OPEN SOURCE SOA

Jeff Davis

SAMPLE CHAPTER

MANNING

Open Source SOA

by Jeff Davis

Chapter 1

Copyright 2009 Manning Publications

brief contents

Part 1

History and principles

Service-oriented architecture (SOA) has emerged over the past several years as one of the preferred approaches for systems design, development, and integration. Leveraging open standards and the ubiquity of the internet, SOA is premised on the notion of reusable services that correspond to self-contained, logical units of work. The promise is that these services can be quickly pieced together using common patterns to form new applications that are tightly aligned with the needs of the business. The upshot? Improved business agility and cost-effective utilization of IT resources and assets.

In part 1, we'll examine the history behind SOA and explore some of the commonalities that it shares with earlier architectural and technology approaches. We'll then identify some of the core characteristics of SOA, and explain how they're manifested in actual technologies that can be used in your own enterprise. Collectively, these technologies will combine to form what we are calling the *Open SOA Platform*. Once these technologies, such as business process management (BPM), are identified, our attention will turn to surveying the landscape of possible open source products that can be used to satisfy these technology requirements.

The maturity and adoption of open source products within the enterprise has become widespread. Many of these products are now suitable for use in crafting a technology stack that can support SOA. Some of the major challenges that have precluded more widespread adoption of these solutions in the past pertain to how they can be rationally assessed, and then integrated, within an organization. We'll present requirements for analyzing the product categories of the SOA technology stack, and using them, select what we consider to be the "best of breed" open source solutions for each category. The selection criteria, as we'll see, are also based on how well they can be integrated to form a complete SOA solution. What's more, this can be accomplished at a fraction of the cost of commercial alternatives—an important consideration in today's challenging economic environment.

1

SOA essentials

This chapter covers

- Origins of SOA in distributed computing
- Requirements of a SOA environment
- Key technologies supporting SOA

Ponce de León's early quest to find the "Fountain of Youth" in Florida is one of the most frequently told stories of American folklore. Although he failed in his journey to find the "healing waters," it turns out that he was in good company, for throughout history we can find tales of similar adventures that never materialized. The history of computing bears some resemblance. Every decade or so, a new "silver bullet" emerges that promises to heal the problems that have plagued software development in the past. Those problems include protracted development cycles; solutions that fail to achieve expectations; high maintenance costs; and, of course, the dreaded cost overruns.

The quest is to find a solution that simplifies development and implementation, supports effective reuse of software assets, and leverages the enormous and low-cost computing power now at our fingertips. While some might claim that service-oriented architecture (SOA) is just the latest fad in this illusive quest, tangible results have been achieved by those able to successfully implement its principles.

According to a recent article in the *Harvard Business Journal*, companies that have embraced SOA "have eliminated huge amounts of redundant software, reaped major cost savings from simplifying and automating manual processes, and realized big increases in productivity" [HBJ]. Further, SOA has achieved greater staying power than many earlier alternatives, which does say something of its merits. Perhaps this is because SOA is a more nebulous concept and embraces technologies as much as it does principles and guidelines—thus refuting its benefits becomes more difficult.

Until recently, achieving a technology infrastructure capable of sustaining a SOA generally required purchasing expensive commercial products. This was especially true if an enterprise desired a well-integrated and comprehensive solution. While several early SOA-related open source products were introduced, they tended to focus on specific, niche areas. For example, Apache Axis was first introduced in 2004 and became a widely adopted web services toolkit for Java. As we'll discover, however, web services represent only a piece of the SOA puzzle. Fast-forward to 2008 and we now see commercially competitive open source products across the entire SOA product spectrum. The challenge now for a SOA architect wanting to use open source is how to select among the bewildering number of competing products. Even more challenging is how to integrate them.

The goal of this book is to help you identify the core technologies that constitute a SOA and the open source technologies that you can use to build a complete SOA platform. Our focus will be on how to integrate these core technologies into a compelling solution that's comparable in breadth and depth to the expensive offerings provided by the commercial vendors. SOA is now attainable for even the smallest of enterprises using high-quality open source software. This book will present a technology blueprint for open source SOA. Of course, thanks to the plethora of high-quality open source solutions, you can naturally swap out the solutions I'm advocating with those you deem appropriate.

Before jumping headfirst into the technology stack, let's establish some context for where SOA originated and develop a common understanding of what it is.

1.1 Brief history of distributed computing

The mainframe systems of the 1960s and '70s, such as the IBM System/360 series, rarely communicated with each other. Indeed, one of the main selling points of a mainframe was that it would provide you with everything necessary to perform the computing functions of a business. When communications were required, the process usually amounted to transferring data by way of tape from one system to another. Over time, though, real-time access between systems became necessary, especially as the number of systems within an organization multiplied. This need was especially apparent in financial markets, where trading required real-time transactional settlements that often spanned across companies.

Initially, real-time access was accomplished via low-level socket communications. Usually written in assembly language or C, socket programming was complex and required a deep understanding of the underlying network protocols. Over time, protocols such as Network File System (NFS) and File Transfer Protocol (FTP) came on the scene that abstracted out the complexity of sockets. Companies such as TIBCO emerged that developed "middleware" software explicitly designed to facilitate messaging and communications between servers. Eventually, the ability to create distributed applications became feasible through the development of remote procedure calls (RPCs). RPCs enabled discrete functions to be performed by remote computers as though they were running locally. As Sun Microsystems' slogan puts it, "The Network is the Computer."

By the 1980s, personal computers had exploded onto the scene, and developers were seeking more effective ways to leverage the computing power of the desktop. As the price of hardware came down, the number of servers within the enterprise increased exponentially. These trends, coupled with the growing maturity of RPC, led to two important advances in distributed computing:

- *Common Object Request Broker Architecture (CORBA)*—Originated in 1991 as a means for standardizing distributed execution of programming functions, the first several releases only supported the C programming language. Adoption was slow, as commercial implementations were expensive and the ambiguities within the specification made for significant incompatibilities between vendor products. The 2.0 release in 1998 was significant in that it supported several additional language mappings and addressed many of the shortfalls present in the earlier standards. However, the advent of Java, which dramatically simplified distributed computing through Remote Method Invocation (RMI), and finally, through XML, has largely led to the demise of CORBA (at least in new implementations).
- *Distributed Computing Object Model (DCOM)*—DCOM is a proprietary Microsoft technology that was largely motivated as a response to CORBA. The first implementations appeared in 1993. While successful within the Microsoft world, the proprietary nature obviously limited its appeal. The wider enterprise class of applications that were emerging at the time—Enterprise Resource Planning (ERP) systems—generally used non-Microsoft technologies. Later, Java's Enterprise JavaBeans (EJB) platform could be construed as Java's alternative to DCOM, as it shared many of the same characteristics.

By the late 1990s, with the widespread adoption of the internet, companies began recognizing the benefits of extending their computing platform to partners and customers. Before this, communications among organizations were expensive and had to rely on leased lines (private circuits). Leased lines were impractical except for the largest of enterprises. Unfortunately, using CORBA or DCOM over the internet proved to be challenging, in part due to networking restrictions imposed by firewalls that only permitted HTTP traffic (necessary for browser and web server communications). Another reason was that neither CORBA nor DCOM commanded dominant market share, so companies attempting communication links often had competing technologies.

When the Simple Object Access Protocol (SOAP) first arrived (in January 2000), it was touted as a panacea due to its interoperable reliance on XML. SOAP was largely envisioned as an RPC alternative to CORBA and DCOM. Since RPCs were the predominant model for distributed computing, it naturally followed that SOAP was originally used in a similar capacity. However, RPC-based solutions, regardless of their technology platform, proved nettlesome. (It is worth noting that SOAP's RPC was an improvement over earlier RPC implementations, as it relied on XML as the payload, which facilitates a much higher degree of interoperability between programming languages.)

1.1.1 *Problems related to RPC-based solutions*

While RPC-based distributed computing was no doubt a substantial improvement over earlier lower-level socket-based communications, it suffered from several limitations:

- Tight coupling between local and remote systems requires significant bandwidth demands. Repeated RPC calls from a client to server can generate substantial network load.
- The fine-grained nature of RPC requires a highly predictable network. Unpredictable latency, a hallmark of internet-based communications, is unacceptable for RPC-based solutions.
- RPC's data type support, which aims to provide complete support for all native data types (arrays, strings, integers, etc.), becomes challenging when attempting to bridge between incompatible languages, such as C++ and Java. Often, incompatibilities result, greatly complicating its use.

SOAP RPC-style messages also suffered from the same inherent limitations as those mentioned here. Fortunately, SOAP offers alternative message styles that overcome these shortcomings.

1.1.2 *Understanding SOAP's messaging styles*

In addition to the RPC-style SOAP messaging, the founders of the standard had the foresight to create what is known as the document-style SOAP message. As pointed out earlier, the RPC style is for creating tightly coupled, distributed applications where a running program on one machine can rather transparently invoke a function on a remote machine. The intention with RPC is to treat the remote function in the same way as you would a local one, without having to dwell on the mechanics of the network connectivity. For example, a conventional client-server application could utilize SOAP RPC-style messaging for its communication protocol.

Document style, on the other hand, was envisioned more as a means for application-to-application messaging, perhaps among business partners. In other words, it was intended for more "loosely coupled" integrations, such as document or data transfers. The differences between the two styles are defined within the SOAP standard and are reflected in the Web Service Definition Language (WSDL) interface specification that describes a given service.

After the initial flirtation with RPC-based web services, a coalescing of support has emerged for the document-style SOAP messaging. Microsoft was an early proponent of the document style, and Sun likewise embraced it completely when introducing the Java API for XML Web Services (JAX-WS). Web services became viewed as a panacea to achieving SOA. After all, a linchpin of SOA is the service, and a service requires three fundamental aspects: implementation; elementary access details; and a contract [MargolisSharpe]. A SOAP-based web service, with its reliance on the WSDL standard, appeared to address all three. The implementation is the coding of the service functionality; the access details and contract are addressed within the WSDL as the port type and XML schema used for document-style messaging. So if you simply expose all your internal components as SOAP-based services, you then have the foundation by which you can (a) readily reuse the services, and (b) combine the services into higher-level business processes—characteristics that eventually would become cornerstones of SOA. So what exactly is SOA?

1.1.3 Advent of SOA

The concepts that today are associated with SOA began to emerge with the widespread adoption of the internet, and more specifically, HTTP. By 2003, Roy Schulte of Gartner Group had coined the term SOA, and it quickly became ubiquitous. What it was, exactly, remained somewhat difficult to quantify. Through time, some commonalities appeared in the various definitions:

> *Contemporary SOA represents an open, agile extensible, federated, composable architecture comprised of autonomous, QoS-capable, vendor diverse, interoperable, discoverable, and potentially reusable services, implemented as Web services. [Erl2005]*

> *Service-Oriented Architecture is an IT strategy that organizes the discrete functions contained in enterprise applications into interoperable, standards-based services that can be combined and reused quickly to meet business needs. [BEA]*

As you can see, the common theme is the notion of discrete, reusable business services that can be used to construct new and novel business processes or applications. As you learned earlier, however, many past component-based frameworks attempted similar objectives. What distinguishes these approaches from the newer SOA?

- As discussed earlier, CORBA, EJB, and DCOM are all based on RPC technologies. In many ways, this is the exact opposite of SOA, since it introduces highly coupled solutions by way of using distributed objects and remote functions. A central theme of SOA, on the other hand, specifically encourages loosely coupled services (I'll address this concept in greater detail later in this chapter).
- In the case of EJB and DCOM, they are both tied to specific platforms and are thus not interoperable. Unless a homogenous environment exists (which is rare in today's enterprises, as they are often grown through acquisition), the benefits from them couldn't be achieved easily. SOA-based web services were designed with interoperability in mind.

- CORBA, EJB, and, to a lesser degree, DCOM were complicated technologies that often required commercial products to implement (at least in their earliest incarnations). In particular, CORBA required use of Interface Description Language (IDL) mappings, which were tedious to manage, and until recently with the 3.0 release of EJB, complex XML descriptor files were required for its implementation. SOA can be introduced using a multitude of off-the-shelf, open source technologies.
- SOA relies on XML as the underlying data representation, unlike the others, which used proprietary, binary-based objects. XML's popularity is undeniable, in part because it is easy to understand and generate.

Another distinction between a SOA and earlier RPC-based component technologies is that a SOA is more than technology per se, but has grown to embrace the best practices and standards that are rooted in the lessons found through decades of traditional software development. This includes notions such as governance, service-level agreements, metadata definitions, and registries. These topics will be addressed in greater detail in the sections that follow.

So what does a SOA resemble conceptually? Figure 1.1 depicts the interplay between the backend systems, exposed services, and orchestrated business processes.

As you can see, low-level services (sometimes referred to as fine-grained) represent the layer atop the enterprise business systems/applications. These components allow the layers above to interact with these systems. The composite services layer represents more coarse-grained services that consist of two or more individual components. For example, a *createPO* composite service may include integrating finer-grained services

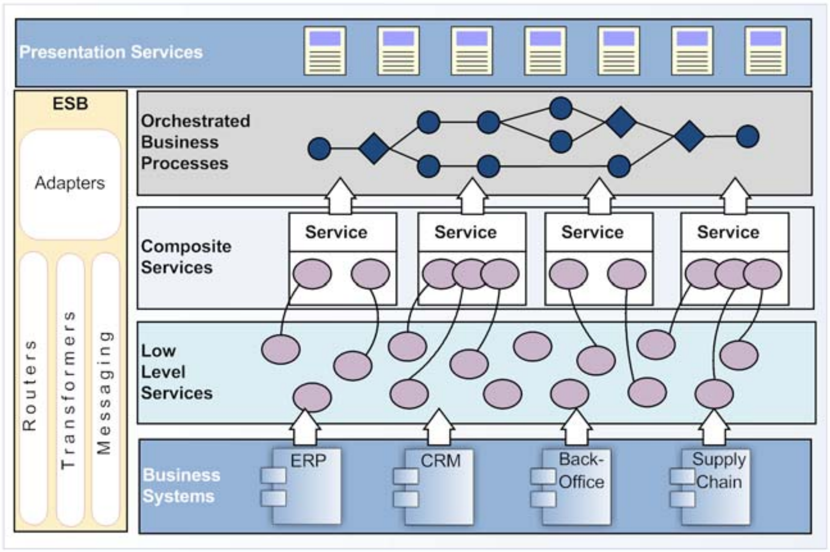

Figure 1.1 Illustration of a SOA environment. Notice the relationships between services and business processes.

such as *createCustomer, createPOHeader,* and *createPOLineItems*. The composite services, in turn, can then be called by higher-level orchestrations, such as one for processing orders placed through a website.

What is interesting is that, in many respects, SOA is a significant departure from older distributed computing models, which centered around the exchange of distributed objects and remote functions. SOA instead emphasizes a loosely coupled affiliation of services that are largely autonomous in nature.

The benefits achieved from embracing SOA are now being realized by the early adopters. When monolithic applications are replaced by discrete services, software can be updated and replaced on a piece-by-piece basis, without requiring wholesale changes to entire systems. This strategy improves flexibility and efficiency. An often-overlooked benefit is that this then enables a company to selectively outsource nonprimary activities to specialists who can perform the function more efficiently and at the lowest cost. Thanks to the advances in connectivity, where a service is housed can be largely transparent to the enterprise.

However, SOA is clearly no silver bullet. According to a recent InformationWeek survey [IW], 58 percent of respondents reported that their SOA projects introduced more complexity into their IT environments. In 30 percent of those projects, the costs were more than anticipated. Nearly the same percentage responded that their SOA initiatives didn't meet expectations. SOAP-based web services do introduce some added complexity to the SOA equation, despite their hype.

1.2 The promise of web services for delivering SOA

The SOAP standard, with its reliance on WSDLs, appeared to address many of the fundamental requirements of a SOA. That being the case, SOA, in many individuals' eyes, became rather synonymous with web services. The major platform vendors, such as Sun, IBM, Microsoft, BEA (now Oracle), and JBoss, developed tools that greatly facilitated the creation of SOAP-based web services. Companies began to eagerly undertake proof-of-concept initiatives to scope out the level of effort required to participate in this new paradigm. Web commerce vendors were some of the earliest proponents of exposing their API through SOAP, with eBay and Amazon.com leading the way (more than 240,000 people have participated in Amazon Web Services). Software as a Service (SaaS) vendors such as Salesforce emerged that greatly leveraged on the promise of web services. Indeed, Salesforce became the epitome of what the next generation of software was touted to become.

Within organizations, the challenge of exposing core business functionality as web services turned out to be daunting. Simply exposing existing objects and methods as web services often proved ill advised—to do so simply embraces the RPC model of distributed computing, not the SOA principles of loosely coupled, autonomous services. Instead, façade patterns or wrappers were often devised to create the desired web services. This approach often entailed writing significant amounts of new code, which contrasted with the heady promises made by vendors. The challenges were compounded by the vast number of choices that were available, even within a particular language environment. In the Java world alone, there were a bewildering number of choices for creating web services: Apache Axis (and Axis 2); Java-WS; Spring-WS, JBossWS, and CXF (previously known as XFire)—and these are just the open source products! Knowing which technology to use alone required significant investment.

Other factors also served to dampen the interest in SOAP web services. The perceived complexity of the various WS-* standards led to a movement to simply use XML-over-HTTP, as is the basis for Representational State Transfer (REST)-based web services (for more on this raging controversy between REST and SOAP, see [RESTvs-SOAP]). The nomenclature found in the WSDL specification, such as port types and bindings, is alien to many developers and strikes them as overly convoluted, especially for simple services (in the WSDL 2.0 standard, some of this arcane nomenclature has been removed, for instance, replacing *port type* with *interface* and *port* with *endpoint*, which is a big improvement, especially for Java and C# developers who are already familiar with such terms and their meaning). Interestingly enough, some convergence between REST and SOAP is taking place, such as the acknowledgment among some REST advocates that the metadata description capabilities of a WSDL are important. Towards this end, REST advocates have devised a new metadata specification for REST-based web services called the Web Application Description Language (WADL) [WADL]. While I may sometimes appear to be a bigot of SOAP, that's primarily because of the metadata features of WSDL, and REST coupled with WADL creates a compelling alternative.

The early enthusiasm for SOAP-based web services as the springboard for SOA began to wane as alternatives such as Web-Oriented Architecture (WOA) began to emerge, which promises a simpler, non-SOAP-based SOA architecture (see [Hinchcliffe]). Truth be told, there's likely room for both, with large enterprises opting for the WS-* stack due to its well-defined interface support, security, and reliable messaging provisions.

1.3 Understanding the core characteristics of SOA

As it turns out, achieving SOA requires more than SOAP-based web services. The characteristics of SOA transcend a particular technology. SOA is an amalgamation of technologies, patterns, and practices, the most important of which I'll address in this section.

1.3.1 Service interface/contract

Services must have a well-defined interface or contract. A contract is the complete specification of a service between a service provider and a specific consumer. It should also exist in a form that's readily digestible by possible clients. This contract should identify what operations are available through the service, define the data requirements for any exchanged information, and detail how the service can be invoked. A good example of how such a contract can be crafted can be found in a WSDL. Apart from describing which operations are available through a given network "endpoint," it also incorporates XML Schema support to describe the XML message format for each of the service operations. Figure 1.2 illustrates the relationship between WSDL and XML Schema.

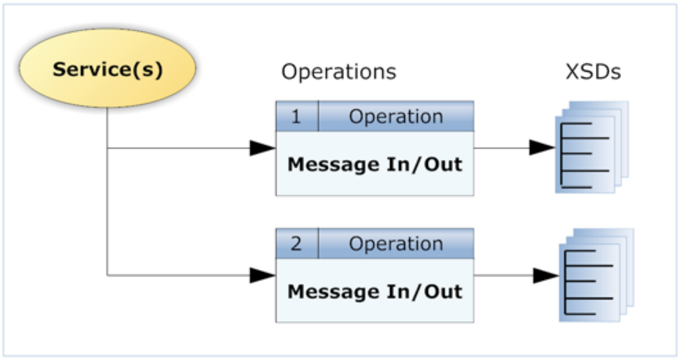

Figure 1.2 WSDL usage of XML Schema for defining the specification of an operation

Multiple operations can be defined, each of which can have its own schema definition associated with it. While the WSDL nomenclature can be confusing (particularly the 1.1 specification, with its rather arcane concepts of ports and bindings), it has, arguably, been the most successful means for defining what constitutes an interface and contract for a service. Commercial vendors, in particular, have created advanced tooling within their platforms that can parse and introspect WSDLs for code generation and service mapping. The WSDL 2.0 specification is intended to simplify the learning curve and further advance its adoption.

One of the early criticisms of the WSDL specification was that the specific service endpoint was “hardwired” into the specification. This limitation was largely addressed in the WS-Addressing standard, which has achieved widespread adoption. It supports dynamic endpoint addressing by including the addressing information within the body of the SOAP XML message, and not “outside” of it within the `SOAPAction` HTTP header. The endpoint reference contained with the WS-Addressing block could also be a logical network location, not a physical one. This enables more complex load-balancing and clustering topologies to be supported. We’ll explore the issue of why such “service transparency” is beneficial next.

1.3.2 Service transparency

Service transparency pertains to the ability to call a service without specific awareness of its physical endpoint within the network. The perils of using direct physical endpoints can be found in recent history. Nearly all enterprise systems began offering significant API support for their products by the mid-1990s. This trend allowed clients to begin tapping into the functionality and business rules of the systems relatively easily. One of the most immediate, and undesirable, consequences of doing this was the introduction of point-to-point interfaces. Before long, you began seeing connectivity maps that resemble figure 1.3.

An environment punctuated by such point-to-point connections quickly becomes untenable to maintain and extremely brittle. By making a change in something as simple as the endpoint connection string or URI, you can break a number of applications,

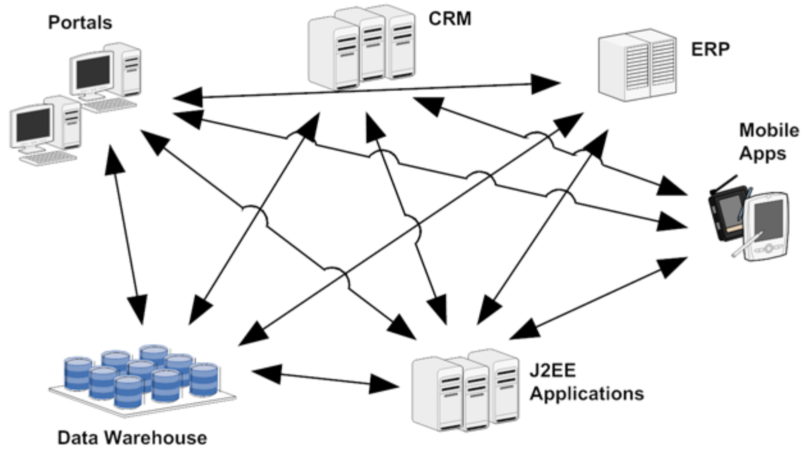

Figure 1.3 Example of how point-to-point connections greatly complicate service integration

perhaps even unknowingly. For example, in figure 1.3 imagine if the CRM system's network address changed—a multitude of other apps would immediately break.

An enterprise service bus (ESB) is often touted as the savior for avoiding the proliferation of such point-to-point connections, since its messaging bus can act as a conduit for channeling messages to the appropriate endpoint location. It no doubt performs such functionality admirably, but the same thing can be accomplished through a simple service mediator or proxy. The scenario depicted in figure 1.3 could then be transformed to the one shown in figure 1.4.

Obviously, figure 1.4 is an improvement over figure 1.3. No longer does the client application or API user have to explicitly identify the specific endpoint location for a given service call. Instead, all service calls are directed to the proxy or gateway, which, in turn, forwards the message to the appropriate endpoint destination. If an endpoint address then changes, only the proxy configuration will be required to be changed.

The WS-Addressing specification, one of the earliest and most well-supported of the WS-* standards, defines an in-message means for defining the desired endpoint or action for SOAP-based web services. It is significant in that, without it, only the transport protocol (typically HTTP) contains the routing rules (it's worth noting that SOAP supports more transports than just HTTP, such as JMS). WS-Addressing supports the use of logical message destinations, which would leave the actual physical destination to be determined by a service mediator (to learn more about WS-Addressing, see the [WSAddressing] reference in the Resources section at the end of this book).

Until fairly recently, no true open source web service proxy solution was available. However, Apache Synapse, although sometimes positioned as an ESB, is designed largely with this capability in mind. It supports outstanding proxy capabilities and can also serve as a protocol switcher. For instance, Synapse can be easily configured to receive a SOAP HTTP message and deposit it for internal consumption by a Java JMS queue. Synapse will be covered in depth in upcoming chapters.

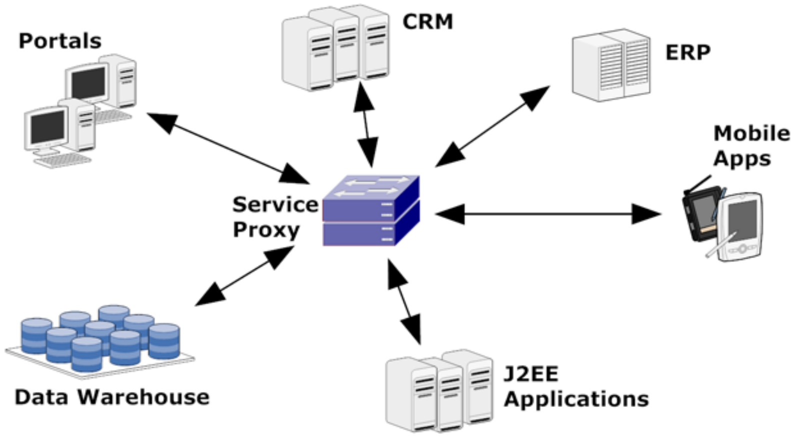

Figure 1.4 Example of mediator or proxy-based service endpoint environment

1.3.3 Service loose coupling and statelessness

Simply exposing a service as a SOAP-based web service, defined by a WSDL, does not, by itself, constitute service enablement. A key consideration is also whether the service is sufficiently self-contained so that it could be considered stand-alone. This factor is sometimes referred to as the level of "service coupling." For example, let's assume that we want to create a new service to add a new customer to your company's CRM system. If in order to use the service you must include CRM-specific identifiers such as `OrganizationId`, you have now predicated the use of that service on having a prior understanding of the internals of the CRM. This can greatly complicate the use of the service by potential consumers and may limit its audience potential. In this case, it would be preferable to create a composite service that performs the `OrganizationId` lookup first, and then performs the call to insert the new customer.

Related to this issue is granularity, which refers to the scope of functionality addressed by the service. For instance, a *fine-grained* service may resemble something like *addCustomerAddress*, whereas a *coarse-grained* service is more akin to *addCustomer.* The preponderance of literature advocates the use of coarse-grained services, in part for performance reasons as well as convenience. If the objective is to add a new customer to your CRM system, calling a single service with a large XML payload is obviously preferable to having to chain together a multitude of lower-level service calls. That said, maximizing reusability may sometimes warrant the construction of finer-grained services. In our example, having the ability to *addCustomerAddress* can be used in a variety of cases, not limited to just creating a new customer. Indeed, composite services that are coarser grained in function can then be crafted based on the lower-level services.

Finally, if possible, a service should be *stateless.* What would be an example of a *stateful* service? Imagine a service that includes a *validation* operation that first must be called prior to the actual action operation. If successful, the validation call would return a unique identifier. The action operation would then require that validation ID as its input. In this scenario, the data input from the validation call would be stored in a session state awaiting a subsequent call to perform the desired activity. While this solution avoids forcing the client user to resubmit the complete data set twice (one for the operation, the other for the action), it introduces additional complexity for the service designer (though various service implementations, both open source and proprietary, do attempt to simplify building stateful services). In particular, scalability can be adversely impacted, as the application server must preserve session state and manage the expiration of unused sessions. Performance management is complicated if appliance-based load balancing is being used, as it must pin the session calls to specific application servers (software clustering can overcome this, but it introduces its own challenges).

In the previous scenario, *statefulness* can be avoided by requiring the client to again send all relevant data when making the action call, along with the validation ID retrieved from the validation call. The validation ID would be persisted in a database and provided a timestamp. The action call would have to take place within a given number of minutes before the validation ID became invalidated.

1.3.4 Service composition

One of the main objectives of a SOA is the ability to generate composite services and/or orchestrations using service components as the building blocks. A *composable service* is largely a function of how well it is designed to participate in such a role. As was illustrated in figure 1.1, there are two general types of composite services. The first type, which could be classified as *simple* or *primitive*, simply wraps one or more lower-level services together into a more coarse-grained operation. This process can usually be accomplished by defining a simple data flow that stitches together services and then exposes the new functionality as a new service. Another goal may be to simply impose a new service contract for an existing service while leaving the underlying target endpoint unchanged. In any case, the underlying service or services participating in the simple composition must adhere to these attributes we've already addressed (and some of which will follow). They include a well-defined service contract; stateless in design, loosely coupled, and offer high availability. A composite service should be no different, and should be treated like any other service, as shown in figure 1.5.

The second type of composite services is the *complex* or workflow-type business processes, often referred to as business process management (BPM). These processes are generally multistep creations that may optionally include long-running transactions. The WS-BPEL (Business Process Execution Language) set of standards defines an XML-based language for describing a sequence flow of activities, or process. Within a process definition, a rich set of nodes can be used for routing, event handling, exception management (compensation), and flow control. The core WS-BPEL standard is tailored for working with SOAP-based web services. Because of this orientation, the

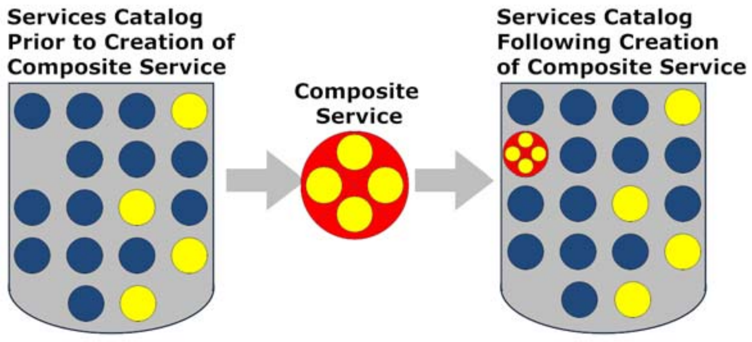
Services Catalog Prior to Creation of Composite Service
Composite Service
Services Catalog Following Creation of Composite Service

Figure 1.5 A composite service is added to an existing catalog of services.

entry point for invoking a WS-BPEL process is most typically a SOAP web service (other possibilities may include a timer service, for example). This can be either a blessing or a curse, depending on whether SOAP services are a standard within your environment.

How does a composite service author have visibility into which services are available for use when constructing such processes? This is the role of the service registry, which we'll cover next.

1.3.5 Service registry and publication

Unlike in the movie *Field of Dreams*, "if you build it, they will come" doesn't apply to services. Clients must be aware of the existence of a service if they're expected to use it. Not only that, services must include a specification or contract that clearly identifies input, outputs, faults, and available operations. The web services WSDL specification is the closest and most well-adopted solution for service reflection. The Universal Description, Discovery, and Integration (UDDI) standard was intended as a platform-independent registry for web services. UDDI can be used as both a private or public registry. Further, using the UDDI API, a client could theoretically, at least, "discover" services and bind to them. Unfortunately, UDDI suffered from an arcane and complex nomenclature, and its dynamic discovery features were myopic and predicated on naive assumptions. Today, relatively few enterprise customers are using UDDI and fewer still public registries. In practice, UDDI is rarely used today, except behind the scenes in a handful of commercial products where its complexity can be shielded from the user. Unfortunately, no standards-based alternative to UDDI is in sight.

The failure of UDDI doesn't obviate the need for a registry, and most companies have instead devised a variety of alternatives. For SOAP-based web services, a comprehensive WSDL can often be adequate. It can list all the available services and operations. Others have used simple database or Lightweight Directory Access Protocol (LDAP) applications to capture service registry information. Simply storing a catalog of services and their descriptions and endpoints in a wiki may suffice for many companies. Recently, there has also been an emergence of new open source registry solutions, such as MuleSource's Galaxy and WSO2's Registry, which attempt to fill this void; we'll discuss these solutions in the next chapter.

Now that we've identified some of the core characteristics of SOA, let's turn our attention to how those higher-level objectives can be decomposed into specific technologies that, when combined, can comprise a complete SOA technology platform.

1.4 Technologies of a SOA platform

As pointed out earlier, it's a mistake to assume that SOA is all about technology choices. Issues like governance, quality of service, and so forth are all major contributors to crafting a complete SOA. That said, our intention is to focus on the technical aspects, as the other areas largely fall outside the scope of this book. Figure 1.6 depicts the various technologies that constitute a SOA technology platform, which, moving forward, I will refer to as the Open SOA Platform. We'll explore each in greater detail along with an explanation of how the technologies tie together.

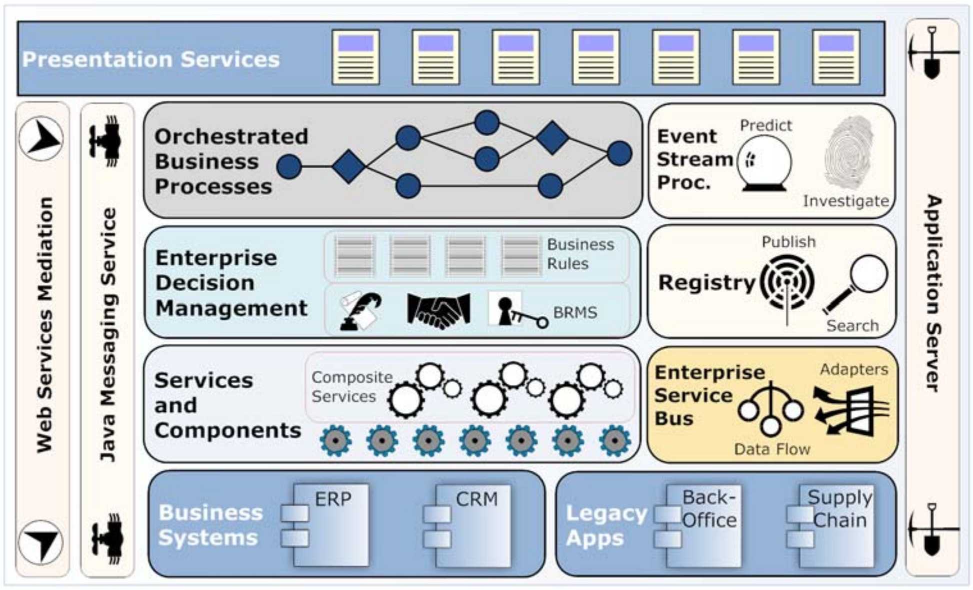

Figure 1.6 SOA technology platform. In chapter 2, we begin identifying applicable technologies for many of these areas.

1.4.1 Business process management

Business process management (BPM) is a set of technologies that enables a company to build, usually through visual flow steps, executable processes that span across multiple organizations or systems. In the past, such systems were less elegantly referred to as workflow processing engines. The promise of BPM, as optimistically stated by Howard

Where are the applications?

In looking at figure 1.6, you may be wondering, "Where are the applications?" The presentation layer can be considered your typical application, but with such a variety of different delivery models (mobile, web, gadgets, hybrids like Adobe AIR, RSS feeds, and so forth), the very notion of what constitutes an application is changing. Hence, we use "Presentation Services," which represent anything that can be considered an interface to computing services.

Smith and Peter Finger is that, "BPM doesn't speed up applications development; it eliminates the need for it" [SmithFinger]. This is because business applications, in this historical context, create stovepipes that are separated by function, time, and the data they use. The *process* in BPM refers to a holistic view of the enterprise, which incorporates employees, partners, customers, systems, applications, and databases. This also serves to extract the full value of these existing assets in ways never before possible.

Many consider BPM to be the "secret sauce" of SOA, insofar as the benefit it provides to companies that adopt it. In the book *The New Age of Innovation*, the authors identify business processes as the "key enablers of an innovation culture" [Prahalad]. To be competitive in a dynamic marketplace, business processes must change at a rapid pace, and this can only be achieved through BPM systems that enable defining, visualizing, and deploying such processes.

For a system to participate in a BPM process, services or functionality must be made externally accessible. For this reason, SOA is often considered a prerequisite for BPM, since SOA is fundamentally about exposing services in a way that enables them to participate in higher-level collaborations. Theoretically at least, BPM allows business users to design applications using a Lego-like approach, piecing together software services one-upon-another to build a new higher-level solution. In reality, it's obviously not quite so simple, but skilled business analysts can use the visual design and simulation tools for rapid prototyping. These design primitives can also be highly effective at conveying system requirements.

The fundamental impetus behind BPM is *cost savings* and *improved business agility*. As TIBCO founder Vivek Ranadivé notes, "The goal of BPM is to improve an organization's business processes by making them more efficient, more effective and more capable of adapting to an ever-changing environment" [Ranadivé]. Integrating many disparate systems and linking individuals across organizational boundaries into coherent processes can naturally result in significant return on investment (ROI). A useful byproduct of such efforts is improved reporting and management visibility. Agility, or the ability of a company to quickly react to changes in the marketplace, is improved by enabling new business processes to be created quickly, using existing investments in technology.

1.4.2 Enterprise decision management

An enterprise decision management (EDM) system incorporates a business rule engine (BRE) for executing defined business rules and a Business Rule Management

System (BRMS) for managing the rules. What exactly is a business rule? It is a statement, written in a manner easily digestible by those within the business, which makes an assertion about some aspect of how the business should function. For example, a company's policy for when to extend credit is based on certain business rules, such as whether the client has a Dun & Bradstreet number and has been in business for *x* number of years. Such rules permeate most historical applications, where literally thousands of them may be defined within the application code. Unfortunately, when they are within application code, modifying the rules to reflect changing business requirements is costly and time consuming.

A rules-based system, or BRMS, attempts to cleanly separate such rules from program code. The rules can then be expressed in a language the business user can understand and easily modify without having to resort to application development changes. This also serves to make business rules an "enterprise asset" that represents the very lifeblood of an organization. Figure 1.7 illustrates how a centralized decision service can be used by services and applications.

One of the biggest challenges when building applications is bridging the knowledge gap that exists between the subject matter experts (SMEs) who have an intimate understanding of the business, and the developers who often possess only a cursory awareness (and sometimes desire no more than that). Developers are faced with translating business requirements into abstract representations in code. This gap is often responsible for the disappointing results that too often surround the rollout of new applications. As Taylor and Raden note, "Embedding business expertise in the system is hard because those who understand the business can't code, and those who understand the code don't run the business" [TaylorRaden].

What differentiates a BRMS from an EDM? To be honest, it's probably mostly semantics, but EDM does emphasize centralized management of *all* business rules, including those considered operational, which may range in the thousands for a given company. According to Taylor and Raden, this includes heretofore "hidden" decisions that permeate a company, such as product pricing for a particular customer, or whether a customer can return a given product.

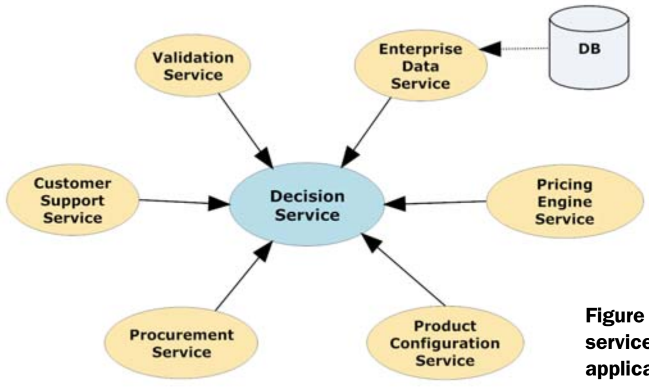

Figure 1.7 A centralized decision service can be used by other services and applications.

In chapters 11 and 12 we cover EDM in more detail, and describe how the use of domain-specific languages (DSLs) can be used to create business-specific, natural language representations of rules most suitable for maintenance by SMEs.

1.4.3 Enterprise service bus

An enterprise service bus (ESB) is at its core a "middleware" application whose role is to provide interoperability between different communication protocols. For example, it's not uncommon for a company to receive incoming ASCII-delimited orders through older protocols such as FTP. An ESB can "lift" that order from the FTP site, transform it into XML, and then submit internally to a web service for consumption and processing. Although this can all be done manually, an ESB offers out-of-the-box adapters for such processing, and most commonly, event-flow visual modeling tools to generate chained *microflows*. The cost savings over conventional code techniques is often substantial.

How does such a microflow (or what could be alternatively called a *real-time data flow*) differ from a BPM-type application? After all, at first glance they may appear similar. One key distinction is that BPM applications are typically designed for support of long-running transactions and use a central orchestration engine to manage how the process flow occurs. A real-time data flow, however, typically uses a model more akin to what's known as choreography. In a choreographed flow, each node (or hop) encapsulates the logic of what step to perform next. In addition, a real-time data flow typically passes data by way of message queues, and thus there's a single running instance of the process, with queues corresponding to each node that consume those messages. A BPM, on the other hand, typically instantiates a separate process instance for each new inbound message. This is because, as a potentially long-running transaction, the sequential queuing method would not be appropriate. To keep the number of running processes to a reasonable number, a BPM engine will "hydrate" or "dehydrate" the process to and from running memory to a serialized form, which can then be stored in a database.

Table 1.1 describes a typical set of services provided in an ESB. Because of the number of services provided by an ESB, it sometimes is described as a "backplane" or central nervous system that ties together the various SOA technologies.

Table 1.1 Core ESB features and capabilities

Feature	Description
Data Connectivity/Adapters	HTTP (SOAP, XML), FTP, SFTP, File, and JMS connectivity.
Data Transformation	XSLT for XML-based transformations.
Intelligent Routing	Content-based routing based on message properties or inline XML via XPath. Some include additional, more advanced rule-based routing using a rules engine.

Table 1.1 Core ESB features and capabilities *(continued)*

Feature	Description
Service Management	Administrative tools for managing deployments, versioning, and system configuration.
Monitoring & Logging	The ability to monitor, in real time, document and message flows. Beneficial is the capability to put inline interceptors between nodes and specifically target individual nodes for more verbose logging.
Data-flow Choreography	The ability to visually (or through editing declarative XML files) create graphs or chains to describe a sequence of steps necessary to complete a data flow.
Custom API	The ability to add custom adapters or components to the ESB.
Timing Services	The ability to create time-based actions or triggers.

Figure 1.8 depicts the role that an ESB plays in integrating various protocols and how they can be exposed through a standard messaging bus.

The flexibility of an ESB to tap into a variety of communication protocols lends some merit to an ESB-centric architecture. However, if an organization can successfully expose its business services as web services, the central role that an ESB plays is diminished (in any case, it certainly has a role in a SOA technology stack).

Let's now turn our attention to how analytical information can be drawn by the messages that flow through an ESB.

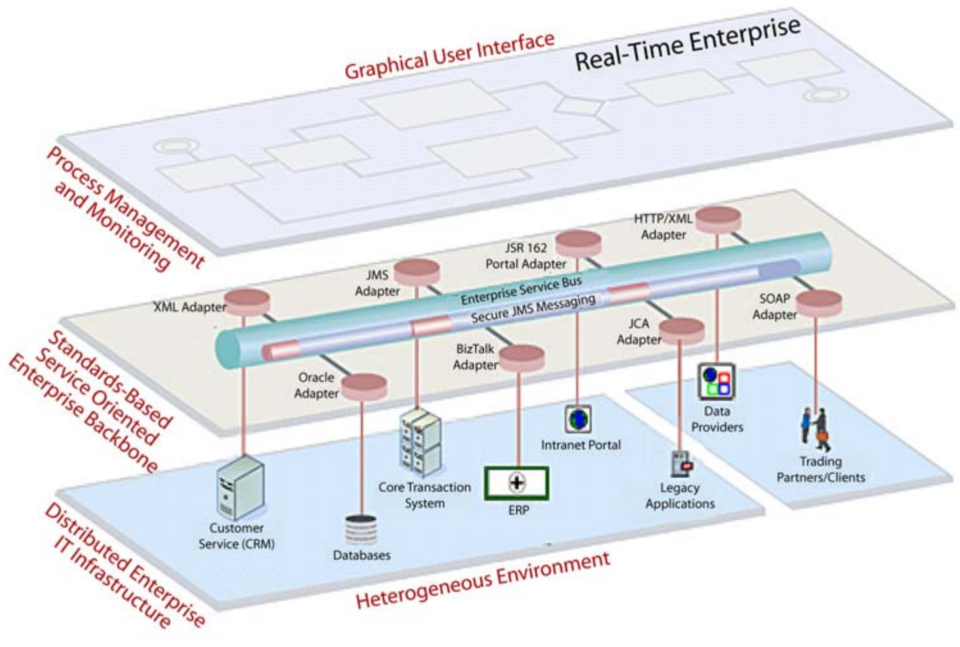

Figure 1.8 Example of an ESB-centric approach for enterprise architecture

1.4.4 Event stream processor

An *event* is simply something of interest that happens within your business. It may be expected and normal, or abnormal. An event that doesn't occur may have as much importance as those that do. Too many events may also indicate a problem. Why is it relevant to SOA? Event stream processing (ESP) support can be integrated into the implementation of your services so that real-time visibility into systems becomes a reality. This operational intelligence arms your enterprise with the ability to quickly spot anomalies and respond accordingly. Adding such capabilities into legacy solutions is often not feasible, and instead you must rely on data warehouse and business intelligence tools, neither of which provides real-time visibility.

Event stream processing is considered part of a relatively new technology sometimes referred to as complex event processing (CEP). TIBCO's Ranadivé defines it as

> *…an innovative technology that pulls together real-time information from multiple databases, applications and message-based systems and then analyzes this information to discern patterns and trends that might otherwise go unnoticed. CEP gives companies the ability to identify and anticipate exceptions and opportunities buried in seemingly unrelated events. [Ranadivé]*

The role of an ESP is to receive multiple streams of real-time data and to, in turn, detect patterns among the events. A variety of filters, time-based aggregations, triggers, and joins are typically used by the ESP to assist in pattern detection. The interpreted results from the ESP can then be fed into business activity monitoring (BAM) dashboards.

In *Performance Dashboards*, Wayne Eckerson identifies three types of business intelligence dashboards: operational, tactical, and strategic [Eckerson]. *Operational dashboards* generate alerts that notify users about exception conditions. They may also utilize statistical models for predictive forecasting. *Tactical dashboards* provide high-level summary information along with modeling tools. *Strategic dashboards*, as the name implies, are primarily used by executives to ensure company objectives are being met. Operational dashboards rely on the data that event stream processors generate. As the saying goes, you can't drive forward while looking in your rearview mirror. For a business to thrive in today's competitive landscape, real-time analysis is essential. This provides a company with the ability to immediately spot cost savings opportunities, such as sudden drops in critical raw materials; proactively identify problem areas, such as a slowdown in web orders due to capacity issues; and unleash new product offerings.

An event architecture strategy must be part of any SOA solution and must be designed from the get-go to be effective. Bolting on such capabilities later can result in expensive reengineering of code and services. Service components and backbone technologies (such as the ESB) should be propagating notable events. While a process may not be immediately in place to digest them, adding such capabilities later can be easily introduced by adding new Event Query Language (EQL) expressions into the ESP engine. We'll examine EQL in more detail in chapter 8.

The messages that carry event data that flow into an ESP are, within a Java environment, most likely to arrive by way of the Java Message Service (JMS), which is addressed next.

1.4.5 Java Message Service

The Java Message Service is one of the fundamental technologies associated with the Java Platform Enterprise Edition. It is considered message-oriented middleware (MOM) and supports two types of message models: (1) the point-to-point queuing model, and (2) the publish and subscribe model. The queuing model, which is probably used most frequently, enables a broadcaster to publish a message to a specific queue, whereby it can then be consumed by a given client. It is considered point-to-point because once the message is consumed by a client, it is no longer available to other clients. In the publish/subscribe model, events are published to one or more interested listeners, or observers. This model is analogous to broadcast television or radio, where a publisher (station) is sending out its signal to one or more subscribers (listeners).

JMS typically is ideally suited for asynchronous communications, where a "fire-and-forget" paradigm can be used. This contrasts with SOAP-based web services, which follow a request/response type model (this isn't a concrete distinction—there are variations of JMS and SOAP that support more than one model—but a generalization). JMS is typically used as one of the enabling technologies within an ESB and is usually included within such products.

Since JMS is rather ubiquitous in the Java world and well documented through books and articles, I won't cover it directly in this book. It is, however, a critical technology for Java-based SOA environments. Let's now address an often-overlooked but critical technology for building a SOA platform: a registry.

1.4.6 Registry

The implementation artifacts that derive from a SOA should be registered within a repository to maximize reuse and provide for management of enterprise assets. Metadata refers to data about data, so in this context, it refers to the properties and attributes of these assets. Assets, as shown in figure 1.9, include service components and composites, business process/orchestrations, and applications. It may also include typical LDAP objects such as users, customers, and products.

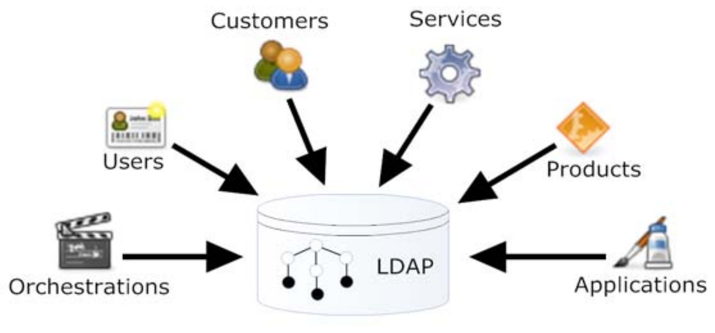

Figure 1.9 Example of an LDAP repository used as a registry. Notice that it's not just used for users, but also for products and even applications.

For smaller organizations, more informal repositories may be utilized and could be as simple as wiki articles or a simple database that describes the various assets. As organizations grow in size, however, having an appropriate technology like LDAP simplifies management and assists in reporting, governance, and security profiling. It's important to treat the SOA artifacts as true corporate assets—this represents highly valuable intellectual property, after all.

The metadata attributes for a given asset type will vary, so a flexible repository schema is essential. For example, a service component's attributes include the following:

- Service endpoint (WS-Addressing)
- Service description
- WSDL location
- Revision/version number
- Source code location
- Example request/response messages
- Reference to functional and design documents
- Change requests
- Readme files
- Production release records

Orchestrations and application may share a similar, if expanded, set of attributes, whereas those relating to a user will obviously vary significantly. A bonus chapter available at http://www.manning.com/davis includes coverage of registries.

We're nearly completed with our whirlwind overview of critical SOA technologies. One essential technology, indeed a cornerstone of SOA, is addressed next: services.

1.4.7 Service components and compositions

Service components and composites represent the core building blocks for what constitutes a SOA platform. A service can be construed as an intelligent business function that combines data and logic to form an abstract interaction with an underlying business service. This service is often a discrete piece of functionality that represents a capability found within an existing application. An example of such a service might be a customer address lookup using information found within a CRM system. The service component, in this instance, "wraps" CRM API calls so that it can be called from a variety of clients using just a customer name as the service input. If the CRM API had to be called directly, a multistep process of (a) first identifying the `customerId` based on the customer name, (b) performing code-list lookups for finding coded values, and (c) using the `customerId` to then call a `getAddress` operation may be necessary. The service component abstracts the methods and objects of the CRM into generic methods or objects and makes the underlying details transparent to the calling client. An illustration of such a service façade or wrapper is shown in figure 1.10.

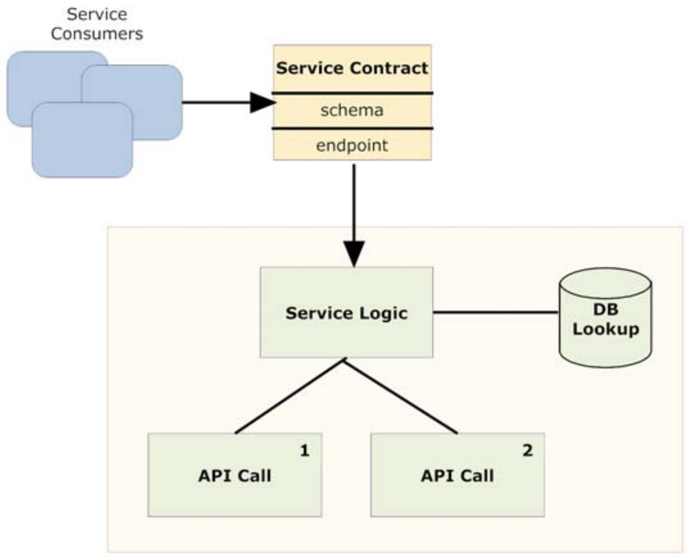

Figure 1.10 Using a façade/wrapper pattern for exposing service functionality

A service must support two fundamental requirements: a well-defined interface and binding. The interface is the contract that defines the service specification and is represented as a WSDL for SOAP-based web services. The binding is the communications protocol for how the client will interact with the service. Examples of such protocols are SOAP over HTTP; JMS; Java RMI (RMI); and EJB. Using a combination of those two requirements, a developer who wants to create a client that uses a service should be able to do so. Of course, how well the interface is designed will dictate how truly useful the service is.

A composite service, as the name suggests, is created by combining the functionality of one or more individual components. Composites may serve to further abstract functionality and are often considered coarse-grained services (such as a service to create a new customer). A composite service, in turn, may then be combined with other services to create even higher level composites. In any event, composites share the same requirements as components—an interface and binding.

Thomas Erl classifies compositions into two distinct types: primitive and complex [Erl2007]. A *primitive* type might be used for simple purposes such as content filtering or routing and usually involves two or three individual components. A *complex* composition could be a BPEL-based service that contains multiple nodes or sequence steps. Chapters 3 and 4 provides in-depth coverage of service components and composites.

Regardless of what protocol and standards your services use, there will likely be scenarios, particularly when integrating with outside organizations, that deviate from your best laid plans. One way to bridge such differences, and to improve service availability and performance, is through web service mediation technology—the topic of the next section.

1.4.8 Web service mediation

Mediation refers to bridging the differences between two parties. Consistent with that definition, web service mediation (WSM) refers to bridging between different communications protocols, with the result being a SOAP-based web service that can be redirected to an appropriate endpoint. For example, a web mediation engine might be used to process authenticating the credentials of inbound calls from an external partner's SOAP message using WS-Security (WSS). If approved, the message can then be forwarded, minus the WS-Security heading, to an internal web service to process the request. Or, perhaps a partner is unwilling or unable to use SOAP, and instead prefers a REST (XML over HTTP) solution. Using a mediator, the inbound REST call can be easily transformed into SOAP by adding the appropriate envelope. Even transformations between entirely different protocols, such as FTP to SOAP, are typically possible. Figure 1.11 depicts the role of the mediator.

A mediator serves other purposes as well, such as logging of all requests and responses, internal load balancing, advanced caching, and support of advanced WS-* features such as WS-ReliableMessaging. Another important feature is the ability to act as a proxy server. This allows the WSM to transparently intercept outbound messages, log them, and apply a WS-Security envelope, for example. The publisher of the originating message can let such policies be applied externally in a consistent fashion and not have to worry about implementing such complex details. Compliance and security can be managed independently of the application—a major benefit.

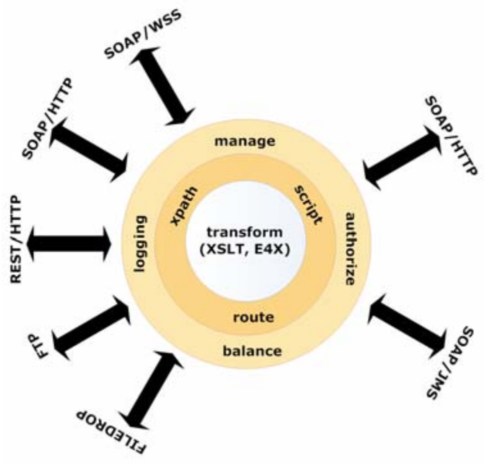

Figure 1.11 The role of web services mediator in bridging between protocols

Many of the web mediation capabilities we've talked about can now be found in modern-day ESBs. In fact, as you'll see moving forward, the ESB we've selected from the Open SOA Platform can perform both conventional ESB duties as well as the mediation features we've identified.

Does implementing SOA require all of the technologies we've alluded to in this section? Of course, the answer is no. In large part, it depends on your particular needs and requirements, so let's explore this further.

1.5 Introducing a SOA maturity model

A maturity model can be useful when you're analyzing the readiness of an IT organization in embracing the various levels of SOA that can be achieved. Figure 1.12

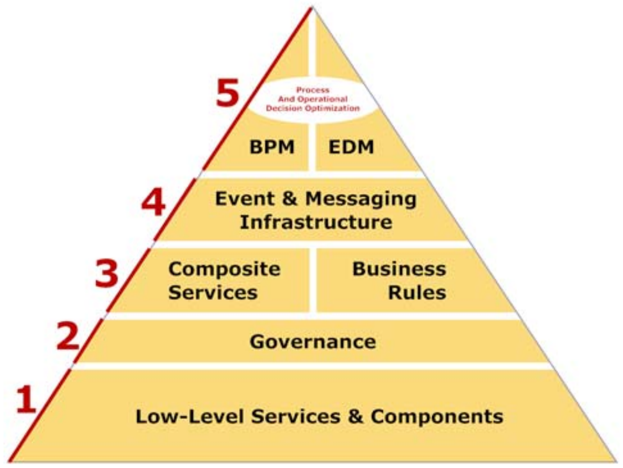
5
Process And Operational Decision Optimization
BPM
EDM
4
Event & Messaging Infrastructure
3
Composite Services
Business Rules
2
Governance
1
Low-Level Services & Components

Figure 1.12 SOA maturity model. Not all levels are required for every environment

depicts such a model, and as the pyramid suggests, each stage, at least in part, depends on the former.

Level 1 begins with the foundation of services and related components. Moving forward to level 2 requires a governance program to ensure that these services are consistently developed using a common framework, along with associated security policies and a means for publication (i.e., think registry). After all, a service isn't really a service unless it's discoverable and reusable. The next tier, level 3, is where significant benefits begin to be realized. With a governance program in place, it now becomes possible to build more coarse-grained, composite services, whose audience may span beyond just the technical team. Business "power users" may begin consuming the services by using simple, end-user integration products like Jitterbit (http://www.jitterbit.com), OpenSpan (http://www.openspan.com), or Talend (http://www.talend.com). While using a business rule engine may make sense at any level, it often becomes a requirement when composite services become introduced, which is why it's also shown in level 3. This is because composite services often require business rule logic to determine how to logically combine lower-level services.

Similar to business rules, a message- and event-driven architectural orientation can be introduced earlier in the pyramid—it's a requirement for those aspiring to level 5. The ability to monitor, in real time, events that occur within your enterprise is essential for optimizing business processes and operational decisions. This capability represents level 4, and without it, decisions and processes are optimized in a vacuum and may not accurately reflect either the business bottom line or relevant trends.

This brings us to level 5, which is where BPM and EDM can really flourish. Although you can attempt to introduce these technologies lower in the maturity model, both benefit immensely by having the prior layers in place. BPM almost always requires the ability to tightly integrate with applications, data, and business rules, and when these assets are exposed as services using SOA principles, implementing BPM is greatly simplified. Centrally managing business rules through EDM exposes business rule assets to a wider audience of business users, who are best positioned to align them to the dynamic changes of the marketplace, which can detect more accurately when events can be assessed in real time through complex event processing (CEP) filters.

For those just undertaking their first SOA projects, attempting to embrace all of the technologies we talk about in this book may seem overly ambitious. By treating SOA as a journey, you begin benefiting quickly as you build reusable services and marry them with the introduction of a business rule engine. Since SOA isn't just about technology but also process, wrapping a governance layer is essential but not difficult (it just requires some discipline). Once these pieces are in place, you can decide whether you want to move further up the pyramid. If you achieve layer 5 on an enterprise basis, the benefits through tighter alignment between IT and business will make your organization much more agile, productive, and frankly, a more fun place to work!

1.6 Summary

In this chapter, we covered the historical origins of SOA, dating back from its roots in earlier distributed computing architectures. The emergence of SOAP-based web services is a critical enabler for a SOA, but it turns out that it's only one, albeit critical, part. Simply "exposing" an application's operations as a web service provides little more than earlier RPC-based models. Instead, a deeper dive into what constitutes SOA revealed five main technologies and principles that are the bedrock of a SOA environment: service interfaces; service transparency; service loose-coupling and statelessness; service composition; and service registry and publication. With that broad understanding of what constitutes a SOA, we then focused on the technical requirements to form the Open SOA Platform. Nine specific technologies were identified that were essential platform building blocks: application server; business process management; enterprise decision management; enterprise service bus; event stream processing; Java Message Service; metadata repository; service composition and composites; and web service mediation.

Until recently, there hasn't been a robust and complete set of open source technologies that addressed each of these nine areas. Instead, only the commercial vendors, with their deeper pockets and pricy products, appeared able to provide a comprehensive SOA environment. That has changed. Compelling open source solutions now exist for each of those eight technologies, and the next chapter provides an overview of them. Following that, we revisit these eight core technologies individually, with substantive examples provided so that you can implement your comprehensive open source SOA platform. The benefits of SOA are no longer limited to big companies with big budgets. Instead, even the smallest of enterprises can participate in this exciting new paradigm by enjoying the fruits of dedicated, and very bright, open source developers. In chapter 2 we assess the open source landscape for the SOA technology platform and identify those that will be the focus for the remainder of the book.

ENTERPRISE JAVA

OPEN SOURCE SOA

Jeff Davis

Free ebook
SEE INSERT

You can now build an enterprise-class SOA solution using just open source applications. But there's a catch. You'll have to decide which products to use and how to integrate them into a working whole. The areas to integrate range from business process management, complex event processing, messaging and middleware, and ESBs, to business rules. The task can be daunting.

If you are a developer or architect who'd like some help with this task, then **Open Source SOA** is the guide for you. You'll learn key SOA concepts and how these technologies fit into the SOA equation. You'll learn valuable ways to integrate them, based on hard-won experience by the author. And you'll discover just why these open source products are a competitive alternative to expensive commercial solutions, and are in many cases superior.

What's Inside

- Full lifecycle coverage of building an SOA system
- Mix, match, and blend different tools
- Hard-to-find case studies and unique solutions
- Introductions to JBoss jBPM, Drools, Apache Tuscany, Synapse, Esper, and more
- An integrated Eclipse project, with all libraries packaged for running the examples

Jeff Davis is Director of Software Architecture at HireRight.

For online access to the author, code samples, and a free ebook for owners of this book, go to www.manning.com/OpenSourceSOA

"A survival guide in the complex landscape of open source SOA."
—Alberto Lagna, whitebox.it

"An invaluable guide ... excellent examples."
—Rick Wagner, Acxiom Corp.

"The in-depth comparisons of various open source SOA products are worth the price of the book."
—Peter Johnson, Unisys

"... applicable to any SOA project, regardless of the platform."
—Irena Kennedy, Microsoft

"Practical SOA solution that integrates key open source technologies."
—Doug Warren, Java Web Services

ISBN 13: 978-1-933988-54-2
ISBN 10: 1-933988-54-1
54999
9 781933 988542

MANNING

$49.99 / Can $62.99 [INCLUDING eBOOK]