

The SOA Magazine

Feature Article



Web Service Contract Versioning Fundamentals Part I: Versioning and Compatibility

by Thomas Erl, David Orchard and James Pasley

Published: November 17, 2008 (SOA Magazine Issue XXIII: October-November 2008)

[Download this article as a PDF document.](#)

Rate this document: • [Digg this](#) • [De.Licio.us](#) • [Slashdot](#) • [Technorati](#)

Abstract: Web services have made it possible to build services with fully decoupled contracts that leverage industry standards to establish a vendor-neutral communications framework. Though powerful and flexible, this framework brings with it an on-going responsibility to effectively manage those service contracts as the services continue to evolve over time. One of the most critical aspects of service governance is contract versioning. This first of two articles, comprised of excerpts from the recently released "Web Service Contract Design & Versioning for SOA" book, explores fundamental service contract issues, including compatible and incompatible versioning requirements.

The following article is an excerpt from the new book "Web Service Contract Design and Versioning for SOA" [REF-1] Copyright 2008 Prentice Hall/Pearson PTR and SOA Systems Inc. Note that chapter references were intentionally left in the article, as per requirements from Prentice Hall.

Introduction

After a Web service contract is deployed, consumer programs will naturally begin forming dependencies on it. When we are subsequently forced to make changes to the contract, we need to figure out:

- whether the changes will negatively impact existing (and potentially future) service consumers
- how changes that will and will not impact consumers should be implemented and communicated

These issues result in the need for versioning. Anytime you introduce the concept of versioning into an SOA project, a number of questions will likely be raised, for example:

- What exactly constitutes a new version of a service contract? What's the difference between a major and minor version?
- What do the parts of a version number indicate?
- Will the new version of the contract still work with existing consumers that were designed for the old contract version?
- Will the current version of the contract work with new consumers that may have different data exchange requirements?
- What is the best way to add changes to existing contracts while minimizing the impact on consumers?
- Will we need to host old and new contracts at the same time? If yes, for how long?

The upcoming chapters address these questions and provide a set of options for solving common versioning problems. This chapter begins by covering some basic concepts, terminology, and strategies specific to service contract versioning in preparation for what's ahead.

Basic Concepts and Terminology

The Scope of a Version

As we've established many times over in this book, a Web service contract can be comprised of several individual documents and definitions that are linked and assembled together to form a complete technical interface.

For example, a given Web service contract can consist of:

- one (sometimes more) WSDL definitions
- one (usually more) XML Schema definitions
- some (sometimes no) WS-Policy definitions

Furthermore, each of these definition documents can be shared by other Web service contracts. For example:

- a centralized XML Schema definition will commonly be used by multiple WSDL definitions
- a centralized WS-Policy definition will commonly be applied to multiple WSDL definitions
- an abstract WSDL description can be imported by multiple concrete WSDL descriptions or vice versa

So when we say that we're creating a new version of a contract, what exactly are we referring to?

Of all the different parts of a Web service contract, the part that establishes the fundamental technical interface is the abstract description of the WSDL definition. This represents the core of a Web service contract and is then further extended and detailed through schema definitions, policy definitions, and one or more concrete WSDL descriptions.

When we need to create a new version of a Web service contract, we can therefore assume that there has been a change in the abstract WSDL description or one of the contract documents that relates to the abstract WSDL description. How the different constructs of a WSDL can be versioned is covered in Chapter 21. The Web service contract content commonly subject to change is the XML schema content that provides the types for the abstract description's message definitions. Chapter 22 explores the manner in which the underlying schema definitions for messages can be changed and evolved.

Finally, the one other contract-related technology that can still impose versioning requirements but is less likely to do so simply because it is a less common part of Web service contracts is WS-Policy. How policies in general relate to contract versioning is explained as part of the advanced topics in Chapter 23.

Fine and Coarse-Grained Constraints

Versioning changes are generally related to the increase or reduction of the quantity or granularity of constraints. Therefore, let's briefly recap the meaning of the term constraint granularity in relation to a type definition. Note the highlighted parts of the following example:

```
<xsd:element name="LineItem" type="LineItemType"/>
<xsd:complexType name="LineItemType">
  <xsd:sequence>
    <xsd:element name="productID" type="xsd:string"/>
    <xsd:element name="productName" type="xsd:string"/>
    <xsd:any minOccurs="0" maxOccurs="unbounded"
      namespace="##any" processContents="lax"/>
  </xsd:sequence>
  <xsd:anyAttribute namespace="##any"/>
</xsd:complexType>
```

Example 1: A complexType construct containing fine and coarse-grained constraints.

As indicated by the bolded text, there are elements with specific names and data types that represent parts of the message definition with a fine level of constraint granularity. All of the message instances (the actual XML documents that will be created based on this structure) must conform to these constraints in order to be considered valid (which is why these are considered the absolute "minimum" constraints).

The red text shows the element and attribute wildcards also contained by this complex type. These represent parts of the message definition with an extremely coarse level of constraint granularity in that messages do not need to comply to these parts of the message definition at all.

The use of the terms “fine-grained” and “coarse-grained” is highly subjective. What may be a fine-grained constraint in one contract may not be in another. The point is to understand how these terms can be applied when comparing parts of a message definition or when comparing different message definitions with each other.

Versioning and Compatibility

The number one concern when developing and deploying a new version of a service contract is the impact it will have on other parts of the enterprise that have formed or will form dependencies on it. This measure of impact is directly related to how compatible the new contract version is with the old version and its surroundings in general.

This section establishes the fundamental types of compatibility that relate to the content and design of new contract versions and also tie into the goals and limitations of different versioning strategies that we be introduce at the end of this chapter.

Backwards Compatibility

A new version of a Web service contract that continues to support consumer programs designed to work with the old version, is considered backwards-compatible. From a design perspective, this means that the new contract has not changed in such a way that it can impact existing consumer programs that are already using the contract.

A simple example of a backwards-compatible change is the addition of a new operation to an existing WSDL definition:

```
<definitions name="Purchase Order" targetNamespace=
  "http://actioncon.com/contract/po"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://actioncon.com/contract/po"
  xmlns:po="http://actioncon.com/schema/po">
  ...
  <portType name="ptPurchaseOrder">
    <operation name="opSubmitOrder">
      <input message="tns:msgSubmitOrderRequest"/>
      <output message="tns:msgSubmitOrderResponse"/>
    </operation>
    <operation name="opCheckOrderStatus">
      <input message="tns:msgCheckOrderRequest"/>
      <output message="tns:msgCheckOrderResponse"/>
    </operation>
    <operation name="opChangeOrder">
      <input message="tns:msgChangeOrderRequest"/>
      <output message="tns:msgChangeOrderResponse"/>
    </operation>
    <operation name="opCancelOrder">
      <input message="tns:msgCancelOrderRequest"/>
      <output message="tns:msgCancelOrderResponse"/>
    </operation>
    <operation name="opGetOrder">
      <input message="tns:msgGetOrderRequest"/>
      <output message="tns:msgGetOrderResponse"/>
    </operation>
  </portType>
</definitions>
```

Example 2: The addition of a new operation represents a common backwards-compatible change.

In this example we're borrowing the abstract description of the Purchase Order service that was initially built at the end of Chapter 7. By adding a brand new operation, we are creating a new version of the contract, but this change is backwards-compatible and will not impact any existing consumers.

An example of a change made to a schema for a message definition that is backwards compatible is the addition of an optional element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType" />
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string" />
      <xsd:element name="productName" type="xsd:string" />
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 3: In an XML Schema definition, the addition of an optional element is also considered backwards compatible.

Here we are using a simplified version of the XML Schema definition for the Purchase Order service. The optional available element is added to the LineItemType complex type. This has no impact on existing consumers because they are not required to provide this element in their messages. New consumers or consumer programs redesigned to work with this schema can optionally provide the available element.

Changing any of the existing elements in the previous example from required to optional (by adding the minOccurs="0" setting) would also be considered a backwards-compatible change. When we have control over how we choose to design the next version of a Web service contract, backwards compatibility is generally attainable. However, mandatory changes (such as those imposed by laws or regulations) can often force us to break backwards compatibility.

Note that both the Flexible and Loose versioning strategies explained in Part II of this article series support backwards compatibility.

Forwards Compatibility

When a Web service contract is designed in such a manner so that it can support a range of future consumer programs, it is considered to have an extent of forwards compatibility. This means that the contract can essentially accommodate how consumer programs will evolve over time. The most common means by which forwards compatibility is attempted in message definitions is through the use of wildcards:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType" />
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string" />
      <xsd:element name="productName" type="xsd:string" />
      <xsd:any namespace="##any" processContents="lax"
        minOccurs="0" maxOccurs="unbounded" />
    </xsd:sequence>
    <xsd:anyAttribute namespace="##any" />
  </xsd:complexType>
</xsd:schema>
```

Example 4: To support forwards compatibility within a message definition generally requires the use of XML Schema wildcards.

In this example, the xsd:any and xsd:anyAttribute elements are added to allow for a range of unknown elements and data to be accepted by the Web service contract. In other words, the schema is being designed in advance to accommodate unforeseen changes in the future. Chapter 22 explains in detail how wildcards can be used in support of forwards compatibility.

There are limited options in support of forwards compatibility when it comes to WSDL definitions. These are

discussed at the end of Chapter 21.

It is important to understand that forwards compatibility is by no means an exact science. A service with a forwards-compatible contract will often not be able to process all message content. Its contract is simply designed to accept a broader range of data unknown at the time of its design.

Note: Forwards compatibility forms the basis of the Loose versioning strategy that is explained at the end of this chapter.

Compatible Changes

When we make a change to a Web service contract that does not negatively affect its existing consumers, then the change itself is considered a compatible change.

Note that in this book, the term “compatible change” refers to backwards compatibility by default. When used in reference to forwards compatibility it is further qualified as a forwards-compatible change.

A simple example of a compatible change is when we set the minOccurs attribute of an element from “1” to “0”, effectively turning a required element into an optional one, as shown here:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType" />
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string" />
      <xsd:element name="productName" type="xsd:string"
        minOccurs="0" />
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="0" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 5: The default value of the minOccurs attribute is “1”. Therefore because this attribute was previously absent from the productName element declaration, it was considered a required element. Adding the minOccurs=“0” setting turns it into an optional element, resulting in a compatible change.

This type of change will not impact existing consumer programs that are used to sending the element value to the Web service, nor will it affect future consumers that can be designed to optionally send that element.

Another example of a compatible change was provided earlier in Example 3, when we first added the optional available element declaration. Even though we extended the type with a whole new element, because it is optional it is considered a compatible change.

Here is a list of common compatible changes that we will be discussing in the upcoming chapters:

- adding a new WSDL operation definition and associated message definitions (Chapter 21)
- adding a new WSDL port type definition and associated operation definitions (Chapter 21)
- adding new WSDL binding and service definitions (Chapter 21)
- adding a new optional XML Schema element or attribute declaration to a message definition (Chapter 22)
- reducing the constraint granularity of an XML Schema element or attribute of a message definition type (Chapter 22)
- adding a new XML Schema wildcard to a message definition type (Chapter 22)
- adding a new optional WS-Policy assertion (Chapter 23)
- adding a new WS-Policy alternative (Chapter 23)

We will also be exploring techniques whereby changes that are not normally compatible can still be implemented as compatible changes.

Incompatible Changes

If after a change a contract is no longer compatible with consumers, then it is considered to have received an incompatible change. These are the types of changes that can break an existing contract and therefore impose the most challenges when it comes to versioning.

Note that as with the term “compatible change,” this term also indicates backwards compatibility by default. When referring to incompatible changes that affect forwards compatibility, this term will be qualified as forwards-incompatible change.

Going back to our example, if we set an element’s minOccurs attribute from “0” to any number above zero, then we are introducing an incompatible change:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://actioncon.com/schema/po"
  xmlns="http://actioncon.com/schema/po">
  <xsd:element name="LineItem" type="LineItemType" />
  <xsd:complexType name="LineItemType">
    <xsd:sequence>
      <xsd:element name="productID" type="xsd:string"/>
      <xsd:element name="productName" type="xsd:string"
        minOccurs="3" />
      <xsd:element name="available" type="xsd:boolean"
        minOccurs="3" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

Example 6: Incrementing the minOccurs attribute value of any established element declaration is automatically an incompatible change.

What was formerly an optional element is now required. This will certainly affect existing consumers that are not designed to comply with this new constraint, because adding a new required element introduces a mandatory constraint upon the contract.

Common incompatible changes that are explained in the next set of chapters include:

- renaming an existing WSDL operation definition
- removing an existing WSDL operation definition
- changing the MEP of an existing WSDL operation definition
- adding a fault message to an existing WSDL operation definition
- adding a new required XML Schema element or attribute declaration to a message definition
- increasing the constraint granularity of an XML Schema element or attribute declaration of a message definition
- renaming an optional or required XML Schema element or attribute in a message definition
- removing an optional or required XML Schema element or attribute or wildcard from a message definition
- adding a new required WS-Policy assertion or expression
- adding a new ignorable WS-Policy expression (most of the time)

Incompatible changes cause most of the challenges with Web service contract versioning.

Conclusion

Part II of this article series demonstrate the use of version identifiers and discuss three common versioning strategies, each of which has its own rules and conventions.

References

[REF-1] "Web Service Contract Design and Versioning for SOA" by Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, Umit Yalcinalp, Canyon Kevin Liu, David Orchard, Andre Tost, James Pasley for the "Prentice Hall Service-Oriented Computing Series from Thomas Erl", Copyright 2008 Prentice Hall/Pearson PTR and SOA Systems Inc., <http://www.soabooks.com/wsc/>

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About](#) [Legal](#)

Copyright © 2006-2009
SOA Systems Inc.