SERVICE ORIENTED ARCHITECTURE AND WEB SERVICES

Presented by

P.PUNEETH
B.Tech III CSE, A.I.T.S, Tirupati
Email: puneeth.pulla560@gmail.com

E.VISHNU VARDHAN
B.Tech III CSE, A.I.T.S, Tirupati
Email: vishnuvardhan.eyunni@gmail.com

ABSTRACT:

*In software engineering, a **Service-Oriented Architecture (SOA**) is a set of principles and methodologies for designing and developing software in the form of interoperable services. These services are well-defined business functionalities that are built as software components (discrete piece of code and/or data structures) that can be reused for different purposes. SOA design principles are used during the phases of systems development and integration.*

SOA also generally provides a way for consumers of services, such as web-based applications, to be aware of available SOA-based services. For example, several disparate departments within a company may develop and deploy SOA services in different implementation languages; their respective clients will benefit from a well-understood, well-defined interface to access them. XML is often used for interfacing with SOA services, though this is not required. JSON is also becoming increasingly common.

SOA defines how to integrate widely disparate applications for a Web-based environment and uses multiple implementation platforms. Rather than defining an API, SOA defines the interface in terms of protocols and functionality. An endpoint is the entry point for such a SOA implementation.

Service-orientation requires loose coupling of services with operating systems, and other technologies that underlie applications. SOA separates functions into distinct units, or services, which developers make accessible over a network in order to allow users to combine and reuse them in the production of applications. These services and their corresponding consumers communicate with each other by passing data in a well-defined, shared format, or by coordinating an activity between two or more services.

*A **Web service** is a method of communication between two electronic devices over the web (internet).*

The World Wide Web Consortium defines a "Web service" as "a software system designed to support interoperable machine-to-machine interaction over a network". It has an interface described in a machine-processable format (specifically Web Services Description Language, known by the acronym WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services platform elements are

- *SOAP (Simple Object Access Protocol)*

SOAP is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks

- *UDDI (Universal Description, Discovery and Integration)*

UDDI is a platform-independent, Extensible Markup Language (XML)-based registry for businesses worldwide to list themselves on the Internet and a mechanism to register and locate web service applications.

- *WSDL (Web Services Description Language)*

WSDL is an XML-based language that provides a model for describing Web services.

Service Oriented Architecture:

In considering the term *service-oriented architecture*, it is useful to review the key terms.

- Architecture is a formal description of a system, defining its purpose, functions, externally visible properties, and interfaces. It also includes the description of the system's internal components and their relationships, along with the principles governing its design, operation, and evolution.
- A **service** is a software component that can be accessed via a network to provide functionality to a service requester.
- The term **service-oriented architecture** refers to a style of building reliable distributed systems that deliver functionality as *services*, with the additional emphasis on loose coupling between interacting services.

Technically, then, the term SOA refers to the *design* of a system, not to its implementation. Although it is commonplace for the term to be used in referring to an implementation—for example, in phrases such as "building an SOA"—in this paper we avoid this use, and we use the adjective *service- oriented* in contexts such as "service-oriented environment" or "service-oriented grid."

We regard SOA as an *architectural style* that emphasizes implementation of components as modular *services* that can be discovered and used by clients. Services generally have the following characteristics:

Services may be individually useful, or they can be integrated—composed—to provide higher-level services. Among other benefits, this promotes re-use of existing functionality.

- Services communicate with their clients by exchanging messages: they are defined by the messages they can accept and the responses they can give.
- Services can participate in a workflow, where the order in which messages are sent and received affects the outcome of the operations performed by a service. This notion is defined as "service Choreography."
- Services may be completely self-contained, or they may depend on the availability of other services, or on the existence of a resource such as a database. In the simplest case, a service might perform a calculation such as computing the cube root of a supplied number without needing to refer to any external resource, or it may have pre-loaded all the data that it needs for its lifetime. Conversely, a service that performs currency conversion would need real-time access to exchange-rate information in order to yield correct values.
- Services advertise details such as their capabilities, interfaces, policies, and supported communications protocols. Implementation details such as programming language and hosting platform are of no concern to clients, and are not revealed.

Figure 1 illustrates a simple service interaction cycle, which begins with a service advertising itself through a well-known registry service (1). A potential client, which may or may not be another service, queries the registry (2) to search for a service that meets its needs. The registry returns a (possibly empty) list of

suitable services, and the client selects one and passes a request message to it, using any mutually recognized protocol (3). In this example, the service responds (4) either with the result of the requested operation or with a fault message.

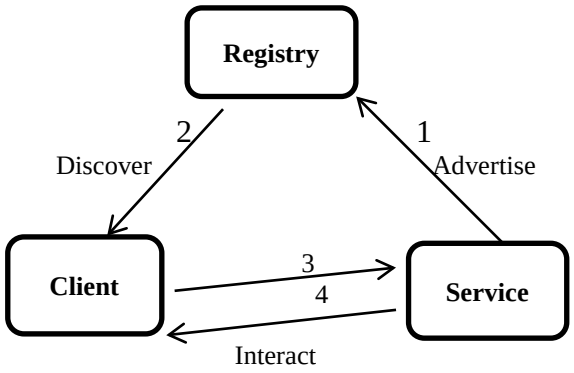

Figure 1: Service interaction in a service-oriented environment

The illustration shows the simplest case, but in a real-world setting such as a commercial application the process may be significantly more complex. For example, a given service may support only the HTTPS Protocol, be requested to authorized users, require Kerberos authentication, offer different levels of performance to different users, or require payment for use. Services can provide such details in a variety of ways, and the client can use this information to make its selection. Some attributes, such as payment terms and guaranteed levels of service, may need to be established by a process of negotiation before the client can make use of the service it has selected. And, while this illustration shows a simple synchronous, bi-directional message exchange pattern, a variety of patterns are possible—for example, an interaction may be one-way, or the response may come not from the service to which the client sent the request, but from some other service that completed the transaction.

LOOSE COUPLING:

In our definition of SOA, we included the term *loose coupling*. This term implies that the interacting software components minimize their in-built knowledge of each other: they discover the information they need at the time they need it. For example, having learned about a service's existence, a client can discover its capabilities, its policies, its location, its interfaces and its supported protocols. Once it has this knowledge, the client can access the service using any mutually acceptable protocol. The word "frictionless" has been used to describe the ultimate goal of loose coupling, and the word aptly conjures up a vision of components that communicate almost without contact.

The benefits of loose coupling include:

Flexibility:

Loosely-coupled services are typically more flexible than more tightly-coupled applications. In a tightly-coupled architecture, the different components of an application are tightly bound to each other, sharing semantics, libraries, and often sharing state. This makes it difficult to evolve the application to keep up with changing business requirements. The loosely-coupled, document-based, asynchronous nature of services in an SOA allows applications to be flexible, and easy to evolve with changing requirements.

Scalability:

The services in an SOA are loosely coupled, applications that use these services tend to scale easily -- certainly more easily than applications in a more tightly-coupled environment. That's because there are few dependencies between the requesting application and the services it uses. The dependencies between client and service in a tightly-coupled environment are compounded (and the development effort made significantly more complex) as an application that uses

these services scales up to handle more users.

Reusability:
Developers within an enterprise and across enterprises (particularly, in business partnerships) can take the code developed for existing business applications, expose it as web services, and then reuse it to meet new business requirements. Reusing functionality that already exists outside or inside an enterprise instead of developing code that reproduces those functions can result in a huge savings in application development cost and time. The benefit of reuse grows dramatically as more and more business services get built, and incorporated into different applications.
A major obstacle in taking advantage of existing code is the uniqueness of specific applications and systems. Typically, solutions developed in different enterprises, even different departments within the same enterprise, have unique characteristics. They run in different operating environments, they're coded in different languages, they use different programming interfaces and protocols. You need to understand how and where these applications and systems run to communicate with them.

Cost Efficiency:

Other approaches that integrate disparate business resources such as legacy systems, business partner applications, and department-specific solutions are expensive because they tend to tie these components together in a customized way. Customized solutions are costly to build because they require extensive analysis, development time, and effort. They're also costly to maintain and extend because they're typically tightly-coupled, so that changes in one component of the integrated solution require changes in other components. A standards-based approach such as a web services-based SOA should result in less costly solutions because the integration of clients and services doesn't require the in-depth analysis and unique code of customized solutions. Also, because services in an SOA are loosely-coupled, applications that use these services should be less costly to maintain and easier to extend than customized solutions. This is potentially the biggest cost saving of all.

STATE AND STATELESSNESS:

A key notion of loose coupling is *statelessness*—a topic that has been much-discussed and is often mentioned as a critical requirement, sometimes without a clear understanding of its significance.

Simply, the benefits of loose coupling, as listed above, are derived from the fact that a client can choose to go to *any* service that is capable of fulfilling its need. If its choice is restricted to a single service then a tight coupling exists between the client and the server, and the benefits of loose coupling are diminished.

For a more complex transaction that requires several steps, however, the design of the service might be such that the service retains in its local memory some information (“state”) about the first step, expecting to make use of it when the client contacts it for the next step. In this case, the service is “stateful,” and the client *must* return to the same service for the next step. This might result in a delay if many clients are using the same service or in a transaction failure if the node hosting the service fails between steps.

A better approach to the design of the service is for it not to retain state about the transaction, but to be “stateless.” This implies that in a multi-step transaction:

- At the end of each intermediate step the service must hand back to the client sufficient state information to enable *any* qualified service to identify and continue the transaction.

- The client must hand the state information to whichever service it selects to process the next step of the transaction.
- The selected service must be able to accept and handle the state information supplied by the client, regardless of whether it processed the earlier steps itself.

Figure 2 shows a client engaged in a three-step transaction with several services, each of which might be capable of handling any part or all of the transaction. The service that handles Step 1 stores the details of the in-progress transaction in the database, and returns requested information to the client, along with a transaction identifier. The client might request confirmation from the user before passing the transaction identifier to another service, which uses it to retrieve the state information from the database and initiates Step 2. This service then updates the database and returns additional information to the client. Finally, the client passes the transaction identifier back to a third service with a request to complete the transaction.

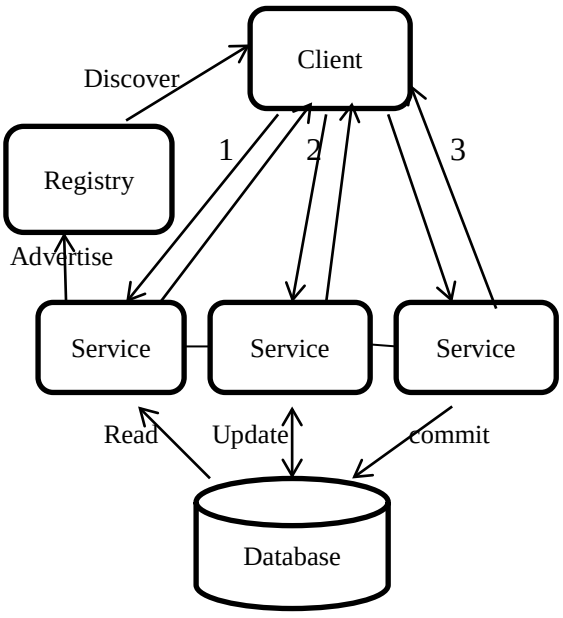

Figure 2: A multi-step client/service interaction

The approach outlined above enhances loose coupling by separating the transaction's state from the services that operate on it. In the example, both the account data and the details of the transaction can be considered to be state information, but the account data is permanent, while the transaction details only need to exist while the transaction is in progress.

WEB SERVICES:

Most people are familiar with accessing the Web through a Web browser, which provides a human-oriented interface to information and user-oriented services such as on-line auctions and retail stores. When a user requests a Web page, the request is handled by a remote Web server, which returns the information in *hypertext mark-up language* (HTML)—a form that allows the browser to present it using a selection of fonts, colours and pictures, all factors that make it more useful and appealing to a human.

Web services are distributed software components that provide information to *applications* rather than to humans, through an application-oriented interface. The information is structured using *extensible Markup Language* (XML), so that it can be parsed and processed easily rather than being formatted for display. In a Web-based retail operation, for example, Web services that may be running on widely separated servers might provide account management, inventory control, shopping cart and credit card authorization services, all of which may be invoked multiple times in the course of a single purchase.

Web services publish details of their functions and interfaces, but they keep their implementation details private; thus a client and a service that support common communication protocols can interact regardless of the platforms on which they run, or the programming languages in which they are written. This makes Web

services particularly applicable to a distributed heterogeneous environment.

Although it is common to talk about Web-service “instances,” this can be misleading as it implies that a service at a given address is a clone of other “instances.” It may be true that at a particular moment several running services are derived from the same source code, but if the code is changed and some services continue to run as they were and others are stopped and re-started, are they still all instances of the same service? What if the change added new operations? The answer is that they are all independently running services that offer common functionality, and the word “instance” is superfluous.

The key specifications used by Web services are:

XML:

Extensible Markup Language (XML) has become the de facto standard for describing data to be exchanged on the Web. As its name indicates, XML is a mark-up language. It involves the use of tags that "markup" the contents of a document, and in doing so, describe the contents of a document. An XML tag identifies information in a document, and also identifies the structure of the information.

SOAP:

Simple Object Access Protocol (SOAP) is an XML-based protocol for exchanging information in a distributed environment. SOAP provides a common message format for exchanging data between clients and services. The basic item of transmission in SOAP is a SOAP message, which consists of a mandatory SOAP envelope, an optional SOAP header, and a mandatory SOAP body.

The envelope specifies two things: an XML namespace and an encoding style. The XML namespace specifies the names that can be used in the SOAP message. Namespaces are designed to avoid name clashes -- the same name can be used for different items, provided that the names are in different namespaces. The encoding style identifies the data types recognized by the SOAP message. If a header is provided, it extends the SOAP message in a modular way. It's important to understand that as a SOAP message travels from a client to a service, it can pass through a set of intermediate nodes along the message path. Each node is an application that can receive and forward SOAP messages. An intermediate node can provide additional services such as transform the data in the message or perform security-related operations. The SOAP header can be used to indicate some additional processing at an intermediate node, which is, processing independent of the processing done at the final destination. Typically, the SOAP header is used to convey security-related information to be processed by runtime components. The body contains the main part of the SOAP message, that is, the part intended for the final recipient of the SOAP message.

WSDL:

Web Service Description Language (WSDL) defines an XML schema for describing a web service. A WSDL document describes a web service as a collection of abstract items called "ports" or "endpoints." A WSDL document also

defines the actions performed by a web service and the data transmitted to these actions in an abstract way. Actions are represented by "operations," and data is represented by "messages." A collection of related operations is known as a "port type." A port type constitutes the collection of actions offered by a web service. What turns a WSDL description from abstract to concrete is a "binding." A binding specifies the network protocol and message format specifications for a particular port type. A port is defined by associating a network address with a binding. If a client locates a WSDL document and finds the binding and network address for each port, it can call the service's operations according to the specified protocol and message format.

SOA AND WEB SERVICES: STYLE VS. IMPLEMENTATION:

We initially described SOA without mentioning Web services, and vice versa. This is because they are orthogonal: service-orientation is an architectural style, while Web services are an implementation technology. The two can be used together, and they frequently are, but they are not mutually dependent.

For example, although it is widely considered to be a distributed-computing solution, SOA can be applied to advantage in a single system, where services might be individual processes with well-defined interfaces that communicate using local channels, or in a self-contained cluster, where they might communicate across a high-speed interconnect.

Similarly, while Web services are well-suited as the basis for a service-oriented environment, there is nothing in their definition that *requires* them to embody the SOA principles. While statelessness is often held up as a key characteristic of Web services, there is no technical reason that they should be stateless—that would be a design choice of the developer, which may be dictated by the architectural style of the environment in which the service is intended to participate.

THE ROLE OF STANDARDS:

The success of Web services is due in large part to the development of standards by bodies such as the World Wide Web Consortium (W3C), the Organization for the Advancement of Structured Information Standards (OASIS), and the Distributed Management Task Force (DMTF). These bodies continue to develop and enhance standards for some of the essential underpinnings of Web services, such as describing service interfaces, implementing security and policy management services, and, at a higher level, for implementing business processes.

REFERENCES:

1. World Wide Web Consortium (W3C): http://www.w3.org

2. Chatterjee & Webber: Developing Enterprise Web Services – An Architect's Guide (Prentice Hall)

3. WebServices.org: http://www.webservices.org

4. Wikipedia.org: http://www.wikipedia.org