Service-Oriented Architecture expands the vision of web services, Part 1

Characteristics of Service-Oriented Architecture

Mark Colan

April 21, 2004

Today's web services implementations are typically simple and often similar to a client-server model. However, platform-neutral interchange is supported, which allows a diverse range of client implementations to interact with new or legacy code as server functions. Much has been written about the technologies that make such applications straightforward to implement. It is now time to look at the bigger picture of what we can do with them. The author addresses the question of how to move forward from simple models to those that represent real-world business models of arbitrary complexity.

In 221B.C., Emperor Qin unified several formerly warring states into a new country, which we now call China. Perhaps one reason China has endured as a nation was Qin's introduction of standards, which consolidated cultures and facilitated trade: a standard distance for wheels on carts which allowed them to travel efficiently on any road, a common written language that everybody could use to exchange messages (even if they did not speak the same language), and a strong defense against outside attacks (like the Great Wall of China). You could say that he developed the models for standardized transport, message exchange, and the firewall.

In the same way, business integration of modern times has benefited from standards that enable heterogeneous computer systems to interoperate efficiently. These technologies are referred to, collectively, as web services. The dawn of web services was marked by the introduction of SOAP 1.1, which defined the use of XML content for interaction in distributed systems while hiding the implementation details. Four years later, many companies are using web services, and you could safely say that the industry is at the beginning of the web services mainstream era.

IBM sees Service-Oriented Architecture (SOA) as key to interoperability and flexibility requirements for its vision of on demand business. SOA supports end-to-end integration across the enterprise and among business partners. This provides a flexible business process model that allows customers to respond quickly to new customer requirements, new business opportunities, and competitive threats.
**What is SOA?**

SOA presents the big picture of what you can do with web services. Web services specifications define the details needed to implement services and interact with them. However, SOA is an approach to build distributed systems that deliver application functionality as services to end-user applications or to build other services. SOA can be based on web services, but it may use other technologies instead. In using SOA to design distributed applications, you can expand the use of web services from simple client-server models to systems of arbitrary complexity.

Thus, individual software assets become the building blocks to develop other applications. You can reduce the complexity of systems by using a common style of interaction that works with both new and legacy code (Lawrence Wilkes, of CBDi, has joked that SOA could stand for “Save Our Assets”). There is a standard way of representing and interacting with these software assets; now the focus shifts to application assembly based on these building blocks.

While discussing SOA for business applications here, SOA is also being used for other distributed systems such as grid computing and advanced web services specifications (for example, WS-DistributedManagement, WS-Trust, and UDDI).

**What is a service?**

A *service* in SOA is an application function packaged as a reusable component for use in a business process. It either provides information or facilitates a change to business data from one valid and consistent state to another. The process used to implement a particular service does not matter, as long as it responds to your commands and offers the quality of service you require.

Through defined communication protocols, services can be invoked that stress interoperability and location transparency. A service has the appearance of a software component, in that it looks like a self-contained function from the service requester's perspective. However, the service implementation may actually involve many steps executed on different computers within one enterprise or on computers owned by a number of business partners. A service might or might not be a component in the sense of encapsulated software. Like a class object, the requester application is capable of treating the service as one.

Web services are based on invocation using SOAP messages which are described using WSDL over a standard protocol such as HTTP. Use of web services is a best practice when communicating with external business partners.

**Loose coupling**

The binding from the service requester to the service provider should loosely couple the service. This means that the service requester has no knowledge of the technical details of the provider’s implementation, such as the programming language, deployment platform, and so forth. The service requester typically invokes operations by way of messages -- a request message and the response -- rather than through the use of APIs or file formats.

This loose coupling allows software on each side of the conversation to change without impacting the other, provided that the message schema stays the same. In an extreme case, a service
provider could replace an early implementation based on legacy code, such as COBOL, with an entirely new code base based on Java language without having any impact on the service requester. This case is true as long as the new code supports the same message schema.

**Well-defined interfaces**

The service interaction must be well-defined. Web services Description Language (WSDL) is a widely-supported way of describing the details required by a service requester for binding to a service provider. The service descriptions focus on operations used to interact with the following:

- A service
- Messages to invoke operations
- Details of constructing such messages
- Information on where to send messages for processing
tails of constructing such messages

WSDL does not include any technology details of the implementation of a service. The service requester neither knows nor cares whether the service is written in Java code, C#, COBOL, or some other programming language. It can describe a SOAP invocation using HTTP. Because of its extension mechanisms, it can also define other styles of interaction such as XML content delivered via JMS, direct method calls, calls handled by an adapter that manages legacy code (CICS), and so forth.

The common definition for WSDL allows development tools to create common interfaces for various styles of interaction, while hiding the details of how it invokes the service from the application code. The Web Services Invocation Framework (WSIF), for example, exploits this capability by allowing a run-time determination of the best way to invoke a quality service if the service is exposed in more than one interaction style.

**Stateless service design**

Services should be independent, self-contained requests, which do not require information or state from one request to another when implemented. Services should not be dependent on the context or state of other services. When dependencies are required, they are best defined in terms of common business processes, functions, and data models, not implementation artifacts (like a session key). Of course, requester applications require persistent state between service invocations, but this should be separate from the service provider.

Here is an example of the **wrong way** to define a conversation:

Requester: “What is Bruce’s checking account balance?"
Provider: “$x"
Requester: “And what is his credit limit?"
Provider: “$y"

The provider is required to remember Bruce's account between requests, which introduces complexity into the service implementation. Stateless service design would redefine the conversation as follows:
Service granularity

The granularity of operations is an important design point. The use of coarse-grained interfaces for external consumption is recommended, whereas fine-grained interfaces might be used inside the enterprise. A coarse-grained interface might be the complete processing for a given service, such as `SubmitPurchaseOrder`, where the message contains all of the business information needed to define a purchase order. A fine-grained interface might have separate operations for: `CreateNewPurchaseOrder`, `SetShippingAddress`, `AddItem`, and so forth.

While the fine-grained interface offers more flexibility to the requester application, it also means that patterns of interaction may vary between different service requesters. This can make support more difficult for the service provider. A coarse-grained interface guarantees that the service requesters will use the service in a consistent manner. SOA does not require the use of coarse-grained interfaces, but recommends their use as a best practice for external integration. Service choreography can be used to create a coarse-grained interface that runs a business process consisting of fine-grained operations.

Quality of service considerations

SOA designs will span computer systems and might cross enterprise lines. You have to think about the security capabilities and requirements when using the Internet and how to link across partners’ security domains. Internet protocols are not designed for reliability (guaranteed delivery and order of delivery), but you have to ensure that a message is delivered and processed one time. When this is not possible, the requester must know that a request has not been processed.

For example, you may need to consider metering, availability, and response time of services you deploy in order to ensure that they are within the promised range. As you design systems that use services from other business partners, you have to consider service-oriented management to cooperatively manage services among partners.

Conclusion

This article has focused on the advantages of SOA for expanding the value of software assets, while managing complexity in distributed applications. It has also defined the nature of services and examined their characteristics, as well as a few best practices.

The next article will examine architectural concepts of SOA from the following list:

- Service invocation and description to discovery
- The broker model
- Internal integration
- Externalization of services