Solution design in WebSphere Process Server and WebSphere ESB, Part 2: Solution design in WebSphere Process Server and WebSphere ESB

Designing an ESB Gateway in WebSphere Process Server and WebSphere ESB

Kim J. Clark

August 12, 2009

Introduction

Part 2 of the series looks at the ESB Gateway architectural component. How is it different from the broader ESB concept? Why is it important? It then discusses how to design and implement one in WebSphere® Process Server and WebSphere Enterprise Service Bus.

Part 2 of this series described how service-oriented architectures (SOAs) mature through a number of stages, and showed what solution styles are used at each level. This article focuses on the advancements made moving from Enterprise Application Integration (EAI) to Service Exposition, in other words from Service Integration Maturity Model (SIMM) level 3 to 4.

We will discuss what it means to expose a mature service on an Enterprise Service Bus (ESB), and look at the features you might use from WebSphere Process Server (hereafter called Process Server) or WebSphere Enterprise Service Bus (WESB). Specifically, we'll be looking at the new Service Gateway pattern available in WebSphere Integration Developer (hereafter called Integration Developer) from v6.2 and at some of the new mediation flow component capabilities such as the Policy Lookup primitive.

As an enterprise architecture moves in to SIMM maturity level 4, a selection of key business and technical functions surfaces as candidates for re-usable services. This re-use might be at many different levels, from just within the IT environment, or perhaps wider, at the enterprise level, right through to exposing services on the Internet. As an SOA moves toward SIMM level 4, two key advances are happening:

- **Assessing service candidates**: The right services need to be selected for exposition to a wider audience. Some of this will come from bottom up initiatives, such as the functions that...
have naturally surfaced in SIMM level 3. Conversely, a top down approach may take feeds from previous exercises such as Component Business Modeling. Methodologies such as SOMA combine such techniques to find service candidates. While this step is critical, it is well beyond the scope of this article.

- **Exposing the services**: Introduction of the architecture and design patterns needed to expose these services appropriately. These services need to have special treatment in the way the architecture makes them available. *This is the main focus of this second article.*

So why do true enterprise services need special treatment in the way they are exposed? A service exposed to multiple, potentially unknown consumers will need to be strengthened in terms of security, reliability, maintainability, scalability, and governance. Much, although not all of this, can be done on exposure of the service. Choosing an interoperable standard protocol, such as SOAP/HTTP or REST/HTTP, is only a fraction of what we mean by "exposing" a service.

**How is exposing enterprise services different from traditional integration?**

Part 1 briefly discussed the differences between traditional integration (SIMM level 3) and service exposition. Indeed, this is the defining concept of how on a technical level SOA is different to the EAI that came before it. Let's recall the traditional hub and spoke integration architecture (Figure 1) of SIMM level 3.

**Figure 1. Traditional Integration (SIMM level 3)**

Now, let's remind ourselves how that architecture is amended to expose mature enterprise services (Figure 2) in SIMM level 4.

**Figure 2. Service Exposition (SIMM level 4)**

The primary changes between the two relate mainly to the introduction of the ESB Gateway:

- **Standardized exposition**: The connectors on the requestor side of the hub have disappeared in the SIMM level 4 diagram (Figure 2). This is because we only allow the ESB Gateway to
use standard mechanisms for transmission protocols, security, transactionality, and interaction styles to ensure the service can be used by the largest possible range of consumers.

- **Common request handling**: The ESB Gateway has a specific role in enforcing generic, and ideally, policy-based handling of requests by the ESB Gateway. This allows you to perform aspects such as monitoring, accounting, administration, and diagnostics in a common way for groups of services that will use a shared or common policy.
- **Standardized operation specific request handling**: Some capabilities are similar in style, but different in the specifics for each operation. Where these can be implemented in a standardized way, they may also be considered appropriate for the ESB Gateway. Ideally, these too are configured through operation specific policy in a registry. Examples might be routing, flow/concurrency control, auditing, access control, and semantic mapping.
- **Service registry**: A registry has been introduced. At a minimum, this provides consumers with the information they need to use the service. As noted above, it may also contain policy details used by the ESB Gateway to enable centralized configuration of policies concerning the service exposition.

What's the purpose of the ESB Gateway?

Since this concept of the ESB Gateway is central to exposing services, we're going to focus on that alone for the rest of this article.

**Figure 3. Formalizing the responsibilities of the ESB Gateway**

Notice from Figure 3 how different the intent of the ESB Gateway is from that of the hub: virtualization, visibility, security, and traffic management. It is entirely focused around how the service is exposed, and how all of the capabilities could be standardized and perhaps policy driven. We say "could" because only some of these standards exist today. These are sometimes referred to as aspect oriented capabilities as they apply across a suite of service interfaces. In time, the configuration of many of these will migrate into the service registry.

Looking in a little more detail at the intent of the ESB Gateway:
• Protocol translation (standardized protocols only)
• Versioning

<table>
<thead>
<tr>
<th>Security</th>
<th>Traffic Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Authentication</td>
<td>• Throttling</td>
</tr>
<tr>
<td>• Authorization</td>
<td>• Prioritization</td>
</tr>
<tr>
<td>• Accounting (overlaps with auditing from visibility)</td>
<td>• Load distribution</td>
</tr>
<tr>
<td>• Identity propagation</td>
<td>• Store/forward</td>
</tr>
<tr>
<td>• Encryption</td>
<td>• Re-try</td>
</tr>
<tr>
<td>• Access control</td>
<td></td>
</tr>
<tr>
<td>• Threat protection</td>
<td></td>
</tr>
<tr>
<td>• Accounting</td>
<td></td>
</tr>
<tr>
<td>• Identity propagation</td>
<td></td>
</tr>
<tr>
<td>• Encryption</td>
<td></td>
</tr>
<tr>
<td>• Access control</td>
<td></td>
</tr>
<tr>
<td>• Threat protection</td>
<td></td>
</tr>
<tr>
<td>• Versioning</td>
<td></td>
</tr>
</tbody>
</table>

Virtualization

An exposed service should be virtualized from the consumer's point of view. They should be sufficiently decoupled from its implementation that they do not know where is or how it is implemented.

• **Routing**: This enables requestors to call a central source (the ESB Gateway) and for the gateway to route the request to the ultimate provider such that the provider's location can change without affecting the requestor. There may even be multiple providers that you need to choose between based on geography, time of day, quality of service, and so on.

• **Mapping**: If the provider performs the right function with the right data fields, but the structure of the data model is different, it is acceptable for the ESB Gateway to perform the mapping. However, this is just for mapping of the data model present in a standardized protocol, such as from XML to XML, or XML to JSON. Do not expect the ESB Gateway to perform low level formatting or parsing of non-standard data formats.

• **Protocol translation**: This may be necessary to expose a service in a different protocol to the one its current provider offers. However, note an important restriction of the ESB Gateway here. It only talks in the standardized protocols agreed for the ESB, both for exposition and also for talking to providers. If translation is required to a non-standard protocol, this should not be done by the ESB Gateway. It should be done by the hub or connectors.

• **Versioning**: This is a special case of the capabilities already mentioned. Over time, services will require maintenance or provide new features, but you need to avoid affecting existing consumers. A versioning strategy will make creative use of the routing, mapping, and protocol translation already mentioned. For example, it will allow versions of the service to appear to still exist via the interface mapping between old and new data models, and intelligent routing between old and new versions based on the header information.

**Will all exposed services require all of the ESB Gateway capabilities described?**

The requirements of each individual service operation will be different. Not all services, or indeed all SOAs, require all of these capabilities. Essentially, the more consumers that are likely to use the service, and the more decoupled you are from those consumers, the more these capabilities become relevant and necessary. There is a significant cost associated with the proper exposure of a service, so you need to ensure that a service has a strong case for re-use before you expose it formally.
Visibility
The ESB Gateway's primary purpose is to provide functionality to service consumers. However, there are a number of other interested parties that can gain benefit indirectly from the exposure of services, and for this, they need visibility of the activity happening in the gateway.

- **Business application monitoring (BAM):** This provides you the business with management information directly relevant to their business. If the services are at the right level of granularity, they perform business meaningful actions, and therefore statistics about their activity are of interest to the business. It can be anything from information on the sales of products, to the spread of the workload received by a call center.
- **Service levels:** Nobody will use a service they cannot trust. So your reputation, at least within the enterprise, and possibly beyond, depends on making promises about service levels. To do that with confidence, you need to measure your compliance to them. You're typically interested in response times, throughput rates, and downtime.
- **Diagnostics logging:** This provides information for diagnosing issues with services. Unlike the other mechanisms described here, diagnostics logging is typically configurable at runtime such that it can provide more information (data content), less information (requests made, invocation path), or no information about the requests being made through the gateway. The consumer of diagnostics logging will be the IT-literate operations staff, and as such, the presentation of the data may not be as sophisticated as for the other categories. It may be no more than a log file.
- **Accounting:** This information may need to be captured for any number of reasons. For example, if you charge for the use of your services, you need to know who's using them and how often to bill them appropriately. Note that there is a specialized form of accounting called auditing discussed in the Security section below.

Security
The more re-usable the service, the broader the audience to which it will be exposed. As a result of this, there is a greater opportunity for the service to be misused, whether accidently or maliciously. In summary, the further an SOA matures, the more attention you need to pay to security. The following security characteristics are particularly relevant:

- **Authentication** attempts to validate that the requestor is who they claim to be, by receipt of some form identifier (for example, username/password, or certificate), and to check that with some form of directory (for example, LDAP).
- **Authorization** assesses whether the requestor is allowed to use the service. Depending on the level of security, they may need to be authenticated, or they may need to belong to a particular group or role. For complex authorization scenarios, you can externalize this capability to products such as Tivoli® Access Manager.
- **Auditing** is essential if there is sensitive information, such as customer's personal data and financial transactions passing through your services. You need to know who made requests, what they were trying to do, and when. You may even need to prove the integrity of this data. Often, this is a hard requirement based on regulatory compliance rules.
- **Identity propagation** means that not all domains share the same directories of users. It's possible that all authenticated users are simply translated to a single system user. More
complex translations of user identity from one domain to another are typically performed by an external capability, such as Tivoli Federated Identity Manager.

- **Encryption** of data is vital if sensitive data is being transmitted over a public medium. Encryption may be at a channel level (HTTPS) or at a message level (encryption of the credit card number).
- **Threat protection** against a bewildering variety of attacks may be required such as **denial of service (DoS) and XML threats**. While these may be blocked by components upstream of the ESB Gateway, it's worth assessing whether the new threats posed by newer protocols, such as Web services, have been fully considered.
- **Data validation** goes hand-in-hand with threat protection. Having established that the XML is valid and well-formed, ensure that the content conforms to the XML schema. Quite apart from security issues, this can significantly simplify diagnosis of problems during testing and runtime.

**Traffic management**

In the **Visibility** section, it was noted that you needed to collect information about service levels and whether the agreed thresholds were being met. Of course, you also need to act on that information by controlling the traffic passing through the ESB Gateway:

- **Throttling** allows you to control the flow of requests. This might be based on throughput rate or on the concurrency. This may be to protect the Process Server and WESB infrastructure itself, but it's equally likely that you do not want to overload the downstream systems.
- **Prioritization** means that not all requests are equal. You may need a way to ensure that more important requests are processed first.
- **Load distribution** of requests is important when you have multiple places where a request is serviced. This may be distribution across a cluster of homogeneous servers, although that is normally handled by the infrastructure of the cluster itself. Another example is switching between providers located in differing geographical or network locations, depending on a time schedule.
- **Store/forward** allows requests to stack up during periods when the provider is unavailable. It can also be used to assist with throttling storing quests to ensure the concurrency capabilities of the back end system are not exceeded.
- **Re-try** is particularly useful where a category of error response from the back end system is known to be transitory, for example, a later attempt at exactly the same request may succeed. Retry needs to be used with caution and with the knowledge of the transactionality and idempotence of the provider.

**How is the ESB Gateway implemented?**

So we have covered the theoretical aspects of the ESB Gateway, now let's take a more practical look at how and where the capabilities of an ESB Gateway is implemented within Process Server and WebSphere ESB.

If you remember from the beginning of the article, the **four key capabilities** that we introduced in SIMM level 4 were:

1. Standardized exposition
2. Common request handling
3. Standardized operation specific request handling
4. Service registry

Figure 4 shows that the ESB Gateway is well suited to handling the first three of these capabilities. Each one can roughly be mapped to specific components within an assembly diagram.

**Figure 4. Inside the ESB Gateway**

We’ll look at each of these capabilities and discuss the various options around their implementation.

**Standardized exposition**

Standardizing service exposition means exposing the service in a sufficiently common way that ensures most requestors no longer require a connector to talk to the hub. As Figure 5 shows, this is done where the request arrives at the gateway.

**Figure 5. Standardized exposition**

The following are the primary aspects of service exposition that you need to consider:
• **Transport/Protocol**: Primarily, this is done by choosing standard interoperable protocols and transports, such as SOAP/HTTP or REST/HTTP. However, simply choosing a protocol and transport is only part of the story.

• **Security**: A topic in its own right, but for example, you can specify the security of the channel through which consumers call (SSL and so on), or propagate identities and trust relationship using standard headers. The broader the re-use of the service, the more important this becomes.

• **Transactionality**: Some questions you need to answer are: Can this service be called transactionally by consumers, so that they will never be left in doubt as to whether the transaction has committed or not? Can this service be combined in transactions with other services? Transactionality simplifies error handling significantly for both the requestor and provider of the service, but at the cost of some additional performance overhead. You must also consider whether it is desirable to allow potentially unknown consumers of the service such low level control of your internal resources.

• **Interaction style**: Should consumers interact synchronously or asynchronously with the service? Synchronous interaction provides a simpler programming model to the consumer of a service and is the simplest way to provide transactionality. Asynchronous communication provides benefits in terms of assured delivery and assured responses. It can be argued also that it provides more efficient use of resources in very high concurrency situations since threads are not held open and more complex flow control can be implemented. It is also better suited to interactions that take a significant amount of time to complete, especially where this is longer than the typical timeouts for transactions (120 seconds by default in WebSphere Application Server) or transports. However, asynchronous interaction pushes more complex error handling onto the consumer.

Now remember, we’re talking about these capabilities specifically from a perspective of how you can provide them in a **standardized** way to ensure your consumers can make use of them easily. The primary way to implement these features today is using the plethora of Web services standards.

**Figure 6. Defining the default policy set for a Web service export**

![Figure 6. Defining the default policy set for a Web service export](image)

You can implement them on Web service exports in WebSphere Integration Developer 6.2 using Policy Sets as shown in Figure 6. Policy sets group standards together into common combinations. A given policy set will contain a combination of standards related to security (LTPA, secure
conversation, WS Security, SSL, HTTPS, and so on), transactionality (WS Transaction and so on), and interaction style (RAMP, WS Reliable Messaging, and so on).

**Figure 7. Policy set definition in the Process Server administration console**

On the server, many predefined Policy Sets are available, and it is possible to create further custom sets (Figure 7).

**Alternatives to Web services and SOAP for service exposition**

Web services are not the only standard for exposing services. For example, using HTTP bindings can also expose services using HTTP/REST. However, HTTP/REST does not at the time of writing provide the deeper standards available to Web services. Therefore, its core benefit is its simplicity with a functional intent embedded in the combination of URL and HTTP operation and lighter data formats (JSON). How a given service is exposed is driven by a clear understanding of the requirements in relation to the above exposition capabilities.

*What's the difference between bindings and adapters?*

If you've ever wondered why some connectivity to Process Server is provided using export/import bindings, and others are done using WebSphere Adapters, you can probably answer that now. Export and import bindings are typically for “standardized exposition”. For example, they are provided for the protocols that are used commonly for exposing services in an SOA. WebSphere Adapters are used for the more unusual connectivity methods, which typically use lower level APIs and more complex interaction patterns. Generally, you can assume that an ESB Gateway uses only export/import bindings and does not need adapters as it deals only in standardized protocols.

Before you get too carried away with all the ways to provide these capabilities with Web services, let's look a little closer to home. Some candidate services might be only re-used within a localized
domain. Under these circumstances, you have it much easier to use protocol/transport that provides transactionality, security, and synchronous/asynchronous interaction patterns out-of-the-box without dabbling in the complex WS-* standards. By default, these are the SCA native import/export bindings.

SCA bindings provide many advantages in terms of performance too. So if the re-use scope is relatively local, we strongly recommend that you consider exposing it over SCA. If the consumers are not necessarily Process Server or WESB, you might also consider JMS over the service integration bus (SIB) or EJB.

It's worth noting that for services exposed within an enterprise, consider any transport that has become sufficiently pervasive. It is common, for example, to find internal services exposed over WebSphere MQ.

**What if you needed to expose more than one protocol?**

It is not uncommon to see a requirement for multiple exposure mechanisms. The most common is to make services available over SCA for local requestors and over Web services for the wider world. However, you need to exercise caution here. You must not slip into providing dedicated exposure mechanisms for each consumer or you will fall back to SIMM level 3 and lose the re-use and decoupling benefits of true mature services.

For the sake of simplicity and maintainability of the SOA, you need to aim for the smallest possible number of exposure styles that will serve a reasonably wide consumer base.

**Let the standards do the work**

Let's reinforce here why we have not discussed a number of capabilities often associated with inbound requests, such as data handlers and adapters. This is mostly because at SIMM level 4, you are focused only on exposing requests over standardized protocols and transports. You can assume that the subtleties of the data handling and protocol management come to some extent for free as an implicit part of the standards you have chosen and the product's support for them. Strong governance of the standards used is critical in ensuring the SOA is not only consumable, but also maintainable going forward.

**Common request handling**

The services exposed, by definition of the way they were chosen, are important business transactions. There will be many different interested parties in the services exposed, all wanting different information, or even control over the service. You need to ensure that all services are treated the same for this type of handling, or at least handling of them conforms to a known style or policy.
Figure 8. Common request handling

It is the primary job of the ESB Gateway to provide this common request handling. As shown in Figure 8, the service gateway pattern made available in version 6.2 of WebSphere Process Server and WebSphere ESB is the most logical way to do it. We'll describe the service gateway in more detail in a moment, but let's first describe the types of information and control that you will need to provide:

- **System monitoring**: Is the service up and running? What is its transaction rate and response time? Is it meeting its Service Level Agreement (SLA)?
- **Accounting**: If you are charging people for the use of the services on your SOA, or even if you're just trying to gather statistics around the general usage of the services, you will need to log all requests that occur and probably who they are from.
- **Business activity monitoring (BAM)**: These are aggregate business relevant statistics about the service. Where do my customers come from? How many sales am I making in relation to key dates?
- **Administration**: How do I control access to the service? How do I switch it on or off? How do I re-direct the service when the implementation is re-located?
- **Diagnostics**: Where's my message? What did the data structure look like on the wire? These are general logging and tracing activities for various diagnostics purposes.

You want to make your SOA as homogeneous as possible, so you want to implement these capabilities in the same way for all requests. Hence, the reason this is classed as *common* request processing, rather than the latter *operation specific* request handling. There are likely to be categories of service that require the same types of request handling so ideally, you want to impose request handling based on policies rather than code it into each request.

So if you take a simple service exposed by a Process Server module, how do you implement this standard request handling? There are two primary options:

- Mediation flow components and the service gateway pattern
- SOAP handlers and logical handlers

These are described in the following sections.

**Mediation flow components and the service gateway pattern**

The obvious choice of a component within Process Server for performing the type of capabilities noted above is clearly a mediation flow component. Within the mediation's request and response flow, you have mediation primitives that will provide capabilities noted above, such as:
• **Message Logger** for logging, tracing, and auditing via files and databases.
• **Event Emitter** for auditing via the Common Event Infrastructure (CEI) for business application monitoring if using WebSphere Business Monitor.
• **Service Invoke** for making requests, for example, to a JDBC adapter for a custom auditing solution and so on.

All good so far, but wait a minute, didn't we say common request handling? In a normal mediation flow, you're adding these primitives separately into the mediations for each individual operation, so in what way is this common? With all that repeated code, you're bound to have inconsistencies, and it will be hard to maintain. To ensure a common approach, document development standards to define exactly what primitives are always placed in the flow, and how they are configured. But, they still have to be done separately for each operation and that leaves too much room for human error and difficult maintenance. It is desirable if to do this in a more generic and centralized way for all operations.

One option we'll discuss later is to use mediation subflows. However, since these have to be placed in each operation individually, this is actually more appropriate for enforcing commonality in the standardized operation specific request handling discussed later. For common request handling, there is a better option.

WebSphere Process Server 6.2 provided a new capability known as the **service gateway** pattern. It's a special usage of the mediation flow component. The difference is that where a normal mediation flow component defines flows for each individual operation on the interface, a service gateway channels all operations through the same mediation. This allows you to generically define the processing of all operations passed to a given interface, exactly the capability you need for your common request handling.

It is possible to create a service gateway yourself. It is essentially made of standard components and a few important configuration settings. However, if it's the first time you're trying this, it is better to use the service gateway pattern as shown in Figure 9.

**Figure 9. Creating a new service gateway based on the supplied pattern**

The service gateway pattern presents a wizard offering choices on:

- How the gateway will perform logging
- What sort of routing is required
- What protocol will be used to exposed the gateway
- What service providers it will connect to
The following two articles provide an excellent introduction to the specifics of the service gateway, so we will not repeat that here.

- What's new in WebSphere Enterprise Service Bus V6.2, Part 2: Service gateway patterns

For the purposes of this article, we'll just look at a simplistic example of a service gateway with a diagnostics logging capability that you can configure remotely in the registry.

**Figure 10. Service gateway with policy driven diagnostics logging**

LogForDiagnostics in Figure 10 is a Message Logger primitive added to the mediation for optional logging when diagnostics are required. Most of its properties are pre-configured. As you can see from Figure 11, some of the properties have been made *promotable properties*, which means that you can set them via the administration console.

**Figure 11. Promoted properties on the LogForDiagnostics primitive**

Using the promoted properties, you can now administratively switch the logging on and off (*Enabled* property), change the logging level at which the logging occurs (*Level* property), and decide whether the logging occurs in the same transaction as the request (*Transaction mode* property).

More interestingly, these properties can gain their values from a registry by using a policy lookup primitive, which is the purpose of the RetrievePolicy primitive in Figure 10. For more information about how a policy lookup can be used to do this, see What's new in WebSphere Enterprise Service Bus V6.2, Part 3: Mediation policy.

You can prepare a single service gateway that exposes all (or at least groups of) services and that performs all common processing, such as monitoring, diagnostics, accounting, and security.

**SOAP handler/logical handler**

There's another alternative that we will briefly discuss. It may be the common request handling that you need to do is at the level of the SOAP or JAX-WS protocols. If so, Integration Developer v6.2
can add custom written handlers directly to a Web services export. There are two types of handler you can write: a **SOAP handler** or a **logical handler**.

**Figure 12. Creating a new JAX-WS SOAP handler**

SOAP handlers and logical handlers are just Java™ classes. Figure 12 shows how you can get Integration Developer to create a template handler for you, so that you only need to write the methods that handle the request messages and faults.

The SOAP handler allows access to the message via an API that is aware of the SOAP protocol and provides easy access SOAP headers, for example. In most instances, this makes for simpler coding of the handler.

The logical handler is used when you want a handler that is agnostic of the protocol. The handler is unaware that you are dealing, for example, with a SOAP message and sees the message purely as XML content. This is more flexibility, but harder to code and maintain.

**Standardized operation specific request handling**

Ultimately, the ESB Gateway will have to know what operations it’s dealing with to route them to the appropriate downstream mediations for further integration, or directly to the service providers. Along with the routing itself, there are some other operation specific things that are appropriate for the ESB Gateway to perform.
Essentially, anything that is done in a standard way, but with operation specific knowledge, is a candidate for the ESB Gateway. If it is sufficiently well standardized, you can add it to the service gateway component as shown in Figure 13.

There is a subtle division of responsibilities between the standardized operation specific work you might do here, and normal operation specific functionality you may be doing in normal mediation flow components downstream. As there are no industry standards to follow in this area, this aspect of the design is still specific for each customer. However, we are starting to see the following as common capabilities:

- **Routing**: The fact that you now know what operation you are dealing with may already provide enough information to wire the request to the appropriate endpoint. Routing might also be based on the URL, which for a service gateway, is available in the HTTP header. You route on this and any other message header content using a message filter (router). You might also choose to externalize the routing using an endpoint lookup primitive. You can configure and administer alternate endpoints via promoted properties on the service invoke and callout primitives.

- **Flow/concurrency control**: Downstream systems may have constraints in terms of the number of requests they can receive. It may be that at certain times of the day that requests must be sent to an alternate endpoint, be handled asynchronously, or even refused altogether. You might need to promote properties that enable administrative control over, for example, invocation style, alternate routing, and retry configuration.

- **Auditing**: Who performed this transaction on this date and time? What other transactions have they performed? This may provide an audit trail for compliance reasons, or for other purposes, such as charging for the use of the service. You would likely do this using the message logger, event emitter primitives, or service invokes to auditing services or JDBC adapters.

- **Access control**: Fine-grained access control might be necessary via a database lookup primitive, or a service invoke to an application specific authorization service.

- **Semantic mapping**: If the data model of the consumers that you are exposing the service to is different than the one in common use behind the ESB Gateway, you may use business object map or XSLT transformation primitives to do the translation. You need to be cautious not to allow complex conditional mapping to creep into the ESB Gateway. There should definitely be no data formatting or parsing of non-standard formats.
Some collection of the above might be coded into the service gateway flow. All requests pass through the flow, but a policy lookup primitive has set operation specific values for all the relevant promoted properties so that it behaves differently in each specific case.

What you shouldn’t do in the service gateway flow, however, is filter by the operation name and start creating significant branches of operation specific logic. These are then clearly not standardized. They should be done in a normal mediation flow, which is designed for that purpose and has clearly separated flows for each operation. In other words, these aspects live outside the ESB Gateway component.

**Other ways to standardize request handling**

What if you simply cannot have all requests coming through the same service gateway component, but you still want to add some standardized code into every request? Well, there are some other alternatives that you can use either on their own, or in conjunction with the service gateway to provide other forms of standardization.

**Using mediation subflows to standardize request handling**

You can make standardized code easier for developers to use by embedding the required sequence of primitives into a mediation subflow. You can store these in a library, as in Figure 14, such that they can be used on many different operations and even in other modules.

**Figure 14. Creating a mediation subflow in a library**

Subflows, introduced in v6.2 of Process Server and WESB, are a way of capturing a set of mediation steps (primitives, like the ones in Figure 15) in a re-useable flow fragment.

**Figure 15. A common sequence of actions captured in a subflow**

You want the subflow as re-usable as possible, so you need to set it to receive the most generic type (Figure 16) possible for the operations where it is likely to be used. This needs some thought during the design of the interfaces.
Figure 16. Setting the message type for the subflow

![Image showing the setting of message type for a subflow](image)

You might have expected to see the `policy lookup` primitive in the subflow, but actually that's currently not possible. If you think about it for a moment, it's clear why. The subflow does not know what operation it's performing. However, the subflow's promoted properties become visible to the parent flow that implements it and so it is in the parent flow you expect the `policy lookup` to be performed.

**Custom primitives and custom visual snippets**

Some final options worth mentioning to standardize the flow implementation is the use of custom visual snippets or mediation primitive plug-in. Both of these allow common complex capabilities to be made available to implementers as just another item on the palette of visual snippets or mediation flows. This brings benefits in consistency and maintainability. This is not specific to the ESB Gateway. It is a general good practice to wrap common complex code in this way to make it easier for implementers to follow your development standards. Along with the links to the Information Center in the text above, there are also numerous developerWorks articles on this topic.

**Other tools that assist in standardized request handling**

Let's not forget also that there are plenty of hooks that are already available either within the product or using associated products. Here's a selection:

- Standard Java logging and tracing capabilities provided in the WebSphere administration console allow the setting of logging on specific component types.
- The Performance Monitoring Infrastructure (PMI) and the Tivoli Performance Viewer are a standard part of the underlying WebSphere Application Server product and provide numerous statistics, such as component response times. Application Response Measurement (ARM) statistics are also available.
- Business Process Choreographer Explorer, which from v6.2 also includes Business Process Choreographer Observer reports, provides historical statistics on requests passing through WebSphere Process Server if they relate to long running business processes.
- Using the test client remotely and cross component trace provide sophisticated debugging capabilities that may remove the need for diagnostics logging in many instances. However, it is recognized that these are rarely appropriate for a production environment.
- Tivoli ITCAM for SOA provides more sophisticated system monitoring of Web services requests and integrates with the WebSphere Services Registry and Repository.
When do you reach the limits of the ESB Gateway?

Alongside the ESB Gateway that we have looked at in detail, the integration "hub" is still present on the SIMM level 4 diagram (Figure 2). What does it do? When do we need it? In other words, what shouldn't you do in the ESB Gateway?

Where you have operation specific needs that are completely customized to the operation, this is where you resort to traditional integration with all the tricks matured in SIMM level 3. This needs to be performed in the hub, not the ESB Gateway. You do not want to burden the ESB Gateway with these advanced integration problems. Although these advanced integration patterns are a topic for a later article, here are some examples of when operation specific handling should definitely not be performed in the ESB Gateway:

- **Non-standard protocols/formats**: The ESB Gateway should only talk in standard protocols. It should not talk in non-standard protocols, low level APIs, or data formats. You can assume, therefore, that it should not use adapters.
- **Orchestration/choreography**: The ESB Gateway should never be required to make several calls to satisfy one request. It should always be one request in, one request out.
- **Stateful logic**: The ESB Gateway should never be required to persist or own any form of state between requests.

Conclusion

This article focused on exposing services in a consistent way and explored the ESB Gateway in some detail because it is so critical for an enterprise to successfully establish its SIMM level 4 maturity. Formalizing of the ESB Gateway concept within an SOA provides the standardized administration and visibility of services that a maturing SOA must have. If you are to have any chance of governing your SOA as it expands, consistency in service exposure is paramount.

We have separated out the role of the ESB Gateway so that there is benefit in placing it into a separate layer in the architecture. This is becoming more common, and it is not unusual in a large SOA to see a separate product, such as WebSphere DataPower, performing the role of the ESB Gateway and saving the integration hub to focus on deeper advanced integration. While this is beyond the scope of this article, there is some discussion around the placement of WebSphere DataPower in an SOA in Chapter 9 of the following IBM® Redbook, *Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB*.

While looking primarily as a service exposition, we have ignored the deeper integration challenges that come into play when connecting to real, and typically aging, backend systems. In a later article, we will explore all those fascinating advanced integration patterns that happen outside the ESB Gateway in the integration hub. The specifics of a complex integration with these non-standard backend systems often take up a significant portion of the project in a maturing SOA.

We have also neglected the "process" side of WebSphere Process Server in this article. We will return to that in the next article and look at the different process implementation types, what they are used for, and how they should be implemented.
Acknowledgements

The ideas within this article were primarily developed in collaboration with Brian Petrini, who will be authoring other articles within this series.

Furthermore, the conclusions above are gathered from discussions with people on design topics, project experiences, and also from our conversations with people involved in the creation of the product. Our thanks to the following colleagues (alphabetical by first name): Andrew Ferrier, Andy Garratt, Bobby Woolf, Callum Jackson, Chris Markes, Geoff Hambrick, Greg Flurry, Helen M. Wylie, Jonathan Adams, Peter Lambros, Rob Phippen, Stephen Cocks, Paul Verschueren and in reality, many more.
Related topics

- The new service gateway discussed in this article is described in the WebSphere Integration Developer Information Center.
- The following two articles describe in detail the service gateway and its use with a policy lookup primitive, respectively:
  - What’s new in WebSphere Enterprise Service Bus V6.2, Part 2: Service gateway patterns
  - What’s new in WebSphere Enterprise Service Bus V6.2, Part 3: Mediation policy
- Enterprise Integration Patterns defines the base integration patterns, most of which have progressively been productized in products, such as WebSphere Enterprise Service Bus, WebSphere Message Broker, and WebSphere DataPower.
- Enterprise Connectivity Patterns: Implementing integration solutions with IBM's Enterprise Service Bus products puts some important detail behind the broad range of architectural integration patterns.
- The developerWorks article series, Exploring the Enterprise Service Bus, provides core definitions for the term Enterprise Service Bus and practical advice, both architecturally and in terms of IBM products.
- Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB was an early attempt to describe the ESB Gateway in more detail, but it is superseded by this article. It does still provide some useful notes on WebSphere DataPower, WebSphere Service Registry and Repository, and WebSphere Transformation Extender in the context of an ESB.
- The article ESB-oriented architecture: The wrong approach to adopting SOA is a valuable wake-up call that an ESB is only part of a broader picture.
- Increase flexibility with the Service Integration Maturity Model (SIMM) provides a basic description of the SIMM, which has since been adopted by the open group.