

The SOA Magazine

Feature Article



Building Reusable Services

by Vijay Narayanan

Published: April 30, 2009 (SOA Magazine Issue XXVIII: April 2009)

[Download PDF](#)

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

Abstract: The ability to reuse services is one of the principal benefits for adopting SOA in an enterprise. This statement will remain a promise for your development efforts unless some foundational concepts are practiced. This article offers several practices for increasing the likelihood of achieving service reusability including decoupling of transport, message and interface standardization, encapsulating horizontal cross cutting concerns, and validation of service contracts to avoid interoperability issues.

Introduction

There are a host of practices to adopt when designing services that will increase the odds for service reusability. If care isn't taken a service can be coupled to a transport, access mechanism, and distribution channel that will inhibit our ability to reuse it. This article will explore some of the practices that I have used to facilitate reuse of services across projects and initiatives. The golden rule for building reusable services is to not place functionality that is specific to a consumer alongside the core service logic. In addition to this overall guideline, there are several practices that will help foster service reuse.

Decouple the Physical Transport from the Service Logic

The physical transport over which a service is exposed should be decoupled from the service functionality. This will ensure that the service can be bound to additional transports gracefully and provide flexibility offer transports on an as needed basis. On a related note, design the service's functional contract to be identical across transports. This will make it easier for consumers to switch transports or consume multiple transports for the same service depending on their requirements. Transport decoupling also means not tying transport specific headers or processing to service logic. Let us assume that HTTP headers were used to store service parameters and your service implementation was relying on them for proper functioning. A service might start out getting consumed over HTTP and a new use case might require the same service to be available over a reliable transport (such as WebSphere MQ). The use of HTTP headers will inhibit reuse unless it is modified to remove the coupling.

Decoupling the service logic from physical transport is also critical to ensure that difference message exchange patterns can be supported by the same underlying service. Maybe you have a service that is initially accessed in a synchronous request/reply fashion over HTTP. A new consumer might want to access the service in an asynchronous request/reply fashion using dedicated send and receive queues using a reliable transport such as Websphere MQ. If the transport and service logic are decoupled, such a requirement can be supported gracefully. Consider another common scenario - there might be event notifications that get generated out of the SOA infrastructure that will cause messages to be published to downstream systems via subscription queues. If you need to support on demand and event driven message exchanges for the same service transport decoupling is an absolute must. The service logic can be reused reducing development time and maintenance effort.

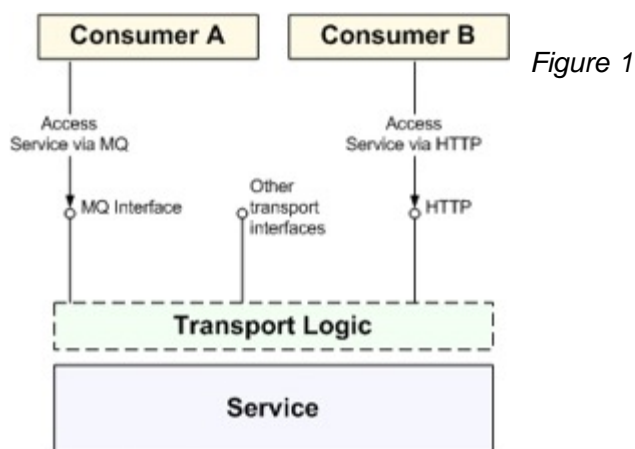


Figure 1

Provide Standard Interfaces for Service Access

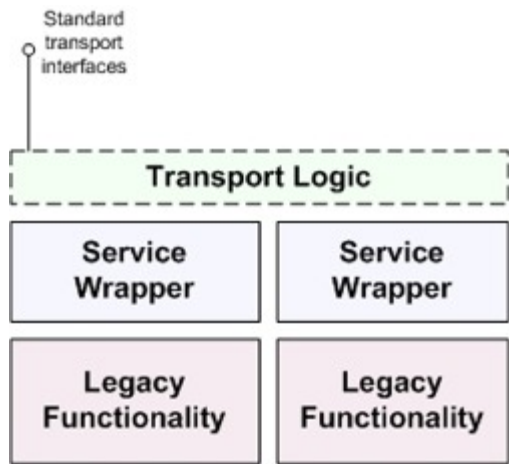
By encapsulating service access the provider gets the flexibility to change implementations over time or offer multiple implementations based on SLA requirements. When exposing any service to your consumers always integrate them via a standard interface. You can route requests based on volume, data characteristics, and transport characteristics from a single point in your infrastructure. This is useful also when you know your service implementation is going to grow and get more sophisticated. For example, you might initially have a simple lookup for a string input against a single system. Over time, you might want to add a query against a search engine, or multiple applications etc. The standard interface will keep this evolving complexity out of sight from your consumers. Additionally, this practice is very relevant when exposing legacy services that eventually might get migrated to a newer platform. You definitely want to leverage and reuse legacy functionality but at the same time you don't want point to point coupling between consumers and legacy services. By offering multiple transport interfaces for the service you will make it easier for your consumers to integrate for a variety of scenarios. This will additionally facilitate reuse of XML schema contracts as well as horizontal capabilities (e.g. metrics, monitoring, auditing, security, and logging) that are part of your SOA infrastructure.

Legacy services might have different naming conventions, behavioral characteristics, error handling semantics, inconsistent error codes, and programming interface definitions. If you have invested effort in XML schema definitions following a contract-first development approach, business data types, naming conventions, schema definition styles, business object structures, and other idioms would have been created for consistency across your service inventory. If legacy services are exposed as-is they will by pass all the value addition happening in the SOA infrastructure.

There are scenarios where multiple Service Level Agreement (SLA) norms and policies have to be supported by a service. Based on the SLA requirement you can reuse the core service logic but offer it via a dedicated interface meant for a certain segment of your consumer population. This approach also can facilitate data enrichment or special processing that needs to be performed for a certain consumer segment. Standard interfaces could take the core functional logic and wrap additional layers of value added services for your consumers. If you permit direct access to services bypassing the standard interfaces, yet again, the value addition in your SOA infrastructure will be under utilized.

Standard interfaces are immensely useful for another key reason as well and that is they act as the gatekeeper into the SOA infrastructure. This is important because horizontal concerns such as security and data validation can be performed at this layer. Metrics can be captured and queues and HTTP ports can be monitored via an administrative console. Service requests can thus be handled at a single strategic point in your SOA to keep bad data out, apply consistent processing semantics, and increase consistency.

Figure 2



Offer Standardized Publications of Your Business Processes and Entity Services

Event notifications arising from milestones from a business process or publications from updates to critical enterprise data need to be offered for service consumers using 'standard' publication messages. These standard publication messages need to reuse your business schema data types, object definitions, as well as underlying service components. If you support entity and task services (such as business process orchestrations) publications can be supported from both these entities. Entity messages could include data attributes and audit trails while the task publication messages could include approvals, SLA adherence metrics, escalations, exceptions, and delegations. Ensure that your publications contain no information about the technical platform, programming language, persistence mechanism etc. as part of its structure and content. This will increase the likelihood of having the publication messages generic enough thereby increasing the potential for reuse. The idea is to let the salient milestones from your process publish standard messages. A particular consumer may not need all the publications in which case subscriptions can be managed or unwanted messages can be filtered out.

The most critical aspect of this guideline is to resist the temptation to make a message too specific to a consumer. If you do, the reusability of the publication will be significantly reduced. The idea with these standard publications is to be able to provide a message that contains the right number of data attributes that will meet most consumers' requirements. This doesn't mean you have to get it absolutely perfect the first time - service and schema versioning schemas have to be leveraged in order to support multiple standard messages. However, this should be a deliberate, planned effort in conjunction with the rest of your SOA governance processes. As a working principle, new consumers should be subscribed to standard publications unless the existing standard message doesn't meet their needs or cannot be consumed due to other limitations. For example, a service consumer might want custom XML tokens or headers added as part of your standard service publications. In such cases do not modify the core service logic to format any destination specific information. Instead, either persuade consumers to take your standard message or subscribe to your own publication and modify for the consumer as an exception. Whatever approach you take, if you refrain from putting consumer specific logic into your service it will facilitate reuse naturally.

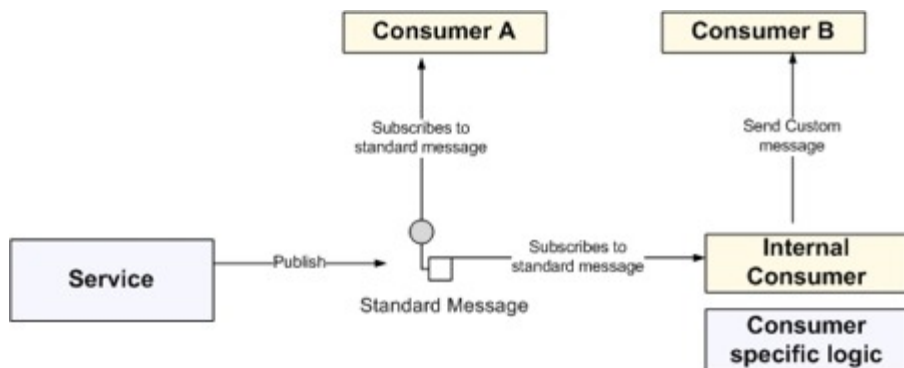
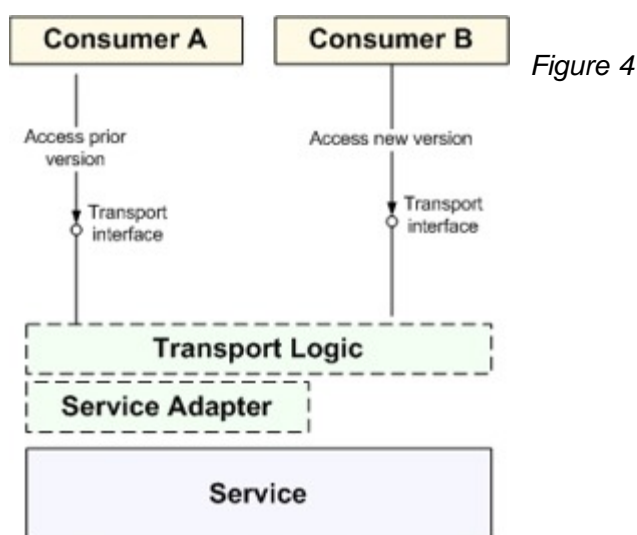


Figure 3

Create Service Adapters for Backwards Compatibility

This practice is related to the above but looks at facilitating service reuse for existing consumers as well as new ones. Maybe you offered a service that has existing consumers. If you want to upgrade your service you can either create an entirely new version or upgrade your existing one and still be backward compatible to the previous version. In case you choose the latter option, you can reuse your service logic across your consumer base by creating adapters that will perform necessary backward compatibility logic. Once all your consumers upgrade to the newer version, the adapters get eliminated. The idea here is to avoid the need to support multiple service versions if you can. If data transformations need to support older namespaces and data types, simple data lookups/validations, and error descriptions are the key differences between the versions it is possible to fully contain the changes in this adapter layer. If there is a need to support non-standard transport interfaces that could be done in this layer as well.

The advantage with this approach is manifold. Firstly, the code base is thinner due to the reused service logic. Secondly, because the service adapter layer is outside the core service logic it is isolated and encapsulated. When it is time to get rid of the service adapter, the code can be easy to locate and eliminate without fearing unintended consequences to the core service. Thirdly, as alluded to earlier in this article, this adapter can be reused across transports as well. So you can avoid implementing custom solutions for every transport your consumers use. Finally, the adapter itself if made modular can be reused across several services. For example, the prior version of a service could have supported a particular payload structure or a standard set of processing steps. Those pieces of the logic can be centralized in this adapter layer and new services can be supported with the same overall processing flow.



Apply Cross-Cutting Concerns Horizontally

Any cross functionality such as logging, metrics, security, and monitoring, should be reused across your catalog of services. For security, there could be an existing mechanism that is in use in your enterprise that you need to leverage. Regardless of solution, logic for authenticating a request or decrypting a payload should not be at the service level. For instance, your existing authentication may be driven off a LDAP provider. Day two, you might want to move to digital certificate based authentication or start taking advantage of WS-Security. Whatever the reason, making authentication centrally will make it easier. Also, there are situations where your core service might be invoked using multiple authentication mechanisms. Think of a situation where a web application user needs to use their login credentials to access a particular service while a batch process needs to use a digital certificate to access the same service. If the authentication mechanism is tightly coupled to a service supporting multiple mechanisms or swapping technologies will become an arduous exercise. Reuse the core service and build this capability outside of it.

For debugging, runtime diagnostics, monitoring, and production support a robust set of metrics will need to be captured. This is information every service will need to capture and as an architect we should strive to encapsulate as much of this logic as possible horizontally. Take special care to record metrics that your support team needs so it can ease their ability to monitor and react to issues that arise in the SOA platform. There could be metrics common to all services and metrics that are specific to a particular technology platform. Also important is the ability to capture transport and message exchange pattern specific

information. For example, you will want to capture incoming queue name, message pickup time, correlation id etc. for asynchronous request/reply via queues. In the same vein, the JMS Topic that you are publishing to, the time of publication, and business process name or entity name are key to publish/subscribe message exchanges. Most applications will need to preserve metrics for a duration based on your application and/or business requirements. In addition to capturing metrics, a reporting facility to get a variety of canned and custom reports needs to be provided as well. Logging is another key feature that is a cross cutting concern. At a bare minimum, the request and response payloads will need to be logged and key integration points in your orchestrations will need to be captured as well. If your enterprise has an existing API for this do reuse that. If not, create it as a generic feature that all your services can take advantage of. Consider the need to control logging levels at runtime as well.

Never put logic for these capabilities in a single service because chances are you will surely need them for another one! Making these cross cutting concerns truly horizontal will guarantee that these concerns will be reused for the majority of consumer usage patterns. This will also provide you with the flexibility to change the implementation of a cross cutting concern without impacting any of the services. The services themselves can then be reused with a newer version or implementation of these horizontal capabilities.

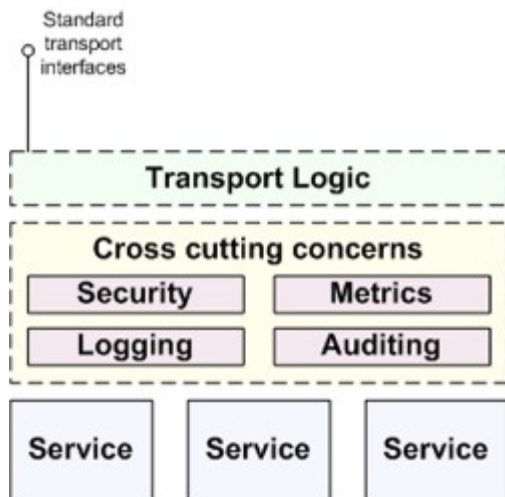


Figure 5

Ensure that Your Services are Interoperable

One of the biggest impediments to service reuse is unintended coupling. A service might be designed with reuse as an objective but may not be able to do so. The contract might include platform specific schema structures, transport headers, or annotations that cause coupling to a specific technology. In fact, the service may expect the consumers to be on the same platform as well inhibiting reuse for consumers outside the platform. Unsupported transports, bindings, or SOAP attributes could also inhibit a service's interoperability and so its reusability. There are situations when the WSDL document generated by a SOA toolset cannot be consumed by a particular consuming technology platform. Take no chances and ensure that the WSDL document can be consumed successfully by the major technology platforms and that your target consumers can generate proxies and XML data bindings from the WSDL. It is essential to use a tool such as the WS-I WSDL Analyzer to validate your web service interfaces and ensure that your services are reusable. This could be linked to your continuous integration process where the latest contracts are validated for interoperability and reusability via a script that invokes the WSDL analyzer.

Conclusion

The idea of service reusability being a natural by product of service oriented architecture is very comforting. However, unless care and conscious design decisions are made to decouple services from transport, distribution channel, access pattern and standard messaging interfaces are offered high degree of service reusability cannot be a reality for your enterprise. Cross cutting concerns must be decoupled from the service functional logic in order for them to be reusable across your service inventory. Ensure metrics are adequate and comprehensive to account for multiple transports and message exchange patterns. Logging and auditing is key as well and should be reused across all your services. Finally, ensure that your service

contracts such as WSDL documents are thoroughly validated for interoperability issues that determine the service's reusability.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About](#) [Legal](#) Copyright © 2006-2009
SOA Systems Inc.