Redpaper

Andre Tost
Greg Flurry

Building a Service Versioning Gateway with WebSphere ESB and WebSphere Service Registry and Repository V7.5

Service-oriented architecture (SOA) successfully addresses the need for business flexibility and resiliency in many separate enterprises. Like any system, SOA must inevitably address change that occurs due to business or technical reasons. Because SOA promotes business agility, and agility implies change, one might suggest that SOA even encourages change. Such change can be painful. However, SOA also promotes resilience, the ability to adapt to change, so SOA can deal with this change gracefully.

Change in a system takes place at many levels of granularity. Because SOA concerns itself with services, the most natural focus for examination of change in SOA is on such services. When something changes, we often think of the result as a new version. Therefore, here we will discuss service versioning, or *service evolution*, in the context of SOA.

In SOA, a service consumer interacts with a service provider to accomplish a certain type of business task. Changes in SOA impact services. Many service consumers can, and in an ideal SOA, do use a single service provider, so it is reasonable to expect the perspectives and expectations of the consumer and the provider to vary with service versioning. For example, a provider typically wants to minimize change, but also respond fairly rapidly to requests for change from consumers and changes needed from within. On the other hand, the consumer would typically desire no changes, isolation from changes, or perhaps slow change.

Based on these varying perspectives on change in SOA, service versioning must balance these two needs:

► Facilitating the delivery of change by the provider
► Minimizing disruption to consumers due to change

To achieve these goals, this IBM® Redpaper™ introduces a Service Versioning Gateway. It is, as the name indicates, a gateway that exists between service consumers and service providers and offers consumer-specific virtual service interfaces without requiring a specific service provider implementation for each of the interfaces. In other words, the gateway is capable of handling particular cases of service evolution and versioning.

Containment of multiple approaches to implementation makes possible the gateway described in this document. These approaches share a common architecture but differ in the specifics of how the gateway components are built and work together. The implementation approach outlined in this document leverages IBM WebSphere® enterprise service bus (ESB) and WebSphere Service Registry and Repository (WSRR). We show how the individual components supporting the versioning gateway are designed and implemented.

This publication is intended for potential and actual users of the Service Versioning Gateway.

Introduction: The challenge

The problem of service evolution is primarily driven by the data (or messages) used in requests and responses. Separate "flavors" of the same data objects are required by separate consumers of what is fundamentally the same service.

In many cases, new consumers require additional data fields that are not offered by the interface specification in its existing version. Assume, for example, that a Customer service returns an Address. A new consumer of this service now requires that the address returns two fields for the street. This was not included in the original version of the service, or more precisely, it was not included in the complex type definition of Address, which is used as part of the output message of the Customer service.

To address this new requirement, the schema for Address has to be updated to include the new additional field called street2.[1] The update is deployed as a new version of the service. Either the earlier version of the service must remain deployed too, or existing consumers must be updated to reflect the changed schema.

---

[1] We assume, of course, that the additional field is supported by the back-end system that serves as the system of record for customer data.
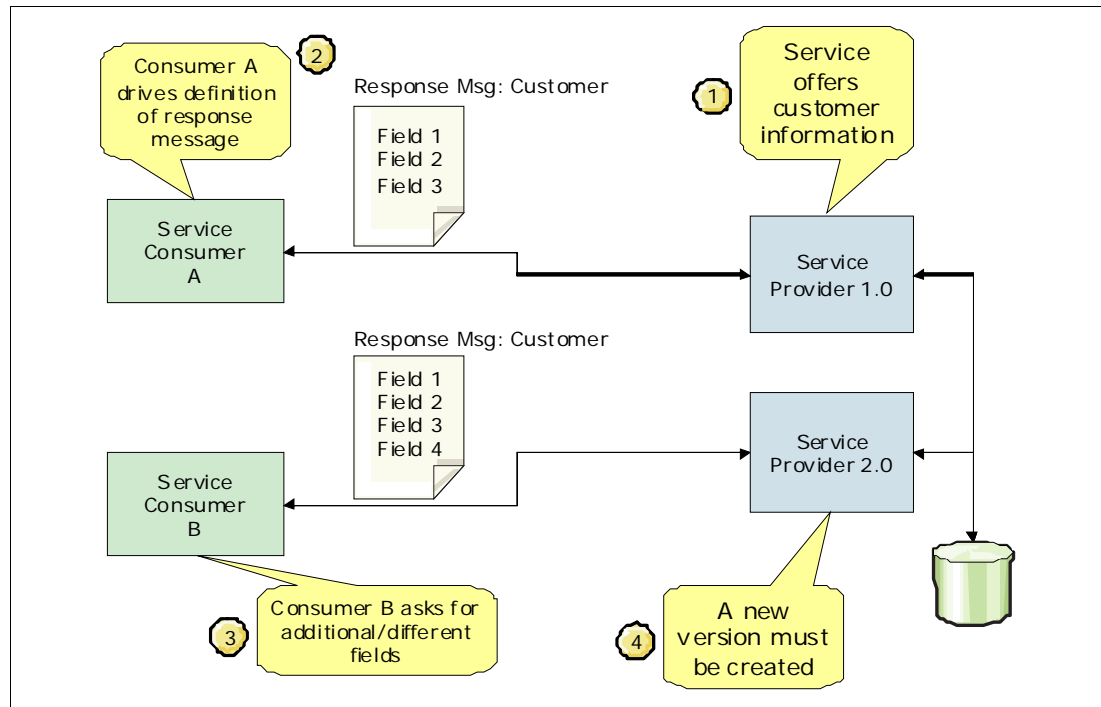
Figure 1 shows an overview of this scenario.

Figure 1   Versioning scenario

Service versioning

Before discussing service versioning, it is useful to discuss the nature of the "service" itself. Depending on the context, *service* can refer to a rather abstract notion of a business task, a physical provider of that business task, or a formal representation of that provider. To clarify the discussion, this Redpaper uses *governed service* to mean the business task (sometimes called the *charter*), *service realization* for the physical provider, and *service specification* for the formal representation.

The following sections describe how these concepts relate to an architectural model for service versioning used to analyze and address the challenge.

Service specification

The lynchpin of the service versioning model is the service specification (SS) because of its business and technical relevance to both service provider and service consumer. The SS, shown in Figure 2, is a set of concise, external semantic and non-semantic characteristics implemented by a service realization. The SS defines only external characteristics to maintain separation of concerns between the service consumer and service realization.
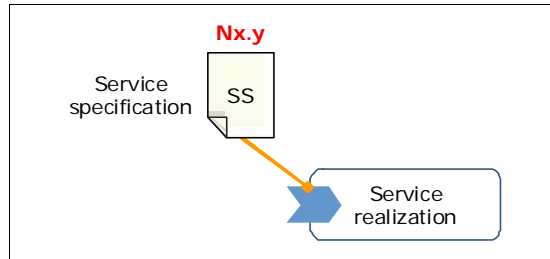

Figure 2   Service specification

At a business level, the SS defines the business task achieved by engaging in a service interaction. At a technical level, the SS describes how a consumer interacts with a service realization and the qualities of service that can be expected in an interaction. Thus the SS describes the "what" and the "how" of a service interaction.

Figure 2 shows that the service specification is a versioned entity. As such, it has an identifier of the form {name, version}, which is unique to one service specification in a domain. The name reflects the relevant large-grained functional and non-functional external characteristics of the governed service and distinguishes different governed services, but not different versions of a governed service. For example, governed service {"CreditScore", ?} is distinguished from {"Customer", ?}. The version distinguishes separate versions of a SS, such as {"CreditScore", 1.2}, as opposed to {"CreditScore", 1.3}. In Figure 2, the identifier is abbreviated as Nx.y, where N is the name and x.y is the version.

The next section examines how changes to the SS impact the version component of the SS identifier.

Backward compatibility

Change in the SS can be evolutionary or revolutionary. Evolutionary changes show a clear heritage to prior SSs. Revolutionary changes do not show such a relationship. Evolutionary change results in essentially the same, although evolved, business or technical characteristics, and thus a new version of a governed service. This leads to a view of the governed service as a continuum of evolved service specifications over the lifetime of the governed service, from its inception to its retirement, as shown for Service 1 in the bottom half of Figure 3.
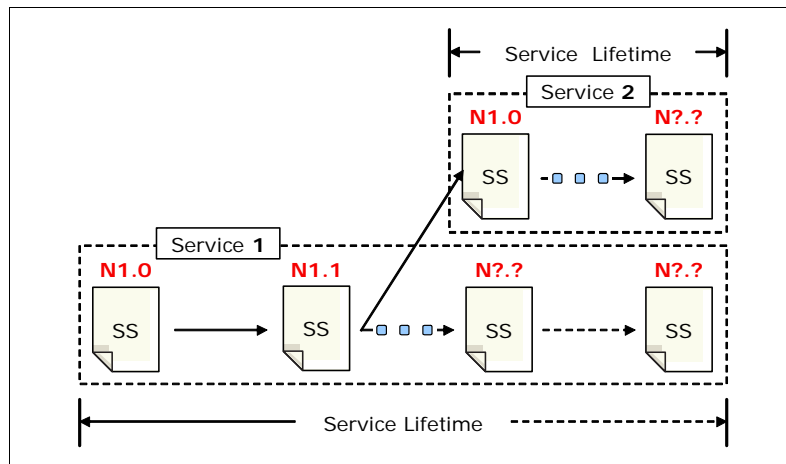
Figure 3   Revolutionary change

Revolutionary changes result in a new governed service, as shown for Service 2 in the top half of Figure 3. Here, Service 2 branches off from Service 1. Typically, revolutionary changes are the result in changes in the semantic characteristics, that is, the business task. When a revolutionary change occurs, the "old" service might continue to live, or might die, depending on the needs of the enterprise.

The provider must determine whether a change is evolutionary or revolutionary based on impact to consumers and consequences to the provider. Note that only evolutionary change that results in new versions are in the scope of this discussion.

Note that the SS version logically has the form of a number **major.minor** and is based on backward compatibility adapted from the Open Services Gateway initiative (OSGI). The version starts at 1.0 for the initial SS. Backward compatible changes cause the **minor** number to increment. Backward incompatible changes cause the **major** number to increment, and the **minor** number to reset to 0.

The provider must determine whether a change is backward compatible based on impact to consumers. "Change" means any difference in the SS. Any characteristic not included in the SS cannot be considered a source of change

A new SS is backward compatible with an ancestor SS if all consumers can, without change, interact with service realizations satisfying the new SS under the terms of the old SS. Listed here are several examples of backward compatible changes:

- ► Adding a new operation to a service (usually)
- ► Adding a new protocol binding to a service (usually)
- ► Fixing a bug in a service realization (usually)
- ► Adding a new, optional business data item to an operation (maybe)

A new SS is backward incompatible with an ancestor SS if any consumer will, without change, be unable to interact with service realizations satisfying the new SS under the terms of the old SS. Listed here are several examples of backward incompatible changes:

► Adding a new, required business data item to an operation (always)
► Adding confidentiality to the interaction protocol (always)
► Modifying the business task (usually)
► Modifying the qualities of service (maybe)[2]

Note: "Placeholders" at the end of each complex type defined in XML schema can be represented by <xsd:any/> elements. To prepare for future change is typically not practical, because it leads to an inflated number of such elements.

Service description

The SS defines the "what" and the "how" for a service interaction, but it is actually not enough to truly enable an interaction. In particular, there is no information that indicates the address of a service realization. The entity *service description (SD)*, shown in Figure 4, contains all information needed for a SS to interact with a service realization. It comprises, by inclusion or reference, the SS to define relevant external semantic and non-semantic characteristics (the "what" and 'how"), and a service realization address (the "where").
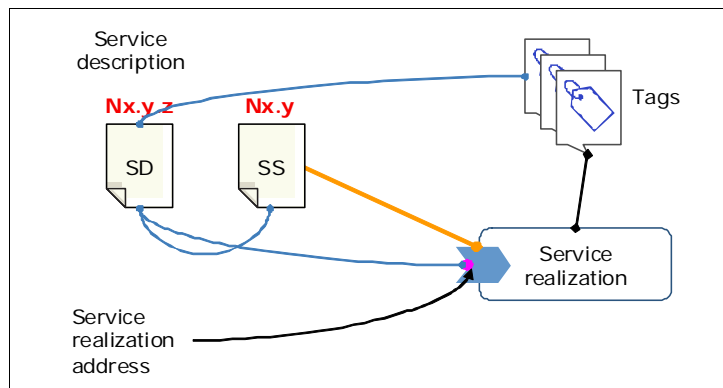

Figure 4   Service description

Note that the SD itself has a unique identifier of the same form as an SS. In fact, the SD identifier reflects the SS identifier, but the version adds a <micro> field to SS version to account for "maintenance changes." The **micro** field increments when internal changes are made that can affect the tacit characteristics relevant to a particular version of an SS. Examples include separate configuration files or rules and even bug fixes in the implementation or assembly for the service realization. An SD can include various forms of tags to distinguish a service realization from others enacting the same SS.

The SD is typically published (as metadata) in a service registry. In general, the associated SS is published (as metadata) as well.

In realizations of the model, the tags or **micro** field, or both, are sometimes unused. In effect, the SD and the SS are merged.

---

[2] For example, adding a new field to a response message, even if optional, can lead to problems with consumers who have not been built with technology that can handle unexpected fields in a response message.

Service version

The previous section suggested that many service realizations can implement the same version of a service specification. This involves another entity in the model called the *service version (SV)*, as shown in Figure 5.
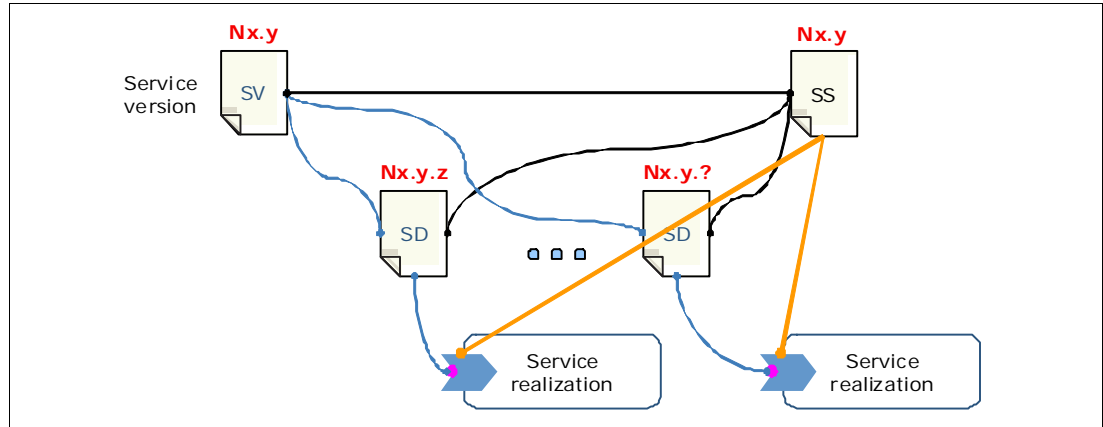

Figure 5   Service version

The SV aggregates all service realizations that implement the same version of a SS for a governed service. The SV comprises (by inclusion or reference) one version of a SS and one or more SDs representing service realizations.

The SV itself has a unique identifier that is the same as the SS identifier. The SV, however, is not itself a versioned entity in the model; it simply represents all realizations of a particular service version.

Like the other entities, the service version is typically published (as metadata) in a service registry.

Governed service

The final entity in the model is the *governed service (GS)*, as shown in Figure 6. As previously suggested, a GS can be considered a continuum of SSs and the implementations of those SSs.
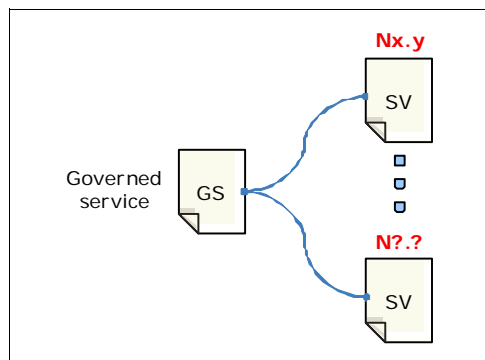

Figure 6   Governed service

The GS aggregates all service versions, unless retired, from inception of the service to most current version, resulting in a continuum of service versions. The service also indirectly aggregates all service realizations that implement all versions of an SS. The GS comprises (by inclusion or reference) one or more SVs, each identifying the corresponding SS and SD.

The GS has an identifier that is simply the name of the service, as in the SS. The GS itself is not a versioned entity in the model.

As with the other entities, the GS is typically published (as metadata) in a service registry.

Service subscription

So far, our model has focused on what might be the provider side of the interaction. This means all of the information about the GS, SV, SD, and so forth, is generic to all consumers of the GS. However, as suggested above, there can be many consumers of a GS, and each consumer might want to use the GS a bit differently. For example, each consumer might want to use another version of the GS. The service subscription, shown in Figure 7, facilitates this capability.
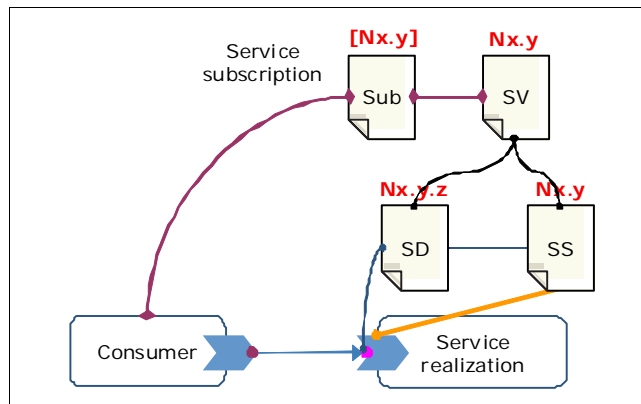

Figure 7   Service subscription

The *subscription* is an agreement or contract between an instance of a consumer and a SV, and indirectly, between a consumer instance and a GS. Thus, the subscription ties a consumer to a specific version of a GS. Although beyond the scope of the current discussion, the service subscription also enables QoS management and impact analysis for the provider.

The service subscription itself is not versioned, but only references a SV. That said, the service subscription can be considered to have the same identifier as the SV it references, indicated by [Nx.y].

Mediation and service versioning

In modern SOAs, the service realization is really constructed from a provider exposed as a GS by a mediation flow hosted in an enterprise service bus (ESB). Mediation is used to ensure separation of concerns, such as to compensate for non-semantic differences in characteristics demanded by a service specification and offered by a provider.

Mediation can be used to support service versioning, buffer consumers from change, and allow greater flexibility for change for the service realization. In particular, a mediation pattern called the *service gateway* is extremely useful in service versioning. A service gateway mediation is interface agnostic, meaning that it acts as any interface, allowing it to accept any message. The gateway then routes the message to the correct service realization. This allows the service gateway to appear to be any version of a GS, compatible or incompatible.

Figure 8 shows a summary of our model, including a service gateway mediation. A GS comprises a number of SVs. Each SV has its external semantic and non-semantic characteristics defined in an SS, and any SV can have many service realizations that implement the corresponding SS. Each service realization also has a SD referenced by the SV.
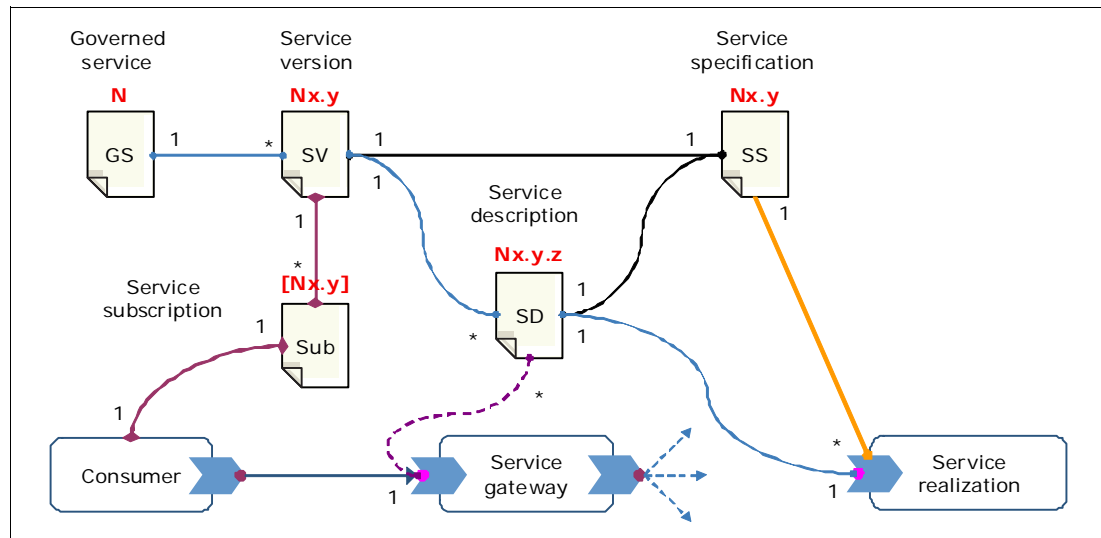

Figure 8   Summary of service versioning model

To enable the service gateway function, each SD in effect also references the service gateway that emulates the corresponding service realization.

Finally, a consumer has a service subscription that references a particular SV. Through the linkages described above, requests from the consumer to the service realization are initially send to the service gateway. The gateway uses various means to determine the correct service realization to which to send the requests.

Figure 9 on page 10 shows a specific use case. The governed service A has three versions:

- ► The initial version, A1.0,
- ► A backwards compatible version, A1.1,
- ► A backwards incompatible version, A2.0.

There is a single consumer subscribed to each version, but each consumer sends requests to the service gateway. The gateway can route requests appropriately so that requests go to the appropriate service realization.
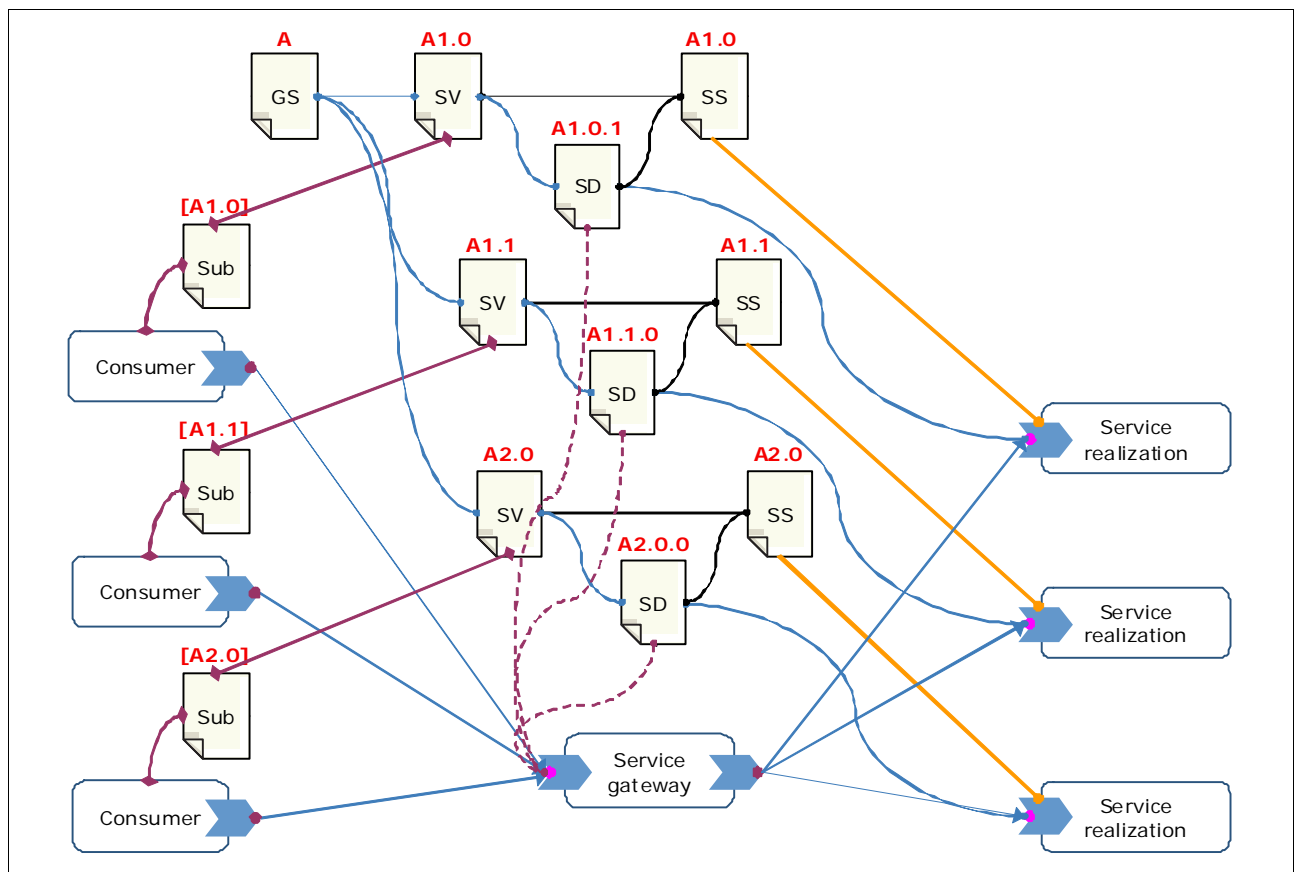

Figure 9   A Service gateway example

It is important to recognize that a service gateway can do more than the simple routing described above. For example, because A1.1 is backward compatible with A1.0, A1.0 can be retired and requests simply routed instead to A1.1. Sophisticated service gateways can translate request and response messages. Under certain circumstances, it is actually possible for the gateway to perform that translation between compatible and incompatible message types. Thus, in Figure 9, requests for A1.0, A1.1 and A2.0 can be sent to the service realization for A2.0, with requests and responses translated as appropriate.

Use case scenario

To illustrate the role of the service versioning gateway and its design, we will assume a typical setup of services and related components, as shown in Figure 10.
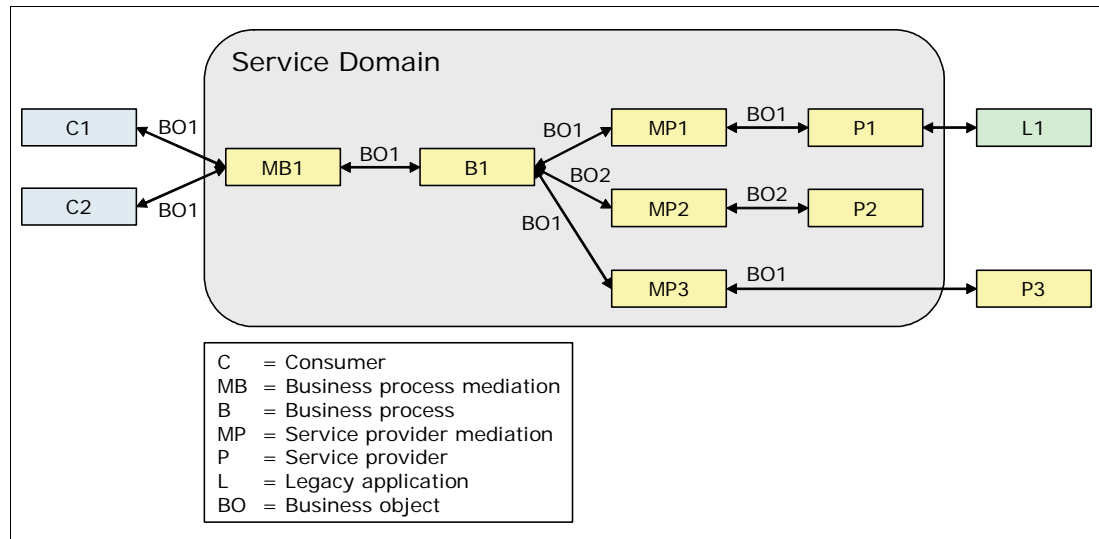
Figure 10   Typical service configuration

Here we see two consumers, C1 and C2, using a service. They access a mediation called MB1, which acts as a gateway in front of the actual service, B1. B1 happens to be implemented as a Business Process Execution Language (BEPEL) flow that is hosted in WebSphere Process Server. The flow uses three other services as part of its processing. All are accessed through additional mediations, MP1, MP2 and MP3. These mediations are located in front of the actual services needed by the flow, namely P1, P2 and P3. P1 is wrapping an existing older function, named L1, such as an existing IBM CICS® transaction. P2, on the other hand, is a service fully implemented in Java and hosted in WebSphere Application Server. P3 is a service that is implemented and hosted outside of the local IT environment, or service domain.

The separate components shown in the picture all have their own Web Services Description Language (WSDL) definition file. However, almost all of them use a complex type called BO1. In other words, BO1 describes a business object (that is, a customer) that flows all the way from the service provider P1 to the consumers C1 and C2. In the example, P3 is using it also.

Let us further assume that a new consumer requires additional data to be retuned from the service. The service operations are the same, the flow implementing B1 is the same, and the same back end is invoked. The only difference is that one or two new fields are carried back to the consumer.
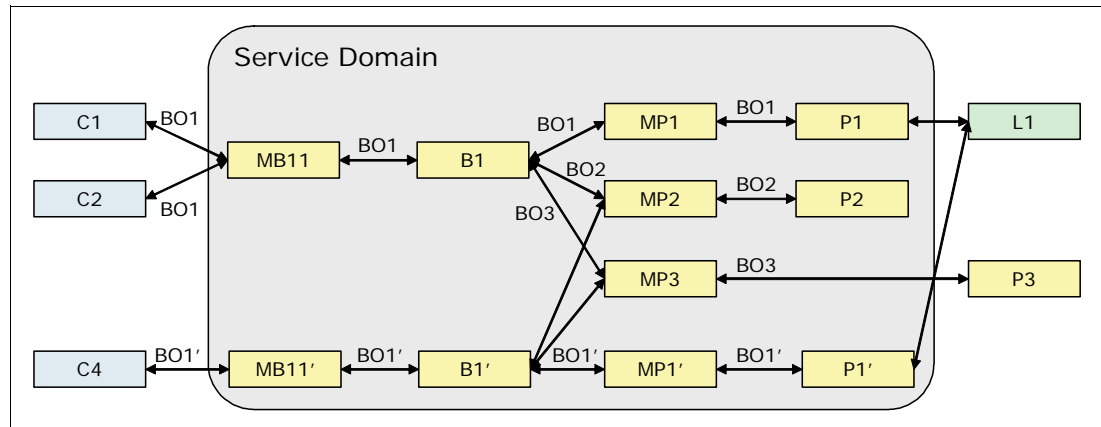
Figure 11 illustrates this concept.

Figure 11   Adding a new consumer

To address this new requirement, the schema for BO1 must change, leading to BO1'. It might have a new namespace, and we assume it is not backward compatible with the previous version (see "Backward compatibility" on page 5). This change now affects all involved components, namely P1 (which becomes P1'), MP1 (MP1'), B1 (B1'), and MB1 (MB1'). The actual logic in all of these components is (nearly) identical; they merely transport a slightly modified business object. Because the existing consumers cannot be changed to use the new interfaces (at least not immediately), the previous versions of all components are sustained. Figure 11 shows the related components after the new version has been introduced.

Whenever a new consumer is built that needs more data or other data, many of the runtime components have to be rebuilt and redeployed, and all of them stay in production, which creates an operational challenge. Moreover, rebuilding, retesting, and redeploying all changed components might take a long time.

In order to address this problem (to accelerate the time it takes to let a service evolve), we must understand the types of message mismatches that might occur during service evolution:

► Additional optional elements are added to a business object that is used, although not exclusively, inside a response message of a service. In other words, elements might have to be removed from a message before being delivered to a consumer.

► The namespace of an element in a request or response message for a service changes.

Service Versioning Gateway

The Service Versioning Gateway directly addresses the use case requirements by providing a gateway mediation component that exists between service consumers and service providers and applies appropriate transformations to request and response messages.

Overview

The solution described in this document uses the notion of an enterprise service bus (ESB), which provides the runtime environment (plus applicable tooling and administration) for the gateway. The gateway receives messages from a variety of consumers who target different services and dynamically adjusts them to fit the expected message schema on either side using Extensible Stylesheet Language Transformation (XSLT).

Figure 12 shows an overview of the Service Versioning Gateway, applied to the example scenario in Figure 1 on page 3.
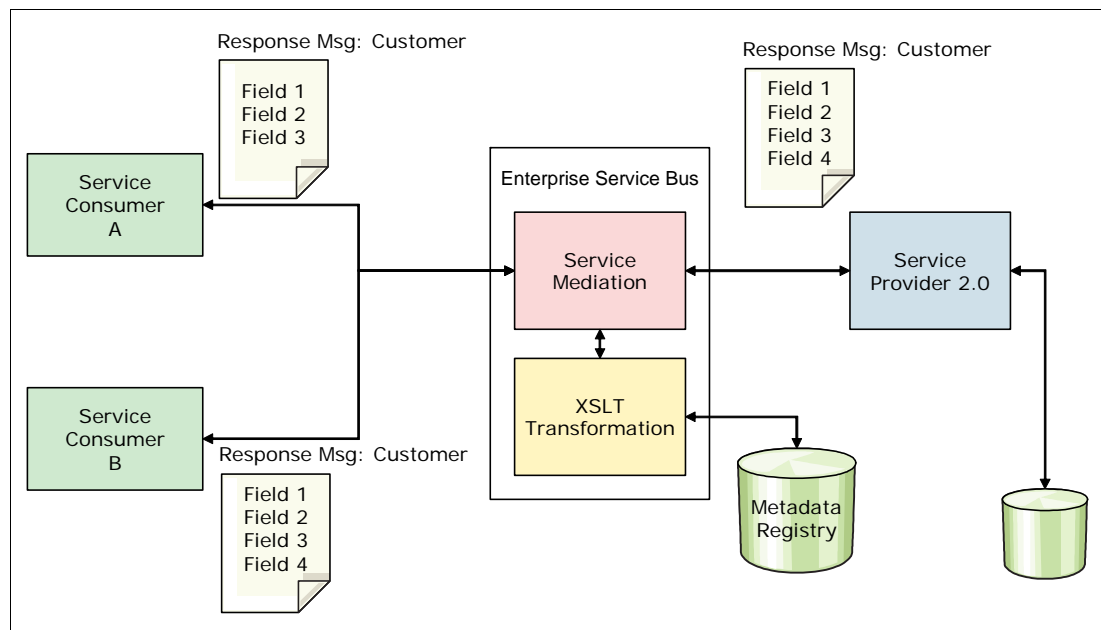
Figure 12   Service Versioning Gateway

Figure 12 shows that only the latest service provider version is actually deployed. All consumers go through the gateway mediation, where it is determined whether or not messages have to be transformed to satisfy the contract they were built for. The mediation uses metadata stored in the registry to make this determination. The transformation itself is done through XSLT.[3]

---

[3] Note that the relevant registry content is cached so that no retrieval and compilation of the XSLT is required for every request.

Putting this solution into the larger context of the use case described in "Use case scenario" on page 11, we can place the Service Versioning Gateway into the mix of components as shown in Figure 13.
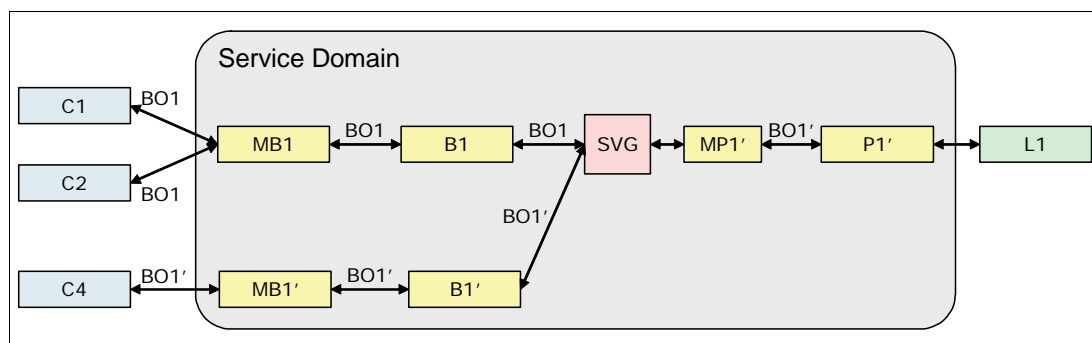

Figure 13   Placing the Service Versioning Gateway into the component view

Putting the Service Versioning Gateway in place allows you to update the P1 and MP1 components with the new schema for BO1 and avoids having to include both P1 and P1' in production. At the same time, the existing consumers C1 and C2, and components MB1 and B1, remain unchanged.

An alternative location for the gateway component is even closer to the consumers, as shown in Figure 14.
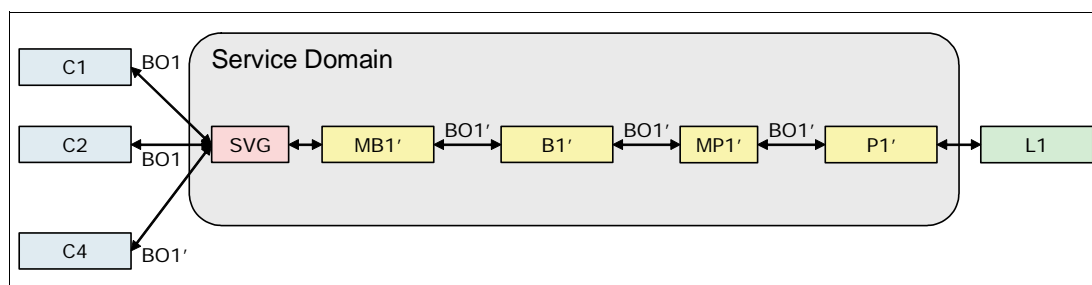
Figure 14   Placing the Service Versioning Gateway close to consumers

Although this setup promises to maximize reduction of the number of existing components in production, it might not always be ideal. In this case, we assume that B1 can be replaced by B1', even though that is not always necessarily the case. B1 might include processing that is based on BO1 and which cannot easily be replaced. The Service Versioning Gateway must support being placed anywhere in the environment, and the decision about where it goes is made based on the concrete situation.

High-level design

The high-level flow that occurs within the Service Versioning Gateway is shown in Figure 15.
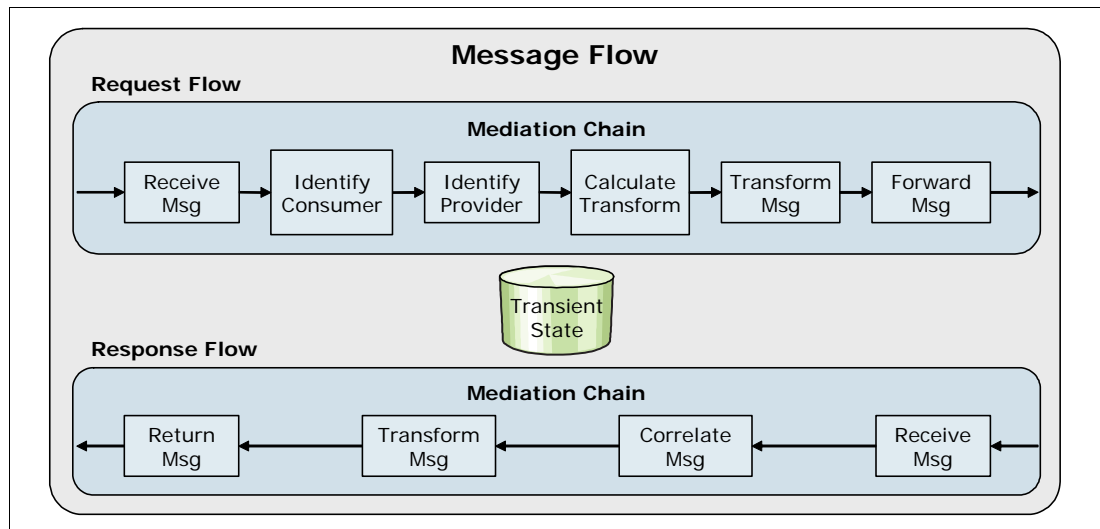

Figure 15   High level flow

Here we list then describe the component operations within the message flow and their responsibilities in this process:

► Receive Msg / Forward Msg / Return Msg
► Consumer Identification
► Provider Identification
► Transformation Identification
► Transformation
► Request and Response Correlation

Receive Msg / Forward Msg / Return Msg

Support for receiving and sending messages over a variety of protocols is fully supported by WebSphere ESB so that no or only limited custom code is needed for these components. They are represented by predefined mediation primitives in the mediation flow whose interfaces are in turn linked to Export and Import components in the mediation module's Service Component Architecture (SCA) assembly.

Consumer Identification

The Consumer Identification operation determines the version of the service consumer that originated an incoming request message. To be more specific, this is the version of the service that the consumer is invoking or the version of the service contract that this consumer was built against.

There are several implementation alternatives for this determination. For example, each message can be expected to include a **version** element that contains the version number of the service (or the default to 1.0 in cases where the element is missing). This requires that each service includes a common element definition in its contract that carries this number. In the case of a SOAP-based service, this element might be in the SOAP Body or the SOAP Header part of the envelope. Although this way of identifying a service version is relatively straightforward, it often requires that the consumer code actively inserts the correct version number into the message.

An alternative to this method is to embed the version number in the SOAP action header definition in the WSDL binding definition. Consider the extract of a RetrieveCustomer service definition shown in Example 1.

Example 1   RetrieveCustomer service definition

```
<wsdl:binding name="WS_Export_RetrieveCustomerHttpBinding"
              type="this:RetrieveCustomer">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="retrieveCustomer">
    <soap:operation soapAction="CustomerService/Version1.0"/>
    <wsdl:input name="retrieveCustomerRequest">
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="retrieveCustomerResponse">
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Service proxies that are generated against this WSDL will automatically include the correct SOAP action header in the request message, without any manual coding required. A downside of this approach is that certain SOAP engines might reserve the use of the SOAP action header for their own purposes (the WebSphere SOAP engine does not). Also, this option requires a SOAP binding.

Another alternative for identifying the version of the service consumer is to append the version number to the endpoint address that is used to invoke the service. Take, for example, the following WSDL endpoint definition:

```
<soap:address location="http://MyCo.com/CustomerService/sca/WS_Export/v1.0"/>
```

Here, the last part of the URL used is not actually indicating a concrete HTTP endpoint; instead it carries the version number. This approach requires that you use a SOAP engine that supports extracting this part of the URL (as WebSphere ESB does).

There are other options on top of the ones mentioned above. Later, we will show how our implementation of the Service Versioning Gateway executes an XPath statement against the incoming message to find the right version. This XPath can select a version based on any differentiating element in the message, be it an explicit version element, the SOAP action header, or any other part of the message.

In any case, it is important that a consistent approach towards consumer version identification is defined as part of your service versioning strategy early on. For the examples described later, we will use the SOAP action header.

Provider Identification

The Provider Identification operation retrieves the identity of the service provider the message will eventually be forwarded to. This includes calculating the appropriate target endpoint address.

Transformation Identification

The Transformation Identification operation determines the appropriate transformation that is to be executed, based on the identified consumer version and target service provider.

Transformation

The Transportation operation handles the execution of the actual transformation. Transformation maps are represented in XSLT. They can be built by mapping the input format (for example, the "older" version) into the output format (for example, the "newer" version) for the request message and the reverse for the response message. There might be cases where only a request transformation is required, and others where only a response is needed, or both.

Request and Response Correlation

The operation described in "Transformation Identification" on page 16 is executed only once and returns the appropriate transformations for both the request and the response flow. The information is stored in the context of one service invocation and can be correlated when a response message arrives. WebSphere ESB offers ready-for-use support for this correlation.

Implementation alternatives

The design for the Service Versioning Gateway outlined in the "Service Versioning Gateway" on page 13 can be implemented in a variety of ways. You can build your own custom implementation or develop it for a IBM ESB offering, namely WebSphere ESB, WebSphere Message Broker, or the WebSphere IBM DataPower® SOA Appliance. They all support the scenario and design described here equally well. Note that there are well-defined criteria for selecting one over the other, but describing those criteria is beyond the scope of this paper.

There are various ways to define and store the information about which transformations apply to which consumer-provider pair and how to detect the consumer version after a request message arrives. You can store this information in a database or in a collection of files and retrieve and process them through custom code at run time. The option chosen for this implementation is to use a service registry to store this information. Using the registry in this way prevents having to develop a lot of custom code because the retrieval of information that is stored in the registry from within the ESB is a well-defined and well-supported pattern. In fact, you will see later that the chosen implementation using WebSphere ESB and WebSphere Service Registry and Repository does not require a single line of Java code.

Implementation with WebSphere ESB and WebSphere Service Registry and Repository V7.5

In this section, we walk you through a concrete implementation of the Service Versioning Gateway, based on the design outlined above. We will use WebSphere ESB V7.5 and WebSphere Service Registry and Repository V7.5.

The additional material supplied with this Redpaper provides a Project Interchange file which contains all of the components described in this section. Import this file into an IBM Integration Designer workspace to see the solution described in this section in more detail.

The additional material is available for download here:

ftp://www.redbooks.ibm.com/redbooks/REDP7447

> **Note:** The exact same implementation can be executed in version 7.0 of both products. In other words, we are not explicitly using any functionality that was introduced in version 7.5.

Example service: RetrieveCustomer

The service that we will use to validate that the implementation actually works as designed is called RetrieveCustomer. The service contains only one operation, also named retrieveCustomer. Not surprisingly, this operation returns a Customer object that contains information like the name and the address of a customer. Figure 16 shows what the service interface looks like. (This picture is taken from the IBM Integration Designer Interface Editor.)
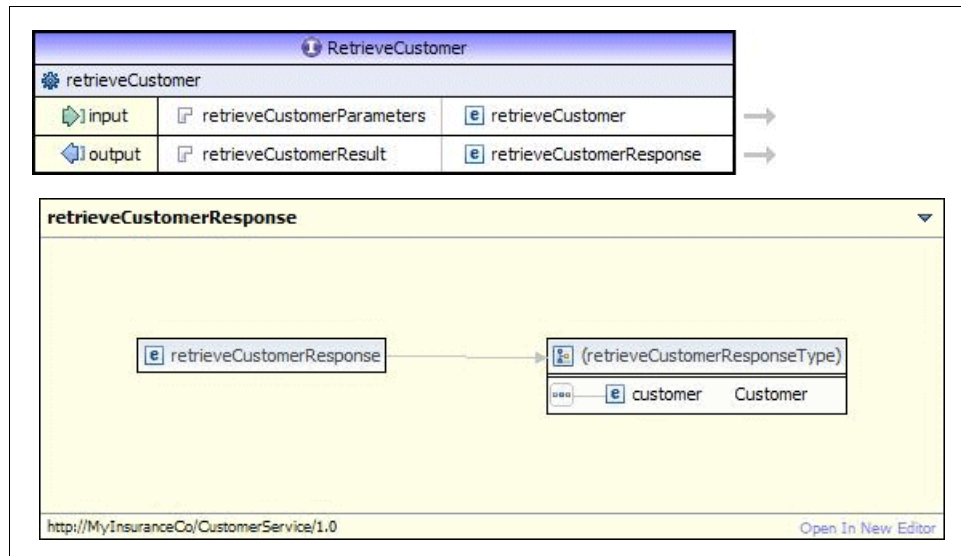

Figure 16   The RetrieveCustomer service interface

The Customer object that is returned has the XML schema definition shown in Figure 17.
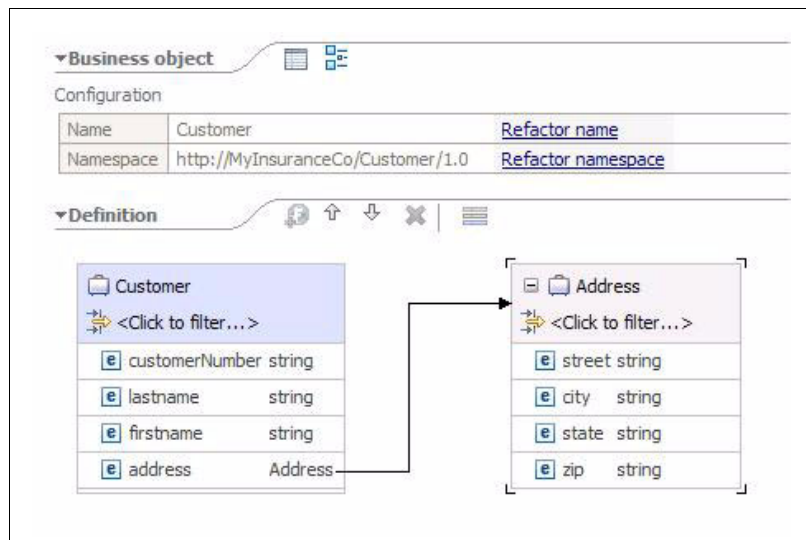
Figure 17   Customer business object

The target namespace for the schema is `http://MyInsuranceCo/Customer/1.0`. The last part of the name includes the major version number for the schema, in this case, 1.0. One of the elements in the schema, address, has a complex type definition that is defined in a separate XML schema. Its target namespace is the same.

The request message, which is not shown here, is a simple parameter named customerNumber, which is of type **string**.

In our scenario, a new consumer for the RetrieveCustomer service comes along. This new consumer is quite happy with the interface that is provided, with the exception that an additional field is required in the Address object. Because this additional field is not expected by the existing consumers, simply replacing the existing service with a new version that returns additional fields is not possible.

Thus, a new major version of the Address business object schema must be created. This has a ripple effect because now also the containing Customer schema and finally, the RetrieveCustomer service itself, must receive new major version numbers.

Figure 18 shows the new version of the schema.
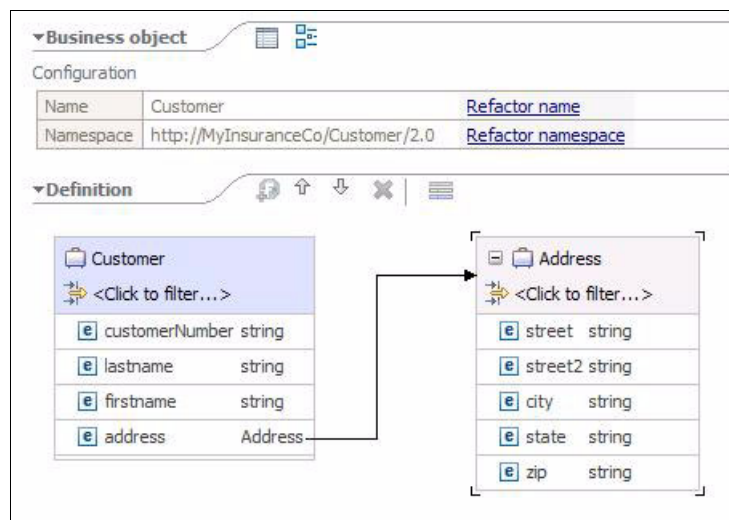
Figure 18   Version 2.0 of the Customer business object

Note that the target namespace has now changed to `http://MyInsuranceCo/Customer/2.0` to reflect the new major version. The only difference to the previous version is that the Address type now has an additional field called **street2**. The containing WSDL definition is no different than before, but also has a new target namespace because it now uses version 2.0 of the Customer and Address schemas.

The request part of the service does not change; it continues to be a string called customerNumber.

If it were not for the Service Versioning Gateway, our sample company would now have to deploy and maintain two services in production, one for the existing consumers and one for the new one that requires the additional address field.

Transformation mappings

The Service Versioning Gateway will execute XSLT-based transformations to adjust request and response messages according to their respective version needs.

A Word on Namespaces

Before we move on to look at the actual XSLT mapping, let's briefly revisit the relevant namespaces that we are dealing with.

There are two types of namespaces that are important when looking at a service contract in WSDL: the target namespaces of the WSDL elements (such as the port type, the binding, and so on) and the target namespaces of the schema elements that are used in the WSDL.

WSDL namespace

Each WSDL definition has a target namespace. A service contract can be contained in one WSDL file with one <definition> element and then have only one namespace. It can also be split into multiple files, each with a <definition> element and possibly a separate namespace. However, it is recommended to have one namespace per service, regardless of how many separate WSDL files exist.

If you create your WSDL definitions through the editors and generators in IBM Integration Designer, you will have only one namespace. In our sample, the WSDL namespace of the RetrieveCustomer service is `http://MyInsuranceCo/CustomerService/1.0`, and the updated version is `http://MyInsuranceCo/CustomerService/2.0` as its namespace.

XML Schema namespace

Quite similar to WSDL definitions, each XML Schema has a target namespace, which covers the type and element and other declarations within the <schema> element. We've already seen how the version 1.0 Customer business object, for example, uses `http://MyInsuranceCo/Customer/1.0` as its namespace.

The services we define follow the so-called "wrapped document literal" style. In this style, the WSDL definition contains a wrapper element for each of the service's operations. Take, for example, the RetrieveCustomer service definition. The service includes an operation called retrieveCustomer, for which there is a corresponding element defined, as shown in Example 2.

Example 2   WSDL retrieveCustomer

```
<wsdl:definitions
  name="RetrieveCustomer"
  targetNamespace="http://MyInsuranceCo/CustomerService/1.0" ...>
    <wsdl:types>
    <xsd:schema
      targetNamespace="http://MyInsuranceCo/CustomerService/1.0">
      <xsd:import namespace=http://MyInsuranceCo/Customer/1.0
                 schemaLocation="Customer.xsd"/>
    <xsd:element name="retrieveCustomer">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="customerNumber"
                      nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    ...
    </xsd:schema>
    </wsdl:types>
...
</wsdl:definitions>
```

Note that the definition imports the Customer business object schema with its namespace. However, the wrapper element is defined locally, with a target namespace of http://MyInsuranceCo/CustomerService/1.0 (highlighted in the example). This is the same namespace the entire service uses as its target namespace.

Thus, there are two sets of XML namespaces at play in a conventional WSDL definition: the target namespace of the service, which is also used by the wrapper elements, and the namespaces defined by the business objects used in the input and output messages of the service. Both types of namespaces will appear in the mappings because both are versioned.

Request Mapping

The request mapping in our example is simple because the request parameter does not change at all between the old and the new versions. So why do we need a mapping at all? It is because the target namespace of the service has changed, so has the target namespace of the wrapper element (retrieveCustomer). This means that the request mapping does nothing but change the namespace of the request message body from `http://MyInsuranceCo/CustomerService/1.0` to `http://MyInsuranceCo/CustomerService/2.0`. See Figure 19 for an example.
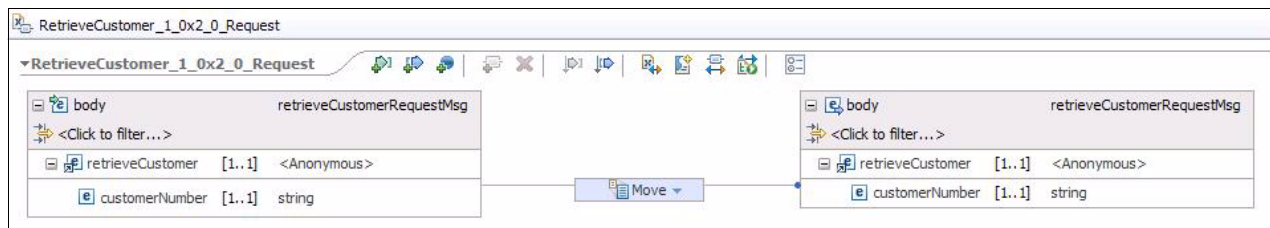
Figure 19   Request mapping from RetrieveCustomer Version 1.0 to RetrieveCustomer Version 2.0

Response Mapping

The mapping of the response message is more interesting, of course, because the change in the format of the Customer business object is what triggered the version change in the first place. See Figure 20 for an example.
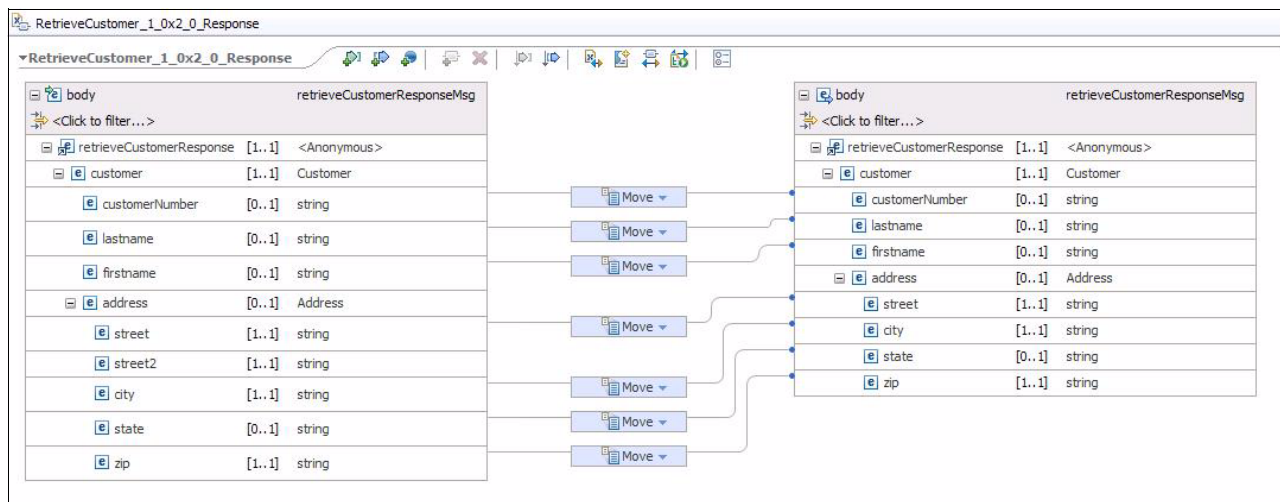
Figure 20   Response mapping from RetrieveCustomer Version 1.0 to RetrieveCustomer Version 2.0

The picture shows how all of the fields of the incoming message are mapped to the outgoing message, with one exception: the street2 field is not mapped. By not mapping it, street2 is effectively removed from the message. The namespaces are also changed (the picture does not show that explicitly), namely from `http://MyInsuranceCo/CustomerService/2.0` to `http://MyInsuranceCo/CustomerService/1.0` and from `http://MyInsuranceCo/Customer/2.0` to `http://MyInsuranceCo/Customer/1.0`, respectively.

As a result, after running this mapping, the outgoing message matches the response message expected from version 1.0 of the service, even though version 2.0 has been executed at this point.

ServiceVersionGateway module

We are now ready to build the module in WebSphere ESB V7.5 that will contain the implementation of the Service Versioning Gateway. We are taking advantage of a feature of the product called the *service gateway*. This feature allows building a mediation that uses an untyped interface and therefore can serve many separate service messages. It simply declares the message element of any incoming message as an `xsd:anyType`. See Figure 21 for an example.
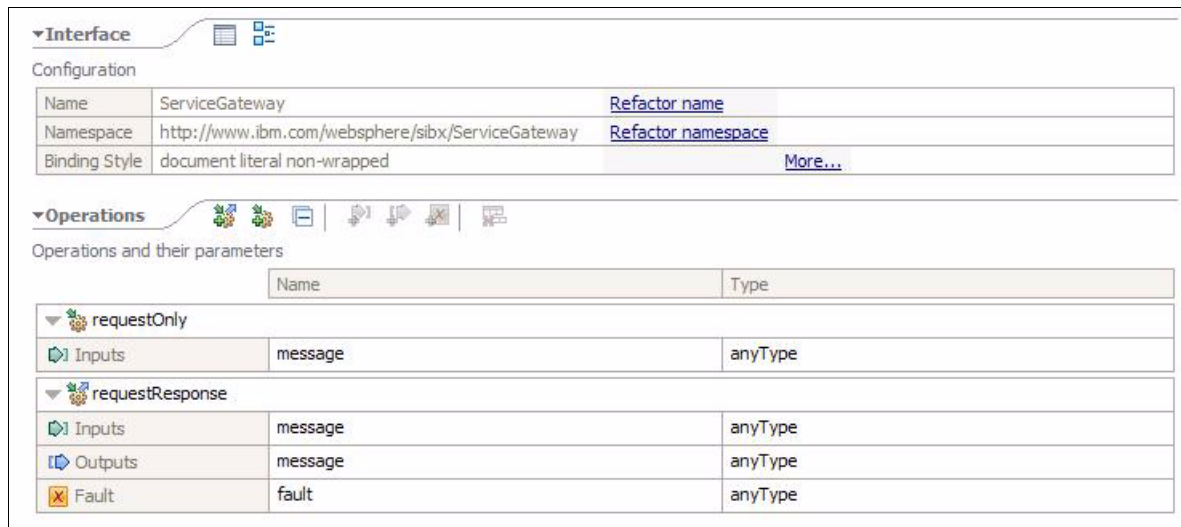
Figure 21   ServiceGateway interface

Figure 22 shows the Service Gateway module assembly. In the assembly, we see a mediation flow component, which contains the logic to transform the messages, and the module's Import and Export. Both the Import and Export have a SOAP/HTTP web service binding so that we assume, in this case, that the consumer uses SOAP over HTTP to communicate with the service provider.

We can also generate a Java Message Service (JMS) binding for either the Import, Export or both. However, in that case, we cannot use the SOAP action header as the identifier of the consumer version because there is no SOAP message to begin with. Instead, we have to use another mechanism, for example, a JMS header field that has a version number in it.

We cannot use SCA bindings at all, because the WebSphere ESB service gateway feature does not support this binding style yet.

Figure 22   Service Gateway assembly

This takes care of the "Receive Msg / Forward Msg / Return Msg" step of the high level design as described in "High-level design" on page 15.

ServiceVersionGateway mediation flow component

We are now ready to build the flow that executes the remaining steps, namely the consumer identification and transformation.

Request flow

Figure 23 shows an example of the request flow.
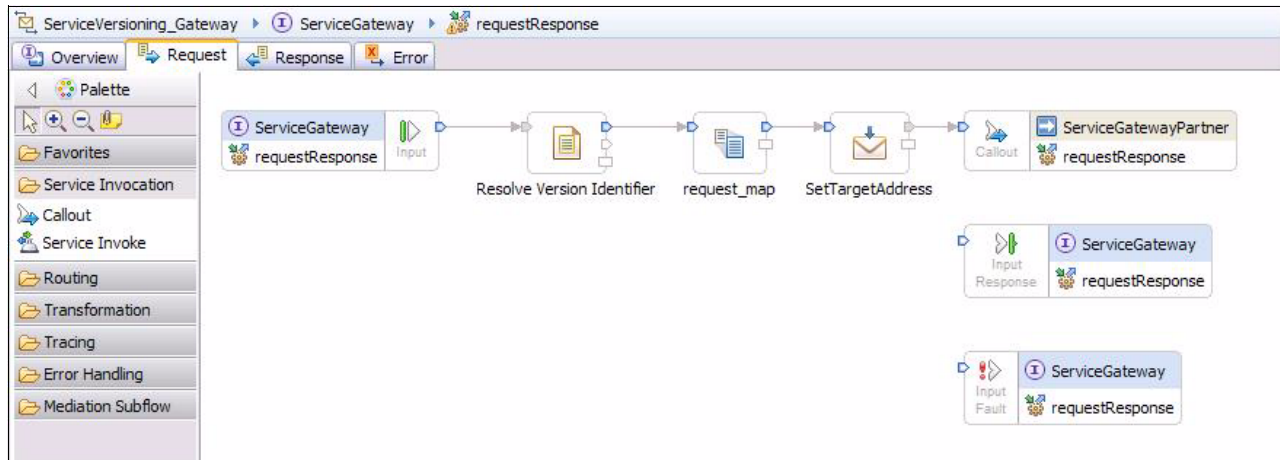
Figure 23   Service Versioning Gateway request flow

The first step in the flow, labeled Resolve Version Identifier, represents identification of the consumer version. For this step, we are using a mediation policy, which we explain in detail in "Mediation policies" on page 27. Effectively, this primitive will retrieve a set of dynamic properties from the service registry, that is, WSRR.

The next step, labeled request_map, is the actual XSLT transformation. The name of the stylesheet that is executed by this primitive is a configurable property. By promoting this property, we allow it to be changed at run time. See Figure 24.

Figure 24   Promotable properties of the XSLT transform primitive

Note that we set the name of the stylesheet to `empty.xsl`, which is obviously not the transformation we want to perform at run time. Even though we will overwrite the value of this property at run time, the module will check at startup time that the default value contains the name of a stylesheet that actually exists, so that a dummy XSLT stylesheet named `empty.xsl` is included in the mediation module.

There are two ways of overwriting a promoted property at run time: it can either be set through the WebSphere ESB administrative console, or it can be changed for an individual message by setting it in the service message object (SMO). Example 3 shows an extract of an SMO that contains a dynamic property for the request XSLT transformation shown in Figure 24 on page 24.

Example 3   SMO extract

```
<smo …>
  <context>
    <dynamicProperty isPropagated="true">
      <propertySets>
        <group>
            ServiceVersioning_Gateway.ServiceVersioning_Gateway
        </group>
        <properties>
          <name>request_map.XSLTransform</name>
          <value>
              xslt/RetrieveCustomer_1_0x2_0_Request.xsl
          </value>
        </properties>
      </propertySets>
    </dynamicProperty>
  </context>
…
</smo>
```

Setting the `isPropagated` attribute to `true` will ensure that the dynamic property will also be set on the response message.

We can set this context programmatically from within the mediation flow component, using, for example, a custom mediation primitive. In our example, however, we leave it to the mediation policy primitive to do this, and will explain in detail later how that works.

The last primitive in the request flow is labeled SetTargetAddress. It sets the endpoint address of the actual service provider. This step is important because after all, we are building a service gateway that handles many services, and it has to determine the address of the service provider for each request message. The endpoint address for a service invocation is set in the `/headers/SMOHeader/Target/address` element of the SMO, as shown in Figure 25.
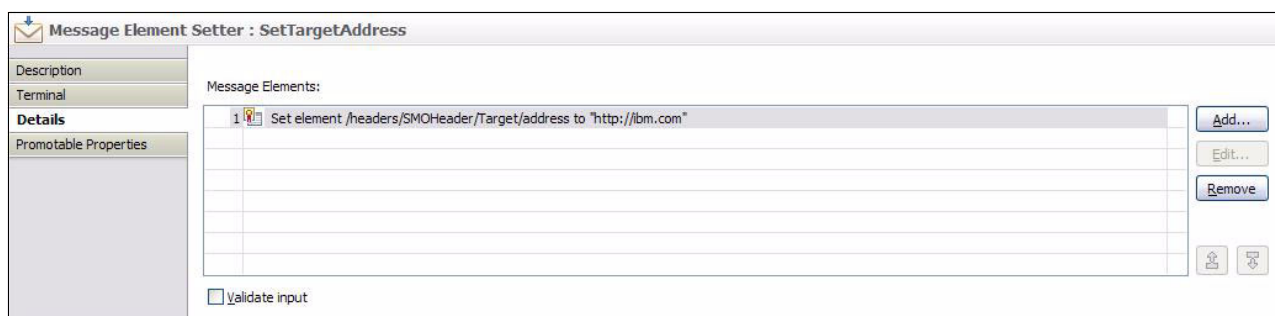
Figure 25   Setting the SMO endpoint address

In Figure 25 on page 25, the endpoint address is set to a default value. We can now set it to the correct value for each message by using the same mechanism we have used for the XSLT stylesheet name: we promote the property and set it programmatically in the SMO. See Figure 26.
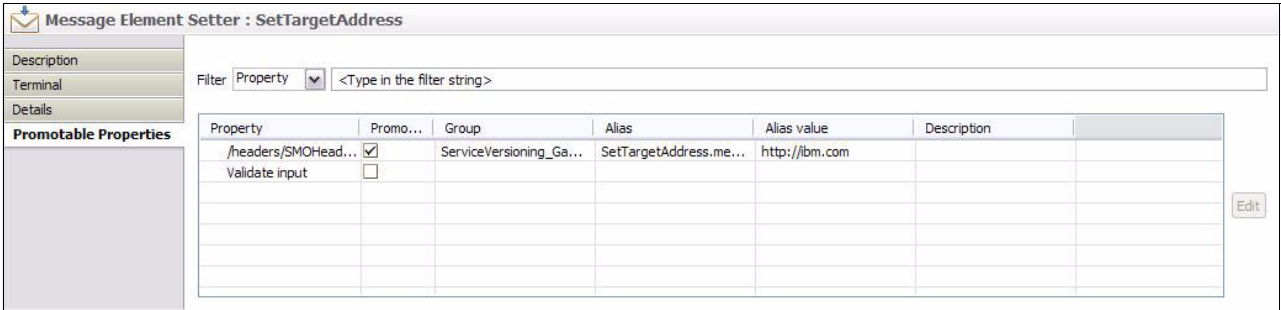
Figure 26   Endpoint address as a promotable property

We set the right value as a dynamic property using the same mediation policy primitive.

Response flow

Figure 27 shows an example of the response flow.
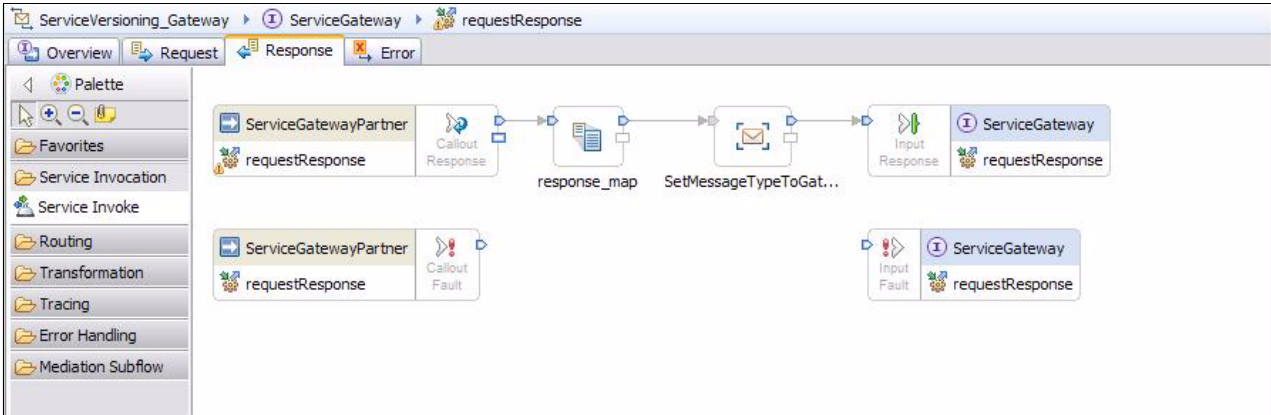
Figure 27   Service Versioning Gateway response flow

We do not need another mediation policy retrieval step here because we have specified that all dynamic properties are promoted to the response flow. Hence, we can go straight to the transformation of the response message. The mechanism is exactly the same as before, namely that we promote the "Mapping file" property of the response_map primitive and set the right value at run time.

This means that we now have to set three dynamic properties for the component:

► The name of the mapping file for the request message
► The endpoint address to which the request message will be forwarded
► The name of the mapping file for the response message

All three properties are set in the SMO context as shown in "Request flow" on page 24.

The final primitive in the mediation flow component is labeled `SetMessageTypeTo-Gateway`. It sets the message coming out the transformation (which is "any") to the proper message type used by the service gateway. See Figure 28.

Figure 28   Setting the message type to gatewayMessage

Mediation policies

The last remaining piece of the puzzle is the "Identify Consumer" step. In the mediation flow component, this step is represented by the mediation policy primitive labeled "Resolve Version Identifier". Explaining what this primitive does deserves a more detailed explanation.

Policy standards: WS-Policy and WS-PolicyAttachment

Two core standards exist that allow defining policies for web services, namely WS-Policy and WS-PolicyAttachment.

For more information about WS-Policy, see the following website:

http://www.w3.org/TR/ws-policy/

For more information about WS-PolicyAttachment, see the following website:

http://www.w3.org/TR/ws-policy-attach/

The first standard defines a generic language for policy assertions to be expressed in XML, and the second standard defines how these policy assertions can be associated with elements of a service contract. These standards enable policy documents to be attached to more than one service, and more than one policy can be attached to a service.

Changes to a policy do not require an update to the service contract and vice versa. Figure 29 shows the relationship between service contract, policy, and policy attachment.
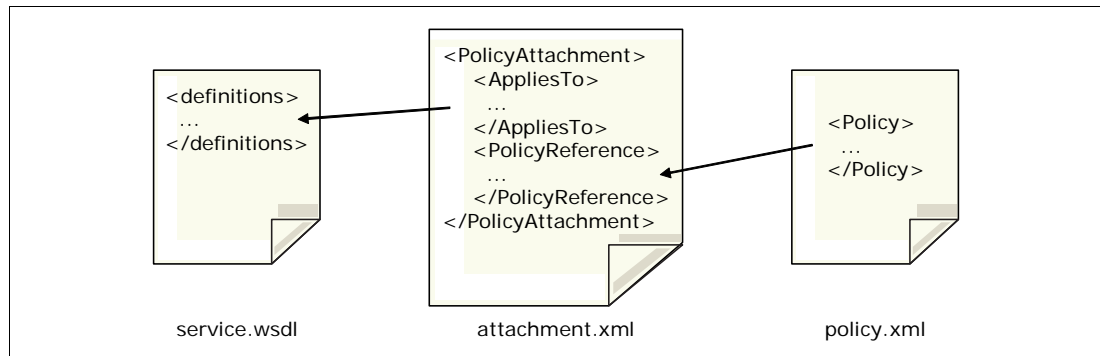
Figure 29   Relationship of service contract, policy, and policy attachment

A policy assertion contains basically any type of key-value pair in XML, or a combination of nested policies. Describing the details of the relevant standards is beyond the scope of this paper.

Policy support in WebSphere Service Registry and Repository

The WebSphere Service Registry and Repository (WSRR) product offers first-class support for documents following the WS-Policy and WS-PolicyAttachment standards. It supports loading and maintaining these documents and applying governance lifecycles and relationships to other entities in the registry. A policy can be attached not only to a WSDL service contract, but also to an entire SCA module.

Moreover, policies and policy attachments can be edited directly using the WSRR administrative console.

Mediation Policy primitive in WebSphere ESB

WebSphere ESB takes advantage of the support for policies in WSRR to allow overwriting promoted properties of a mediation flow component at run time. As mentioned earlier, a WS-Policy contains assertions. and WebSphere ESB translates assertions that are stored in a policy into dynamic (promoted) properties. How does it know which policy to retrieve from the registry? There are two criteria:

► The policy must be attached to the SCA module in which the mediation flow component is contained. To do this, the SCA module must be loaded into the registry, and a policy attachment must exist that connects the SCA module with the policy.

► If more than one policy is attached to the module, then a *gate condition* is used to select the right one. The gate condition is based on an XPath statement that is executed against the SMO. In other words, the actual content of each message determines which policy is applied. Effectively, this means that the content of the message is used to define how certain promoted properties are set.

We are using this functionality in the service gateway to define the appropriate transformations to be executed and the endpoint where the message should be sent. Figure 30 shows an example.
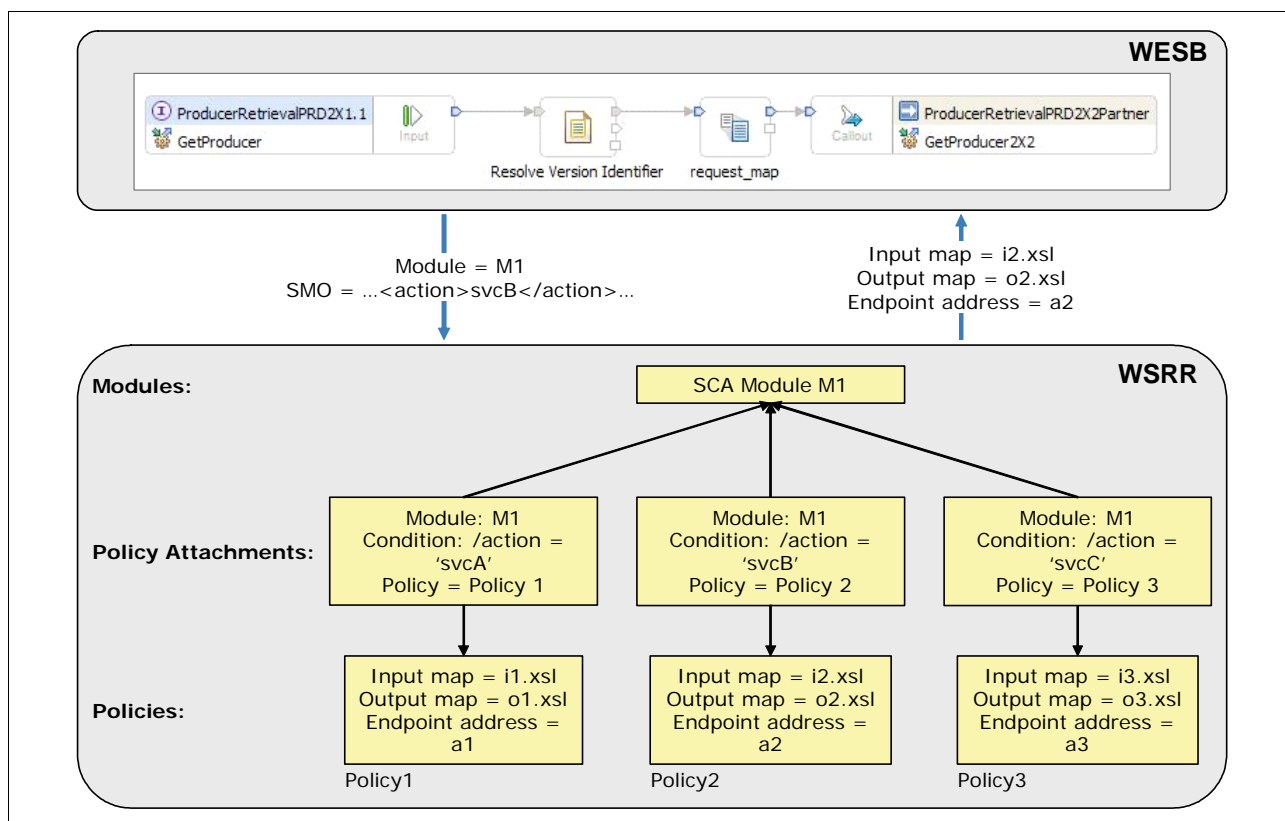
Figure 30   Retrieving transformation map names from WSRR

In Figure 30, we see that several elements exist in the registry:

► The SCA module (here called M1).

► Three policies (Policy1, Policy2, and Policy3). Each policy has three assertions, one each for the input map, the output map, and the endpoint address.

► Three policy attachments. These attachments link each of the policies to the module. Moreover, each attachment has a "condition." Each attachment condition contains an XPath statement (/action) and a value (svcA, svcB, and svcC).

At the top of Figure 30, you can see that the mediation flow executed in WebSphere ESB contains a mediation policy primitive labeled Retrieve Version Identifier. The primitive defines an XPath statement for the gate condition and, during execution, executes that statement against the incoming SMO. In our example, the gate condition determines the value of /action. It then sends a query that contains the module name and the value of the gate condition statement. In the example, that value is svcB.

The registry retrieves the appropriate policy using the condition defined in the policy attachment. In the example, Policy2 is selected, and its assertions are returned to the mediation flow component. Finally, the assertions are transformed into dynamic properties stored in the SMO. The input map is used by the request_map primitive to determine the right transformation to be executed. The endpoint address is used by the SetTargetAddress primitive to send the message to the correct service provider endpoint, and the output map is used to execute the transformation of the response message through the response_map primitive in the response flow.

Putting It All Together

Now that we've covered the theory of mediation policies, we are ready to build and deploy the mediation, load the appropriate artifacts into the registry, and run the service gateway.

Defining the mediation policy

We have already shown the makeup of request and response flows. What is still missing is the mediation policy primitive. Figure 31 shows the detail settings for this primitive.
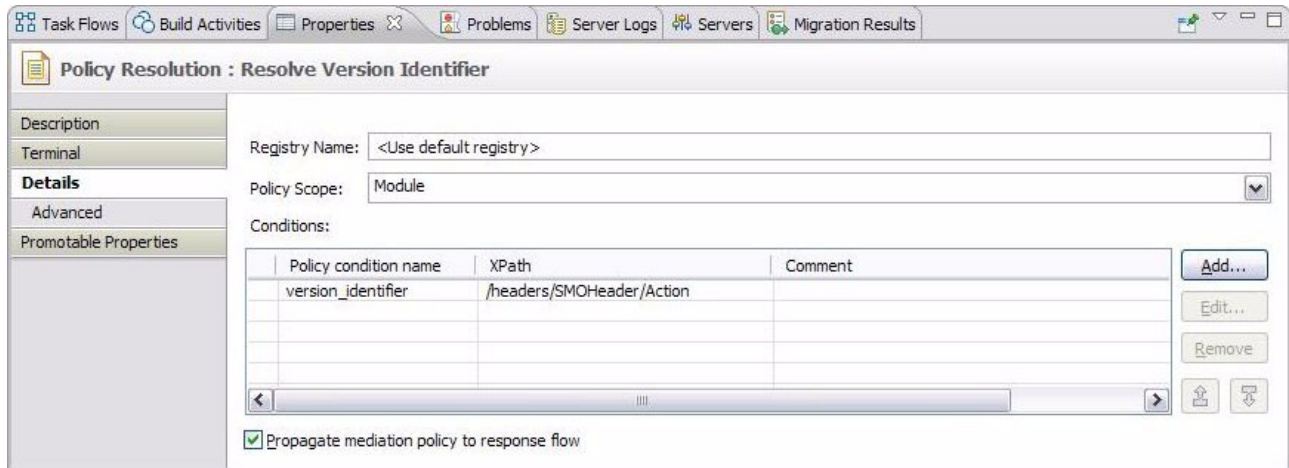
Figure 31   Mediation Policy primitive properties

The following properties are set for each primitive:

► Registry name
► Policy scope
► Conditions
► Propagate mediation policy to response flow

Registry name

The Registry name property is set to the default registry. You can define multiple WSRR instances in the WebSphere ESB administrative console. In Figure 32, you see the connection properties for a secure connection to a locally deployed registry. Note that to establish a secure connection, you might have to import the registry's security certificates.
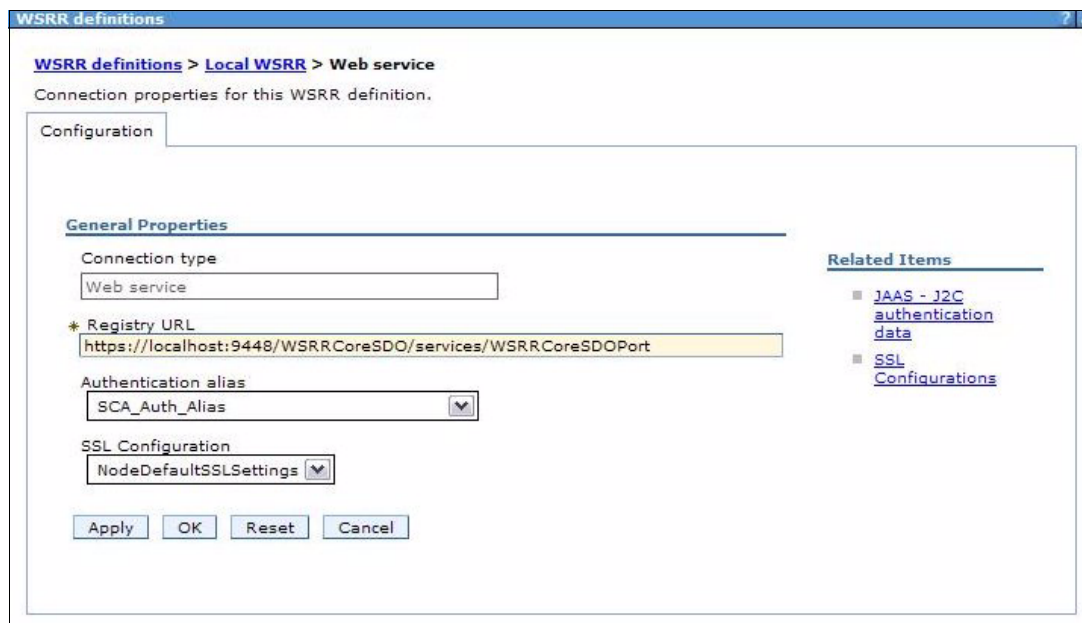
Figure 32   WSRR definition properties for WebSphere ESB

Policy Scope

Leave the Policy Scope setting to Module scope. It defines the scope of the query that is sent to the registry.

Conditions

In the Conditions property, we define the gate condition that will allow selecting the appropriate policy. The XPath statement `/headers/SMOHeader/Action` points to the content of the SOAP action header of the incoming request message. Remember how this header is set in the WSDL definition of the service, as shown in "Consumer Identification" on page 15.

The policy condition name is used in the policy attachment, which we will create later.

Propagate mediation policy to response flow

Setting this flag allows us to retrieve the policy only once per service invocation because the resulting dynamic properties are set in the request flow and restored for the response flow.

Export and deploy the module
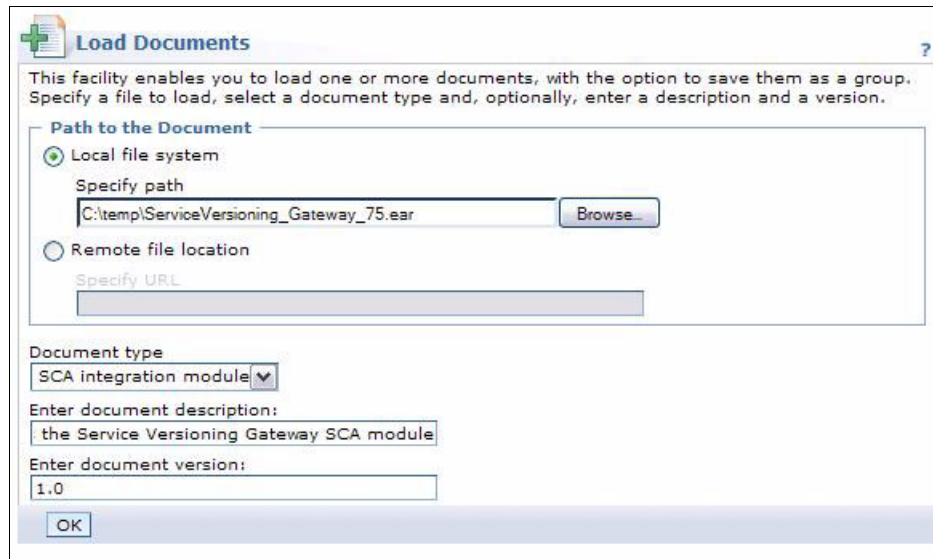
The mediation module for the service gateway is now ready to be deployed. However, before we do that, we have to export it for so it can be loaded into the registry. In the Integration Designer tool, use the **Export → Business Integration → Integration modules and libraries** path to create an enterprise archive (EAR) file that contains the complete SCA module.

You deploy the Service Versioning Gateway module to a target WebSphere ESB server just like you would deploy any other module.

Loading the module into the registry
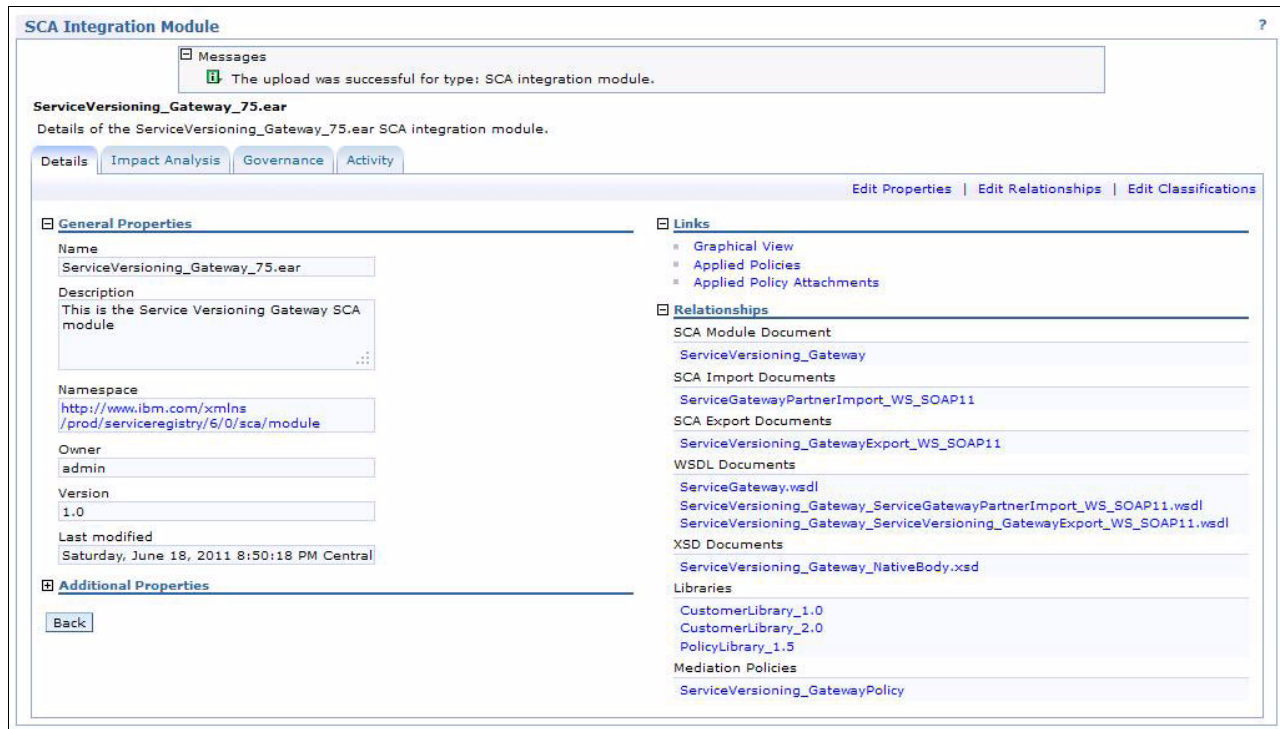
The next step is to load the SCA module with the gateway into the registry. You can use the Load Documents option, as shown in Figure 33.

Figure 33   Loading the SCA module

Make sure the Document type is set to SCA integration module. Figure 34 shows how the contents of the Service Versioning Gateway are now loaded into the registry. There are multiple libraries, WSDL and XML schema, and one mediation policy. This policy is a default policy that is created by the tool, but is not actually attached to the module. We do not use this default policy at all.

Figure 34   The SCA module loaded in the registry

Creating policies and policy attachments

Finally, it is time to create the policies that will tell the gateway which transformations to perform on a message at run time. The simplest way to do this is to use the Business Space that comes with WebSphere ESB, and within that Business Space, the Service Administration page.[4]

In Figure 35, you can see how three modules are loaded in the WebSphere ESB server. Two of them are actual service providers, and one is the Service Versioning Gateway.
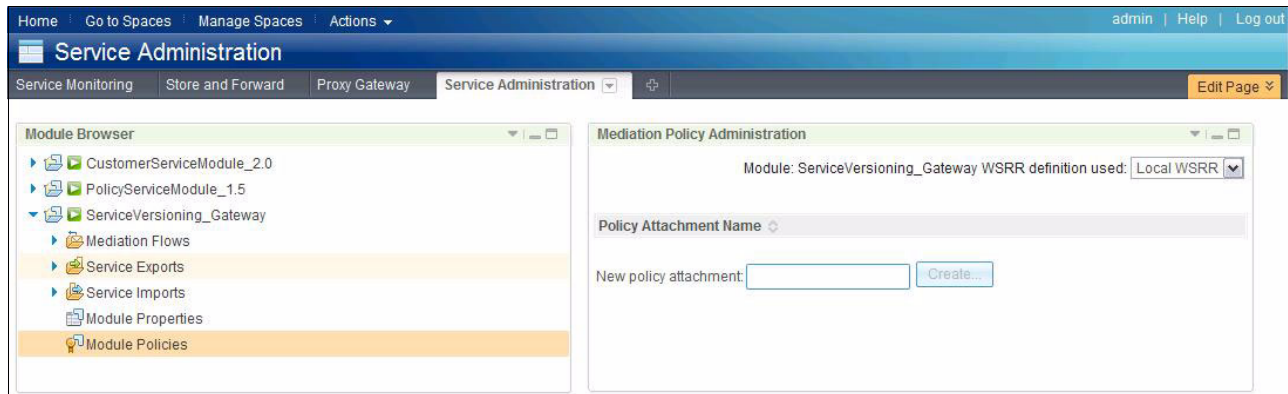

Figure 35   WebSphere ESB Business Space

When selecting the Module Policies link for the gateway module, we see that there are no policies or policy attachments defined yet. In order to create a policy and policy attachment, we can use the Mediation Policy Administration widget on the same page.

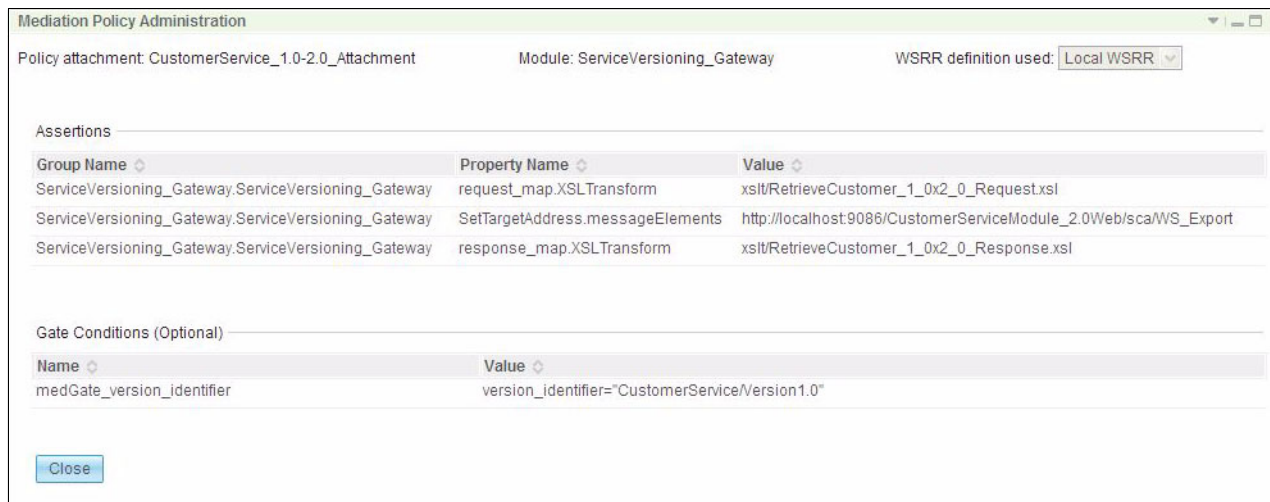Figure 36 shows a completed policy and policy attachment pair for the gateway module.

Figure 36   A new policy and policy attachment

The values used here are all directly related to other definitions and settings we made earlier in the module.

---

[4] Note that both WebSphere ESB and WebSphere Service Registry and Repository come with a Business Space application. Here, we use the WebSphere ESB Business Space application.

Group Name

The Group Name value relates directly to the group name of the promoted properties that we defined in the mediation flow component, as shown in Figure 24 on page 24.

Property Name

The Property Name value relates to the Alias name of the property, which you can also see in Figure 24 on page 24.

Value

.The Value field must equal the value to which we set the promotable property if the containing policy is retrieved.In the example shown in Figure 36 on page 33, we define two XSLT stylesheets and the endpoint address of the PolicyService service provider (which, in this case, is deployed locally on the same server as the gateway).

Note: The stylesheets defined here *must* exist in the gateway module. The XSLT transformation primitive we use for the implementation cannot load the stylesheet from an external source, so we copy it into a library that is included in the gateway module.

Gate Conditions (optional): Name

This value is set to "medGate_" with the name of the condition appended to it. Remember we set the name of the condition to "version_identifier" (see Figure 26 on page 26), thus the name specified here is `medGate_version_identifier`.

Gate Conditions (optional): Value

The value of this field has the name of the condition again, plus the result of the XPath statement (which we defined in the mediation policy primitive when applied to the SMO. See Figure 26 on page 26.

In our concrete example, we have defined the gate condition XPath statement to point to the SOAP action header of the request message. Here, we specify that if that header field contains `CustomerService/Version1.0`, the current policy should be returned.

At this point, things all come together. We have a request message with a SOAP action header. This header refers us to the policy attachment through the gate condition. The policy attachment points to the policy. In the policy, we define values for promoted properties. The mediation flow component picks up these properties at run time and executes the right transformations for both the request and response message and uses the appropriate endpoint address for the service provider.

We can now support many service consumers and many service providers and their respective version-to-version transformations. The Service Versioning Gateway is complete.

Service Versioning with the Governance Enablement Profile in WSRR V7.5

Before we conclude this paper, we want to point out an additional level of support for service versioning that is available in the WebSphere Service Registry and Repository (WSRR) V7.5. The product comes with a predefined "Governance Enablement Profile," which defines a fine-grained service meta-model that includes the definition of entities as described in "Service versioning" on page 3. It allows modeling governed services, service descriptions, service versions, and so on and linking these entities to the appropriate artifacts like WSDL or XML schema.

Furthermore, WSRR V7.5 supports basic versioning of all elements that are stored by storing a version name property with them.

Using the Service Versioning Gateway described here with the Governance Enablement Profile is easy if you add appropriate version numbers to the policies and policy attachments that drive the gateway and making sure that all service artifacts (WSDL, XML schema, and so forth) are given the correct version number. These artifacts should then be linked with the service version descriptions that are part of the Governance Enablement Profile.

Summary

In this paper, we introduced a solution that addresses a common challenge in service-oriented environments, namely that of service evolution or service versioning. We defined the various elements of a "service," how they play a role in supporting multiple versions of that service, and the reasons for creating service versions in the first place.

To reduce the need to support multiple versions of a service in production, which brings with it challenges around maintainability and operability of that service, we then described the Service Versioning Gateway, which is an architectural component that mediates the differences in expectation (that is, the versions) between consumers and providers. This gateway allows limiting the number of service providers while fully supporting new and existing service consumers, using message transformations.

Finally, we showed an actual implementation of the Service Versioning Gateway, which is based on WebSphere ESB V7.5 and WebSphere Service Registry and Repository V7.5. It leverages the service gateway feature supported in WebSphere ESB, combined with mediation policies that are stored in the registry.

The team who wrote this paper

This paper was produced by a team of specialists from around the world.

Andre Tost works as a Senior Technical Staff Member in the IBM Software Services for WebSphere organization, where he helps IBM customers establishing service-oriented architectures. His special focus is on web services and enterprise service bus technology. Before his current assignment, he spent ten years in various partner enablement, development, and architecture roles in IBM software development. Andre is an IBM developerWorks® Master Author.

Greg Flurry is a Distinguished Engineer in the IBM Business Process Optimization group. His responsibilities include working with clients on service-oriented solutions and advancing IBM service-oriented product portfolio. Greg is a developerWorks Professional Author.

Thanks to the following people for their contributions to this project:

- ► Gary Gershon
  Architect, Chubb Insurance Company

- ► Adam Aronoff
  Architect, Chubb Insurance Company

- ► Callum Jackson
  Software Engineer, IBM Hursley, Great Britain

This Redpaper on the web

This Redpaper is available on the web at the following address:

http://www.redbooks.ibm.com/redbooks/REDP4774

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at the following address:

http://www.ibm.com/redbooks/residencies.html

Stay connected to IBM Redbooks

► Find us on Facebook:

http://www.facebook.com/IBMRedbooks

► Follow us on Twitter:

http://twitter.com/ibmredbooks

► Look for us on LinkedIn:

http://www.linkedin.com/groups?home=&gid=2130806

► Explore new IBM Redbooks® publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

► Stay current on recent Redbooks publications with RSS Feeds:

http://www.redbooks.ibm.com/rss.html

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

This document REDP-4774-00 was created or updated on September 28, 2011.

Send us your comments in one of the following ways:
- ► Use the online **Contact us** review Redbooks form found at:
  ibm.com/redbooks
- ► Send your comments in an email to:
  redbooks@us.ibm.com
- ► Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400 U.S.A.

IBM ®

Redpaper ™

Trademarks