WSDL First

For platform independence and Web Service interoperability, the WSDL interface should not reference or have dependencies upon any technical API other than XML Schema. However, the complexity of XML schema and over-verbosity of WSDL is a major concern in practical development of Web services. As a result, two divergent practices for developing Web services and WSDL have emerged, the ‘code first’ approach (also known as ‘bottom up’) and 'WSDL first' approach (also know as ‘top down’ or ‘contract driven’). The code first approach, which is often implemented in JAX-RPC environments, involves auto-generation of the WSDL file from service implementation classes using tools that leverage reflection and introspection. Alternatively, the WSDL first approach involves writing the original WSDL and XML Schema, and generating service implementation classes from the WSDL file.

5.1 Code First

Advantages

The 'code first' approach is often appealing because of its simplicity. Developers are hidden from the technical details of writing XML and WSDL.

Disadvantages

Generating WSDL files from source code often introduces dependencies upon the implementation language. This is especially apparent when relying upon the SOAP engine to serialize objects into XML, which can lead to interoperability issues across different platforms (e.g. differences in how Java and .NET serialize types that may be value types in one language but are reference objects in the other). WSDL created from source code is less strongly typed than WSDL that is created from the original XML Schema. Indeed, the more powerful features of XML Schema are often not supported by automatic WSDL generators.

5.2 WSDL First

Advantages

Platform and language interoperability issues are prevented, because both the client and server are working from a common set of interoperable XML Schema types. Defining a common platform-independent type system also facilitates separation of roles, whereby client side developers can work in isolation from server side developers. In our experience, this greatly increases productivity and simplifies development, especially for large distributed applications where developers may be geographically separated.

Disadvantages

The developer requires at least a reasonable knowledge of XML Schema and of WSDL.

In our experience, the WSDL first approach is the most suitable for developing robust, interoperable services. However, we also found the code first approach convenient for rapid

prototyping, especially when using loose data typing.

6. WSDL Modularisation

The WSDL specification and the WS-I basic profile recommend the separation of WSDL files into distinct modular components in order to improve re-usability and manageability. These modular components include;

1. XML Schema files.
2. An Abstract WSDL file.
3. A Concrete WSDL file.

6.1 XML Schema Files

Moving the type declarations of a Web Service into their own documents is recommended as data can be modeled in different documents according to namespace requirements. XML Schema declares two elements; <xsd:include> and <xsd:import> which are both valid children of the <wsdl:types> element (<xsd:include> is used when two schema files have the same namespace and <xsd:import> is used to combine schema files from different namespaces). In doing this, complex data types can be created by combining existing documents.

6.2 Abstract WSDL File

The abstract.wsdl file defines what the Web Service does by defining the data types and business operations of the Web Service. The file imports XML schema(s) as immediate children of the <wsdl:types> element, and defines different <wsdl:message> and <wsdl:portType> elements.

6.3 Concrete WSDL File

The concrete.wsdl file defines how and where to invoke a service by defining network protocol and service endpoint location with the <wsdl:binding> and <wsdl:service> elements. The concrete.wsdl file incorporates the abstract.wsdl file using <wsdl:import> or <wsdl:include>. These elements should be the first immediate children of the <wsdl:definitions> element (<wsdl:include> is used when two wsdl files have the same namespace and <wsdl:import> is used to combine wsdl files from different namespaces). This approach greatly improves component re-usability as the same abstract.wsdl file can have multiple service bindings.

7. Conclusions

Developments in the field of Web services have largely focused on the RPC style rather than on Document style messaging. This is apparent in the tools provided by vendors of JAX-RPC/ SOAP engine implementations. RPC style services have serious limitations for the description of data and can lead to interoperability issues. Real applications also require complex data modeling and validation support. For these applications, RPC simple typing and SOAP encoded complex types are inadequate. The RPC style can produce interoperability issues as many automatic WSDL generation tools introduce technical dependencies upon implementation languages. As a result, the RPC style is increasingly being referred to as ‘CORBA with brackets’ in the Web Service community. Loosely typed RPC services provide an alternative approach by encapsulating data within generic types. Loosely typed services are easy and convenient to develop and are suitable in a number of scenarios. However, loose typing introduces a different set of limitations, mainly associated with the additional manual negotiation between consumer and provider to establish the format of encapsulated data. In contrast to RPC, Document style services use 100% standard XML schema as the type system. This facilitates complex data modeling, loose and tight data typing where necessary, and full validation support. Use of a platform independent type system also ensures transport agnosticity, where abstract WSDL definitions can be bound to different transport protocols defined in concrete WSDL files. We also recommend the use of dedicated data binding/ validation frameworks for the construction of XML documents/ messages rather than relying on the SOAP engine. In doing this, the more powerful features of XML Schema can be levered, and the roles of the SOAP engine and data binding/ validation framework become clearly separated. The main disadvantage of the Document style is its increased complexity over RPC; developers require at least a reasonable understanding of XML and WSDL and are required to take the ‘WSDL first’ approach to Web Service design.

References / Resources

[1] WSI; http://www.ws-i.org/
[2] JAXB; http://java.sun.com/webservices/jaxb/
[3] XML Beans; http://xmlbeans.apache.org/
[4] Axis; http://ws.apache.org/axis/
[5] XFire; http://xfire.codehaus.org/
[6] The e-HTPX project; http://www.e-htpx.ac.uk