

The SOA Magazine

Feature Article



The Case for Single-Purpose Services: Understanding the Non-Agnostic Context and a Strategy for Implementation

by Herbjorn Wilhelmsen

Published: December 17, 2008 (SOA Magazine Issue XXIV: December 2008)

[Download this article as a PDF document.](#)

Abstract: Justifying the extra investment for developing a single-purpose service ? a service expected to solve only one large business problem - instead of putting the single-purpose logic inside a non-service-oriented application can be challenging. Reuse, the most popular motivation for creating services, will not apply. So where's the business case? Acceptable justifications can include: enabling support for multiple providers, isolating logic from change, centralizing IT-support for a given business process, service composition optimization, and separation of concerns. Although performance is commonly referenced as a reason to not create services, that line of thought is not always valid.

With the help of patterns referenced from the recently published SOA Design Patterns book [REF-1] and the soapatterns.org site [REF-17], this article will delve into these issues as we explore the case for the single-purpose service.

Introduction

Services are useful, but they come with a price tag. The cost of developing a service is higher than the cost of developing a traditional (non-service-oriented) application, primarily due to the extra work and infrastructure required. Another common concern when creating and consuming services is the possibility of a performance hit. Together these issues hint that even if you've decided to wholeheartedly adopt SOA, you may not want or need to move all your functionality into services. This is where the application [Service Encapsulation](#) becomes a focal point as we need specific criteria to determine what should and should not be encapsulated into services.

To make this determination, we will take a look at three different aspects:

- acceptable reasons for creating a service
- costs associated with creating a service
- and, of course, the performance issue

After covering these aspects specifically in relation to single-purpose services, I will introduce an implementation strategy.

Reasons to Create a Service

Any service that is created needs to have a cleanly defined responsibility. The capabilities it exposes should clearly fall within this defined responsibility. In the case of a single-purpose service, it can be argued that it may be better to implement single-purpose logic as a non-service-oriented application. Let's take a closer look at some of the more important considerations:

Reuse

When logic is incorporated into a service, it is potentially available for reuse by multiple applications, some of which may themselves be services. Reuse leads to reduced development and maintenance effort, which translates into a lower cost

of ownership and can further result in improved quality and lower risks [REF-2]. Reuse is also an important part of the agile IT enterprise. Composing existing logic to solve larger business problems is more efficient than writing all of the logic from scratch.

Alas, enabling this kind of reuse is not as easy as just incorporating logic into services; it often requires a lot of thinking and design effort to create a service that is truly reusable. But, it can be done. With regards to single-purpose services, reuse is usually not a consideration. These services are specific to parent business process logic and therefore serve just that one purpose. Figure 1 illustrates some common service categories [REF-3] and how they relate to each other and business processes.

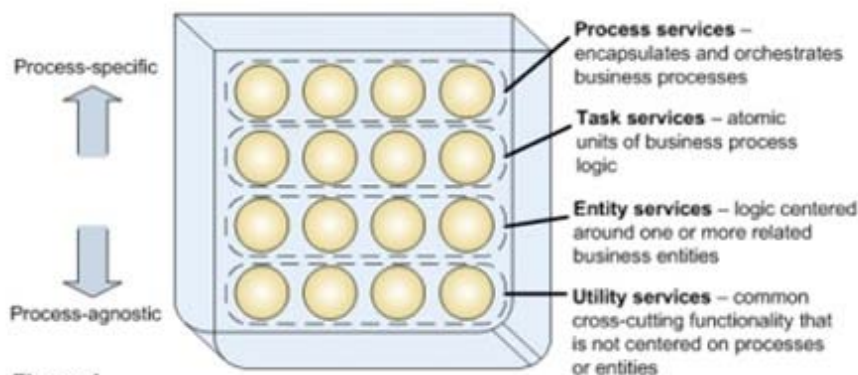


Figure 1: A Service Inventory typically consists of services from multiple categories. Process-specific services can not be reused when implementing support for other business processes. The more process-agnostic a service is the higher it's reusability

As just stated, services that cannot be repurposed to automate another business process, as per Non-Agnostic Context, are not considered reusable. However, an important realization here is that logic that solves only one large business problem may still be used by multiple consumers.

Let's explore this notion with a simple analogy. Due to technological advances, the manner in which people perform their jobs today is very different compared to 20 years ago and in the years to come we will probably witness an increase in the rate of technological progress. One kind of change that we have seen is that companies want to enable employees to perform their jobs using different tools in different settings. When we are at our desk we typically expect rich functionality and applications that make the best of our hardware, such as advanced large screens with high resolution and many colors and advanced keyboards with many functions, to name but a few.

On the other hand, we want to be able to do at least some of the same tasks when we are traveling and in that setting we may only have access to, say, a Smartphone. This type of mobile device is much more difficult to work with than a desktop application, and it requires a very different kind of user interface. Processing power, development environments and support for frameworks, among other things, are more limited for Smartphone applications. In spite of these differences, the two applications (desktop and mobile) can still be designed to automate the same task (see Figure 2). As a result, both applications could call the same single-purpose service (which actually does lead to a form of reuse).

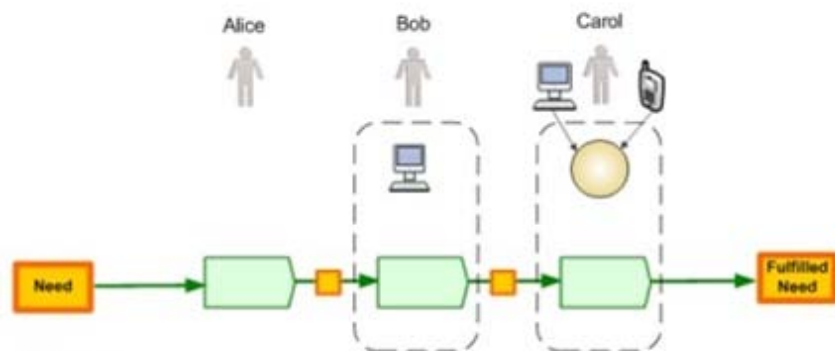


Figure 2: Alice, Bob and Carol work for different departments but are responsible for different activities that make up a business process. Some of the process steps need IT support and some don't. Carol owns a desktop application and a smartphone application. She creates a service to encapsulate the process-related logic that she is responsible for and lets her applications call the service.

Even though reuse is an important criterion for creating a service it is by no means the only one. In a recent blog post [REF-4] Paul C Brown argues that the main criteria for determining if a capability ought to be put inside a service (apart from reuse) are multiple providers and isolation against change. These are discussed briefly below.

Multiple Providers

The reuse of a service can be thought of as the existence of two or more service consumers. The reverse of this is when you have two or more providers of the service (Figure 3).

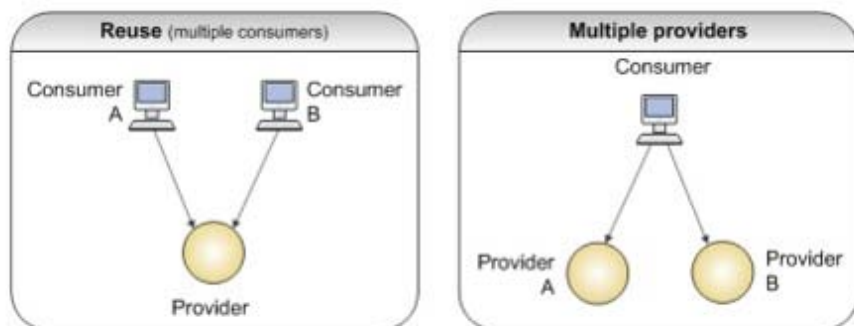


Figure 3: Reuse compared to multiple providers.

Figure 3

Choosing between providers, calling multiple providers, or merging together results from different providers might be non-trivial tasks, and applications can be shielded from this kind of logic by placing it inside a service. Corporate mergers and acquisitions often lead to this kind of scenario, but it can also occur within a company when multiple departments work with overlapping tasks and information. Due to the commonality of this situation, there are design patterns available that describe exactly how to construct such services (e.g. Enterprise Domain Repository [REF-5]).

A problem that must be handled when using multiple providers is that of partial failures. As the number of network links and composed services increases, the probability for failure in one of these links or services increases. To address this, a single-purpose service might securely store messages and make a number of retries to accomplish the delivery of messages to composed services, as per [Reliable Messaging](#). Shielding a consumer from these complex tasks is a good enough reason to create a service, even if it cannot be reused.

Isolation Against Change

Being able to handle change successfully is one of the biggest promises of service-orientation. SOA can help us achieve increased business or organizational agility [REF-6] in several ways, one of which is by developing services and consumers in parallel. This approach requires that we (the service and consumer developers) have first have agreed upon the contract.

Another way is to create new functionality by composing existing services, either our own or services provided by someone else. Yet another important aspect is enabling change by limiting the parts that need development effort to bring about the desired changes. This means making sure that when you change something that the change only affects a limited and preferably isolated part of your software assets. All these issues relate to being able to develop new functionality or change existing functionality with less effort and in a shorter period of time.

To be able to quickly adapt to change can be essential for a business. As Jim Webber so eloquently puts it: "Business people are spaghetti-heads" [REF-7]! Behind this statement lies the profound understanding of the fact that business people need to make new decisions - sometimes even unexpected decisions - in the light of new business demands and opportunities.

Changes in a business process can lead to changes in services, consumers, or both, but there are also justifiable reasons for changing a service even when the business process it encapsulates has not changed.

When implementing a service, there are a variety of realization options ranging from buying or building applications hosted on premise via different hosting options and placing services in the cloud to options that haven't even been conceived yet [REF-8]. These ever-evolving options lead to never-ending opportunities for change that are further influenced by the cost associated with the options, your company's business strategy, and many other factors.

One factor worth calling out is the business strategy, because this strategy itself can be subject to change over time. When a company's strategy changes so will the strategic importance of its services and other IT assets. What this means is that outsourcing may become an option for a particular service today (e.g. to cut costs) but it may be then be necessary to in-source it tomorrow when it becomes more strategically significant (Figure 4).

By encapsulating logic into a service, these kinds of changes will become much easier to handle. A single-purpose service might swap out the current implementation of one of its composed services with an alternative implementation without changing its contract. To be able to accomplish this you may have to apply Data Format Transformation inside the single-purpose service.

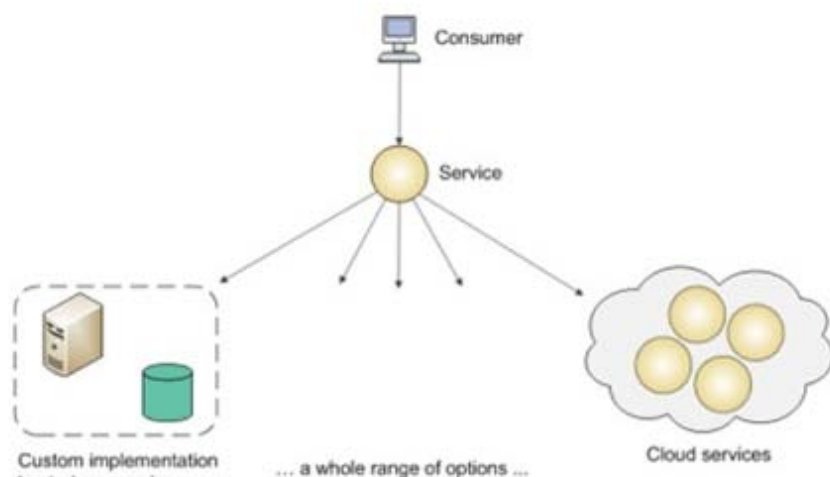


Figure 4

Figure 4: By creating a service that encapsulates single-purpose functionality it becomes possible to quickly outsource or insource functionality - or choose between any of the options in between - without affecting the consumer

Although it can be tempting to always plan for this kind of flexibility, it is important to keep in mind that agility may not be the most important factor for all processes or companies. Balancing the need for business flexibility in proportion to the cost of IT flexibility can become the true key to success.

Keeping it together

Getting IT support for a business process right may involve some technical challenges. A business process can be considered a collection of related activities that solves a large business problem which can span departments and even business domains. To successfully implement a business process, the main focus must be on the customer and the result of the process - not on its organizational boundaries [REF-9]. Attaining a consensus about organizational issues and focusing on processes rather than personal fiefdoms within the enterprise is usually a far greater challenge than creating viable technical solutions. Some companies handle these problems by appointing an owner for every business process who is responsible for accommodating the customer of the process, developing the process, staffing the process, as well as developing, buying and maintaining any IT-support that the process requires.

To really understand how a business process works in an established company, you must practice both anthropology and archeology [REF-10]! As an anthropologist you study something mainly by talking to the natives - that is, asking the people that work with the business process what they do and how they do it. (Unfortunately people don't always remember those nitty gritty details.) Also, parts (and sometimes even the majority) of a business process might be automated or at least have some IT support. This support may be provided by multiple IT systems (with integrations) that are used by people working for different departments.

In these cases, the person that does the real work does not know enough of the story to really help you understand the complete business process. When this happens, archeology is the answer to your problems. You have to dig through the ruins (dissect legacy systems - especially databases - and the workarounds people have created) to see what is really going on. If you find hard evidence in the legacy environment that contradicts the results of your interviews with the users, further investigation is required.

So if you do need to know how a business process really works (e.g. to evaluate if it should be changed or to locate a defect), you might be in for a surprisingly big job. An elegant way to save yourself from such problems is to encapsulate all the logic that relates to a specific business process into a single-purpose service, as shown in Figure 5. The owner of the business process will then really be the owner of the system that encapsulates and composes all of the relevant logic (which is the way it should be), as per Process Abstraction.

Figure 5: Alice, Bob and Carol work for different departments but are responsible for

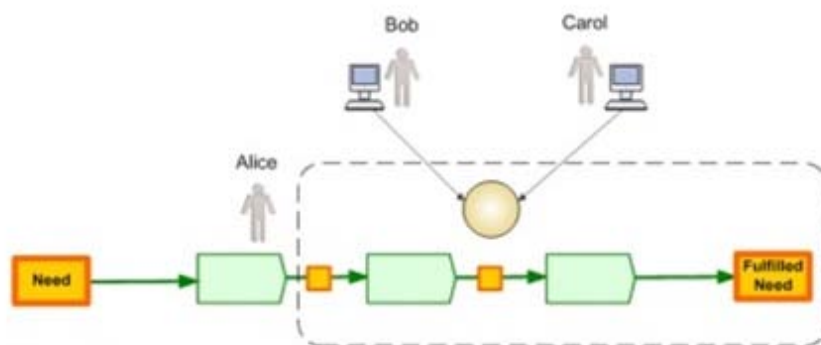


Figure 5

different tasks that make up one business process. As the owner of the process, Alice creates a service that encapsulates all IT-support that the business process requires. She lets Bob's and Carol's applications call her service.

Service Composition Optimization

The modeling and design of service compositions is a science onto itself. It represents that stage at which you can truly harness the power of your services by assembling them together into sophisticated solutions. Reusable (agnostic) services are generally built to accommodate this by taking Service Composability considerations into account during their modeling and design stages. However, the single-purpose logic that is generally responsible for hosting the bulk of the parent composition logic, may not receive the same attention (especially when it is placed into a non-service-oriented application).

As a result, the single-purpose logic can jeopardize the performance and stability of the service composition because the it was never subjected to the rigor of service-orientation. When isolating this logic into a service, it benefits from the same design considerations as any other service and therefore becomes a much more effective part of the composition, thereby enhancing the performance and quality of service-oriented applications.

Offloading Work from Client Computers

Sometimes single-purpose logic will require computationally intensive computing. Offloading computationally intensive work from client computers has a number of advantages. Equipping servers, which typically host services, with computing resources (such as multi-core CPUs or multiple multi-core CPUs) is significantly more cost-effective than equipping a lot of workstations with more hardware resources. The probability of being able to make good use of such hardware resources is much higher for a server than a client computer, due to the fact that multiple client computers and users can utilize the same server in a timesharing fashion. However, this approach is not suitable for all kinds of computations. If the work can be broken down in large chunks that require few or ideally only one data exchange encapsulating the logic in a single-purpose service might work out well. If multiple data exchanges are necessary, network delays might make it unfeasible.

Separation of Concerns

Another reason for creating single-purpose services is to establish a clean separation of concerns. This well-known software engineering theory is all about breaking logic into pieces that solve distinct concerns. The aim is to create software with features that have little or no overlap in functionality. Eric Evans even recommends that we separate logic that deals with different concepts into separate modules or components [REF-11]. The argument for doing this is that it separates the domain layer from other kinds of code.

Without this separation, the design choices of the developers can become limited and will ultimately lead to reduced software quality [REF-12]. Creating a separate component with all of the service-oriented code results in a low coupling between the service-oriented code and the domain code, as well as a high cohesion within the separate components (Figure 6).

Figure 6: By putting domain code and service-oriented code into different components we introduce low coupling between the two concepts and high cohesion within the components

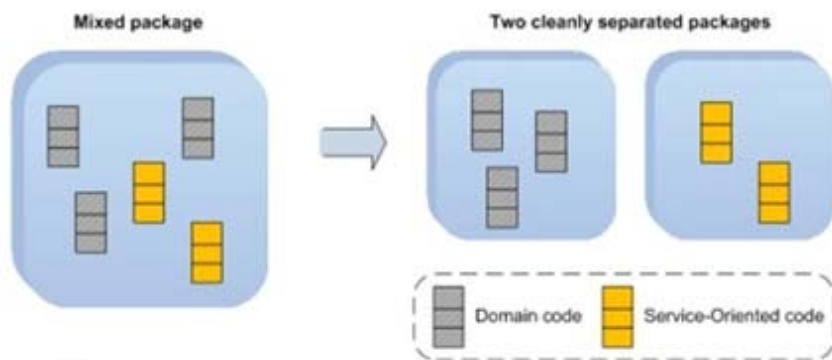


Figure 6

Right now you're probably thinking: "Sure enough, the arguments for doing this are valid, but it refers to components not services"! And that brings up an important question that we need to discuss before continuing: What is a service?

What a Service Is

There are a surprisingly large number of definitions of the term "service". The discrepancies between these definitions are equally surprising. Defining what a service is in the face of such diversity is not something that I want to even attempt in this article. I do, however, want to make some notable distinctions.

A Web Service Is Not Automatically a Service

There is an unfortunate tendency among practitioners to use Web service as a synonym for service. Web services are currently the most popular choice for implementing services. However, that is not the same thing as claiming that a Web service is automatically a service.

Exposing functionality as a Web service without any consideration of how it will contribute to the whole architecture of services will probably not lead to a "real" service - that is, a real unit of service-oriented logic. Web services that are produced in this manner are more likely to have poor semantic interoperability with other Web services within the same architecture. They can therefore not easily be reused and they cannot communicate effectively with other software programs.

JBOWS is the word of choice to describe an architecture comprised of these poor excuses for services. The acronym comes from the phrase "Just a Bunch of Web Services" [REF-13]. JBOWS is radically different from service-oriented solutions, even though both may be built using Web services technology.

A Service is Not Automatically a Web Service

If a Web service isn't automatically a service, then we might just as well ask ourselves if a service must be implemented as a Web service. There are, in fact, quite a few references to developing services as components and the consequences thereof [REF-14]. Another more recently popular option is the delivery of services as RESTful services. See Figure 7 for a simple comparison.

Figure 7: The currently three most popular ways of implementing a service

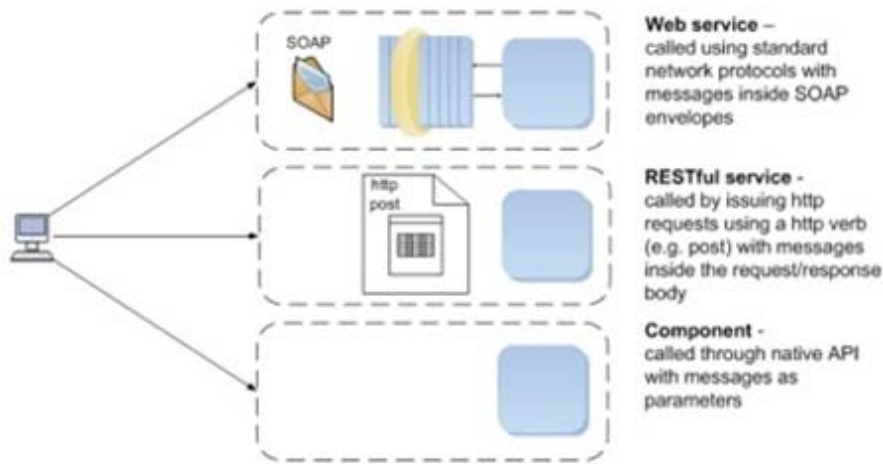


Figure 7

The Cost of Building Services

When practitioners are asked how expensive it is to build services, they tend to have different opinions. Some of the increased cost comes from buying and maintaining the extra infrastructure that is required. Other costs have to do with educating developers and IT professionals so that they know how to take advantage of this new infrastructure. You may also have to spend both time and money educating business people and developers to help them grasp concepts like service-orientation and service-oriented computing. Teaching developers how to handle schemas, versioning, and new security requirements can take some time. Achieving agreement as to how your business should adopt SOA can take much longer.

Agreeing on things like what kind of information is relevant (how the real world should be abstracted into business entities, for example), how this information should be structured and how it should be used by business processes, also can take quite some time. But this effort is required if you want to be able to build business services with intrinsic interoperability and stable contracts. If you take shortcuts and end up with contracts that are non-standardized and prone to change you will find that subsequent business change will have significant ripple effects throughout your architectures (which can be much more expensive than building services properly to begin with). So, while services are more expensive to deliver initially, they are an investment that pays off and actually results in a dramatically more cost-effective enterprise than one based on traditional monolithic applications.

These considerations apply just as much to single-purpose services, except that these services are often less expensive to build than multi-purpose (reusable) services because they do not need to take reuse considerations into account.

The Performance Issue

A common perception is that the use of a Web Service or a RESTful service will affect performance negatively. It is difficult to argue against this because calling a resource over a network will always be slower than accessing that same resource locally. Additionally, calling a service over the network means that you must serialize and deserialize data. There is no way we can claim that this will be faster than not serializing and deserializing the same data.

If performance is our only concern then we can certainly ensure that all resources are available locally. Unfortunately, this is not an option in most enterprises because it would add a lot of cost and we actually need to share a lot of the resources, like data stores.

To assess the performance consideration in relation to single-purpose services, we first need to agree that performance is not something we can calculate based upon processor power and how much work needs to be carried out. Performance, at least for the sake of this article, is the time it takes for the consumer of the service to get the job done.

Network roundtrip time [REF-15] is the time it takes to transmit something to and get a response back from two points in a network. This measurement doesn't take any of the processing that our applications or services do into account. It is purely about network latency. This latency consists of several parts, two of which are: distance delay (the amount of time it takes for a message to travel along the network) and queue delay (the amount of time a message has to spend in a network buffer).

If network latency is considerably higher between the consumer and the service than it is between services, then the network roundtrip time can be reduced by creating an extra service (see Figure 8 for a thought experiment on this).

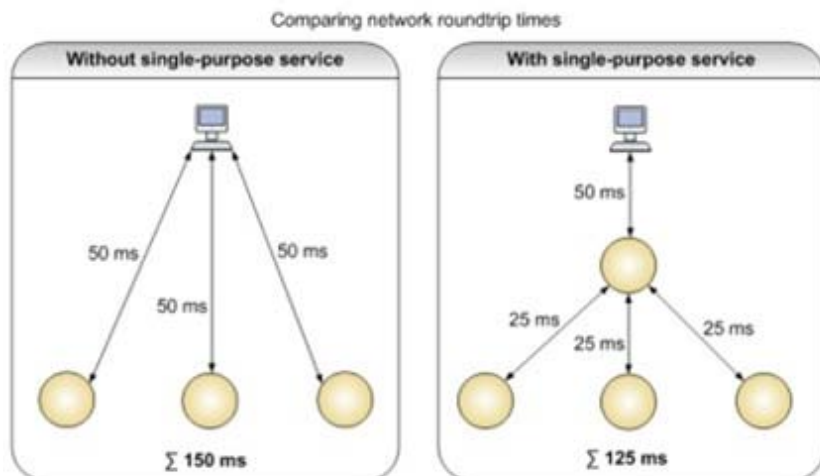


Figure 8

Figure 8: The network roundtrip time might be positively affected by positioning a single-purpose service between the consumer and a number of multi-purpose services if the network latency between the consumer and the services is higher than that between services

The point is that if the amount of time we save on reduced network latency is greater than the time we spend on the serialization and deserialization a Web Service or a RESTful service, the service can out-perform an application that does the actual orchestration itself. Utilizing asynchronous messaging or [Asynchronous Queuing](#) might further mitigate this problem by not locking the consumer runtime resources and thereby making network latency less of an issue.

The Component-First Strategy

If you do not feel that you are ready to create a Web service or RESTful service implementation for your service, then I have a recommendation for you:

Create a service but implement it as a component. This means applying Erl's [Dual Protocols](#) pattern. Be sure that you understand that you will consequently be unable to fully apply the pattern [Canonical Protocol](#). This approach can reduce some of the costs associated with creating services. For instance you do not need to put a lot of work into handling multiple concurrent calls or dealing with other issues that can often be challenging to implement when you are limited to just one protocol. You can keep these considerations in mind as future improvements when you design your component, but you do not need to actually implement the service so that it is able to handle these things when it is initially delivered. The service contract will be a proprietary API consisting of methods with parameters and should be modeled to suit the underlying services that it is composing.

One of my favorite patterns, which I could discuss at length, is [Canonical Schema](#) and I recommend that you use it in your SOA. It involves establishing standardized, compatible schemas for your service inventory which are positioned as part of the technical service contracts. The aim of this pattern is to avoid having to resort to [Data Model Transformation](#), but there are several other benefits as well. If you have successfully applied [Canonical Schema](#), then creating a relatively stable contract for your single-purpose service will be quite easy.

When you create the API for your component implementation you can pick schemas from your library with the confidence that it will be quite easy to communicate with any services that need to be composed. You could accept XML messages that validate against your schemas as parameters in your component, or you could construct Value Objects [REF-16] that mimic the structure of the schemas.

As the service is implemented as a component, there will be very little extra cost added compared to putting all this code into a non-service-oriented application. Furthermore you have constructed the component in such a way that the transition from component to Web service or RESTful service (Figure 9) requires less work. Obviously you will have to do some extra design and development to realize this. The service might not be reused by a lot of applications, but it could still be used a lot and that means that you have to do some extra work. The good news is that your contract will remain relatively stable and the amount of change that you have to make in your non-service-oriented code is limited.

Figure 9: Start out by implementing the service

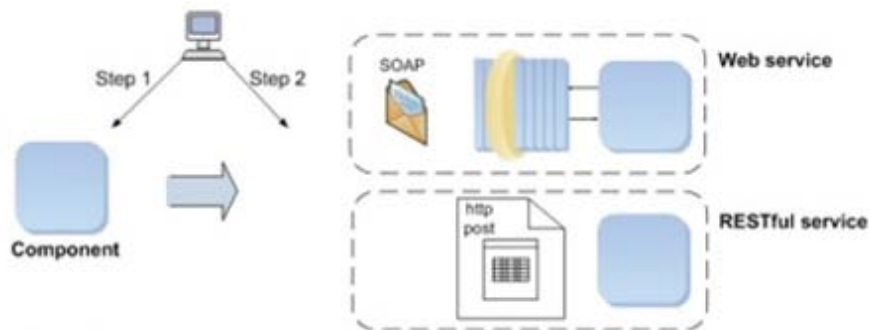


Figure 9

as a component and change it into Web Service or a RESTful services when or if that becomes more suitable.

The performance of service-oriented components will be very similar to the performance you would get if you incorporated the service-oriented logic natively into a regular application (which might be worse than that of a Web service or RESTful service as previously explained). If performance is your main concern I recommend investing more in the testing cycle. When you have test data that compares the performance of different implementation options for your particular environment, you will have enough information to make the right choices.

Conclusion

The decision to put logic into a service or not should be based on many different factors. Unless you investigate requirements, usage, and environments you may not make the correct initial decision. Using a strategy that allows you to rethink your choice with relative ease is sometimes the only acceptable way because many of the factors that affect our choices tend to change over time. Creating single-purpose services as components is a recommended way of responding to situations where you need to build services, but you cannot justify the extra investment that Web services or RESTful services demand.

As a final word of advice I would say that if you do decide to implement your single-purpose service as a Web service or a RESTful service it is a good idea to implement it in such a way that you can easily change it into a component if that makes sense in the future.

References

- [REF-1] Erl, Thomas. SOA Design Patterns. Prentice Hall, 2008.
- [REF-2] See for instance: <http://www.esdswg.com/softwarereuse/Resources/library/reuse-definitions/>
- [REF-3] See Erl, Thomas. Service-Oriented Architecture Concepts, Technology and Design. Prentice Hall, 2006, Appendix B for more categories.
- [REF-4] Brown, Paul C. To SOA or Not to SOA: That is the Question. <http://blog.total-architecture.com/archives/10>
- [REF-5] The "Enterprise Domain Repository" pattern <https://edr.dev.java.net/>
- [REF-6] Erl, Thomas. SOA Principles of Service Design. Prentice Hall, 2008, p 63.
- [REF-7] Interview with Jim Webber: Business people are spaghetti-heads. <http://www.nsilverbullet.net/2008/05/29/JimWebberBusinessPeopleAreSpaghettiheads.aspx>
- [REF-8] See for instance Pace & Carraro. Head in the Cloud, Feet on the Ground. The Architecture Journal #17.
- [REF-9] Whittle & Myric. Enterprise Business Architecture. Auerbach, 2005, p 177.
- [REF-10] McComb, Dave. Semantics in Business Systems. Morgan Kaufman, 2003, pp 167 - 172.
- [REF-11] Evans, Eric. Domain Driven Design. Addison Wesley, 2004, p 109.
- [REF-12] Evans, Eric. Domain Driven Design. Addison Wesley, 2004, p 115.
- [REF-13] McKendrick, Joe. The Rise of the JBOWS Architecture. http://www.webservices.org/weblog/joe_mckendrick/the_rise_of_the_jbows_architecture_or_just_a_bunch_of_web_services
- [REF-14] See for instance Erl, Thomas. SOA Principles of Service Design. Prentice Hall, 2008, p 176.
- [REF-15] All related definitions from Davis, Kevin. Understanding Latency in Network Systems. <http://www.netqos.com/resourceroom/whitepapers/pdf/Sources%20of%20Latency.pdf>
- [REF-16] Evans, Eric. Domain Driven Design. Addison Wesley, 2004, pp 97-103.
- [REF-17] SOAPatterns.org Community Site

I would like to thank Thomas Rischbeck (IPT) and Joshua Anthony (Objectware) for their valuable comments on this article and Peter Tallungs (Objectware) for sharing his insights into uncovering processes in enterprises.

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.