JavaOne

Versioning Strategies for Service-Oriented Architectures

Gopalan Suresh Raj
Software Architect
Sun Microsystems
http://blogs.sun.com/gopalan

Kiran Bhumana
Software Architect
eBay, Inc.
http://www.ebay.com

S313771

Goal of this Talk

What the Audience Will Gain

This presentation provides a Dozen (12) Best-Practices to evolve Messages, Services, and the Service Infrastructure of your SOA

Agenda With Section Highlights

- Objects, Components, Services, and Messages
- Service Contracts
- What is Message Versioning, Service Versioning
- Tip #1 Service Versioning Granularity
- Versioning Hell
- Compatibility Defined

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- Tip #3 Message Router at endpoints
- Tip #4 Mediator/Intermediary Pattern
- Tip #5 Service Bindings to handle change
- Tip #6 Endpoint Pattern for versioning
- Tip #7 Multiple Endpoint Addresses
- Tip #8 Version-aware Registry using UDDI

Agenda With Section Highlights

- Tip #9 Parameter Pattern for versioning
- Tip #10 Using Version Extensibility Points
- Tip #11 Lifecycle Pattern to Service Versioning
- Tip #12 Adapter Pattern to evolve the Service Infrastructure
- Summary
- Q&A

Agenda With Section Highlights

- **Objects, Components, Services, and Messages**
- Service Contracts
- What is Message Versioning, Service Versioning
- Tip #1 Service Versioning Granularity
- Versioning Hell
- Compatibility Defined

Objects and Components

- Binary reusable entities
- Support Encapsulation
- Support Abstraction
- Support Polymorphism
- Support Inheritance

Services

- Messaging Endpoints
- Collections of ***Related***, ***Behavioral***, ***Atomic***, method procedures each representing a Domain-specific behavior
- Act on well-defined Schema-based messages
- Support Encapsulation
- Support Abstraction
- All communication is through ***Idempotent*** messages

What Services are NOT

- Are NOT Components
- Do NOT expose an API
- Do NOT accept parameters
- Polymorphism is NOT supported
- Inheritance is NOT supported

Messages

- Are data defined by a schema
- Bytes on the Wire
- They do NOT encapsulate any behavior
- They do NOT carry any semantic meaning
- In a SOA, the Schema of both the request and response Messages is The REAL INTERFACE

Agenda With Section Highlights

- Objects, Components, Services, and Messages
- **Service Contracts**
- What is Message Versioning, Service Versioning
- Tip #1 Service Versioning Granularity
- Versioning Hell
- Compatibility Defined

Service Contracts

- Service Contracts are of two types:
  1. Formal Service Contracts
  2. Semantic Service Contract
- Formal Service Contracts are of two types:
  1. Collections of related, behavioral, atomic operations or method procedures representing a Domain-specific behavior
  2. Message Schema defining data-types & message signatures
- Each method could have multiple schemas as part of its contract

Agenda With Section Highlights

- Objects, Components, Services, and Messages
- Service Contracts
- **What is Message Versioning, Service Versioning**
- Tip #1 Service Versioning Granularity
- Versioning Hell
- Compatibility Defined

What is Message Versioning?

- XML Schema allows creation of new vocabulary based on a self-describing markup
- Versioning is required to handle the inevitable evolution of such vocabulary gracefully
- This evolution may involve adding, changing, or deleting parts of the vocabulary
- XML Namespaces provide a mechanism for identifying different versions of the vocabulary
- XML schema provides for controlled extensibility of the vocabulary

Message Versioning Defined

- Versioning facilitates the simultaneous co-existence of multiple different implementations of the same thing with each of those implementations uniquely distinguishable and individually accessible
- Message Versioning is a popular approach for dealing with change

Message Schema evolution over time

- New types could be added
- Existing types could be changed
- Existing types could be deprecated
- Existing types could be removed
- The Ideal Message Schema should provide support for both ***Versioning*** and ***Extensibility***

Message Versioning and Extensibility

- Versioning facilities in a Schema should support change ***linearly***
- Extensibility facilities in a Schema should support change ***concurrently***
- Extensions should NOT use the namespace of the XML format
- All elements should allow for extension attributes
- Complex elements should allow for extension elements
- Specify a processing model for dealing with extensions

Service Versioning

- Allows multiple versions of the same service to co-exist
- Allows each consumer to use the correct version of the service for which it has been designed & tested for
- Multiple co-existing versions of the same service in the system allows for independent life-cycles of services and their consumers
- It minimizes the overall impact of introducing changes to the system

Agenda With Section Highlights

- Objects, Components, Services, and Messages
- Service Contracts
- What is Message Versioning, Service Versioning
- **Tip #1 Service Versioning Granularity**
- Versioning Hell
- Compatibility Defined

#1 Service Versioning Granularity

- Two types of Versioning granularity:
  1. Versioning a service as a whole (including all its operations/methods).
  2. Versioning of individual service operations/methods
- Method-based versioning granularity provides enhanced flexibility
- Method-based versioning is the recommended approach

#1 Method-based versioning Pros

- Allows for immutable services. Service can provide additional methods (with many versions) over time, but the name & classification of the service itself never changes
- Minimizes impact of service changes to consumers. Only those consumers of the method being versioned are affected while other service consumers are not affected
- Minimizes overall amount of deployed code. Only those methods that have changed need (re)deployment

#1 Method-based versioning Cons

- Requires that each method be independently deployed, each with its own endpoint address(es).
  - Even then this allows the ability to provide different Service Level Agreements (SLAs) for different methods within the same service
- Requires a more complex service addressing scheme.
  - Rather than just specifying the service name to invoke, consumer has to specify service name, method name, and the version of the method that it requires

Agenda With Section Highlights

- Objects, Components, Services, and Messages
- Service Contracts
- What is Message Versioning, Service Versioning
- Tip #1 Service Versioning Granularity
- **Versioning Hell**
- Compatibility Defined

Versioning Hell

- As long as clients bind to an API contract, they are stuck in Versioning Hell.
- The only two generic Service API contracts clients should bind to are:
  1. responseMessage = operation(requestMessage)
  2. operation(requestMessage)

  wherein, Both request and response Messages should conform to a pre-defined Message Schema

Agenda With Section Highlights

- Objects, Components, Services, and Messages
- Service Contracts
- What is Message Versioning, Service Versioning
- Tip #1 Service Versioning Granularity
- Versioning Hell
- **Compatibility Defined**

What determines a new version

- Analyze any changes and their impact on the consumer's execution
- Identify the changes that will break the consumer's execution
- Any change that will impact consumer execution should result in a new version of the service method or message

What determines a new version ...

- As usual the service interface never ever changes. It is always either
  - responseMessage = operation(requestMessage), OR
  - operation(requestMessage)
- Message Schema changes could force a version change
- Implementation changes could force a version change

Types of Compatible Changes

- Versioning support has NOT been built into the web-services architecture
- Best Practices have to be applied to solve this
- Compatibility changes are of the following types:
  1. Forward compatible changes
  2. Backward compatible changes
  3. Non-backward compatible changes

Compatibility Defined

- Forward Compatibility
  - Older versions of Services can consume newer versions of a request message and still not break
  - Older Services do not implement any new behavior
  - Clients can send a newer version of the request message and still have messages successfully processed by an older version of the service
  - Newer clients can continue to use existing services
  - To support this, changes in the XML vocabulary typically involve adding optional elements and or attributes

Compatibility Defined

- Backward Compatibility
  - A new version of the Service can be rolled out without breaking existing clients.
  - Clients can send an older version of a request to a service that understands a newer version of the message and still have successful message processing
  - Existing clients can use newer/updated services

Backward Compatible Changes

- These are changes that will NOT break backward compatibility.
- For these changes, one could just update the XSD/WSDL document(s) in the repository
- Some examples are:
  - Addition of new atomic operations to an existing WSDL document
  - Addition of new XML schema types that are NOT contained within a pre-existing type

Non-backward compatible changes

- These changes will break compatibility
- Some examples are:
  - Removing an operation
  - Renaming an operation
  - Changing data-types of either the request or response messages or order of parameters
  - Changing the structure of a complex data-type
- One strategy is to use the XML namespace to delineate versions of a document that are compatible, e.g.,
  - **http://company.name/datestamp/Identifier**

Agenda With Section Highlights

- **Tip #2 Namespace Pattern for versioning**
- Tip #3 Message Router at endpoints
- Tip #4 Mediator/Intermediary Pattern
- Tip #5 Service Bindings to handle change
- Tip #6 Endpoint Pattern for versioning
- Tip #7 Multiple Endpoint Addresses
- Tip #8 Version-aware Registry using UDDI

#2 Namespace Pattern for versioning

- In this pattern, Namespaces are used to determine different type versions
- Initial version can start with a target namespace of for e.g., **urn:company.name:serviceName:v1**, and each subsequent major version would increase the version number
- Looking at the version number, the appropriate implementation can be invoked
- Amazon.com® uses this pattern extensively.

#2 Namespace Pattern for versioning

- Takes advantage of XML namespaces
- Works well with backward-incompatible schema changes
- This pattern however, does NOT take advantage of XML Schema Extensibility

#2a Placing new constructs in new namespaces – Not favored

- Additions to XML formats could be in a different namespace from the core types
- To ensure backward compatibility, extensibility model with default Must Ignore rules for items outside of consumer understandable interface along with **mustUnderstand** constructs.
- Core components of format are NOT in the same namespace as extensions and its difficult to differentiate between them in later versions
- This approach is NOT Forward Compatible

Non-backward compatible changes

- Only use a new namespace when backward compatibility is NOT permitted, for e.g., if the new language components wont be understood
- If a later version of a type is backward compatible with older versions, the old namespace name must be used together with the XML's extensibility model
- Formats should specify a **mustUnderstand** model for Backward-Incompatibe changes to the formats that do NOT change the namespace name

Non-backward compatible changes...

- With SOAP Binding and
  - Literal Encoding - Namespace is specified in the definition of messages as part of the XML schema namespace definition.
  - SOAP Encoding - Namespace is specified within the SOAP binding element
- Generate a failure on server if an old namespace is requested
- Employ a router to route messages to appropriate implementation based on namespace received

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- **Tip #3 Message Router at endpoints**
- Tip #4 Mediator/Intermediary Pattern
- Tip #5 Service Bindings to handle change
- Tip #6 Endpoint Pattern for versioning
- Tip #7 Multiple Endpoint Addresses
- Tip #8 Version-aware Registry using UDDI

#3 Message Router at endpoints

- Implement Message Endpoints as Message Routers
- Multiple implementations which accept different schema are available
- Inbound messages are automatically routed to the right implementation based on their schema
- Messages which conform to unrecognized schema will be rejected
- Very similar to the concept of method overloading in the Object-Oriented world

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- Tip #3 Message Router at endpoints
- **Tip #4 Mediator/Intermediary Pattern**
- Tip #5 Service Bindings to handle change
- Tip #6 Endpoint Pattern for versioning
- Tip #7 Multiple Endpoint Addresses
- Tip #8 Version-aware Registry using UDDI

#4 Mediator/Intermediary Pattern

- A specialization of the Message Router pattern is the Mediator pattern
- All versions can be deployed independently
- Mediator is responsible to dynamically resolve endpoint addresses of the desired service version, and dispatch the message to the right instance.
- This pattern must support stringent Service Level Agreements (SLA) of all services accessed through it

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- Tip #3 Message Router at endpoints
- Tip #4 Mediator/Intermediary Pattern
- **Tip #5 Service Bindings to handle change**
- Tip #6 Endpoint Pattern for versioning
- Tip #7 Multiple Endpoint Addresses
- Tip #8 Version-aware Registry using UDDI

#5 Service Bindings to handle change

- Scenarios include:
  - Changes in operations exposed by the Service
  - Changes in the URI of WSDL for the Service
  - Changes to Service interface version
  - Changes to Service Endpoints
- The Binding mechanism should NOT assume a static endpoint

#5 Service Bindings to handle change

- A flexible service binding approach could:
  - Use indirection to acquire a Service Endpoint
  - Avoid assumptions like anything about a Service beyond its interface is static
  - Identify compatible services based on the Interface and the Interface Version that they exhibit

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- Tip #3 Message Router at endpoints
- Tip #4 Mediator/Intermediary Pattern
- Tip #5 Service Bindings to handle change
- **Tip #6 Endpoint Pattern for versioning**
- Tip #7 Multiple Endpoint Addresses
- Tip #8 Version-aware Registry using UDDI

#6 Endpoint Pattern for versioning

- Specify the version in the endpoint URL making it part of the URL being invoked
- For e.g., version 1 of myService could have an endpoint URL as:
  - http://host.com/services/1-0-0/myService
- Alternatively, it could be something like:
  - http://host.com/service/v1
  - http://host.com/service/v2
- JIRA® uses the above pattern

#6 Endpoint Pattern for versioning ...

- The advantage of this pattern is the extensive use of XML Schema
- The disadvantage is that it requires users to update their invocation URL for each new version update
- This will also NOT work well with incompatible schema changes
- This pattern does NOT take advantage of XML Namespaces

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- Tip #3 Message Router at endpoints
- Tip #4 Mediator/Intermediary Pattern
- Tip #5 Service Bindings to handle change
- Tip #6 Endpoint Pattern for versioning
- **Tip #7 Multiple Endpoint Addresses**
- Tip #8 Version-aware Registry using UDDI

#7 Multiple Endpoint Addresses

- Each version of a given service method is deployed at its own endpoint address
- This assumes consumers can resolve these addresses using a registry given the service name, operation name, and its version
- This achieves complete separation of multiple method version deployments
- However, its a more complex addressing schema
- Because of just one network hop, provides better scalability and lowers coupling between versions of the same operation

Agenda With Section Highlights

- Tip #2 Namespace Pattern for versioning
- Tip #3 Message Router at endpoints
- Tip #4 Mediator/Intermediary Pattern
- Tip #5 Service Bindings to handle change
- Tip #6 Endpoint Pattern for versioning
- Tip #7 Multiple Endpoint Addresses
- **Tip #8 Version-aware Registry using UDDI**

#8 Version-aware Registry using UDDI

- Each **wsdl:portType** should be represented by a unique **tModel**
- Clients of that portType version can do a search of UDDI for services that are advertized
- This can be done in a couple of ways:
  1. Advertize compliance with several interfaces
  2. Introduce a version number to qualify the interface in the **tModel**
- #1 above supports version compatible searches using just a **tModel** key, while #2 is slow since it requires queries to analyze many UDDI structures

#8 Advertizing compliance ...

- A Service compatible with both an earlier and later version of interface can reference the **tModel** for each in its **tModelInstanceDetails** collection
- This eliminates need for a version number
- No need to alter way in which queries are done
- As the interface evolves, maintenance of version collection may become unwieldy
- If several implementations exist, multiple **tModel** collections should be maintained

#8 Version Number qualifier for tModel

- Introduce a **instanceDetails** structure carrying version Number and URL for related WSDL in **tModelInstanceInfo** for standard **tModel** reference
- Service versioning achieved without requiring service registrations to carry **tModel** references to all interface versions they are compatible with
- This approach conflicts with OASIS best practices
- All service implementations must agree on same version number
- Query should be modified to include version id

Agenda With Section Highlights

- **Tip #9 Parameter Pattern for versioning**
- Tip #10 Using Version Extensibility Points
- Tip #11 Lifecycle Pattern to Service Versioning
- Tip #12 Adapter Pattern to evolve the Service Infrastructure
- Summary
- Q&A

#9 Parameter Pattern for versioning

- Include a version number in the request message to help determine the response to return
- The advantage of this pattern is that it uses XML Schema extensively
- Does NOT work well with incompatible schema changes
- Does NOT take advantage of XML Namespaces
- Requires user to pass the parameter in every request
- eBay® uses this pattern extensively

Agenda With Section Highlights

- Tip #9 Parameter Pattern for versioning
- **Tip #10 Using Version Extensibility Points**
- Tip #11 Lifecycle Pattern to Service Versioning
- Tip #12 Adapter Pattern to evolve the Service Infrastructure
- Summary
- Q&A

#10 Using Version Extensibility Points

- Any XML vocabulary can be made extensible by providing well-defined wild-card elements at specific points in the content model
- To make use of wild-cards deterministic, delimiters or sentry elements should be placed around those wild-card elements
- This approach keeps all core components of the vocabulary in a single namespace

#10 Using Version Extensibility Points

- This approach is both Forward and Backward compatible
- Forward compatibility is based on not adding any new required constructs in future versions
- It obviates the need for an explicit **mustUnderstand** construct – If an unknown element is encountered, it results in fatal error
- However, this approach makes both XML Schemas and XML instances more verbose

Agenda With Section Highlights

- Tip #9 Parameter Pattern for versioning
- Tip #10 Using Version Extensibility Points
- **Tip #11 Lifecycle Pattern to Service Versioning**
- Tip #12 Adapter Pattern to evolve the Service Infrastructure
- Summary
- Q&A

#11 Lifecycle Pattern to Service Versioning

- Develop a service lifecycle that a Service goes through from conception to its final retirement
- For e.g., eBay® uses this Service Lifecycle
  1. PROPOSED - Service has been proposed. Not accepted for development yet.
  2. PLANNED - Service has been accepted for development. A project or plan (with resources) exists to create the service.
  3. IN-REVIEW - Service contract artifacts are submitted for review.

#11 Lifecycle Pattern to Service Versioning ...

- For e.g., eBay®'s Service Lifecycle continued ...
  4. APPROVED - Service contract artifacts have been approved. Any change to the WSDL after this approval requires a new version.
  5. TESTED - Service has been implemented and tested (QA OK)
  6. DEPLOYED - Service has been deployed and is available for use.
  7. DEPRECATED - Service is only available for use for a limited time. And is in the process of being retired. New versions of the service are no longer allowed.
  8. RETIRED - Service has been retired and is no longer available for use.

Agenda With Section Highlights

- Tip #9 Parameter Pattern for versioning
- Tip #10 Using Version Extensibility Points
- Tip #11 Lifecycle Pattern to Service Versioning
- **Tip #12 Adapter Pattern to evolve the Service Infrastructure**
- Summary
- Q&A

#12 Adapter Pattern to evolve the Service Infrastructure

- Just like Messages and Services evolve, the Service Infrastructure itself may need to evolve:
  - Changes to Transports and Bindings, for e.g, switching from SMTP to HTTP
  - Changes to Message Encoding, upgrading from SOAP to REST
  - Changes in Addressing Schema, moving to WS-Addressing
- It is prudent to implement Backward Compatibility so the new Infrastructure can handle messages produced by or for the older version of the Service Infrastructure

#12 Adapter Pattern to evolve the Service Infrastructure ...

- Moving all service implementations and consumers to the new Service Infrastructure is both prohibitively expensive and time-consuming
- Introduce an Adapter between the consumer and the provider within the new Service Infrastructure
- To older service consumers the adapter will talk the language of the older Infrastructure version
- To the service provider residing within the new version of the Infrastructure, it sends in a modified message to mimic the new Infrastructure version

Summary

- This presentation provided a Dozen (12) Best-Practices to evolve Messages, Services, and the Service Infrastructure of your SOA

Q&A

Gopalan Suresh Raj
Software Architect
Sun Microsystems
http://blogs.sun.com/gopalan

Kiran Bhumana
Software Architect
eBay, Inc.
http://www.ebay.com