webservices.xml.com


# What Is Service-Oriented Architecture

by Hao He
September 30, 2003

"Things should be made as simple as possible, but no simpler." -- Albert Einstein

## Introduction

Einstein made that famous statement many decades ago, and it's still relevant today for building superior software systems. Unfortunately, as anyone who has been in the IT industry for long can point out, far too many software systems have failed Einstein's test. Some are made too simple to carry out the duties they are supposed to perform. Others are made too complex, and the costs of building and maintaining them have rocketed, not to mention the nearly impossible tasks of integrating different systems together. It seems that reaching the right level of simplicity is more like a dream than reality. Where have we gone wrong?

## Loose Coupling

We don't have to look far to find the problems. As we build more and more software systems, we see similar situations and patterns appearing. Naturally, we want to reuse the functionality of existing systems rather than building them from scratch. A real dependency is a state of affairs in which one system depends on the functionality provided by another. If the world only contained real dependencies, Einstein's test would have been satisfied long time ago. The problem is that we also create artificial dependencies along with real dependencies.

If you travel overseas on business, you know that you must bring power adapters along with you or your life will be miserable. The real dependency is that you need power; the artificial dependency is that your plug must fit into the local outlet. Looking at all the varying sizes and shapes of those plugs from different countries, you would notice that some of them are small and compact while many others are big and bulky.

The lesson here is that we cannot remove artificial dependencies, but we can reduce them. If the artificial dependencies among systems have been reduced, ideally, to their minimum, we have achieved loose coupling. In that sense, Einstein was just talking about was loose coupling. We might rework his famous principle thus: "Artificial dependencies should be reduced to the minimum but real dependencies should not be altered."

## SOA Defined and Explained

Now we are able to define a Service Oriented Architecture (SOA). SOA is an architectural style whose goal is to achieve loose coupling among interacting software agents. A service is a unit of work done by a service provider to achieve desired end results for a service consumer. Both provider and consumer are roles played by software agents on behalf of their owners.

This sounds a bit too abstract, but SOA is actually everywhere. Let 's look at an example of SOA which is likely to be found in your living room. Take a CD for instance. If you want to play it, you put your CD into a CD player and the player plays it for you. The CD player offers a CD playing service. Which is nice because you can replace one CD player with another. You can play the same CD on a portable player or on your expensive stereo. They both offer the same CD playing service, but the quality of service is different.

The idea of SOA departs significantly from that of object oriented programming, which strongly suggests that you should bind data and its processing together. So, in object oriented programming style, every CD would come with its own player and they are not supposed to be separated. This sounds odd, but it's the way we have built many software systems.

The results of a service are usually the change of state for the consumer but can also be a change of state for the provider or for both. After listening to the music played by your CD player, your mood has changed, say, from "depressed" to "happy". If you want an example that involves the change of states for both, dining out in a restaurant is a good one.

The reason that we want someone else to do the work for us is that they are experts. Consuming a service is usually cheaper and more effective than doing the work ourselves. Most of us are smart enough to realize that we are not smart enough to be expert in everything. The same rule applies to building software systems. We call it "separation of concerns", and it is regarded as a principle of software engineering.

How does SOA achieve loose coupling among interacting software agents? It does so by employing two architectural constraints:

1. A small set of simple and ubiquitous interfaces to all participating software agents. Only generic semantics are encoded at the interfaces. The interfaces should be universally available for all providers and consumers.
2. Descriptive messages constrained by an extensible schema delivered through the interfaces. No, or only minimal, system behavior is prescribed by messages. A schema limits the vocabulary and structure of messages. An extensible schema allows new versions of services to be introduced without breaking existing services.

As illustrated in the power adapter example, interfacing is fundamentally important. If interfaces do not work, systems do not work. Interfacing is also expensive and error-prone for distributed applications. An interface needs to prescribe system behavior, and this is very difficult to implement correctly across different platforms and languages. Remote interfaces are also the slowest part of most distributed applications. Instead of building new interfaces for each application, it makes sense to reuse a few generic ones for all applications.

Since we have only a few generic interfaces available, we must express application-specific semantics in messages. We can send any kind of message over our interfaces, but there are a few rules to follow before we can say that an architecture is service oriented.

First, the messages must be descriptive, rather than instructive, because the service provider is responsible for solving the problem. This is like going to a restaurant: you tell your waiter what you would like to order and your preferences but you don't tell their cook how to cook your dish step by step.

Second, service providers will be unable to understand your request if your messages are not written in a format, structure, and vocabulary that is understood by all parties. Limiting the vocabulary and structure of messages is a necessity for any efficient communication. The more restricted a message is, the easier it is to understand the message, although it comes at the expense of reduced extensibility.

Third, extensibility is vitally important. It is not difficult to understand why. The world is an ever-changing place and so is any environment in which a software system lives. Those changes demand corresponding changes in the software system, service consumers, providers, and the messages they exchange. If messages are not extensible, consumers and providers will be locked into one particular version of a service. Despite the importance of extensibility, it has been traditionally overlooked. At best, it was regarded simply as a good practice rather than something fundamental. Restriction and extensibility are deeply entwined. You need both, and increasing one comes at the expense of reducing the other. The trick is to have a right balance.

Fourth, an SOA must have a mechanism that enables a consumer to discover a service provider under the context of a service sought by the consumer. The mechanism can be really flexible, and it does not have to be a centralized registry.

### Additional Constraints

There are a number of additional constraints one can apply on SOA in order to improve its scalability, performance and, reliability.

### Stateless Service

Each message that a consumer sends to a provider must contain all necessary information for the provider to process it. This constraint makes a service provider more scalable because the provider does not have to store state information between requests. This is effectively "service in mass production" since each request can be treated as generic. It is also claimed that this constraint improves visibility because any monitoring software can inspect one single request and figure out its intention. There are no intermediate states to worry about, so recovery from partial failure is also relatively easy. This makes a service more reliable.

### Stateful Service

Stateful service is difficult to avoid in a number of situations. One situation is to establish a session between a consumer and a provider. A session is typically established for efficiency reasons. For example, sending a security certificate with each request is a serious burden for both any consumer and provider. It is much quicker to replace the certificate with a token shared just between the consumer and provider. Another situation is to provide customized service.

Stateful services require both the consumer and the provider to share the same consumer-specific context, which is either included in or referenced by messages exchanged between the provider and the consumer. The drawback of this constraint is that it may reduce the overall scalability of the service provider because it may need to remember the shared context for each consumer. It also increases the coupling between a service provider and a consumer and makes switching service providers more difficult.

### Idempotent Request

Duplicate requests received by a software agent have the same effects as a unique request. This constraint allows providers and consumers to improve the overall service reliability by simply repeating the request if faults are encountered.

### Deriving Web Services from SOA

Everyone knows roughly what a "web service" is, but there is no universally accepted definition. The definition of web service has always been under hot debate within the W3C Web Services Architecture Working Group. Despite the difficulty of defining web services, it is generally accepted that a web service is a SOA with at least the following additional constraints:

1. Interfaces must be based on Internet protocols such as HTTP, FTP, and SMTP.
2. Except for binary data attachment, messages must be in XML.

There are two main styles of Web services: SOAP web services and REST web services.

## SOAP Web services

A SOAP web service introduces the following constraints:

1. Except for binary data attachment, messages must be carried by SOAP.
2. The description of a service must be in WSDL.

A SOAP web service is the most common and marketed form of web service in the industry. Some people simply collapse "web service" into SOAP and WSDL services. SOAP provides "a message construct that can be exchanged over a variety of underlying protocols" according to the SOAP 1.2 Primer. In other words, SOAP acts like an envelope that carries its contents. One advantage of SOAP is that it allows rich message exchange patterns ranging from traditional request-and-response to broadcasting and sophisticated message correlations. There are two flavors of SOAP web services, SOAP RPC and document-centric SOAP web service. SOAP RPC web services are not SOA; document-centric SOAP web services are SOA.

## SOAP RPC Web Services

A SOAP RPC web service breaks the second constraint required by an SOA. A SOAP RPC Web service encodes RPC (remote procedure calls) in SOAP messages. In other words, SOAP RPC "tunnels" new application-specific RPC interfaces though an underlying generic interface. Effectively, it prescribes both system behaviors and application semantics. Because system behaviors are very difficult to prescribe in a distributed environment, applications created with SOAP RPC are not interoperable by nature. Many real life implementations have confirmed this.

Faced with this difficulty, both WS-I basic profile and SOAP 1.2 have made the support of RPC optional. RPC also tends to be instructive rather than descriptive, which is against the spirit of SOA. Ironically, SOAP was originally designed just for RPC. It won't be long before someone claims that "SOAP" actually stands for "SOA Protocol".

## REST Web Services

The term REST was first introduced by Roy Fielding to describe the web architecture. A REST web service is an SOA based on the concept of "resource". A resource is anything that has a URI. A resource may have zero or more representations. Usually, people say that a resource does not exist if no representation is available for that resource. A REST web service requires the following additional constraints:

1. Interfaces are limited to HTTP. The following semantics are defined:
    a. HTTP GET is used for obtaining a representation of a resource. A consumer uses it to retrieve a representation from a URI. Services provided through this interface must not incur any obligation from consumers.
    b. HTTP DELETE is used for removing representations of a resource.
    c. HTTP POST is used for updating or creating the representations of a resource.
    d. HTTP PUT is used for creating representations of a resource.
2. Most messages are in XML, confined by a schema written in a schema language such as XML Schema from W3C or RELAX NG.
3. Simple messages can be encoded with URL encoding.
4. Service and service providers must be resources while a consumer can be a resource.

REST web services require little infrastructure support apart from standard HTTP and XML processing technologies, which are now well supported by most programming languages and platforms. REST web services are simple and effective because HTTP is the most widely available interface, and it is good enough for most applications. In many cases, the simplicity of HTTP simply outweighs the complexity of introducing an additional transport layer.

## Related Reading

A Web Services Primer
Representational State Transfer (REST), Chapter 5
Architecture of the World Wide Web
Web Services Architecture
Web Services are not Distributed Objects: Common Misconceptions about Service Oriented Architectures
REST and the Real World
A Brief History of SOAP
Web Services and Sessions
Web Architecture: Extensible Languages
Web Services Glossary

## Acknowledgments

The author would like to thank Bill Donoghoe and Mark Baker for their valuable comments. The author is grateful to all the people who participated the SOA thread in the W3C Web Services mail list. Thanks also go to Janet Aylwin, Ian Campbell, Adam Davis and Peter Drummond for proof reading and their suggestions that have made the article more enjoyable to read.

*The opinions expressed herein are those of the author and do not necessarily reflect the opinions of The Thomson Corporation with regard to this subject.*