

# The SOA Magazine

## Feature Article



### Web Service Contract Versioning Fundamentals Part II: Version Identifiers and Versioning Strategies

by Thomas Erl, David Orchard and James Pasley

Published: January 19, 2009 (SOA Magazine Issue XXV: January 2009)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

*Abstract: There are different ways of versioning service contracts based on policies, priorities, and requirements. This, the second article in a two-part series from the book "Web Service Contract Design & Versioning for SOA", introduces three common versioning strategies: strict, flexible, and loose. The pros and cons of each approach are discussed and further ranked in relation to strictness, governance impact, and complexity. The role of version identifiers is also explored through a series of examples.*

*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.*

### Version Identifiers

One of the most fundamental design patterns related to Web service contract design is the Version Identification pattern. It essentially advocates that version numbers should be clearly expressed, not just at the contract level, but right down to the versions of the schemas that underlie the message definitions.

The first step to establishing an effective versioning strategy is to decide on a common means by which versions themselves are identified and represented within Web service contracts.

Versions are almost always communicated with version numbers. The most common format is a decimal, followed by a period and then another decimal, as shown here:

```
version="2.0"
```

Sometimes, you will see additional period + decimal pairs that lead to more detailed version numbers like this:

```
version="2.0.1.1"
```

The typical meaning associated with these numbers is the measure or significance of the change. Incrementing the first decimal generally indicates a major version change (or upgrade) in the software, whereas decimals after the first period usually represent various levels of minor version changes.

From a compatibility perspective, we can associate additional meaning to these numbers. Specifically, the following convention has emerged in the industry:

- A minor version is expected to be backwards compatible with other minor versions associated with a

major version. For example, version 5.2 of a program should be fully backwards compatible with versions 5.0 and 5.1.

- A major version is generally expected to break backwards compatibility with programs that belong to other major versions. This means that program version 5.0 is not expected to be backwards compatible with version 4.0.

This convention of indicating compatibility through major and minor version numbers is referred to as the compatibility guarantee. Another approach, known as "amount of work," uses version numbers to communicate the effort that has gone into the change. A minor version increase indicates a modest effort, and a major version increase predictably represents a lot of work.

These two conventions can be combined and often are. The result is often that version numbers continue to communicate compatibility as explained earlier, but they sometimes increment by several digits, depending on the amount of effort that went into each version.

There are various syntax options available to express version numbers. For example, you may have noticed that the declaration statement that begins an XML document can contain a number that expresses the version of the XML specification being used:

```
<?xml verison="1.0"?>
```

That same version attribute can be used with the root `xsd:schema` element, as follows:

```
<xsd:schema version="2.0" ...>
```

You can further create a custom variation of this attribute by assigning it to any element you define (in which case you are not required to name the attribute "version").

```
<ListItem version="2.0">
```

An alternative custom approach is to embed the version number into a namespace, as shown here:

```
<ListItem xmlns="http://actioncon.com/schema/po/v2">
```

Note that it has become a common convention to use date values in namespaces when versioning XML schemas, as follows:

```
<ListItem xmlns="http://actioncon.com/schema/po/2010/09">
```

In this case, it is the date of the change that acts as the version identifier. In order to keep the expression of XML Schema definition versions in alignment with WSDL definition versions, we use version numbers instead of date values in the examples throughout the upcoming chapters. However, when working in an environment where XML Schema definitions are separately owned as part of an independent data architecture, it is not uncommon for schema versioning identifiers to be different from those used by WSDL definitions.

Regardless of which option you choose, it is important to consider the Canonical Versioning pattern that dictates that the expression of version information must be standardized across all service contracts within the boundary of a service inventory. In larger environments, this will often require a central authority that can guarantee the linearity, consistency, and description quality of version information. These types of conventions carry over into how service termination information is expressed (as further explored in Chapter 23).

Note: Of course you may also be required to work with third-party schemas and WSDL definitions that may already have implemented their own versioning conventions. In this case, the extent to which the Canonical Versioning pattern can be applied will be limited.

## Versioning Strategies

There is no one versioning approach that is right for everyone. Because versioning represents a governance-related phase in the overall lifecycle of a service, it is a practice that is subject to the conventions, preferences, and requirements that are distinct to any enterprise.

Even though there is no de facto versioning technique for the WSDL, XML Schema, and WS-Policy content that comprises Web service contracts, a number of common and advocated versioning approaches have emerged, each with its own benefits and tradeoffs.

In this chapter we're going to single out the following three known strategies:

*Strict* - Any compatible or incompatible changes result in a new version of the service contract. This approach does not support backwards or forwards compatibility.

*Flexible* - Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility but not forwards compatibility.

*Loose* - Any incompatible change results in a new version of the service contract and the contract is designed to support backwards compatibility and forwards compatibility.

These strategies are explained individually in the upcoming sections and referenced throughout the remaining chapters.

### Strategy #1: The Strict Strategy (New Change, New Contract)

The simplest approach to Web service contract versioning is to require that a new version of a contract be issued whenever any kind of change is made to any part of the contract.

This is commonly implemented by changing the target namespace value of a WSDL definition (and possibly the XML Schema definition) every time a compatible or incompatible change is made to the WSDL, XML Schema, or WS-Policy content related to the contract. Namespaces are used for version identification instead of a version attribute because changing the namespace value automatically forces a change in all consumer programs that need to access the new version of the schema that defines the message types.

This "super-strict" approach is not really that practical, but it is the safest and sometimes warranted when there are legal implications to Web service contract modifications, such as when contracts are published for certain inter-organization data exchanges. Because both compatible and incompatible changes will result in a new contract version, this approach supports neither backwards or forwards compatibility.

#### *Pros and Cons*

The benefit of this strategy is that you have full control over the evolution of the service contract, and because backwards and forwards compatibility are intentionally disregarded, you do not need to concern yourself with the impact of any change in particular (because all changes effectively break the contract).

On the downside, by forcing a new namespace upon the contract with each change, you are guaranteeing that all existing service consumers will no longer be compatible with any new version of the contract. Consumers will only be able to continue communicating with the Web service while the old contract remains available alongside the new version or until the consumers themselves are updated to conform to the new contract.

Therefore, this approach will increase the governance burden of individual services and will require careful transitioning strategies. Having two or more versions of the same service co-exist at the same time can become a common requirement for which the supporting service inventory infrastructure needs to be prepared.

### Strategy #2: The Flexible Strategy (Backwards Compatibility)

A common approach used to balance practical considerations with an attempt at minimizing the impact of changes to Web service contracts is to allow compatible changes to occur without forcing a new contract

version, while not attempting to support forwards compatibility at all.

This means that any backwards-compatible change is considered safe in that it ends up extending or augmenting an established contract without affecting any of the service's existing consumers. A common example of this is adding a new operation to a WSDL definition or adding an optional element declaration to a message's schema definition.

As with the Strict strategy, any change that breaks the existing contract does result in a new contract version, usually implemented by changing the target namespace value of the WSDL definition and potentially also the XML Schema definition.

#### *Pros and Cons*

The primary advantage to this approach is that it can be used to accommodate a variety of changes while consistently retaining the contract's backwards compatibility. However, when compatible changes are made, these changes become permanent and cannot be reversed without introducing an incompatible change. Therefore, a governance process is required during which each proposed change is evaluated so that contracts do not become overly bloated or convoluted. This is an especially important consideration for agnostic services that are heavily reused.

#### Strategy #3: The Loose Strategy (Backwards and Forwards Compatibility)

As with the previous two approaches, this strategy requires that incompatible changes result in a new service contract version. The difference here is in how service contracts are initially designed.

Instead of accommodating known data exchange requirements, special features from the WSDL, XML Schema, and WS-Policy languages are used to make parts of the contract intrinsically extensible so that they remain able to support a broad range of future, unknown data exchange requirements.

For example:

- The anyType attribute value provided by the WSDL 2.0 language allows a message to consist of any valid XML document.
- XML Schema wildcards can be used to allow a range of unknown data to be passed in message definitions.
- Ignorable policy assertions can be defined to communicate service characteristics that can optionally be acknowledged by future consumers.

These and other features related to forwards compatibility are discussed in upcoming chapters.

#### *Pros and Cons*

The fact that wildcards allow undefined content to be passed through Web service contracts provides a constant opportunity to further expand the range of acceptable message element and data content. On the other hand, the use of wildcards will naturally result in vague and overly coarse service contracts that place the burden of validation on the underlying service logic.

Note: All three strategies will be referenced in upcoming chapters as we explore how versioning can be accomplished with the WSDL, XML Schema, and WS-Policy languages.

#### Conclusion

Provided here is a table that broadly summarizes how the three strategies compare based on three fundamental characteristics.

	Strategy		
	Strict	Flexible	Loose
Strictness	high	medium	low

Governance Impact	high	medium	high
Complexity	low	medium	high

Table 1 - A general comparison of the three versioning strategies.

The three characteristics used in this table to form the basis of this comparison are as follows:

- *Strictness* - The rigidity of the contract versioning options. The Strict approach clearly is the most rigid in its versioning rules, while the Loose strategy provides the broadest range of versioning options due to its reliance on wildcards.
- *Governance Impact* - The amount of governance burden imposed by a strategy. Both Strict and Loose approaches increase governance impact but for different reasons. The Strict strategy requires the issuance of more new contract versions, which impacts surrounding consumers and infrastructure, while the Loose approach introduces the concept of unknown message sets that need to be separately accommodated through custom programming.
- *Complexity* - The overall complexity of the versioning process. Due to the use of wildcards and unknown message data, the Loose strategy has the highest complexity potential, while the straight-forward rules that form the basis of the Strict approach make it the simplest option.

Throughout this comparison, the Flexible strategy provides an approach that represents a consistently average level of strictness, governance effort, and overall complexity.

Each strategy also determines how compatible changes, incompatible changes, and version identifiers are used and applied in support of the rules and conventions of the strategy. Chapters 21, 22, and 23 explore the application of these strategies individually to WSDL definitions, XML Schema definitions, and WS-Policy definitions.

## 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

