

White Paper



Designing an Enterprise Application Framework for Service-Oriented Architecture¹

Shyam Kumar Doddavula, Sandeep Karamongikar

Abstract

This article is an attempt to present an approach for transforming Service Oriented Architecture (SOA) principles from concepts to design and then to code, based on our experiences in the development of our Infosys Radien framework. In this article, we present a systematic requirementsdriven approach for designing and building an enterprise application framework for developing applications using the SOA principles.

¹ First published at OnJava.net at
<http://today.java.net/pub/a/today/2005/04/28/soadesign.html?page=last&x-maxdepth=0>

Introduction

Let us examine why we need SOA. There is a lot of literature (see the Resources section for some examples) on what SOA is, and so we will just cover this topic very briefly. SOA concepts are primarily designed to achieve the vision of an *agile enterprise* with a flexible IT infrastructure that enables a business to respond to changes in the best possible way. As the business dynamics change and new opportunities emerge in the market, the IT infrastructure of an enterprise should be designed to be able to respond quickly and provide the applications needed to address the new business needs before the business opportunity disappears. This is possible within reasonable costs only through reuse of existing investments. This is where SOA concepts come in; they are based on the principle of developing reusable business services and building applications by composing those services instead of building monolithic applications in silos.

One of the best ways of enabling application developers to understand concepts and put them to use is by providing an application framework that provides the infrastructure needed while designing and developing applications based on those concepts. Unfortunately, there is not enough literature that can help application architects and developers in the design and implementation phases to build on the SOA concepts, apart from those from product vendors, which mostly explain in terms of their products/technologies. So, naturally, there are fewer options for frameworks that provide all of the basic building blocks needed to build applications using SOA. In this article, we attempt to fill this gap by providing a systematic *requirements-driven* approach to designing an application framework for SOA using our experience in the design of our Infosys Radien framework.

Our Approach to Framework Design

One of the first activities in designing a framework is to have an approach that helps in arriving at the desired objective systematically. There should be a clear vision and goal, a set of core guiding principles, and a systematic process.

The goal is to provide a framework with the infrastructural components needed to develop enterprise applications based on SOA concepts. Some of the core guiding principles that will be used include:

- Should be driven by requirements.
- Should be simple to use.
- Should be standards-based and pattern-driven.
- Should be practical.
- Should not become outdated quickly.
- Should buy/reuse anything existing instead of building it again.

Our strategy is to first identify the significant requirements for developing services, and then to identify the key design elements needed to address those requirements, based on applicable design patterns. Then, define an application framework that provides the basic design elements identified.

Analysis: SOA Design Considerations

From a technical perspective, the core principle of SOA is that, to use some functionality, a service consumer should be able to look up a service that provides that functionality and use it. Ideally, the contract between the user and the provider of the functionality should be the service interface and nothing else except a service locator helping the users locate the service.

The design implications are:

- Service design should be interface-driven.
 - The focus of such an interface should be the requirements of the functionality to be provided exposed as a reusable service.
- There should be a well-defined service lookup mechanism that the service consumers can use to get a handle to the implementation of the service interface.
- The user code shouldn't be tied to the implementation specifics of the service.

- Ideally, user code shouldn't change if the technology used for the service implementation changes, say, from Cobol to a simple Java object, or to an EJB or a .NET-based implementation; or if the underlying implementation logic changes, say, from using one OTS package underneath to another OTS package.
- The user code shouldn't have to deal with the life cycle aspects (ideally all "aspects") of a service like creating, initializing, configuring, deploying, locating, and managing a service.
 - There should be well-defined mechanisms that take care of creating, initializing, configuring, deploying, and managing a service that finally provides a mechanism for the end user to look up the service and use it.
 - There should be mechanisms that will allow for defining other service aspects, like access control to the services or audit of service access where the user can plug in their logic.

Usually, the service needs to be accessible across applications and so should be accessible remotely and through *multiple access mechanisms*, such as an EJB, as a web service, through JMS messages, etc. In some scenarios, multiple instances of a service may be required with different *configurations*; for example, two instances of an audit service with one configured to log messages to database X and another configured to log messages to database Y.

The framework should standardize the *definition, initialization, deployment and configuration, and management* of services to address these requirements.

One of the key aspects that the framework should be addressing is providing a unified interface for all of the multiple technology options available for implementation services such as EJBs, POJOs, .NET, mainframe-technology-based, etc.

A framework that enables an enterprise to implement its applications using SOA should therefore enable the service providers to define the service and the users to look up and use the service in a standard and consistent way and then take care of all of the "aspects" of the services.

Design: The Framework Components Required to Implement SOA

Based on the analysis above, the architecturally significant features identified are:

- A clearly defined *mechanism* to define a *service interface* with the available operations and input and output parameters.
- A *registry of services* that the service providers can use to register their service implementations and that the service consumers can use to look up a service implementation.
- An *enterprise service bus* into which the service implementations can plug in and out, and which supports multiple calling semantics (such as synchronous, asynchronous etc.), and features like transformation, routing, etc.
- A well-defined *service orchestration* mechanism to take care of flow-based and longrunning interactions.
- A well-defined mechanism that takes care of *service* aspects such as configuration, management, access control, audit, etc.
- Well-defined *service invocation mechanisms with adaptors* that will allow the service to be invoked and implemented through multiple technologies (web services, EJBs, Java POJOs, etc.).

Service Definition

One of the first things that the framework should provide is a standard mechanism for defining the service interface with the various requests supported and the request and response parameter data formats.

The enterprise architecture teams usually establish the mechanism to use for defining the service interface in an enterprise. Since most enterprises use applications and systems that are implemented using multiple technologies and platforms, most enterprises select an XML-based mechanism to define the service interface. WSDL is an industry standard for defining the service interface, but it's not necessarily the only way; there are several enterprises that already have existing XML formats for the defining service interfaces.

From the service provider's perspective, the framework should provide support for designing a service implementation. The framework should define a mechanism and, if possible, tools that help keep the *service interface* defined in the Enterprise Message Format in sync with the service implementation. Similarly, from the service consumer's perspective, the framework should provide the components to define a service stub to represent the Service Interface that is specific to the service consumer implementation. So, for a Java-based implementation, there should be a mechanism to create a Java interface, and a stub and proxy implementing the interface and encapsulating the service invocation specifics. The stub and the proxy help insulate the service usage and service provider code from the service definition formats and the invocation specifics. The framework should provide a tool that can generate the implementation code from the service definition and vice versa.

It's possible to design a generic stub that can represent any service, but the first choice should be a strongly typed interface for each service, to help in compile-time checks and in better OO design. If, say, an attribute in the message changes, then the re-generated service interface can help identify problems at compile time instead of causing a costly runtime debug exercise. The dynamic stubs should only be used for scenarios where a service is used as a generic service, such as to disable a service with something like `anyService.disable()`.

The framework components for service definition are shown in Figure 1 below. Having these mechanisms defined helps maintain consistency in service definition and usage across projects and also helps in service specification management later during the maintenance phase.

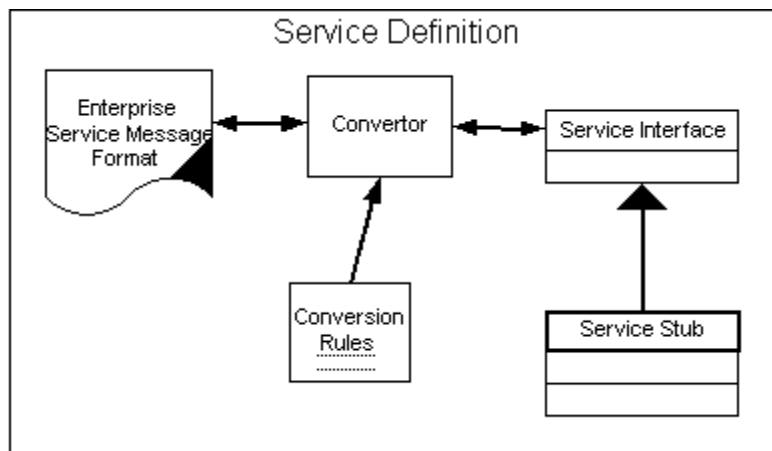


Figure 1. Service definition

Service Registry

One of the important requirements to be addressed by the framework is to provide a *Service Registry* with details of the service interfaces and the service providers. It should also provide a standard mechanism for the service providers to register their implementation of a service interface and for the service consumers to look up the implementation of a service interface. This mechanism is illustrated in Figure 2.

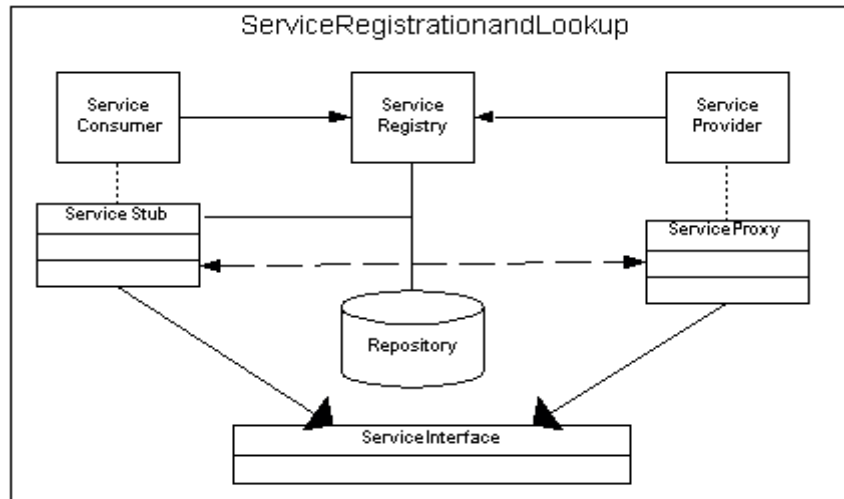


Figure 2. Service registry

The framework needs to provide a Service Registry design element that provides the API to register and look up the Service Stubs that implement the service interface. The Service Stub encapsulates the invocation details for the consumers and interacts with the Service Proxy that encapsulates the invocation details for the service providers. The service invocation details are explained in the next section.

Service Invocation

The next important requirement to be addressed by the framework is to standardize the service invocation mechanism and provide the infrastructural components with clearly defined interfaces that shield the service consumers and the service providers from the underlying implementation details.

Usually, enterprise architecture teams define the communication policies for applications in an enterprise. The communication policies define the strategies on when to use native protocols, when to use point-to-point communication, and when to use message-oriented communication. A common debate while determining communication policies is whether to use message-oriented communication to invoke a service or to directly invoke the service using its implementation specific synchronous protocol such as RMI. One of the fundamental business requirements is to differentiate the service levels offered to the end customers based on their business value to enterprise, so that it helps achieve the desired client experience business objectives. This is technically possible only if the communication mechanism used in invoking services is controlled and the service invocations can be prioritized. Using message-oriented communication instead of directly invoking the service using its implementation-specific synchronous protocols thus provides the mechanism needed to address this requirement. Strategically, the preferred communication mechanism should be one based on a messaging infrastructure instead of point-to-point invocations.

Services are supposed to be coarse-grained and need to be accessible through multiple technologies so the most common invocation mechanism is message-oriented. A high-level logical view of an Enterprise Service Bus (ESB) based on the [Message Broker pattern](#) and the [Message Bus pattern](#) with some of the key logical design elements is shown in Figure 3.

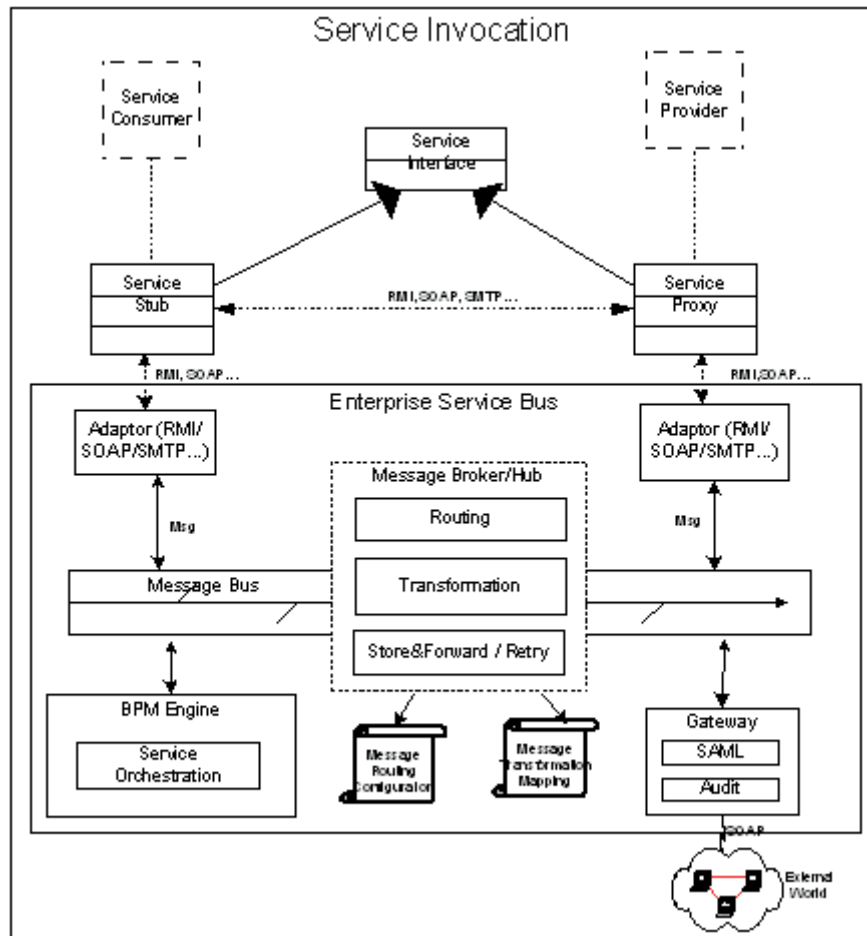


Figure 3. Service invocation

The infrastructural logical components that are needed for service invocation include:

- ServiceStub
- Service Proxy
- Adaptors
- Message Broker
- Message Bus
- Gateways

The *ServiceStub* implements the delegate pattern and provides the service interface to the service consumers, hiding the invocation details.

The *ServiceProxy* implements the proxy pattern and provides the abstraction of the invocation details for the service providers.

The *Adaptors* provide the technology-specific integration mechanisms for the service stubs and proxies. For a J2EE-based implementation, the adaptor can provide the listener mechanisms that the stubs and proxies can use to receive the messages and the API to send a message.

The *Message Broker* and the *Message Bus* provide the transformations, routing, and other such services. The broker and the bus take care of transforming the message representations from the service consumer and service provider internal formats to the Enterprise Message Format and vice versa. They also provide the routing of the messages, store and forward, message retries, prioritizing of messages, etc.

The *Gateways* provide the mechanisms for external integration. The gateways provide the single points of contact for the external partners and transform the invocation protocols and message formats from the external partners to the internal enterprise message formats using a message broker and an adaptor. They also enforce the security checks, audit requirements, etc.

Having clearly defined interface driven components for each of these helps replace implementations with minimal impacts.

Service Orchestration

The next important requirement to be addressed is the orchestration of services for the implementation of a business process.

The framework should provide a mechanism based on COTS BPM tools to define, execute, and manage the service orchestration. The framework should define an *Orchestration Adaptor* that helps abstract the interactions with the orchestration implementations (BPM tools) through an adaptor interface with API to initiate processes, get the list of process instances, get the list of activities and their states, and to manipulate the state of activities, list of exceptions, etc. that provide an abstraction over the implementation specifics. The adaptors can then be implemented for the selected orchestration implementation. Since they all provide the same API, they can be replaced easily as needed without greatly impacting the services interacting with the orchestration component.

Service Management

The next important requirements to be addressed include providing a standard mechanisms for management of the services, for configuration of services, for taking care of the cross-cutting concerns like the access control, audit, etc. that apply to all or most service requests driven by centralized policies.

The diagram in Figure 4 shows some of the components that the framework needs to define/provide to address these requirements.

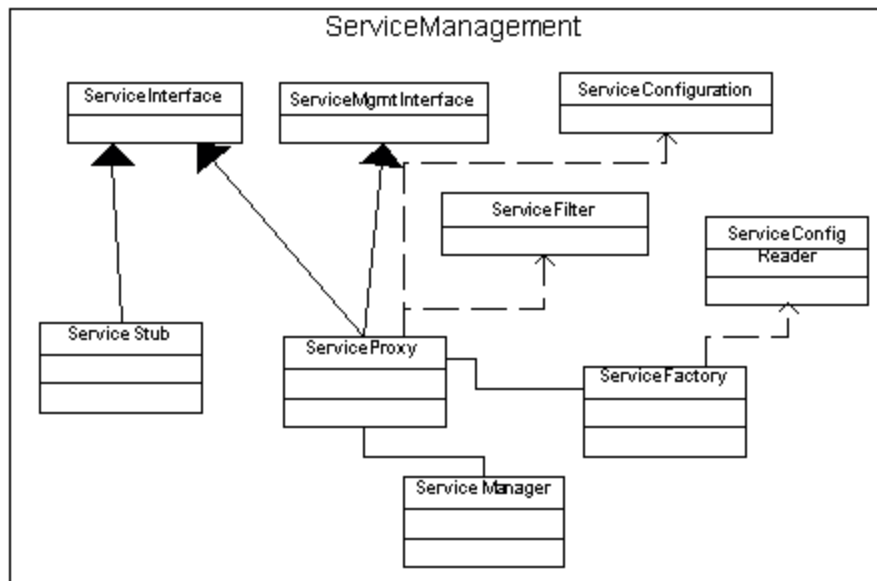


Figure 4. Service management

To address manageability requirements, the framework should define a standardized mechanism to design the *Service Management Interface* for the services and to make the service proxy provide an implementation of the management interface of the service and then provide a Service Manager component with the mechanism to make the service management interfaces accessible through standard management tools.

For J2EE-based implementations, [Java Management Extensions \(JMX\)](#) provides the standard for application management. For each service, an MBean interface with the management API for the service can be defined, and the service proxy is associated with the implementation of the MBean interface. The proxy is then registered with an MBeanServer that allows them to be managed using standard enterprise management tools like Tivoli, etc.

One of the common requirements in service design is to ensure that the service is configurable so that a service instance can be localized to a particular context and deployed. The framework should therefore provide a standard mechanism for service configuration. The framework should define a standard *Service Configuration Format*, a *Service Configuration Reader component*, a *Service Configuration* component to represent and hold the service configuration information, and a Service Factory component that takes care of the creation of the service, loading the service configuration, and initializing the service with the desired configuration.

The framework should provide a mechanism to allow the separation of the service core functional logic from the logic for enforcing the cross-cutting concerns like access controls, audits, etc. The framework should define a *Service Filter* component that can be plugged into the service invocation mechanism at service proxy to intercept the service requests and apply the QoS aspect logic.

Implementation: Framework Development

The next step is to implement the framework with the various design elements identified earlier. Using the “buy instead of build” principle, it makes sense to first evaluate what can be leveraged off the shelf, and then build the missing components on top of the selected implementations. Some of the possible implementation options for a partial list of the design elements identified earlier are discussed below.

Service Definition Components

The [Apache Axis framework](#), the [webMethods Glue framework](#), and the [Web Service Invocation Framework \(WSIF\)](#) provide some of the basic components including stub and proxy/skeleton generators for scenarios where web services and WSDL are used to define services and for J2EE-based implementation scenarios. These, however, may need to be extended to use an ESB-based invocation mechanism, depending on the ESB selected. An equivalent may need to be developed for enterprises that do not use web services and WSDL-based services (this is the disadvantage of not leveraging industry standards).

Service Management Components

The [Spring](#) framework is an option for implementing some of the basic plumbing with a good configuration mechanism. But it would be worth considering a simpler service-locator-based design for the core components identified earlier rather than using the dependency-injectionbased design. You can refer to “[Inversion of Control Containers and the Dependency Injection Pattern](#)” by Martin Fowler for a discussion on these alternatives.

Some of the possible options for implementing ServiceFilters for J2EE-based implementation scenarios include using [AspectJ](#) and [AspectWerkz](#) and using DynamicProxies (see the Resources section for an article explaining how to use dynamic proxies to implement crosscutting concerns). The dynamic proxies provide a simple solution for defining intercepting filters until the AOP products mature in their support for providing the ability (portable across application servers) to define the intercepting filters using a simple configuration mechanism. Note, however, that you can do lot more with AOP than just intercepting filters.

Service Invocation Components

The [Mule](#) project provides an open source implementation option for the ESB that can be leveraged. The article “[An Introduction to Web Services Gateway](#)” provides the details of an example base Commercial Off The Shelf (COTS) solution, which is part of the IBM WebSphere family of products that can be used to implement the gateway to plug into the ESB. For scenarios where web services and WSDL aren't used internally, the solutions described in scenarios 3 and 4 in that article can be used.

Once this evaluation and selection of the COTS solutions for the design elements that suit the enterprise context is performed, the next step is to identify the gaps and build the missing components.

Conclusion

This article presents an approach to transform SOA concepts to implementation through an application framework. This article identifies some of the key design considerations for SOA and then identifies the logical design elements required to address these design considerations and an application framework that provides the basic components needed. Some of the possible implementation options were also considered. The primary intention is to present a systematic approach to developing an enterprise application framework for SOA and take it from concepts to the design-elements level. The next steps would be to continue the next steps of the proposed approach and get into the building of the framework and keep iterating through these steps.

References

- Service-oriented architecture definition at http://en.wikipedia.org/wiki/Serviceoriented_architecture
- “Understanding Service-Oriented Architecture” at <http://msdn.microsoft.com/architecture/soa/default.aspx?pull=/library/enus/dnmaj/html/aj1soa.asp>
- “Service-Oriented Architectures with Web Services” from the Java BluePrints Catalog at <https://bpcatalog.dev.java.net/nonav/soa/index.html>
- “Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)” at <http://java.sun.com/developer/technicalArticles/WebServices/soa/>
- BEA dev2dev online: Service-oriented architecture portal at <http://dev2dev.bea.com/soa/>
- “SOA Blueprints: Best practices for service-oriented architectures” from Middleware Research at <http://www.middlewareresearch.com/soa-blueprints/>
- Apache Axis Framework at <http://ws.apache.org/axis/>
- Service Invocation Framework (WSIF) - <http://ws.apache.org/wsif/>
- Inversion of Control Containers and the Dependency Injection Pattern by Martin Fowler at <http://www.martinfowler.com/articles/injection.html>
- AspectJ at <http://eclipse.org/aspectj/> and AspectWerkz at <http://aspectwerkz.codehaus.org/>
- “Implementing Cross-Cutting concerns using Dynamic Proxies” at <http://www.devx.com/Java/Article/21463>
- The Mule project at <http://www.muleumo.org/>
- An Introduction to Web Services Gateway at <http://www-106.ibm.com/developerworks/webservices/library/ws-gateway/>

About the Authors:

[Shyam Kumar Doddavula](#) is a Senior Technical Architect at the J2EE Center Of Excellence at Infosys. Shyam has over 8 years of experience in software development with expertise in J2EE application framework development, application architecture definition, design and development for the Financial Services, Telecom & Transportation sectors. Shyam holds a Masters degree in Computer Science from Texas Tech University

[Sandeep Karamongikar](#) is a Principal Architect at the J2EE Center of Excellence at Infosys. Sandeep enables application of industry wide best practices to applications development, mentors application development teams in designing distributed, multi-tiered, and highly available Java applications. Sandeep has over 10 years of experience in architecting enterprise frameworks, providing enterprise-class solutions, and implementing large-scale, mission-critical, IT Solutions for Infosys clients across a range of industries



For more information, contact askus@infosys.com

About Infosys

Many of the world's most successful organizations rely on Infosys to deliver measurable business value. Infosys provides business consulting, technology, engineering and outsourcing services to help clients in over 30 countries build tomorrow's enterprise.

For more information about Infosys (NASDAQ:INFY), visit www.infosys.com.