The Definitive Guide to SOA: BEA AquaLogic® Service Bus

Jeff Davies
with Ashish Krishna and David Schorow

Apress®

The Definitive Guide to SOA: BEA AquaLogic® Service Bus

ISBN-13: 978-1-59059-797-2

ISBN-10: 1-59059-797-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin
Technical Reviewer: Jayaram Kasi
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Jeff Pepper, Paul Sarknas, Dominic Shakeshaft, Jim Sumser, Matt Wade
Project Manager: Elizabeth Seymour
Copy Edit Manager: Nicole Flores
Copy Editors: Susannah Davidson Pfalzer and Heather Lang
Assistant Production Director: Kari Brooks-Copony
Production Editor: Katie Stence
Compositor: Dina Quan and Gina Rexrode
Proofreader: Liz Welch
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Contents at a Glance

Contents

Foreword

Enterprise Service Bus (ESB) is a hot topic today. Many vendors are either building new products in this category or dressing up their existing products to pitch them as an ESB. However, there is no clearly accepted definition of what an ESB is, or what its architecture or programming paradigm should be. Definitions range from saying that ESB is wholly unneeded to saying it has all the capabilities of a full integration suite with built-in BPM, data aggregation, and WSM capabilities. Architectures range from being embedded in the clients and endpoints to being a central intermediary. Programming paradigms for the ESB range from writing Java to being completely configuration driven and pliable with graphical interfaces.

BEA did not dress up one of their existing products and pitch it as an ESB but built an ESB from scratch. First introduced in summer 2005, BEA's ESB has a razor sharp focus on where it is positioned as a component in an end-to-end SOA architecture. It complements a BPM service or a data aggregation service but serves a different and distinct role. Much of SOA is about componentization, interconnectivity, and reuse, and the ESB component serves as an intermediary with the clear and distinct role of providing loose coupling between clients and services, a routing fabric, connectivity, and a central point of security enforcement that contribute to the manageability of your SOA network. It can be a central intermediary or a decentralized network of intermediaries. Also, it is completely configuration based with browser-based graphical interfaces.

In this book, Jeff Davies introduces you to ESBs in general and BEA's AquaLogic Service Bus in particular. He includes many examples and clear, understandable explanations of the product and how it can be leveraged to implement a number of ESB use cases. He takes the very practical and useful approach of picking one of the leading products in the ESB category and doing a "show and tell" instead of delving into a lot of philosophical discussions and arguments about various contrasting architectures or definitions for an ESB. The book is very readable and instructive. As one of the architects of the first release of the product, I feel this book is a fine introduction to AquaLogic Service Bus.

Jayaram Kasi
Director of Technical Program Management
AquaLogic Service Bus
BEA Systems

About the Authors

JEFF DAVIES, SOA architect and technical evangelist at BEA, has over 20 years of experience in the software field. Jeff has extensive experience developing retail applications, such as Act! for the Windows and Macintosh platforms, and a number of other commercially available applications, principally in the telecommunications field. His background also includes the development, design, and architecture of enterprise applications. Prior to joining BEA, Jeff was the chief architect at a telecommunications company and responsible for their SOA. Now at BEA, Jeff is focused on the practical application of BEA products to create SOA solutions.

ASHISH KRISHNA is part of the product management team for integration and SOA products at BEA Systems; he's responsible for BEA's integration product. Prior to this, Ashish worked as an SOA architect and was responsible for enabling SOA and EDA solutions at key customers for BEA. Ashish was also part of core engineering team at BEA responsible for architecture and design of various components for BEA's ESB. Before BEA, Ashish was a founding engineering staff member of ECtone, a Silicon Valley start-up that was later acquired by BEA. Ashish has a diverse background in enterprise integration and software development, including legacy systems, EDI, and ERP integration technologies. He was a consultant for SAP R/3 and EDI, responsible for numerous implementations at Fortune 500 companies in telecommunications, automotive, and manufacturing industries. Ashish holds a master's degree in aerospace engineering and a bachelor of engineering degree in mechanical engineering. He is also a PhD candidate in mechanical engineering at Texas A&M University.

DAVID SCHOROW has over 20 years of experience working on enterprise software. David is the chief architect for BEA AquaLogic Service Bus and has guided its development and evolution. Prior to joining BEA, David was the chief Java architect at the NonStop division of HP, overseeing the development of a wide variety of Java projects, including the NonStop Java JVM, NonStop SQL JDBC drivers, the port of WebLogic Server to the NonStop platform, and other enterprise Java products. David has extensive experience in high-performance transaction processing systems, the environments used by the most demanding, mission-critical applications, such as airline reservations, health care, and banking. David has a bachelor of science degree from MIT and a PhD from the University of California, Berkeley.

About the Technical Reviewer

JAY KASI has been an infrastructure architect since 1988, working for Hewlett Packard, Commerce One, and BEA Systems. He has architected the kernel of a relational database management system, system-level high-availability capabilities, a messaging and routing fabric for B2B electronic commerce, and ESBs at both Commerceone and BEA. He has also worked as a distributed OLTP architecture consultant. He was one of the key architects of ALSB and has been with the project since inception. He is now the director of program management for ALSB and is working on a variety of integrations with other products.

Acknowledgments

There are many people who have helped me to make this book a reality. I want to thank my wife for her love and understanding as I spent hours on my computer mumbling incoherently about "namespaces" and the like. There is no finer wife in the world. Similarly, I'd like to thank my children, Eric and Madeline, for putting up with my highly distracted nature while writing this book. Of course, I'd like to thank my parents and my aunt and uncle for enabling me to get to this point in my life with their constant love and support.

I'd like to thank Jay Kasi at BEA for his help and tutelage while writing this book. I have never met a person with such a deep understanding of any software product in my life. Many times when I was stuck on a problem, Jay would quickly look at the code and deliver an exact analysis of the problem within moments.

I'd also like to thank the many folks who helped review the book and provided me with technical answers to the more unusual scenarios. Specifically, I want to recognize (in alphabetical order) Deb Ayers, Stephen Bennett, Naren Chawla, George Gould, David Groves, Dain Hansen, Gregory Haardt, Karl Hoffman, Ashish Krishna, Usha Kuntamukkala, Saify Lanewala, Michael Reiche, Kelly Schwarzhoff, Jeremy Westerman, Mike Wooten, and Bradley Wright.

Finally, I'd like to thank the great mentors in my life, Mark Russell and Gerry Millar. They taught me everything from how to tie a neck-tie to how to "listen to what they are feeling." They both taught me that it's the people who are important; the software is incidental. That's a hard but invaluable lesson for a natural-born geek. Thank you.

Jeff Davies

The BEA AquaLogic Service Bus team is full of innovative people. Their contributions and drive to be the best are reflected in the product. I would like to thank the team for all the hard work. It would not have been possible to write this book without their efforts in producing a world-class product, and I know firsthand, as I was part of the engineering team for the first customer ship (FCS) back in 2005.

I would like to thank my wife Sumina, and my daughter, Isheeta, for their support and patience especially in letting me work on this book at late hours, on holidays, and especially during our month-long vacation to India—our first in four years!

Ashish Krishna

Chapter 13 explains the Transport SDK; this useful extensibility mechanism was designed and implemented by Greg Fichtenholtz, a senior engineer on the ALSB team. It is his design that enables ALSB to be used in new and different environments not addressed in the original implementation. The usefulness of the Transport SDK is because of his good design work.

Greg is only one member of a very talented team that created the ALSB product; however, their names are too numerous to mention (and I'd be afraid of leaving someone out). This group, with its engineering prowess and creative energy, works under the management of Ashok Aletty, who fosters a productive, cooperative, and enjoyable atmosphere; these people

are responsible for making AquaLogic Service Bus such a fantastic product. I consider myself fortunate to have the opportunity to work with such a great team on this exciting product.

I'd also like to thank my sister, Stephanie Schorow, for her thorough review of an early draft of the chapter. She is the real writer of the family—Chapter 13 is much more readable because of her efforts.

Lastly, I'd like to thank my wife, Mona, and my son, Marcus, for their understanding and support when working on this book required my nights and weekends (and cancelling a ski trip).

David Schorow

Introduction

Service-Oriented Architecture (SOA) is rapidly becoming the new standard for today's enterprises. A number of books have appeared in bookstores that discuss various aspects of SOA. Most (if not all) are high-level discussions that provide some strategies for you to consider but very little tactical information. As a software engineer, I am able to grasp these abstract concepts fairly quickly, as I'm sure you are. However, the devil is always in the details. I know that once I begin to implement a new technology, I will discover a whole new dimension of bugs, design issues, and other problems that are never discussed in those strategic books.

SOA is not a technology—it is architecture and a strategy. In order for you to implement your own SOA, you need to learn not a single new technology but a whole series of differing technologies. I thought I knew XML pretty well before I began walking the path to SOA. It didn't take long for me to figure out that there was a lot more to XML than I had previously thought. I had to learn the details of XML, XML Schema, WSDL, XQuery, and XPath before I could begin to make informed design judgments.

While I enjoy reading about new strategies, I enjoy realizing them in code just as much. Code keeps you honest. A lot of things work very well on paper, but once you start flipping bits, the truth will emerge in all of its intolerant glory. What I really wanted to read was a detailed book on SOA development. Since I could not find one, I wrote one. I wrote this book under the assumption that there were thousands of other software developers like myself—people who enjoyed writing code and loved to put theory into practice.

This book is a mix of theory and working code samples. One reason there are so few books on writing real code for an SOA is because few SOA platforms exist that the average developer can download and use. Most SOA (specifically ESB) vendors keep their software locked away, demanding that you purchase it before you can use it. This is like purchasing a car you have never seen or driven based solely on the description provided to you by the salesperson.

Fortunately, BEA Systems provides an enterprise class ESB that anyone can download for free. This book will walk you through many detailed examples of connecting the ALSB to legacy systems, show common design patterns for web services, and generally increase both your development and architectural expertise in ESB and SOA.

What Is AquaLogic?

AquaLogic Service Bus is a single product in the AquaLogic product family. The AquaLogic family includes many products with diverse functionalities; see the BEA web site for a complete listing (`www.bea.com`).

How This Book Is Organized

This book comprises 15 chapters in total. We've written most of the chapters so that they may be read individually. However, we do recommend reading Chapters 2 and 3 so that you can set up your development environment and understand the basic principles of an enterprise service bus.

Chapter 1, "Why Use a Service Bus?," describes the functions and benefits of an enterprise service bus.

Chapter 2, "Installing and Configuring the Software," guides you through installing and configuring the AquaLogic Service Bus and a development environment. By installing the software as described in this chapter, you will be able to run all of the sample code contained in this book.

In Chapter 3, "Hello World Service," following the grand tradition of programming books, we write a web service, test it, and integrate it with the AquaLogic Service Bus. Along the way, we provide a quick tour of AquaLogic Service Bus Administration console.

In Chapter 4, "Message Flow Basics," you'll learn what message flows are, how to create them, and how they are used in AquaLogic Service Bus.

Chapter 5, "A Crash Course in WSDL," introduces you to Web Services Description Language (WSDL), the language of modern web services. Creating (or just reading) WSDL requires a fair bit of skill beyond simple XML. This chapter teaches you the core of what you need to know about WSDLs and leaves out the fluff!

In Chapter 6, "Message Flows," we really put AquaLogic Service Bus through its paces, with sample code for almost every feature available.

Chapter 7, "Advanced Messaging Topics," covers just about every weird integration issue and use of ALSB that you will ever see.

Chapter 8, "Reporting and Monitoring," shows you that there is more to ALSB than just messaging. It can keep you informed about the health of your enterprise and provide automated alerts and sophisticated status reports of services and the servers that host them.

Chapter 9, "Security Models and Service Bus," presents a topic that is often discussed but seldom understood. This chapter will provide you with a solid understanding of how to implement security within your service bus.

Chapter 10, "Planning Your Service Landscape," discusses the considerable planning required to move to SOA. In this chapter, we introduce a methodology that will simplify this planning process and provide you with a taxonomy by which you can quickly classify your services.

Chapter 11, "Versioning Services," is possibly the most controversial chapter in the book! Forget everything you've heard about versioning web services and brace yourself for some heresy!

Chapter 12, "Administration, Operations, and Management," will teach you some best practices for how to keep your service bus running, because there is more to a service bus than development.

Chapter 13, "Custom Transports," explores the Transport SDK. While AquaLogic Service Bus provides many useful transport protocols out of the box, it also contains an API that allows you to create your own customer transports, so it can integrate with any legacy system.

Chapter 14, "How Do I . . . ?," answers some common questions about using AquaLogic Service Bus in the real world.

The Appendix, "AquaLogic Service Bus Actions," is a quick reference for the actions supported by AquaLogic Service Bus.

—Jeff Davies

CHAPTER 1

Why Use a Service Bus?

Enterprise Service Buses (ESBs) are all the rage in modern software development. You can't pick up a trade magazine these days without some article on ESBs and how they make your life wonderful. If you're a software development veteran, you'll recognize the hype immediately. ESBs aren't going to be the magic answer for our industry any more than were XML, web services, application servers, or even ASCII. Each of the aforementioned technologies started life with a lot of fanfare and unrealistic expectations (the result of the inevitable ignorance we all have with any emerging technology), and each technology ended up becoming a reliable tool to solve a specific set of problems.

The same is true for the ESB. Putting the hype aside, let's focus on a bit of software history so we can better understand the problems that the ESB is designed to address.

The Problems We Face Today

Software development is a tough business. We expect modern software systems to have exponentially more functionality than we expected from them only a few years ago. We often develop these systems with ever-dwindling budgets and sharply reduced timeframes, all in an effort to improve efficiency and productivity. However, we cannot lament these issues. These very issues drive us to deliver software that's better, faster, and cheaper.

As we've raced to develop each generation of software system, we've added significantly to the complexity of our IT systems. Thirty years ago, an IT shop might have maintained a single significant software system. Today most IT shops are responsible for dozens, and sometimes hundreds, of software systems. The interactions between these systems are increasingly complex. By placing a premium on delivering on time, we often sacrifice architecture and design, promising ourselves that we'll refactor the system some time in the future. We've developed technologies that can generate large quantities of code from software models or template code. Some of the side effects of this race into the future are a prevalence of point-to-point integrations between software applications, tight coupling at those integration points, lots of code, and little configuration.

Point-to-Point Integrations

Software development today is tactical and project oriented. Developers and architects frequently think in terms of individual software applications, and therefore their designs and implementations directly reflect this thinking. As a result, individual applications are directly integrated with one another in a *point-to-point* manner.

A point-to-point integration is where one application depends on another specific application. For example, in Figure 1-1, the CustomerContactManager (CCM) application uses the Billing application interface. You can say that the CCM application "knows" about the billing application. You also hear this kind of relationship referred to as a "dependency," because one application depends on another application to function correctly.

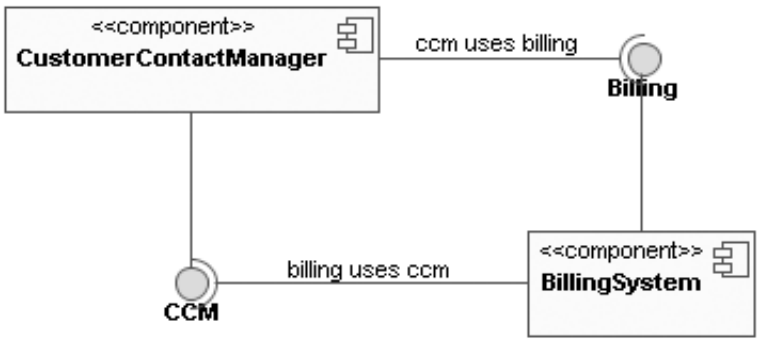

Figure 1-1. *Early point-to-point integrations*

Figure 1-1 illustrates a trivial IT environment, one that has only two applications and two point-to-point integrations. Just to be clear, the first integration allows the CCM system to call the Billing system. The second integration point allows the Billing system to call the CCM system. When your IT department is this small, point-to-point integration is fairly easy to manage.

Figure 1-2 expands on the problem a bit. The IT shop is now home to 8 software systems and a total of 11 integration points. This illustrates a common pattern in integration: the number of integration points grows faster than the number of systems you're integrating!

Even Figure 1-2 is, by modern standards, a trivial IT system. A midsized service provider where Jeff once worked had 67 business systems and another 51 network systems. One hundred eighteen software systems integrated in a point-to-point manner is unmanageable. We know of telcos that have 12 or more billing systems. Having duplicates of certain software systems (such as billing) or having a large number of software systems in general is quite common; large companies can acquire smaller companies (and therefore acquire the software systems of the smaller companies) faster than most IT shops can integrate the newly acquired systems.

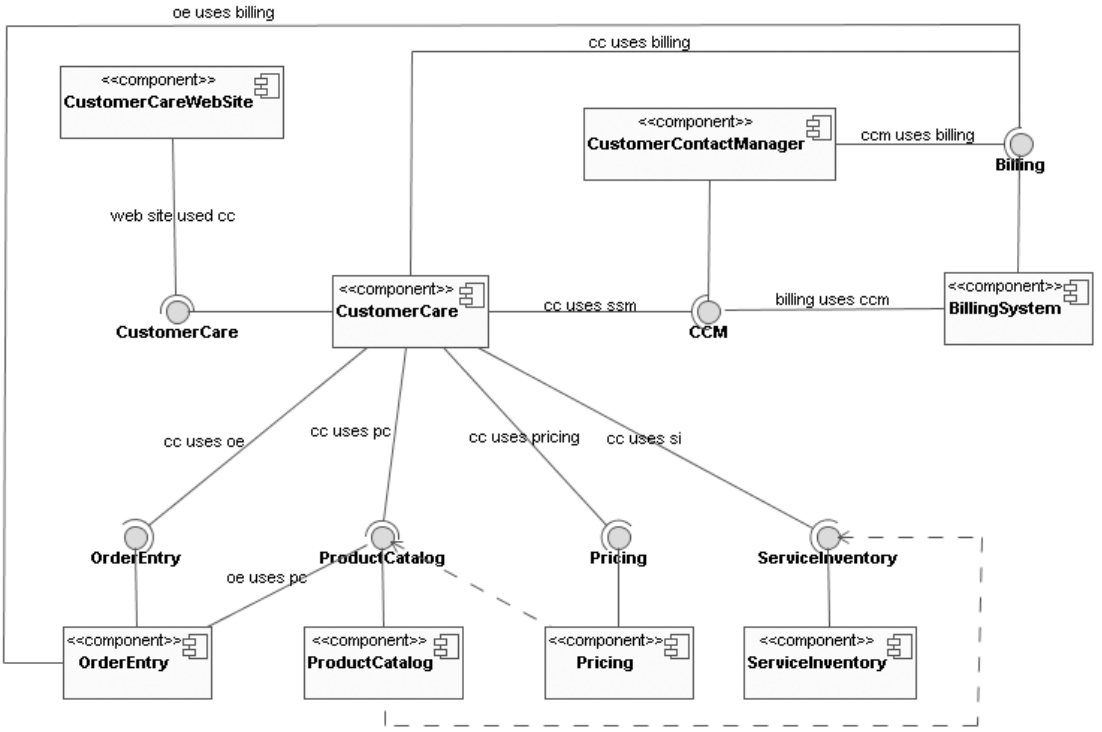

Figure 1-2. *Increasing point-to-point integration*

Tight Coupling

Tight coupling is often a byproduct of point-to-point integrations, but it's certainly possible to develop tightly coupled applications no matter what your integration environment looks like. There are two types of coupling, *tight* and *loose*. Loose coupling is desirable for good software engineering, but tight coupling can be necessary for maximum performance. Coupling is increased when the data exchanged between components becomes larger or more complex. In reality, coupling between systems can rarely be categorized as "tight" or "loose." There's a continuum between the two extremes.

Most systems use one another's APIs directly to integrate. For Enterprise JavaBeans (EJB) applications, you commonly create a client JAR file for each EJB application. The client JAR file contains the client stubs necessary for the client applications to call the EJB application. If you make a change to any of the APIs of the EJB application, you need to recompile and deploy the EJB application, recompile the client JAR, and then recompile and redeploy each of the client applications. Figure 1-3 illustrates this set of interdependencies between the software components and the file artifacts that realize them.

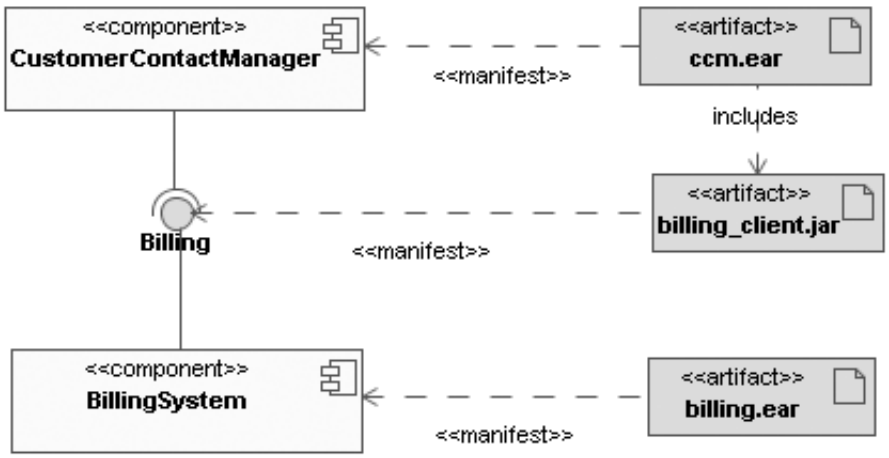

Figure 1-3. *EJB coupling model*

Tight coupling results in cascading changes. If you change the interface upon which other components depend, you must then recompile the client applications, often modifying the client code significantly.

It's a common (and false) belief that you can use interfaces to reduce the coupling between systems. Interfaces are intended to abstract out the behavior of the classes that implement the interfaces. They do provide some loosening of the coupling between the client and the implementation, but their effect is almost negligible in today's systems. This is not to say that interfaces aren't useful; they most certainly are. But it's important to understand the reasons why they're useful. You still end up tightly coupled to a specific interface. For example:

```
package com.alsb.foo;
public interface SampleIF {
   public int getResult(String arg1);
}
```

A client that depends on this interface is tightly coupled. If you change the `getResult()` method to take another argument, all clients of the interface must be recompiled. It's precisely this level of intolerance to change that tightly couples the code. The problem isn't so much in the design of the interface, but with the technology that implements the interface.

More Code Than Configuration

Every medium-to-large sized enterprise runs a lot of code these days. We have so much code as an industry that we needed to invent tools to manage it all. We have source code control (SCC) systems that provide document management of our code. In the last few years we've seen the rise of source code knowledge bases.

Early ESBs

Early ESBs were primarily concerned with making web services available to service consumers. Their implementation was clunky (as new technologies usually are) and didn't embrace many open standards, simply because those standards didn't exist at the time. Furthermore, the developers of early ESBs could only try to predict how web services would affect enterprise computing and IT organizations.

The early ESBs were "ESBs" in name only. As the industry has matured, so has our understanding of the role of an ESB in modern architecture. Today's ESBs must go far beyond simply "service enabling" functionality. An ESB must also provide robust solutions for today's IT challenges.

Modern Solutions

The IT industry is constantly evolving. Our understanding of the issues that surround the management of large IT systems matures on a daily basis. Modern ESBs are simply the latest tools to help us manage our IT problems. They benefit from real-world examples of how Service-Oriented Architecture (SOA) is changing the face of today's advanced corporations. Although early ESBs could only address a handful of the following issues, modern ESBs need to address them all.

Loose Coupling

You might have heard that web services provide you with loose coupling between systems. This is only partially true. Web services, by the very nature of Web Services Description Language (WSDL) and XML Schema Document (XSD), can provide some loose coupling because they formalize a contract between the service consumer and the service provider. This is a "design by contract" model, and it does provide tangible benefits. If you're careful, you can create a schema that's platform neutral and highly reusable.

However, if you take a look at any WSDL you'll see that the service endpoints are written into the WSDL, as you can see in Listing 1-1.

Listing 1-1. *HelloWorld Service Defintion*

```
<service name="HelloWorldService">
   <port binding="s1:HelloWorldServiceSoapBinding"
      name="HelloWorldPortSoapPort">
      <s2:address location="http://www.bea.com:7001/esb/Hello_World" />
   </port>
</service>
```

By specifying a specific machine and port (or a set of machines and ports), you're tightly coupling this service to its physical expression on a specific computer. You can use a Domain Name Server (DNS) to substitute portions of the URL, and therefore direct clients into multiple machines in a server farm. However, DNS servers are woefully inadequate for this, due to their inability to understand and manage the status of the services running on these servers.

So, loose coupling isn't achieved by WSDL or web services alone. A more robust solution is to provide some mediation layer between service clients and service producers. Such a mediation layer should also be capable of bridging transport and security technologies. For example, a service might be invoked through a traditional HTTP transport mechanism, but it can then invoke lower-level services through Java Message Service (JMS), e-mail, File Transfer Protocol (FTP), and so on. This approach is often effectively used to "wrap" older services and their transports from the newer service clients.

Location Transparency

Location transparency is a strategy to hide the physical locations of service endpoints from the service clients. Ideally a service client should have to know about a single, logical machine and port name for each service. The client shouldn't know the actual service endpoints. This allows for greater flexibility when managing your services. You can add and remove service endpoints as needed without fear of having to recompile your service clients.

Mediation

An enterprise service bus is an intermediary layer, residing between the service client and the service providers. This layer provides a great place for adding value to the architecture. An ESB is a service provider to the service clients. When clients use a service on the service bus, the service bus has the ability to perform multiple operations: it can transform the data or the schema of the messages it sends and receives, and it can intelligently route messages to various service endpoints, depending on the content of those messages.

Schema Transformation

The web service published by the service bus might use a different schema from the schema of the business service it represents. This is a vital capability, especially when used in conjunction with a canonical taxonomy or when aggregating or orchestrating other web services. It's quite common that a service client will need to receive its data using a schema that's significantly different from that of the service provider. The ability to transform data from one schema to another is critical for the success of any ESB.

Service Aggregation

The service bus can act as a façade and make a series of web service calls appear as a single service. Service aggregation follows this pattern, making multiple web service calls on behalf of the proxy service and returning a single result. Service orchestration is similar to service aggregation, but includes some conditional logic that defines which of the lower-level web services are called, and the order in which they're invoked.

Load Balancing

Due to their position in any architecture, ESBs are well suited to perform load balancing of service requests across multiple service endpoints. When you register a business web service with AquaLogic Service Bus (ALSB), you can specify the list service endpoints where that business service is running. You can change this list, adding or removing service endpoints without having to restart the ALSB server.

Enforcing Security

You should enforce security in a centralized manner whenever possible. This allows for a greater level of standardization and control of security issues. Furthermore, security is best enforced through a policy-driven framework. Using security policies means that the creation and application of security standards happens outside the creation of the individual web services.

Monitoring

An ESB plays a vital role in an SOA. As such, you must have a robust way to monitor the status of your ESB, in both proactive and reactive manners. The ability to proactively view the performance of the service bus allows you to help performance-tune the service bus for better performance. Tracking the performance over time can help you plan for increasing the capacity of your ESB.

Reactive monitoring allows you to define alerts for specific conditions. For example, if a specific service doesn't complete within a given timeframe, the ESB should be able to send an alert so that a technician can investigate the problem.

Configuration vs. Coding

A modern service bus should be configuration based, not code based. For many engineers the importance of that statement isn't immediately obvious. It took us some time before we appreciated the configuration-oriented capability of ALSB. Most software systems in use today are code based. J2EE applications are a great example of this. In a J2EE application you write source code, compile it into an EAR or WAR file, copy that EAR or WAR file onto one or more J2EE application servers, then deploy those applications. Sometimes it's necessary to restart the J2EE server, depending on the nature of your deployment.

Configuration-based systems work differently. There's nothing to compile or deploy. You simply change the configuration and activate those changes. We would argue that your telephone is configuration based; you configure the telephone number you want to call, and your call is placed. There's no need to restart your phone. Similarly, network routers and switches are configuration based. As you make changes to their configuration, those changes take effect. There's no need for a longer software development life cycle to take place.

Configuration and coding are two different strategies. Neither is superior to the other in all situations. There are times when the J2EE approach is the right approach, and times when the configuration-based approach is best.

Enter AquaLogic Service Bus

BEA released AquaLogic Service Bus in June 2005. ALSB runs on Windows, Linux, and Solaris platforms. ALSB is a fully modern ESB and provides functionality for each of the capabilities expected from today's enterprises, described in the following sections.

Loose Coupling

Aside from the loose coupling benefits from WSDL and XSD, ALSB adds the ability to store WSDL, XSD, eXtensible Stylesheet Language Transformation (XSLT), and other information

types within the ALSB server as "resources." These resources are then made available throughout the ALSB cluster of servers, allowing you to reuse these resources as needed.

The benefit of this might not be immediately clear, so we'll give an example. Many companies define and manage enterprise-wide data types using an XML schema. Because ALSB can store an XML schema as a resource in the service bus, that schema can easily be reused by any number of WSDLs or other XSDs. This enables you to create and enforce enterprise-wide standards for your data types and message formats.

Location Transparency

One of the capabilities of ALSB is to register and manage the locations of various web services within the enterprise. This provides a layer of abstraction between the service client and the service provider, and improves the operational aspect of adding or removing service providers without impact to the service clients.

Mediation

One of the roles for which ALSB is specifically designed is that of a service mediator. ALSB uses the paradigm of "proxy services" and "business services," where the proxy service is the service that ALSB publishes to its service clients, and the business services are external to ALSB. In between the proxy service and the business service is the layer where service mediation takes place. Schemas can be transformed, as can the data carried by those schemas. Intelligent or content-based routing also takes place in this mediation layer.

Schema Transformation

Schema transformation is a central capability of ALSB. ALSB provides a number of ways to transform schemas, depending on your specific needs. You can use XSLT to transform XML data from one schema to another. Similarly, you can use XQuery and XPath to perform XML schema transformations. Additionally, ALSB supports the use of Machine Format Language (MFL) to format schemas to and from non-XML formats.

Service Aggregation

ALSB doesn't match a single proxy service to a single business service. Instead, ALSB allows you to define a many-to-many relationship between proxy services and business services. This approach allows for service aggregation, orchestration, and information enrichment.

Load Balancing

Because ALSB registers the service endpoints of all business services, it's ideally situated for operating as a load balancer. This is especially true because ALSB is configuration based, not code based. As a result, you can add or remove service endpoints from a business service and activate those changes without having to restart your service bus.

Enforcing Security

ALSB, as a service mediator, is ideally situated to enforce the security of the web services because it operates on the perimeters of the enterprise. ALSB is designed to enforce security through the use of explicit security policies. Using ALSB, you can propagate identities, mediate, and transform between different security technologies, such as Basic Authentication, Secure Sockets Layer (SSL), and Security Assertion Markup Language (SAML).

Monitoring

ALSB provides a robust set of features around monitoring. The service bus console allows you to look proactively at the state of your entire ESB.

For reactive monitoring, ALSB allows you to define alerts for conditions that you define. You can send these alerts via Simple Network Management Protocol (SNMP) traps to third-party monitoring programs, such as Hewlett Packard's OpenView or AmberPoint's SOA Management System. Also, alerts can be delivered via e-mail to specified recipients. We'll discuss monitoring more fully in Chapter 8.

Configuration vs. Coding

ALSB is a configuration-based service bus. You don't write Java code for ALSB, although ALSB can recognize and make use of Java code in some circumstances. Instead, you configure ALSB through its web-based console.

One handy feature of the ALSB console is the Change Center. Your configuration changes don't take effect when you make each change. Instead, your configuration changes are grouped together, similarly to a database transaction, and only take effect when you tell ALSB to activate your changes. This is a critical capability, because many times you'll make multiple changes that are interdependent.

Of course, creating these changes by hand can be an error-prone process. As a result, ALSB allows you to make changes in one environment (a development or a test environment) and then export those changes as a JAR file. You can then import that JAR file into your production environment as a set of configuration changes, and activate them as if you had entered those changes directly into the Change Center by hand.

Won't This Lock Me into BEA Technologies?

ALSB is entirely standards based. You configure ALSB through the use of XQuery, XPath, XSLT, and WSDLs. The only aspect of ALSB that might be deemed "proprietary" is the implementation of the message flows (see Chapter 4). However, these message flows are simply graphical constructs for common programming logic, and they're easy to reproduce in just about any programming language. The real heavy lifting in ALSB is done using the open standards for functionality, and WebLogic Server for reliability and scalability.

Because ALSB is standards based, it's designed to integrate with and operate in a heterogeneous architecture. Using ALSB as a service bus doesn't preclude you from using other technologies in any way. ALSB is used to integrate with .NET applications, TIBCO, SAP, Oracle, JBoss, WebSphere, Siebel, and many more. BEA didn't achieve this level of heterogeneity by accident; it's all part of its "blended" strategy: using open standards and open source to achieve the maximum amount of interoperability.

Why Buy an Enterprise Service Bus?

We come across this question frequently. The truth is that an ESB contains no magic in it at all. It's possible to build your own ESB from scratch. In fact, one of the authors has done it twice before joining BEA. There's nothing that the engineers at BEA can write that you cannot write, given enough time, money, and training. This principle holds true for all software. You don't have to use Microsoft Word to write your documents; you could create your own word processor. The same holds true for your web browser. HTML standards are publicly available, and you could use your engineering time to develop your own browser.

Naturally, few of us would ever consider writing our own word processor or web browser. It's a far better use of our time and money either to buy the software or to use an open source version. This is especially true if your company isn't a software company. If you work in an IT shop for a company whose primary line of business isn't software, you'll recognize the fact that building software from scratch is a difficult sell to your executive staff. There simply is no return on investment for such development efforts. Your time and skills are better spent solving problems specific to your company.

There are a number of benefits to purchasing ALSB. First is the fact that it comes from a dyed-in-the-wool software company. BEA has been in business for more than a decade and has a long history of delivering innovative, successful products. Furthermore, BEA supports those products for many years. BEA's first product was Tuxedo, a product that BEA still sells and supports to this day, though it's gone through many versions to keep it current with today's technologies.

A number of open source ESBs are available today. Most are in the early stages of development and functionality. Although we love open source and advocate its use in many areas, we would be hesitant to use an open source ESB. An ESB will become the central nervous system of your enterprise. You should exercise caution and diligence when selecting an ESB. You want one with a proven record of success, from an organization that works hard to keep itself ahead of current market demands.

ALSB is built on BEA's WebLogic Server technology. This gives you enterprise-quality reliability and scalability. On top of this, AquaLogic is built on open standards for maximum interoperability in a heterogeneous environment. It's an ESB that will carry your company into the future.

Summary

In this chapter we reviewed the features and functions that a modern ESB should have, and we've described each feature's importance to the organization. ALSB implements all these features, and possesses many more advanced features that we'll cover in this book. But we've talked enough about ALSB. It's time to start to demonstrate, in working code, exactly how to use these features to their fullest.

CHAPTER 2

Installing and Configuring the Software

This chapter will walk you through the installation process for ALSB and the process of configuring your development environment. By the end of this chapter, you'll be able to compile and run the sample code that comes with this book.

To begin with, you need a computer that runs Java. Specifically, it needs to run Java Development Kit (JDK) version 1.5 or later. All the examples are written using JDK 1.5, and ALSB requires that you have JDK 1.5 installed. Fortunately, ALSB ships with two different JDKs that meet this requirement. One is the JRockit JDK, which is intended for use on production systems that run on Intel (or compatible) CPUs. The second is the Sun JDK, which is recommended for use with development versions of ALSB, or production versions that aren't running on Intel-compatible CPUs.

Naturally, you need to install the ALSB software. You can download ALSB from `http://dev2dev.bea.com/alservicebus/`. It's also a good idea to download the most recent documentation so you can stay informed about recent changes.

Of course, you'll need an editor to edit your sample source code. ALSB ships with WebLogic Workshop, an IDE that's based on Eclipse (`http://www.eclipse.org`), and comes with a suite of Eclipse plug-ins that are preconfigured to make your development with ALSB must faster.

You'll often use Ant to build your software, especially the Java web service that will act as your "business service" (more about business services later). You'll need Ant version 1.6 or later. Like most of the software used by ALSB, Ant is included with the ALSB installer and is preconfigured into the WebLogic Workshop environment. The Ant home page is at `http://ant.apache.org`.

Finally, you'll need two software packages for some of your more advanced work with the service bus. The first is an FTP server that you'll use to demonstrate integrating the service bus with legacy systems via FTP. You can use any FTP server that you like. We selected the FileZilla FTP server, which resides at `http://filezilla.sourceforge.net/`. Also, you'll need access to an e-mail server when testing the e-mail integration. Because your company might not appreciate you sending test e-mails over its e-mail server, we recommend installing your own e-mail server. We selected Java Mail Server, which is available at `http://www.ericdaugherty.com/java/mailserver`. Because both FTP and SMTP are based on well-defined standards, feel free to substitute your own FTP and e-mail servers. However, we do provide a detailed configuration walkthrough of both these programs, so if you aren't accustomed to setting up these types of servers, you're better off using the same ones we've used.

You'll find all the software you need for this book in the Source Code/Download area of the Apress web site at `http://www.apress.com`.

Installing the Software

ALSB comes with most of the software you'll need to compile and deploy the applications you create in this book: Ant, WebLogic Workshop, WebLogic 9.x, and the JDK 1.5. Installing ALSB is a breeze. For the most part, you can safely accept the default values provided by the installation program. However, we do recommend creating a new BEA home directory if you have a previous version of WebLogic 9 installed. On our system we have WebLogic 8.1 installed in the traditional BEA home directory of C\bea. We installed ALSB into C:\bea92ALSB to keep the installations separate.

Once you have the software installed, you need to do a little configuration to complete the setup.

Configuring WebLogic Workshop

ALSB ships with a customized version of Eclipse known as the WebLogic Workshop. This customization is achieved by using Eclipse's plug-in capability to extend Eclipse. Workshop comes entirely preconfigured and ready to run. When you start Workshop for the first time, it will ask you to select a *workspace* (see Figure 2-1). A workspace is a directory where your Workshop projects will be created. Workshop allows you to create as many workspaces as you like. For your purposes, we recommend that you name your new workspace *alsb_book*, and use that workspace as the home for all the projects you'll create in this book.

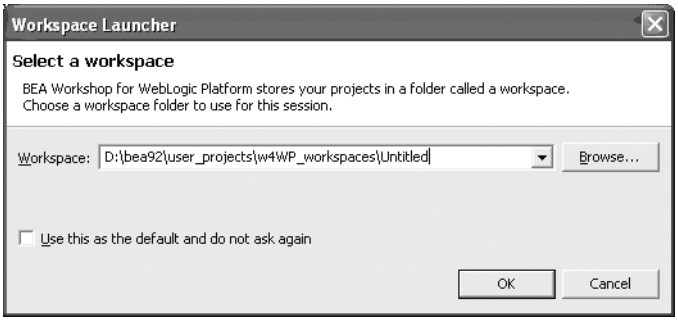

Figure 2-1. *Create a workspace in Eclipse.*

Once you're happy with the name of your workspace, click the OK button, and the WebLogic Workshop IDE loads. If you're familiar with the Eclipse IDE, learning Workshop will be a breeze for you. If you're moving to WebLogic Workshop 9 from WebLogic Workshop 8, or are otherwise unfamiliar with Workshop 9, we'll review the major capabilities of the latest version of Workshop in the following section. Also, you'll find that the first project in Chapter 3 will walk you through the IDE in detail, making it much easier to learn the Workshop IDE as you go.