SAMPLE CHAPTER

Example Implementations in Mule and ServiceMix

OPEN SOURCE ESBs IN ACTION

Tijs Rademakers
Jos Dirksen

Forewords by
Ross Mason and Guillaume Nodet

MANNING

Open Source ESBs in Action

by Tijs Rademakers
Jos Dirksen

Sample Chapter 1

brief contents

1 The world of open source ESBs

In this chapter:

- Typical ESB functionality
- Open source ESB overview
- Mule and ServiceMix

If you ask integration specialists and architects to supply one buzzword used in the integration market today, enterprise service bus (ESB) would be one of the predominant answers. Concepts like service-oriented architecture (SOA) and business process management (BPM) would also be mentioned. These buzzwords sound interesting, but are they just part of the hype in the integration market or do they represent real business value?

As with every buzzword in the integration industry, a sales pitch is involved, but these concepts have a business case. Other books are available that focus on SOA (such as *Understanding Enterprise SOA* by Eric Pulier and Hugh Taylor [Manning, 2005]). In this book we focus on the enterprise service bus, but we also discuss some interesting open source products related to SOA and BPM.

There's a lot of confusion about what an ESB is, so let's start off with an overview of the most important functionality that should be present in a product calling itself an ESB. Many ESB products are available in the market, from vendors like

IBM, TIBCO, Microsoft, and Oracle. Most ESB vendors offer products that have a background in the enterprise application integration (EAI) market. As we'll see in section 1.1, this is not so strange, because ESB functionality has a lot in common with the *older* EAI products.

But there are also a number of products available that have been built from the ground up. In this group of products we see not only commercial vendors but also open source projects that deliver the functionality needed in an ESB.

In section 1.5 we examine two open source ESBs (Mule and ServiceMix) that we use in the examples and case studies presented in this book. These two open source ESBs have gained a lot of attention in the market and are the two most used open source ESBs in businesses around the world today. This means that this book is not a typical cookbook for a specific ESB. Because we show examples involving two ESBs, we're confident that you'll gain the knowledge and experience you need to use any open source ESB.

1.1 Why do you need an ESB?

We can't begin a book about open source ESBs without a good discussion about the use of an ESB within an enterprise. Maybe you've already read articles or books that introduced the concept of an ESB. If you want to have a solid background, we recommend you check out *Enterprise Service Bus* by David A. Chappell (O'Reilly Media, 2004).

A lot of the early ESB products had a history in the enterprise application integration market. It was sometimes hard to tell the difference between some ESB products and their EAI predecessors!

However, we can identify two main differences between EAI and ESB products. The first is the change from the hub-and-spoke model in EAI products to a bus-based model in ESB products. The hub-and-spoke model is a centralized architecture, where all data exchange is processed by a hub, or broker. The hub-and-spoke model can be seen as the successor of the point-to-point model (which we discuss in figure 1.1 in a moment). The bus model, on the other hand, uses a distributed architecture, in which the ESB functionality can be implemented by several physically separated functions.

A second main difference between EAI and ESB products is the use of open standards. EAI products like WebSphere Message Broker, TIBCO BusinessWorks, and Sonic XQ were mainly based on proprietary technology to implement messaging functionality and transformation logic. ESB products are based on open standards, such as Java Message Service (JMS), XML, J2EE Connector Architecture (JCA), and web services standards.

As we mentioned earlier, many current ESB products have a background in the EAI space. So newer versions of WebSphere Message Broker (version 6), TIBCO BusinessWorks (version 5), and Sonic ESB (yes, the name has changed) are now marketed as ESBs but still have a strong foundation in EAI. In addition, a number of ESBs have been built from the ground up, like WebSphere ESB, Cordys, and TIBCO ActiveMatrix Service Grid.

Because open source ESBs were not yet available during the EAI period, they don't have a history in the implementation of proprietary technology. Many integration specifications like JMS and Java Business Integration (JBI) are available, and open source ESBs use these specifications as the foundation for their open source product implementations.

But why do you need an ESB? Let's take some time to explore the benefits of an ESB. Then, in section 1.1.2 we look in greater detail at the ESB from an application perspective.

1.1.1 Benefits of an ESB

In any discussion of the implementation of an ESB within an organization or department, there's a need for a management-level overview of an ESB. In essence, an ESB is a technical product that solves integration problems. But let's try to step back from the technical aspects of an ESB and talk about some high-level benefits. To show the advantages of an ESB, we start with an overview of how applications are integrated *without* the use of an EAI broker or an ESB. This model (see figure 1.1) is known as point-to-point architecture.

The application landscape example shown in figure 1.1 is still a common way of dealing with integration problems. In this example, four existing applications are integrated via point-to-point integration solutions. For example, the enterprise resource planning (ERP) system needs to have billing information from the COBOL application.

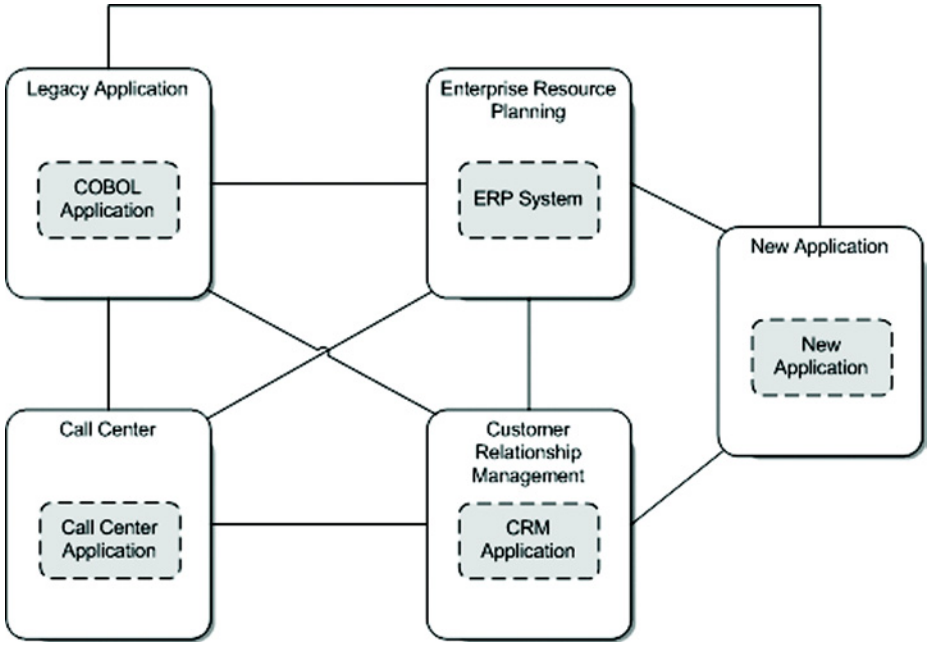

Figure 1.1 The point-to-point model describes an environment where applications are integrated with a unique and custom-made integration solution.

Because the COBOL application is only capable of exporting a file in a batch, a custom-made integration solution is being used to transfer the billing information from the exported file to the ERP system. The information also has to be transformed to a data format that the ERP system is able to process. For every line drawn between the four existing applications in figure 1.1, a custom integration solution is developed. So an important downside to the point-to-point model is the number of custom-made integration solutions that must be developed and maintained.

The complexity and maintenance cost increase when we add a new application to this application landscape. Imagine that this new application must communicate with the ERP, customer relationship management (CRM), and COBOL application as depicted in figure 1.1. This means that we need to implement three new integration solutions to be able to integrate this new application into the existing environment.

In this kind of application environment, there are many reasons to think about an integration solution like an ESB (summarized in table 1.1). Is there a business driver to integrate applications? In most organizations a real business need exists for integrating applications. New products have to be delivered to the market today, not tomorrow. And the IT environment must be able to facilitate the business to be able to do this. An ESB can help to increase the flexibility of an IT environment, and therefore can help to improve the time-to-market for new products.

Here's another reason to consider ESBs: the application landscape is heterogonous when it comes to technologies and protocols. When you have to deal with many different protocols—for example, JMS, FTP, HTTP, SOAP, SMTP, and TCP—it's difficult to implement new integration solutions between applications. An ESB provides protocol or technology adapters, which make it easy to deal with a heterogonous IT environment.

A third reason is the reduction of the total cost of ownership of the full application landscape. In a point-to-point model, the management and maintenance of all the integration points can be time-consuming and therefore expensive. It would be less time-consuming to have an ESB solution to deal with integration problems so that management and maintenance becomes easier.

Table 1.1 Reasons to start thinking about an ESB

Reason	Description
Necessity to integrate applications	There must be a clear business need to integrate applications. Time-to-market and real-time reports are examples of business drivers.
Heterogonous environment	When you have to deal with lots of different technologies and protocols, there is a clear need for a central solution that's made to deal with these challenges.
Reduction of total cost of ownership	IT departments are forced to cut maintenance costs to be able to satisfy demands for new products by the business departments. A central integration solution can help decrease the management and maintenance costs of the full application landscape.

We've discussed the point-to-point model and explained the disadvantages of this model. The introduction of an ESB to an application landscape could help to deal with the maintenance nightmare and make it easier to add new applications. Let's go back to the application environment example described in figure 1.1. The addition of an ESB to this environment is depicted in figure 1.2.

What's most striking in figure 1.2 is the reduction in the number of integration connections among the various applications. Every application is connected to the ESB, and the integration logic needed to integrate the COBOL application with the CRM application is implemented within the ESB. Note that the ESB landscape shown in figure 1.2 is just a high-level picture. The picture hides the complexity of implementing the integration logic by drawing an ESB layer, but complexity remains inside this layer that should be dealt with. The big difference with the point-to-point model is that the ESB is designed to deal with integration challenges. Because an ESB provides all kinds of integration functionality, workbenches, and management environments out of the box, implementing a new integration flow between applications is made much easier.

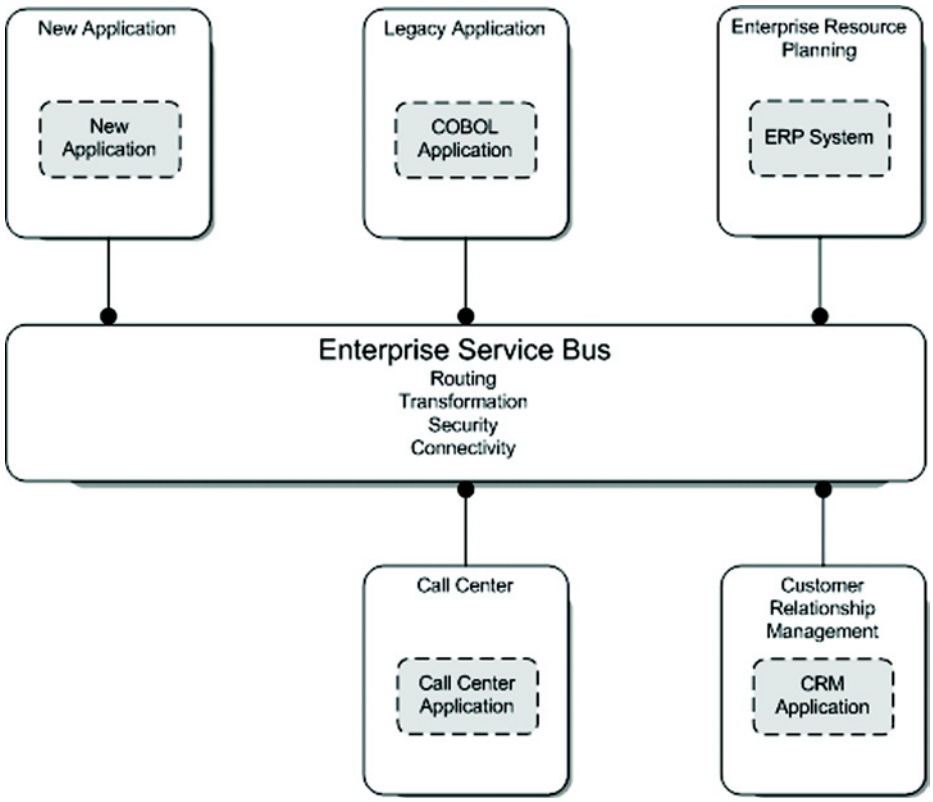

Figure 1.2 An application landscape using an ESB to integrate the applications

As shown in figure 1.2, adding a new application is also simpler than ever before. The new application is connected to the ESB with the transport protocol and technology adapter suited for this application. The integration flows that connect the new application with the three existing applications can be handled within the ESB.

This concludes our discussion about the benefits of an ESB on a high level. Let's focus a bit more on the technology aspects of an ESB, as we take a look at the ESB from an application perspective.

1.1.2 Using an ESB from an application perspective

With the rise of Java Message Service (JMS) as a messaging standard, most of the integration products that are currently available in the market are built with Java technology. The integration vendors, however, hide this Java technology from the integration specialist by offering fancy integration workbenches, which let you use drag-and-drop development. Therefore, integration architects and specialists working with these integration products often don't have a background in Java and Java Enterprise Edition (JEE) development.

This book focuses on a specific kind of ESB, the open source ESB. Open source ESBs are also built on JMS and other Java technologies. Although graphical tools are also available for most of the open source ESBs, as we'll see later in this book, open source ESBs are more focused on Java and XML development. In this book we show many code examples that include Java and XML snippets, because that's the typical place to implement integration logic within an open source ESB. This means that you shouldn't be afraid of doing a bit of Java coding and XML configuration when using an open source ESB.

A COMMON JEE APPLICATION ARCHITECTURE

Because you likely have a Java background, we'll look at the use of an ESB from a Java or JEE application in this section. To start this discussion, let's examine a typical JEE application architecture, as shown in figure 1.3.

The three-tier architecture approach shown in figure 1.3 is common in JEE or Microsoft .NET applications developed in business-critical environments. The division of the application logic into three layers promotes the scalability of an application and should improve the maintainability. All the functionality needed for the application shown in figure 1.3 is implemented in the three layers. The only external part necessary is a relational database to retrieve and store the information that's maintained in the application. So would this application architecture benefit from introducing an ESB?

Well, knowing that the logic implemented in this application isn't used by other applications, the answer is no. This kind of application can be categorized as an isolated application that doesn't have the need to communicate with other applications. In the early years of this century, the architecture of most applications that were developed for large businesses looked like the example shown in figure 1.3.

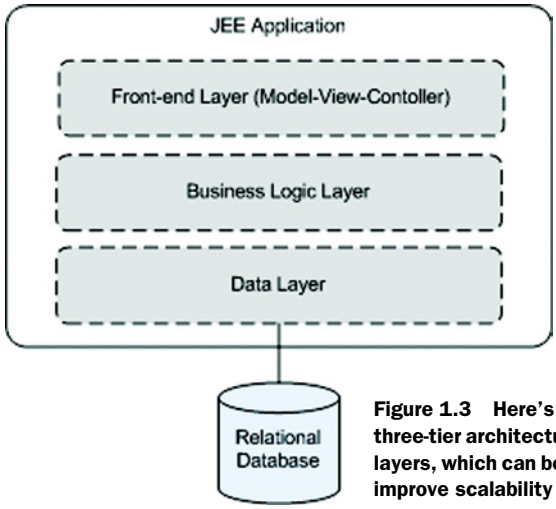

Figure 1.3 Here's a typical example of an application with a three-tier architecture. The application logic is divided into three layers, which can be distributed over multiple physical servers to improve scalability and performance if necessary.

Describing the need for an integration solution

However, with the demand to improve the flexibility of business processes and the time-to-market for new products and other business drivers, applications have to be integrated. The need for a single-client view is an example of the need for application integration. Information about clients exists in many businesses scattered across different applications, like CRM applications, ERP systems, and legacy applications. When the call center of such a business needs a complete client overview, information from all these applications is necessary. And most likely, the call center application is not the only party interested in the client view. Figure 1.4 shows an overview of this single-client view example.

The example given in figure 1.4 requires a solution that's capable of retrieving the information of a specific client from the ERP, CRM, and COBOL applications and that's able to consolidate this information into a single-client view and return it to the call center application. We have multiple options for implementing such an integration solution.

Adding an additional layer to the application

One option is to enrich the call center application with logic necessary to create the single-client view. This would mean that the application architecture shown in figure 1.3 should be extended with an integration layer. This integration layer is responsible for the retrieval of the client information from the three other applications. Although only three applications need to be integrated, a lot of integration logic is necessary. You can imagine that the connectivity necessary to integrate the legacy COBOL application is different from the connectivity needed for the ERP system. This means that the integration layer of the call center application also needs to support different connectivity protocols and likely different message formats as well. The architecture of the call center application would then look like the overview in figure 1.5.

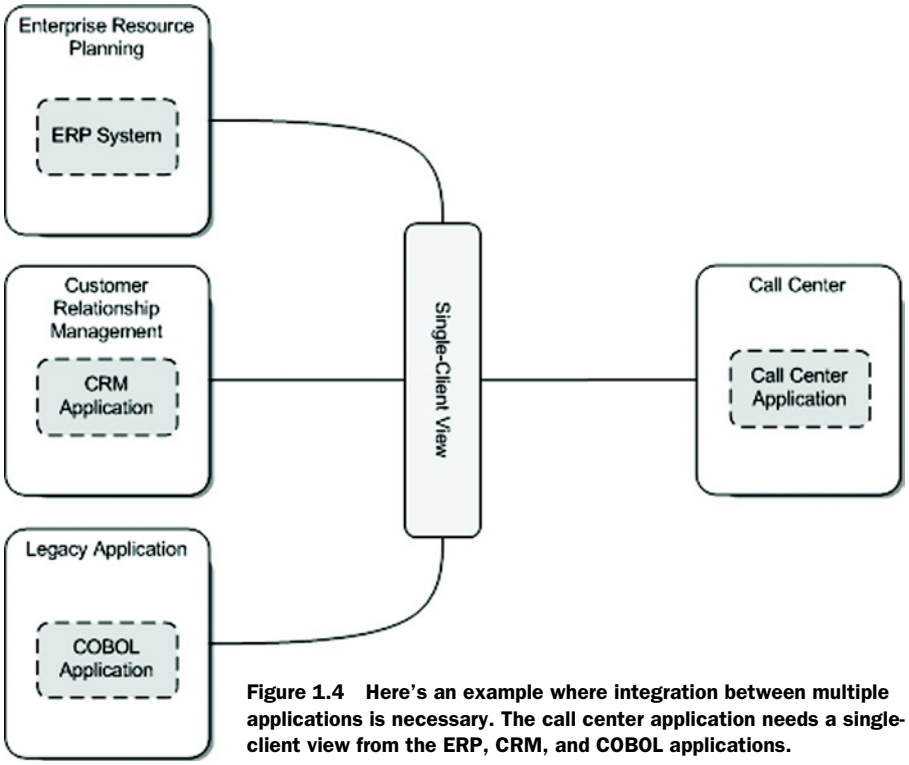

Figure 1.4 Here's an example where integration between multiple applications is necessary. The call center application needs a single-client view from the ERP, CRM, and COBOL applications.

The architecture shown in figure 1.5 is not bad per se. If the logic or data of the applications that needs to be integrated won't be needed in any other application within a department or enterprise, a separate integration solution may not be necessary. But implementing different connectivity protocols; supporting various message formats; and providing messaging, routing, and transformation functionality is a time-consuming exercise. Furthermore, dedicated software is available to solve an integration problem. This is where the ESB product comes into the picture.

USING AN ESB TO IMPLEMENT THE INTEGRATION SOLUTION

When we look at the possibilities for adding an ESB to the architecture shown in figure 1.5, it's clear that the main difference involves the size of the integration layer and the abstraction that an ESB can provide. The integration logic needed for the ERP, CRM, and COBOL applications can be implemented in the ESB solution. Furthermore, the ESB can implement the logic needed to create a single-client view. What remains in the integration layer is connectivity logic to communicate with the ESB. The advantage is that ESBs support a wide range of connectivity protocols, including industry standards like SOAP over JMS or SOAP over HTTP. Figure 1.6 shows the architecture of the call center application with the addition of an ESB for the integration with the three back-end applications.

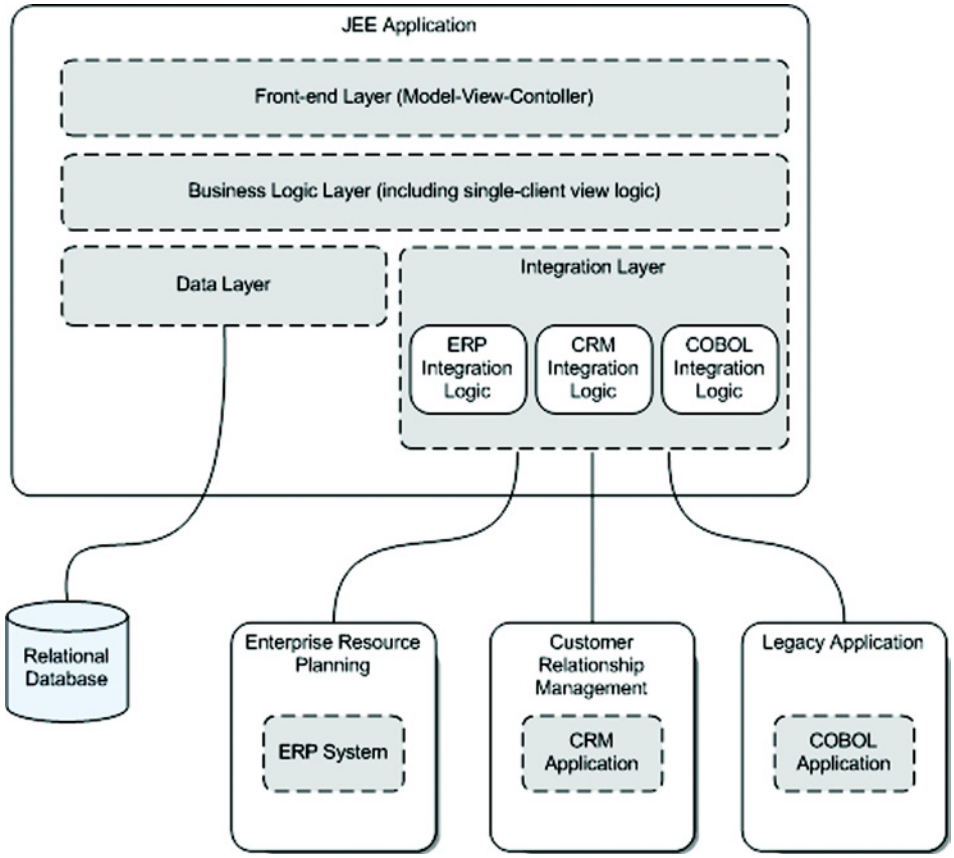

Figure 1.5 The architecture shown in figure 1.3 is extended with an integration layer that provides the logic needed to integrate with other applications.

If we compare figures 1.5 and 1.6, the main difference is where the integration logic for the back-end applications is implemented. In figure 1.5, the integration layer of the call center application implements the integration logic, which translates to a lot of custom development. With the addition of an ESB in figure 1.6, the integration logic is centralized in a software component that isn't part of the call center application. Because ESBs offer an environment that's focused on providing integration functionality, there's no need for much custom development to implement the integration with the three back-end applications.

In figure 1.6 we show a simplified overview of a call center application that's integrated with three back-end applications via an ESB. The advantages of using an ESB become clearer if we consider multiple applications that need to be integrated with, for example, the ERP system and the CRM application. The ESB has already implemented integration logic with these applications, and this logic can be reused for other applications that need information from the ERP system or to update data in the CRM application.

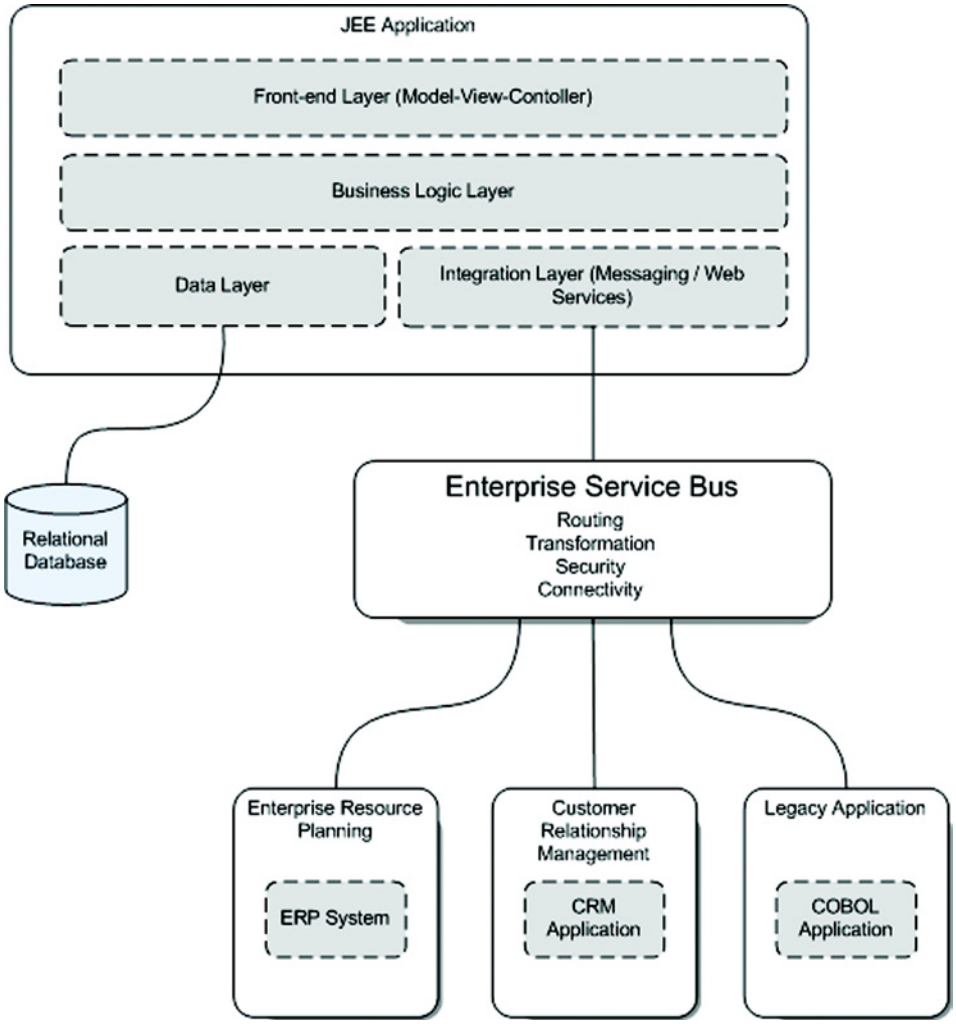

Figure 1.6 Here we introduce an ESB to the call center architecture. The ESB provides functionality to communicate with the three back-end applications and to route the message to the right back-end application.

But before we can decide when and when not to use an ESB, let's first look at the core functionality of an ESB.

1.2 Explaining the core functionalities of an ESB

ESB is a common integration buzzword nowadays, and there are a lot of definitions used by integration vendors, market analysts, and business users. If you want to look for these definitions, just Google "enterprise service bus" and you'll definitely find enough resources for a couple of hours' reading. We provide you with a practical

overview, not an exhaustive list, of what we think are the core functionalities of an ESB. You can then use this overview to create your own definition of an ESB. Table 1.2 provides a short overview of the seven core functionalities.

Table 1.2 Overview of the core functionalities necessary in an ESB

ESB core functionality	Description
Location transparency	The ESB helps with decoupling the service consumer from the service provider location. The ESB provides a central platform to communicate with any application necessary without coupling the message sender to the message receiver.
Transport protocol conversion	An ESB should be able to seamlessly integrate applications with different transport protocols like HTTP(S) to JMS, FTP to a file batch, and SMTP to TCP.
Message transformation	The ESB provides functionality to transform messages from one format to the other based on open standards like XSLT and XPath.
Message routing	Determining the ultimate destination of an incoming message is an important functionality of an ESB that is categorized as message routing.
Message enhancement	An ESB should provide functionality to add missing information based on the data in the incoming message by using message enhancement.
Security	Authentication, authorization, and encryption functionality should be provided by an ESB for securing incoming messages to prevent malicious use of the ESB as well as securing outgoing messages to satisfy the security requirements of the service provider.
Monitoring and management	A monitoring and management environment is necessary to configure the ESB to be high-performing and reliable and also to monitor the runtime execution of the message flows in the ESB.

Next we explore each of these seven core functionalities. The first functionalities that we discuss, location transparency and transport protocol conversion, are typical examples of ESB functionality. The ordering of the other core functionalities is not really relevant.

1.2.1 Location transparency

When a service consumer communicates with a service provider (you can also think of an application here) via the ESB, the consumer doesn't need to know the actual location of the service provider. This means that the service consumer is decoupled from the service provider and that a service provider's new server location has no impact on the service consumer. The core functionality of an ESB that provides this capability is known as *location transparency*.

You can implement the location transparency within the ESB with a simple XML configuration, a database, or a service registry. Your approach depends on your requirements, such as dynamic configuration capabilities and the need for additional information about service providers (e.g., quality of service). The simplest implementation of location transparency is the configuration of service provider endpoints in a static XML file. This is a common way to implement location transparency in an open source ESB. When you need dynamic configuration of service provider locations, you require more advanced configuration options. Dynamic configuration can be implemented with a hot-deployment model for location configuration files or with locations stored in a database. When you have even more requirements, such as the definition of quality of service and business information about a specific service provider, a service registry can provide the necessary capabilities. In this book, we focus on the static XML file and the hot-deployment options. Figure 1.7 shows a graphical overview of the options you have available when implementing location transparency with an ESB.

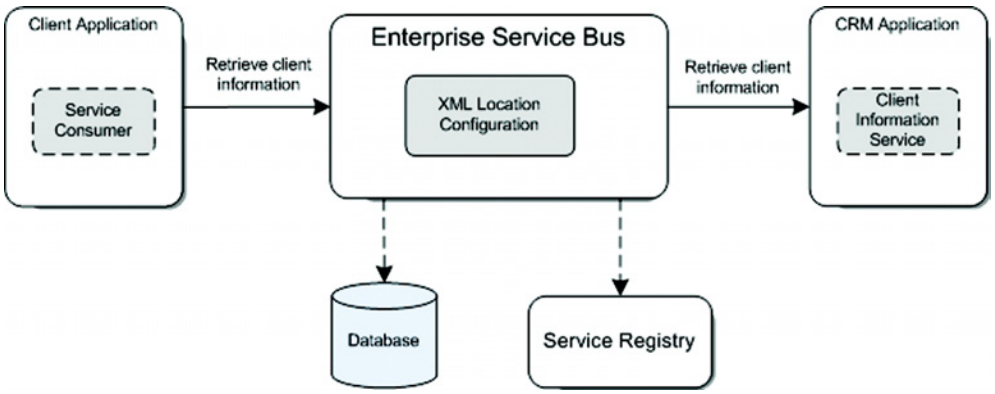

Figure 1.7 The ESB can use several options to configure and store the location of the CRM client information service. A common option is an XML file configuration, but there are alternatives, such as a database or a service registry.

Figure 1.7 shows a simple case in which an application needs client information from a CRM application. Because an ESB is used, the location of the client information service within the CRM application is transparent to the service consumer. Notice that when the location of the client information service changes, only the location configuration within the ESB has to be updated.

1.2.2 Transport protocol conversion

Another common scenario is one in which we have a service consumer that's using a different transport protocol than the service provider is. You can probably think of a number of cases where you have seen this in practice. Let's use an example in which we have a service consumer that's communicating via JMS. The service provider is a

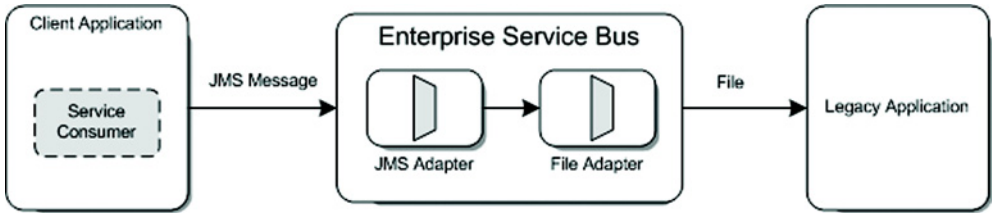

Figure 1.8 In this example a client application sends a JMS message to the ESB. A JMS adapter accepts the JMS message and forwards it to the file adapter, which writes the content of the JMS message to the file system of a legacy application.

legacy system that's only capable of importing and exporting files in a batch. Of course, we can write our own transport protocol conversion logic, but wouldn't it be great if it were offered out of the box? An ESB is capable of converting incoming transport protocols to different outgoing transport protocols; we call this ESB core functionality *transport protocol conversion*. The components in an ESB offering transport protocol conversion are typically referred to as *protocol adapters*. Figure 1.8 shows the transport protocol conversion of the example we have just discussed: JMS to File.

When dealing with environments with many different transport protocols, an ESB can offer transport protocol conversion, as shown in figure 1.8. Of course, a typical ESB doesn't support all of the transport protocols you may come across in complex integration environments, but it does support a wide variety. For protocols that aren't supported out of the box, you can purchase an adapter from third parties or develop a custom adapter.

1.2.3 Message transformation

Besides the support for a set of transport protocols, implementing the integration between a service consumer and a service provider often requires a transformation of the message format. In the example shown in figure 1.8, the content of the JMS message can't be forwarded as is to the legacy application. There is a need for logic that transforms the message format to the expected format of the service provider. The ESB core functionality that helps with changing the message format is known as the *message transformation* functionality.

A common technology to transform a message from the source to the target format is Extensible Stylesheet Language Transformation (XSLT). XSLT is a World Wide Web Consortium (W3C) recommendation widely adopted in the integration industry, which ensures that message transformations written in XSLT are usable in most of the ESBs available in the market. Before the age of open standards like XSLT and the use of ESBs, the EAI products, often referred to as brokers, implemented message transformation most often with proprietary technology. So message transformation is a good example of the evolution of open standards used in integration products. Let's take a look at a graphical representation of message transformation as a core functionality of an ESB in figure 1.9.

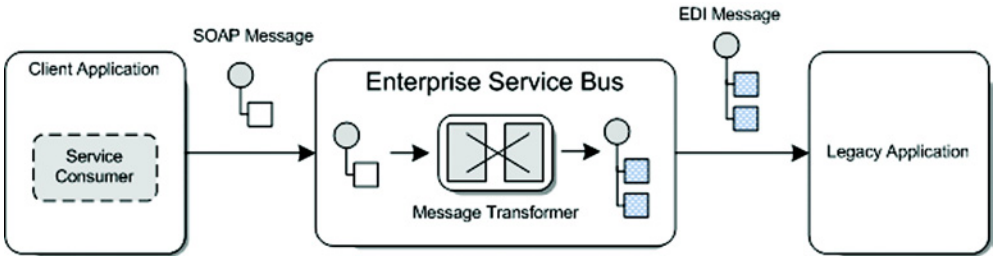

Figure 1.9 An ESB provides the capability to transform the message format of an incoming message to the format expected by the target application. In this example the ESB transforms the SOAP message to an EDI message by using a message transformer.

Message transformation, as shown in figure 1.9, is one of the most used capabilities in ESBs. It's rare that the message format of an incoming message exactly matches the format expected by the target application. The example used in figure 1.9 shows a transformation from a SOAP message to an electronic data interchange (EDI) message. The message transformer that performs the message transformation can be implemented with an XSLT style sheet as we already mentioned, but it can also be a transformation tool from a third party that's dedicated to supporting all kinds of EDI-related transformations. Alternatively, you can write your own with the application programming interface (API) provided with your ESB product. In chapter 5, we explore how message transformation can be implemented with a number of examples.

1.2.4 Message routing

In our examples so far, the target destination of the incoming message was just one possible service provider. But in most integration projects, multiple applications are involved that could be the target application of a particular incoming message. Based on many kinds of rules and logic, the ESB has to determine which service provider(s) a message must be sent to. The core functionality involved with dealing with this kind of logic is known as *message routing*.

This message routing functionality is a classification for different kinds of routing capabilities. There is, for example, content-based routing, which is used for routing messages to their ultimate destination based on their content. But there is also the message filter routing functionality, which is used to prevent certain messages from being sent to a particular destination. A third example is the recipient list routing capability, which can be used to send a particular message to multiple destinations. Message routing is the ESB core functionality needed in almost every integration implementation. Figure 1.10 shows an example of message routing based on the content of an incoming message.

Message routing can be complex and difficult to implement because knowledge of the routing rules and logic involved is often spread across different people. It's difficult to explain the use of routing rules to businesspeople, although their business

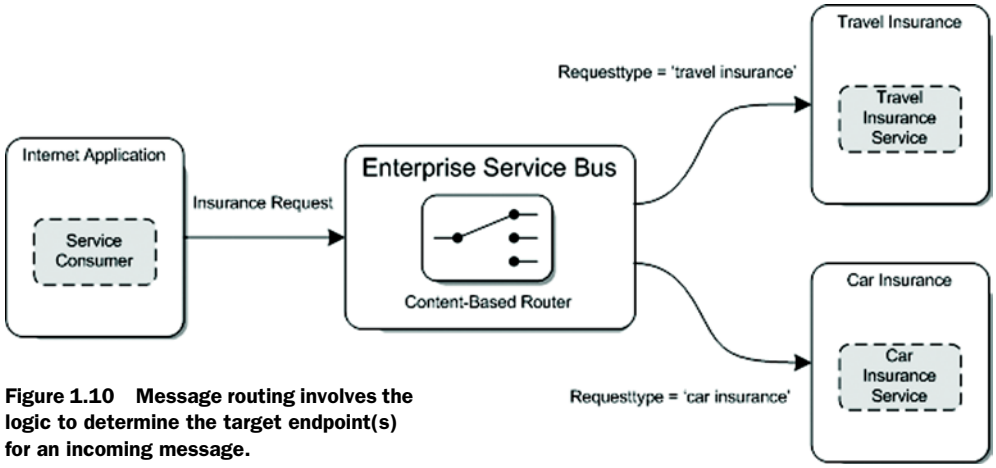

Figure 1.10 Message routing involves the logic to determine the target endpoint(s) for an incoming message.

domain is essential for the definition of a routing rule. The rules and logic involved with message routing is, however, related to the functionality of the applications that are integrated. Therefore, a deep understanding of the applications involved in an integration project is often necessary to specify the proper routing rules. The example given in figure 1.10 is just a simple one, designed to show what we mean by message routing and content-based routing. In figure 1.10 the insurance request consists of an element named `requesttype` that specifies the kind of insurance request applied for by the customer using the internet application. Based on the value of this element, the message is routed to the travel or the car insurance service. We'll look at different routing examples in greater detail later in this book, starting with chapter 5.

1.2.5 Message enhancement

The message transformation process can change a source message format to a target message format. But to be able to create the correct outgoing message that will be sent to the target application, you may have to add additional data or convert the existing data. A common way to add data to a message is by retrieving it from a database based on certain element values of the incoming message. An example of such an element is a *client identifier*. The destination of the incoming message with the client identifier can be an application that requires some extra client information that's not available in the incoming message. The ESB can then retrieve this information from a database based on the client identifier in the incoming message. For data conversion, more custom development is needed in most cases. A data conversion example is where the length of the client name has to be reduced to a maximum length of 40 characters. This functionality requires a clear message-handling API so that the retrieval and update of a particular message element is made easy for a developer.

The functionality described here can be categorized as a *message enhancement* capability and is closely related to message transformation. The main difference between these functionalities is that message transformation deals with data that's already

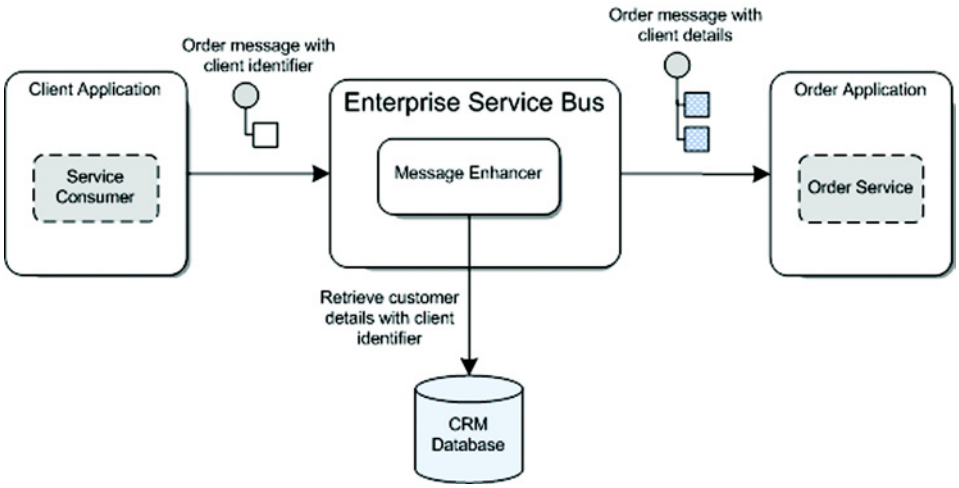

Figure 1.11 In this example of message enhancement, an order message with a client identifier is sent by a client application. The ESB retrieves the customer details from the CRM database using the client identifier with message enhancement capability.

available in the incoming message, and message enhancement deals with data that must be retrieved from a (external) data source, for example a database. Figure 1.11 shows an example of message enhancement.

The example shown in figure 1.11 uses a message enhancer that retrieves client information from a database based on the client identifier provided by the incoming message. In this typical example, the ESB needs to provide functionality to connect to a database and perform a query with parameters provided in the configuration settings. Another functionality that's used often and that's part of message enhancement is that some custom logic is performed against the incoming message. This custom logic can be implemented with, for example, Java code to retrieve data from an external database. We'll discuss the functionality of message enhancement in greater detail in chapter 5.

1.2.6 Security

Because ESBs often deal with business-critical integration logic that involves a substantial number of applications, an ESB must provide ways to authenticate and authorize incoming messages. For messages that may be intercepted for malicious purposes, encryption is an important feature that an ESB must be able to provide. When an ESB doesn't apply a security model for its environment, everybody who can send messages to the starting point of an integration flow, such as a message queue or a web service, is able to start this flow with possibly malicious messages. In most situations, an ESB is an internally oriented environment, which means that it's not accessible from outside the organization boundaries, due to firewall settings. This means that possible malicious

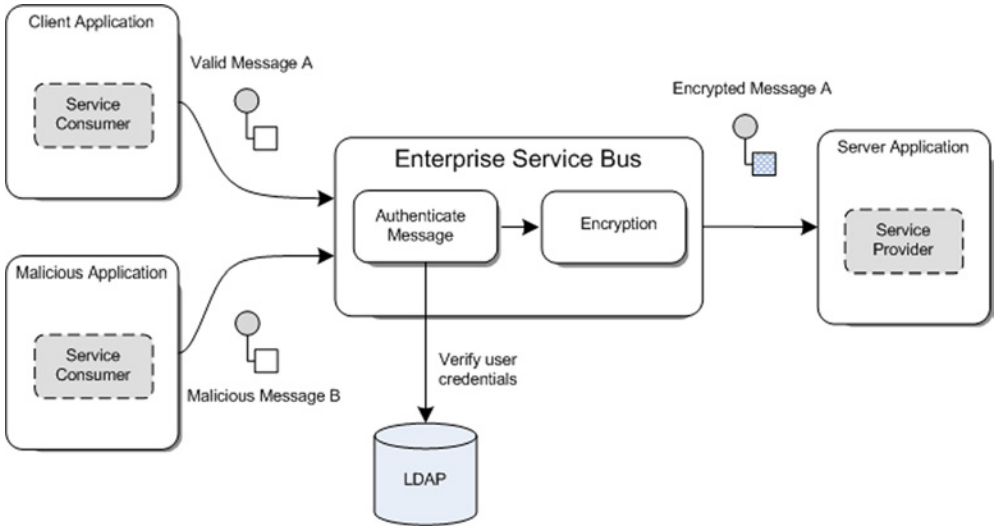

Figure 1.12 Security involves the confidentiality, integrity, and availability of messages sent over the ESB. This example shows an implementation of confidentiality via an authentication mechanism.

messages can only arrive from within the IT environment of the organization. But when an ESB also offers starting points of integration flows to applications outside the boundaries of the organization, security is even more important.

Let's first look at an example of security within an ESB (see figure 1.12).

The example in figure 1.12 shows how the authentication inside an ESB can be implemented. Besides authentication, authorization can also be configured for an integration flow. By using authorization, the functionality of a service provider can be secured on a method level so that, for example, a group of users can be granted different access than an administrator user. Our example also implements encryption for the outgoing message before it's sent to the service provider. This is another part of the security functionality an ESB should be able to implement. Service providers can have all kinds of security measures implemented, and an ESB should be able to construct an outgoing message that has the right security values set. For example, to ensure that a message can't be read by other parties, a message can be encrypted with the public key of the service provider in the ESB, as in the example in figure 1.12. As you can see, security is a broad topic. In chapter 8 we discuss how security can be implemented in an ESB with a number of practical examples.

1.2.7 Monitoring and management

The last ESB core functionality that we examine involves managing an ESB environment. This core functionality is different from the ones we've discussed, as the others were focused on development and runtime capabilities of an ESB. This section focuses on the ability to maintain and manage an ESB.

Because an ESB is a critical piece in a system landscape, the environment must be managed and monitored. This is not that different from application servers hosting JEE applications, but an ESB usually integrates a large set of applications not only limited to a Java domain. Therefore, if the message size in a queue is exceeding a certain limit, for example, that must be detected as early as possible. We categorize this functionality as *monitoring and management*. A graphical representation of this ESB core functionality appears in figure 1.13.

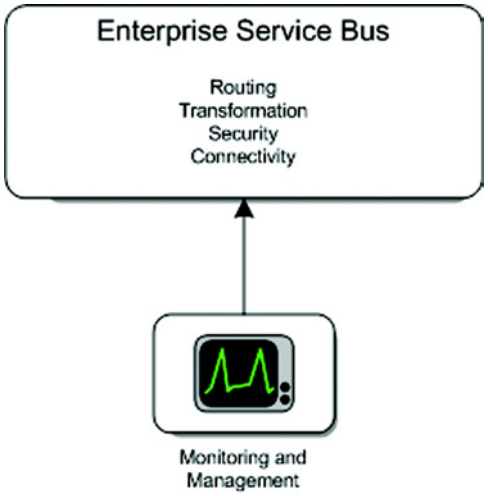

Figure 1.13 The ESB is a central product within the environment and therefore monitoring and management capabilities are vital.

Managing and monitoring an ESB environment can become complex because of the large set of capabilities an ESB provides. Therefore, the management and monitoring functionality consists of multiple parts, and each is responsible for a component of the ESB. For the messaging layer in the ESB, the management and monitoring environment will, for instance, involve managing the queues and monitoring the message size and message throughput of queues. For web services provided by the ESB, monitoring will involve such things as whether the web service is up and running and how many calls are made per minute; management will address the number of instances that are running for a web service. In chapter 10, we explore examples involving the management and monitoring capabilities of an ESB.

1.2.8 Core functionality overview

Well, we covered quite a bit of ground in this section; we hope we didn't overwhelm you with a theoretical discussion of the core functionality of an ESB. We wrote this book with a practical goal in mind: to show how ESBs and, in particular open source ESBs, can be used for your own purposes. But we think that a book about ESBs should begin with a clear definition of what an ESB is and what it should do. This section defines seven core functionalities an ESB should provide at the very least. These core functionalities are by no means a complete list; we didn't yet mention orchestration and transaction handling. These functionalities are important, but we chose the keep the list of core functionalities short—just enough to provide a good picture of what an ESB is. In the remaining chapters, we discuss other functionalities.

You should now be able to arrive at your own opinion as to what an ESB is and how it compares to the definition you found on the internet. You can see that some definitions are difficult to understand, and some define an ESB as the ultimate

integration solution. But let's end our theoretical discussion of ESB functionality and move on to practical implementations. Next we present an overview of available open source ESBs.

1.3 Researching the open source ESB market

A lot of definitions you find on the internet are from vendors who are selling ESBs for a substantial license fee. Because this book is about open source ESBs, we first discuss the differences between *closed source* (products with a license fee and a confidential source) and open source ESBs based on the core functionalities we've discussed.

As we see in section 1.3.2, quite a few options are available in the open source market. To provide a good overview of the available open source ESBs, we introduce the most important ones in short sections. But let's begin with a discussion of some open source ESBs myths. We include this short discussion because there's a lack of clarity within the integration market about open source products.

1.3.1 Demystifying the open source ESB

Let's first specify what we mean by open source and so-called closed source ESBs. When we talk about closed source ESBs, we're referring to ESB products that have a usage-based license fee and for which the source code is not freely available. Open source ESBs do have a license (like the Apache or GPL license), but don't have a usage-based license fee and the source code is freely available. Although the open source ESB itself is available for free, services and support can be provided for a fee.

Therefore, open source ESBs can be provided by a commercial vendor just like closed source ESBs. Good examples of companies making money with open source ESBs are MuleSource (the company behind Mule) and IONA Technologies (which offers support and training for Apache ServiceMix with its FUSE ESB product). So let's explode the myth that open source ESBs lack support and training options. The open source ESBs discussed in this book, Mule and ServiceMix, have company backing that provides 24/7 support and can offer training.

A second myth is that open source projects in general, including open source ESBs, are led by geeks who are developing interesting pieces of software, but lack a quality assurance (QA) model, a decent release roadmap, and a delivery process. Of course, open source development means that developers are often working full-time in their day-to-day job and are developing the open source projects in their spare time. However, there's a movement in which full-time open source developers work for a company to offer support and training for an open source project. Again, good examples include MuleSource and IONA; in addition, WSO_2 (with Apache Synapse) and Sun Microsystems (with Open ESB) fit this picture.

Because all decent open source projects use a bug-tracking environment like Atlassian's JIRA (which identifies all closed and open bugs and also provides information about the release schedule), a solid foundation for QA is laid. In addition, good unit tests, a continuous build environment, and an active community pave the way to

well-tested and community-driven releases. With a release roadmap, which consists of several release candidates, the quality of the open source ESB can be guaranteed. The community behind the open source ESB is involved in the delivery process of a new version. So in conclusion, the great thing about open source projects is that the QA model is open for everyone and that you are able to test new releases early in the release process.

The last myth that we want to discuss is that open source ESBs lack tool support for development and testing. Closed source ESBs provide integration workbenches to give developers an abstraction layer that hides the technical implementation. The integration workbench provides drag-and-drop development interfaces and wizards to create integration flows. This means that the developer is more or less guided in the design and implementation of an integration flow. For open source ESBs, the tool support is more basic, with a focus on XML configuration and Java development. This means more or less that there's no abstraction layer to hide the technical implementation. Developers working with open source ESBs therefore need to have more development knowledge to implement integration flows. But this also gives developers greater freedom in implementing integration logic for their integration solution. And because enterprise integration is difficult and often requires custom integration logic, this can be very welcome.

But does this mean that the myth about tool support is true? No, tool support is available that can ease the development effort when working with open source ESBs. In appendix C we show two examples of tool projects that provide graphical support for constructing message flows for Mule and ServiceMix. And in chapter 11 we examine two tools that provide graphical drag-and-drop support to construct processes that can be deployed on Mule and ServiceMix. So the tool support is growing and will be enhanced in the near future, but admittedly there's some catching up to do when compared to the closed source ESB product offerings. In table 1.3 the myths about open ESBs are summarized.

Table 1.3 Overview of the myths about open source ESBs

Myth	Short description
Lack of support and training	Just like the closed source ESBs, 24/7 support and training are available for open source ESBs. Companies like MuleSource, IONA, WSO_2, Sun, JBoss, and EBM Websourcing provide support and training for specific open source ESBs.
Lack of QA, a decent release calendar, and a delivery process	Open source ESBs that we examine in section 1.3.2 have an excellent bug-tracking system, provide unit tests, and are backed by an active community. In addition, a core team of developers is often working full-time on the development of the open source ESB. Therefore, the QA model and release process are well implemented and also open to everyone who is interested.

Table 1.3 Overview of the myths about open source ESBs *(continued)*

Myth	Short description
Lack of tool support	Open source projects are not famous for their tool support. This is not different for most open source ESBs, so Java and XML skills are mandatory for open source integration developers. Tool support is, however, growing, and the NetBeans support for open ESB is a great example of an open source ESB with good tool support.

Now, let's look at the best-of-breed open source ESBs currently available.

1.3.2 Overview of open source ESBs

In just a couple of years, we've seen quite a few open source ESBs arrive on the market. The adoption of open standards and Java specifications like JMS, JCA, XML, JBI, SOAP, and others paved the way for the development of open source ESBs. With the specifications available for everyone who is interested, the only things lacking were implementations of these specifications. A number of open source projects started to implement specifications like JMS and JBI. These projects provided the foundation to build open source ESBs, and eventually several open source ESB projects were launched.

The problem with providing an overview of open source projects for a particular technology or functionality is that there are so many projects out there. This isn't different for open source ESBs. Therefore, we have only listed the open source ESBs that received a lot of attention on the internet and in the integration market. Another criterion is that we focused on the open source projects provided by a substantial community, such as Apache, Codehaus, Java.net, and JBoss.

MULE

After doing the same donkey work at a number of integration projects for setting up an integration infrastructure, Ross Mason decided to define a reusable integration platform named Mule (http://mule.codehaus.org). Basing his work on Gregor Hohpe and Bobby Woolf's book *Enterprise Integration Patterns* (Addison-Wesley Professional, 2003), Mason implemented a lightweight messaging framework. The central part of Mule is the service definitions, which implement the integration logic.

These services can consist of an inbound and outbound element to configure the input and output connectivity. A service can also consist of a component, which can be implemented with all kinds of technologies, including Java and Spring beans. This is a big selling point for Java developers who are looking for an integration framework. Most of the development work with Mule can be implemented with Java classes, and the messages that flow through the Mule container can be Java messages. Figure 1.14 gives an overview of the functionality provided by Mule.

Mule offers connectivity for more than 20 transport protocols and integrates with a large number of integration projects, including Spring, ActiveMQ, Joram, CXF, Axis,

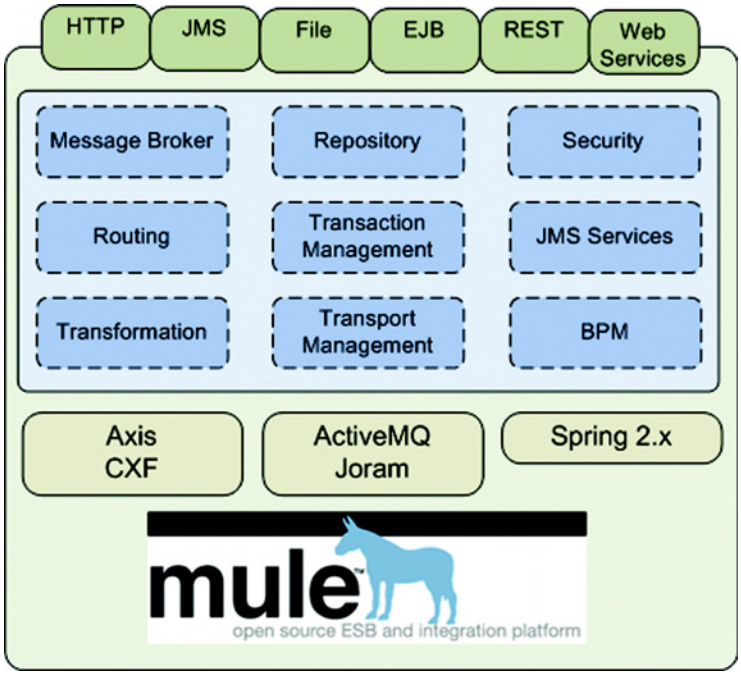

Figure 1.14 Overview of the functionality provided by Mule. The figure shows some examples of open source frameworks that can be integrated with Mule, including CXF and ActiveMQ.

and Drools. Mule chose to not build their architecture on JBI, but implemented their own flexible and lightweight model, focusing on productivity and ease of development. After the 1.0 release of Mule in 2005, Mule received more and more market attention over the years, resulting in the creation of MuleSource (http://www.mulesource.com), which provides professional services and support for Mule. This product is currently used by a large number of companies around the world, including WalMart, Hewlett-Packard, Sony, Deutsche Bank, and CitiBank.

Mule provides connectivity with JBI containers via a JBI adapter implementation. But the next open source ESB, Apache ServiceMix, is fully based on the JBI specification.

APACHE SERVICEMIX

The foundation for Apache ServiceMix is the JBI specification delivered by the Java Community Process (JCP) under Java Specification Request (JSR) 208 in 2005. The purpose of JBI is to define a standard for an integration platform that consists of components from multiple vendors and open source projects (in order to prevent vendor lock-in). For integration products adhering to the JBI specification, it should be possible to build JBI components that can be deployed on all these JBI-based products. A salient detail of the JSR 208 vote was that IBM and BEA abstained, and even today these companies have no integration product adhering to the JBI specification.

For more details about the JBI specification, see chapter 2, where we discuss the architecture of ServiceMix. Figure 1.15 gives an overview of the functionality provided by ServiceMix.

After the JBI specification was accepted by the JCP, in late 2005 the Apache ServiceMix project was introduced as an incubator project at Apache. The goal of ServiceMix is to provide an ESB that implements the JBI specification, with a focus on flexibility, reliability, and breadth of connectivity. ServiceMix includes a large set of JBI components that together supply the ESB core functionalities listed in section 1.2. Included are JBI components that support protocols like JMS, HTTP, and FTP, as well as components that implement Hohpe's patterns of enterprise integration, rules, and scheduling.

In September 2007 the Apache ServiceMix project became a top-level Apache project. The ServiceMix product can be integrated with a number of other Apache projects. Apache ActiveMQ is the messaging foundation, which provides reliability and makes possible a distributed environment and clustering. ServiceMix can also be integrated with Apache CXF, Apache ODE, Apache Camel, Apache Geronimo, JBoss, and any web container. ServiceMix is deployed in many large enterprises around the world, including Raytheon, British Telecom, CVS/Pharmacy, Cisco Systems, and Sabre Holdings, just to name a few.

LogicBlaze was the professional services, support, and training company behind Apache ServiceMix and Apache ActiveMQ. Some of the core developers of ActiveMQ and ServiceMix were employed by LogicBlaze. In 2006, LogicBlaze was acquired by IONA Technologies, which now provides support, services, and training for ServiceMix and other Apache projects via its FUSE ESB product. The FUSE ESB is an open source

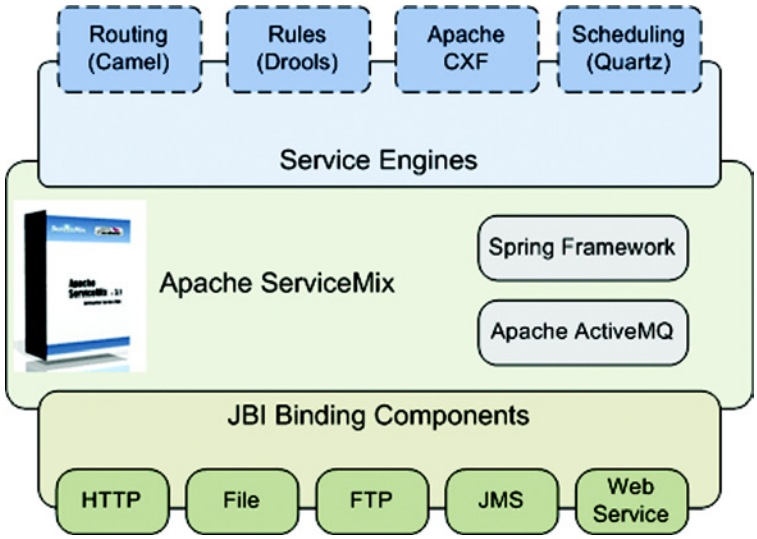

Figure 1.15 Overview of the functionality provided by Apache ServiceMix

product based on ServiceMix and includes other products based on Apache ActiveMQ, Apache Camel, and Apache CXF.

OPEN ESB

The two previous examples of open source ESBs currently available are hosted by open source communities, Mule at Codehaus and ServiceMix at Apache. Open ESB is an ESB initiative started by Sun Microsystems and is hosted as a Java.net project. All of the committers of the Open ESB project are employees of Sun Microsystems. Like the ServiceMix ESB implementation, Open ESB is also an implementation of the JBI specification. The Open ESB project is an umbrella project that includes a number of subprojects. One subproject implements a JBI runtime environment that was started to provide a JBI implementation for Sun's Glassfish application server. Other subprojects provide JBI components such as a JMS binding component and a Business Process Execution Language (BPEL) service engine.

This kind of functionality is also provided by ServiceMix, but one difference is that Open ESB is focused on the Glassfish application server and ServiceMix can be easily deployed on the Apache Geronimo or the JBoss application servers. However, the main difference between Open ESB and other ESB implementations like ServiceMix is the tooling support. Because Sun can build upon its NetBeans IDE, it can provide wizards for building JBI components and design the dependencies between these components. In 2007 the NetBeans IDE was enriched quite impressively as part of Sun's open source strategy. An Enterprise package is available that, among other things, provides a BPEL editor, a Web Services Description Language (WSDL) editor, and a Composite Application Service Assembly (CASA) editor. The CASA editor provides a drag-and-drop interface for designing a JBI service assembly.

APACHE SYNAPSE

One can question if Apache Synapse actually is a true ESB, but based on the core functionalities that we list in section 1.2, Synapse *can* be called an ESB. In essence, Synapse is a web services mediation framework that is built upon Apache Axis2, the web services container. This is quite a difference compared with the previously discussed ESBs (Mule, ServiceMix, and Open ESB). The focus of Synapse is to provide functionality such as routing, transformation, message validation, and a registry based on web services and XML standards.

As a part of standardization organizations such as OASIS (Organization for the Advancement of Structured Information Standards) and W3C (the World Wide Web Consortium), an enormous set of web services specifications is being standardized. A few examples of these web services specifications are WS-Addressing, WS-Security, WS-Policy, and WS-Reliable Messaging. Based on the naming of these specifications, you can pretty much extract the goal: for example, WS-Security provides a specification for things like message encryption and authentication, and WS-Reliable Messaging shows how messages between web services can be exchanged in a reliable way. These web services standards are quite complex in general, and as a developer, you aren't always interested in dealing with the exact syntax and semantics of a web services standard.

So besides offering ESB core functionalities such as routing, transformation, and a registry, Synapse can provide the necessary abstraction to use complex web services standards. Here's an example of this abstraction layer: with only two lines of XML configuration, Synapse is able to execute a message exchange with a WS-Reliable messaging enabled web service. The same abstraction is provided for WS-Security and other web services standards.

The primary connectivity options are SOAP over HTTP and JMS, but other options such as SMTP are also possible. Early in 2007, Synapse graduated from the incubator status to become a full member of the Apache web services project. When dealing with integration problems in a web services area, Synapse provides the necessary functionality.

JBoss ESB

JBoss ESB is widely known for its popular application server and successful open source projects such as Hibernate, an object relational mapping (ORM) framework; Seam, an application framework for building Web 2.0 applications; and jBPM, a process engine. In the area of enterprise integration, JBoss provides a JMS provider called JBossMQ and a rules engine, JBoss Rules. In mid-2006, JBoss acquired an ESB product, Rosetta, that was used as a messaging backbone for a large insurance provider in Canada. Based on Rosetta, JBoss developed a product called JBoss ESB, which provides most of the core functionalities listed in section 1.2.

With the addition of the JBoss ESB product, a complete integration architecture can be implemented based on JBoss products. The messaging layer of JBoss ESB is called JBoss Messaging (the successor of JBossMQ), a JMS provider implementation. The routing functionality of JBoss ESB is based on the rules engine, JBoss Rules, and the orchestration functionality is provided by the jBPM process engine. For custom logic and web services support, Enterprise JavaBeans (EJB) 3 or Plain Old Java Object (POJO) components can be developed that can be deployed on a JBoss application server.

OW2 PEtALS

We've discussed three open source ESBs that implement the JBI specification. With PEtALS we introduce the fourth and last JBI-based open source ESB. PEtALS, which was officially JBI certified by Sun Microsystems in March 2008, provides many connectivity options out of the box. In 2005, architects and EAI experts from EBM Websourcing were unsatisfied with existing integration solutions and decided to launch an open source project named PEtALS. In the following years, the open source ESB gradually supported more connectivity options like JMS, SMTP, file, SOAP, and FTP support.

The PEtALS architecture is focused on providing a distributed environment where different PEtALS instances are seamlessly integrated into one ESB environment. This means that services hosted on different instances are able to access one another without additional configuration. Another selling point is the web-based monitoring tool, whereby JBI components can be managed and your message flows can be deployed and managed. We show an example of this monitoring application in chapter 10.

APACHE TUSCANY

Apache Tuscany is a good example of a product that doesn't use the term *ESB* in its project description but that does provide many of the core functionalities listed in section 1.2. This Apache project is an implementation of the Service Component Architecture (SCA) specification, which is hosted by 18 companies, among them BEA, IBM, Interface21, Oracle, and Sun (http://www.osoa.org). The SCA specification is currently being standardized by OASIS. The goal of the SCA specification is to define services in a vendor-, technology-, and protocol-neutral way.

The specification describes a Service Component Definition Language (SCDL) XML configuration in which you can define service components. A service component has a particular implementation (Java, PHP, BPEL, Ruby script, and so on) and exposes this functionality via service interfaces. A service component is able to communicate with other components and services through service references. Figure 1.16 provides a schematic overview of service components as defined in the SCA specification.

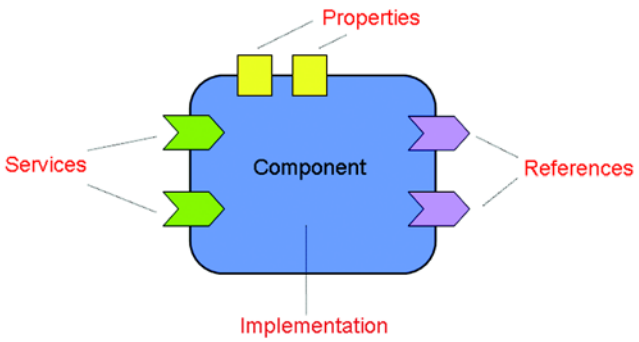

Figure 1.16 A schematic overview of an SCA component as defined in the SCA specification. A component provides services to other components and uses references to invoke other components.

Until this point we haven't seen a lot of overlap with ESB functionality. However, the service interfaces and service references have a binding type attached to them. A binding type can, for example, be a web service, a JMS component, an EJB, or a JCA adaptor. This means that you are able to expose a Java component as a web service. This provides all kinds of connectivity options that can also be seen in ESB products. When we compare this functionality with the JBI specification, it is similar to the binding components concept. On the other hand, an SCA component could be perfectly suited as a JBI service engine implementation.

We decided not to include SCA and Apache Tuscany in the remaining chapters of the book, although SCA is an interesting specification. We consider SCA to be a specification that is focused on implementing services as part of a SOA. Therefore, it doesn't fit in the open source ESB topic of this book.

SPRING INTEGRATION

Another interesting open source integration framework initiative is the Spring Integration framework, which was announced by SpringSource, the company that provides

support and services for the Spring Framework, in December 2007 (http://www.springsource.com/web/guest/2007/springintegration). Spring already provided support for integration projects with JMS templates and web services, but with the Spring Integration project, the strength of the integration functionality is centralized within an abstraction layer, which eases development.

The integration functionality provided by Spring Integration is based on the patterns found in the *Enterprise Integration Patterns* book. With a simple Spring XML and annotations-based model, Spring beans can be utilized in an integration environment. At the time of this writing, there is no 1.0 release yet, but when this book hits the shelves we feel certain there will be a first production release available. In chapter 4 we offer a sneak preview of what you can expect from the Spring Integration framework.

Other open source ESBs

A number of open source ESBs are part of an offering from companies with a smaller community. We've mentioned FUSE ESB (provided by IONA), but there are other examples, such as WSO2's Enterprise Service Bus, which is based on the Apache Synapse product.

Other open source ESBs that we did not mention yet are ChainBuilder ESB, a project by Bostech focused on tool support, and the OpenAdapter platform, which provides support for all kinds of transport protocols and enterprise integration functionality like transformation, validation, and data filtering. Table 1.4 offers an overview of the open source ESBs.

Table 1.4 Open source ESB overview

Open source ESB	Website	Short description
Apache ServiceMix	http://servicemix.apache.org	Apache JBI implementation with a lot of JBI components
Apache Synapse	http://ws.apache.org/synapse	ESB focused on web services support based on Apache Axis2
Apache Tuscany	http://tuscany.apache.org/	Implementation of the (SCA) specification
ChainBuilder ESB	http://www.chainforge.net/	A JBI-based ESB that focuses on providing graphical tools to ease the development effort
FUSE ESB	http://open.iona.com/products/fuse-esb/	IONA's open source ESB offering based on Apache ServiceMix
JBoss ESB	http://labs.jboss.com/jbossesb/	The JBoss implementation of an ESB based on JBoss messaging
Mule	http://www.mulesource.org	Lightweight ESB with a custom implementation model

Table 1.4 Open source ESB overview *(continued)*

Open source ESB	Website	Short description
OpenAdapter	https://www.openadaptor.org/	EAI-based platform that provides a number of adaptors to implement integration solutions
Open ESB	https://open-esb.dev.java.net	JBI implementation provided by Sun that provides great tool support with NetBeans
PEtALS	http://petals.objectweb.org/	Another JBI-based ESB, hosted by OW2 (formerly ObjectWeb)
Spring Integration	http://www.springframework.org/spring-integration	An integration framework that is provided by the well-known Spring Framework
WSO_2 ESB	http://wso2.com/products/esb/	WSO_2's open source ESB offering based on Apache Synapse

As you can see in table 1.4, the open source ESB options are substantial. Now that we've taken a quick look at a number of open source ESBs, it's time to discuss why we chose Mule and ServiceMix for this book.

1.4 Why did we choose Mule and ServiceMix?

To make an objective choice, we define the criteria that we've found are important when choosing an open source ESB.

1.4.1 Defining selection criteria

The first criterion is quite obvious; we defined a list of core functionalities that should be present in an ESB in section 1.2, so the ESB of our choice has to provide support for all these capabilities.

An important criterion that's often neglected by open source projects is the quality of the documentation available. Of course, the source is available for everyone to look into, but developers need good-quality documentation when using an ESB.

Another criterion is the market visibility of the open source ESB. This includes the availability of articles on the internet as well as the number of implementations in businesses.

When working with an open source product, you may run into bugs or enhancement requests. Therefore, it's important that an open source product be able to rely on an active community that can provide support for questions and solve bugs and that's capable of including enhancement requests in future releases.

Another important criterion for open source ESBs is the flexibility and the development effort needed to implement custom logic. Integration projects often need

to deal with unique requirements, due to the specific environment for a particular business, so custom logic may be necessary for an integration solution. It should therefore be a minimal effort to implement your own piece of Java code.

Because an ESB is a central component that must be able to integrate applications implemented with different technologies, the transport and connectivity options provided by an ESB are key. An important criterion when selecting an ESB is whether it can provide the connectivity you need in your own IT environment. Developing custom transports requires a lot of expertise and development effort that could be provided out of the box.

This criterion has some common ground with another important capability of an ESB: the ability to integrate with other open source projects. Because there's a wide variety of open source projects available that provide solutions for all kinds of integration challenges, it's important that the ESB of your choice be able to integrate with these projects. There are, for example, ESBs that integrate with the Spring framework out of the box, which can provide you with a lot of additional functionality you may need.

Another criterion that we want to share with you is the support for implementing your integration solution via an IDE. The tool support for open source projects is something that has room for improvement. However, with the availability of Eclipse and NetBeans as IDE platforms, this support is improving more and more. The development effort needed to build a robust interface capable of configuring an ESB is quite extensive. Therefore, many open source ESBs are focusing more on the runtime functionality than on increasing developer productivity via GUIs. Having said this, a drag-and-drop user interface doesn't necessarily improve a developers' productivity.

1.4.2 Assessing the open source ESBs

We should now be able to classify five of the open source ESBs listed in section 1.3.2 with the eight criteria that we have discussed. Notice we don't include Tuscany, Spring Integration, or any of the vendor-based ESBs like FUSE and WSO_2 ESB. We consider the open source ESBs listed here as the best-of-breed products currently available. The following list discusses the eight criteria (two criteria are discussed together) for the five open source ESBs:

- *ESB core functionality*—Mule and ServiceMix provide good support for the core functionalities. Open ESB isn't yet there with message routing and message enhancement. In the areas of monitoring and management, there is still room for improvement in all ESBs.
- *Quality of documentation*—The documentation of Mule is the most complete. The examples in ServiceMix's documentation are sometimes outdated. Open ESB's documentation is excellent and offers screenshots and clear examples, but they are focused on BPEL. Synapse provides good examples and documentation.

- *Market visibility*—The number of implementations at businesses around the world is starting to increase. Mule has a head start, because it can rely on a large number of implementations of its product and is receiving a lot of attention in the market. ServiceMix has also received a lot of attention, because it is the best-known open source JBI implementation and an Apache project. For the other ESBs, the market visibility is increasing but has some catching up to do compared with Mule and ServiceMix.
- *Active development and support community*—The communities of the five open source ESBs are in general quite active, and support is available from commercial parties as well as forums. Mule has an active development community and is able to provide support via MuleSource. The same is true for ServiceMix, with support offered by IONA. The community for the other open source ESBs is a bit smaller but is growing.
- *Custom logic*—Most of the reviewed ESBs provide good support for adding custom logic. Mule, ServiceMix, and Synapse make it easy to add your own piece of logic, because you can easily integrate POJOs. With Open ESB the focus is not yet on supporting developers to write custom logic, but on writing binding components and service engines.
- *Transport protocols and connectivity options and integration capabilities with open source frameworks*—Most of the transport protocol and connectivity support is offered by integrating other open source products with the ESB. ServiceMix and Mule provide the widest range of connectivity and open source product support. Synapse and PEtALS also offer a nice set of transport protocols and open source products. For Open ESB, the capabilities and support is increasing but is not so impressive yet.
- *Tool support*—Open ESB has excellent IDE support with NetBeans. Some of the other ESBs provide some Eclipse plug-ins, like the Mule 2.0 IDE and the Eclipse STP Enterprise Integration Designer, but these tools don't offer the same level of quality as the Open ESB IDE support.

Table 1.5 shows an overview of five open source ESBs that meet these criteria.

Table 1.5 An assessment summary of the five open source ESBs related to the selection criteria. The notation used in this classification is simple: ++ is very good, + is good, +/– is average, – is not well supported, and — is not supported at all.

Selection criterion	Mule	ServiceMix	Open ESB	Synapse	PEtALS
1. Support for ESB core functionality: location transparency, transport protocol conversion, transformation, routing, message enhancement, security, and monitoring and management	+	+	+/–	+	+
2. Well-written documentation	+	+/–	+	+	+/–

Table 1.5 An assessment summary of the five open source ESBs related to the selection criteria. The notation used in this classification is simple: ++ is very good, + is good, +/– is average, – is not well supported, and — is not supported at all. *(continued)*

Selection criterion	Mule	ServiceMix	Open ESB	Synapse	PEtALS
3. Market visibility	++	+	+/–	+/–	+/–
4. Active development and support community	++	+	+/–	+	+
5. Flexible and easily extendable with custom logic	++	+	+/–	++	+
6. Support for a wide range of transport protocols and connectivity options	+	+	+/–	+/–	+
7. Integration with other open source projects	++	++	+/–	+	+
8. Productivity with IDE support	+	+	++	+/–	+

Notice that the classification is a snapshot in time of these open source ESBs and has some subjective parts in it. Based on the selection criteria we have used, Mule is the winner and ServiceMix is a good second. Because we think that both ESBs provide a unique implementation model and complement each other, we use both Mule and ServiceMix in the examples for this book. We've also seen that a number of open source ESBs are based on JBI. So with the choice of ServiceMix as a JBI implementation example and of Mule as a Java-based model example, we have representatives for both models that currently dominate the open source ESB products. We don't limit this book to only Mule and ServiceMix; we also give short examples of the other ESBs mentioned in section 1.3.2 when appropriate.

Up to this point, we've talked quite a bit about the theory of an ESB, the difference between open source and closed source ESBs, and our choice to use two open source ESBs in this book, Mule and ServiceMix. It's time to see some action, don't you think?

1.5 Hello world with Mule and ServiceMix

We wrap up this chapter with a simple example of both Mule and ServiceMix as a teaser for the examples we give in later chapters. This example isn't about a complex integration challenge; rather, it polls for new files in one directory and writes the polled file to another directory. We use this simple file polling and writing example to illustrate the basics of Mule and ServiceMix without having to go into detail about difficult and challenging steps of the integration solution. A graphical representation of the "hello world" style example is shown in figure 1.17.

In this section, we begin by downloading the Mule and ServiceMix distributions to implement the basic example shown in figure 1.17. You're welcome to join us when

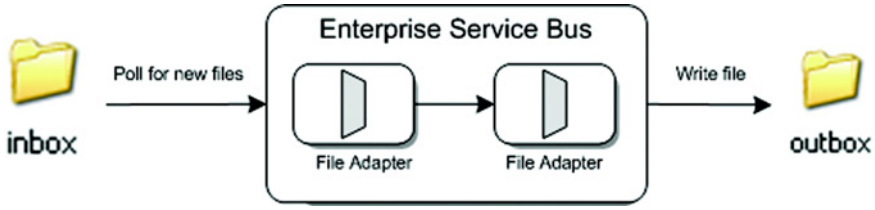

Figure 1.17 A hello world example with an ESB that polls for new files in the inbox directory and writes the polled files to a directory named outbox.

configuring the simple file integration solution, but you can also just look at the code listings. The idea is just to show you a bit of Mule and ServiceMix before we take a deep dive in chapter 2.

1.5.1 Taking a donkey ride with Mule

Our donkey ride starts with the setup of the book's environment by downloading the source code file from the Manning website (or you can use our own website, http://www.esbinaction.com).

> **Setting up the environment**
>
> Once you've downloaded the source code file, you'll notice that there is no actual source code in the downloaded file. The downloaded file contains an Ant build file that will download Mule, ServiceMix, and a lot of other tools and frameworks used in this book. The Ant script will also download the source code of the book and set up the Eclipse projects. Note that the script's execution takes quite some time, due to necessary downloads.
>
> Unpack the downloaded file in a directory like c:/osesbinaction for Windows or /osesbinaction for Unix. Now execute the `ant` command from the root directory of the unpacked file. Mule, ServiceMix, other tools and frameworks, and the source code of this book will be downloaded for you. For detailed information about Java and Ant versions and the resulting environment of this script, you should read chapter 3, but for this chapter you will be good to go.

In this book we use Mule version 2.0.2 (the current production release of Mule version 2). For an overview of the differences between Mule 1.*x* and 2.*x*, check out appendix B of this book.

INSTALLING MULE

In the previous step, by running the Ant script provided in the source code of this book, we already installed the Mule ESB by unpacking the Mule 2.0.2 distribution in the esb/mule-2.0.2 directory. The unpacked distribution should look like the directory structure shown in figure 1.18.

The directory structure shown in figure 1.18 isn't complex. The bin directory consists of scripts to start and stop Mule. For settings like log levels and internal configurations, the files in the conf directory should be explored. Then there are two directories to get more information on how Mule works: in docs, the Javadoc is available, and in the examples directory, you find examples of Mule in use. The lib directory contains the libraries that make up Mule and possibly custom JARs. The licenses directory holds the licenses of the libraries used by Mule, the logs directory contains logging information, and the src directory contains the Mule sources.

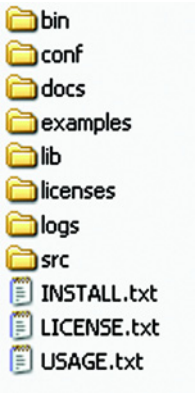

Figure 1.18 The directory structure of the unpacked Mule distribution

Now that we've discussed the contents of the Mule distribution, we've almost finished with the installation. We only have to set the `MULE_HOME` environment variable to the Mule installation directory before we can run Mule from the command line.

Testing the Mule installation

When you've configured the `MULE_HOME` environment variable, you can verify your installation by opening a command line or a console and going to the examples/echo directory within your Mule installation directory. Use the echo shell script (for Unix/Linux) or the batch file (for Windows) to start a simple example that will ask for input via System.in and will forward your input to System.out. When you run the echo script or batch file, you should see the following question once Mule starts up:

```
Please enter something:
```

When you enter the text of your choice and press Enter, the text is output to the console. This means you have succeeded in installing Mule!

Implementing a file polling example with Mule

Now that you've installed Mule, you're ready to implement the simple example shown in figure 1.17. We don't go into much detail about the Mule configuration at this point, because we intend to just give you a first look at Mule. In chapter 2 we talk in greater detail about the Mule architecture and how you can configure Mule. Mule is configured with an XML configuration file that is commonly named mule-config.xml. Without further delay, let's take a look at the mule-config.xml for the file polling example (listing 1.1).

Listing 1.1 The file poller Mule configuration: mule-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.0"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.0">

  <model name="FileExample">
    <service name="FileService">    <-- Describe service
```

```
      <inbound>
        <file:inbound-endpoint path="inbox"          <-- Poll directory
            fileAge="500" pollingFrequency="100"/>
      </inbound>
      <outbound>
        <outbound-pass-through-router>
          <file:outbound-endpoint path="outbox"      <-- Set file output directory
              outputPattern="output.xml"/>
        </outbound-pass-through-router>
      </outbound>
    </service>
  </model>
</mule>
```

The configuration of a Mule implementation starts with a service declaration. The root element of the mule-config.xml file is the `mule` element, which has the namespace declaration Spring (showing the out-of-the-box integration of Mule with the Spring Framework). We don't use any Spring functionality in this short example, however.

The service shown in listing 1.1 configures the name `FileService`. This simple Mule service accepts an incoming message from a configured address and passes it on to a configured outgoing address. Because we don't have to implement any logic for our file polling example, no Mule component is needed.

Within the `service` element we configure two child elements: an inbound router and an outbound router. The inbound router is configured to poll for new files in the inbox directory that is relative to the directory where Mule is started.

The outbound router is configured to write files to the outbox directory, also relative to the directory where Mule is started. The router that we use here is called `OutboundPassThroughRouter`. This Mule component routes the file to one configured endpoint address, in this case the outbox directory, without any filtering.

The remaining part of the configuration instructs Mule which filename to use for the file written to the outbox directory. With a single attribute on the endpoint configuration called `outputPattern`, a filename can be configured. For our hello world example, we use the static filename output.xml, but we could add dynamic parts to the filename such as the current date.

TESTING THE FILE POLLING EXAMPLE WITH MULE

We have now implemented our simple file example for Mule, so let's start Mule with the configuration shown in listing 1.1. In the workspace/workspace-mule/mule/resources/chapter1 folder, you find the file example implementation. Now you can start Mule with the configuration shown in listing 1.1 by issuing a console command from the chapter1 folder.

For Windows:

```
%MULE_HOME%\bin\mule.bat -config file-config.xml
```

For Unix:

```
$MULE_HOME/bin/mule -config file-config.xml
```

Mule now starts with the file polling example configuration. In the current directory (resources/chapter1) you find a file named test-file.xml. To test the file polling example, you can copy this file to the inbox directory within the resources/chapter1 directory in your Mule project. When you've copied the file, you see that an output.xml file containing the same XML content is written to the outbox directory. Congratulations—you have made the first step toward understanding the basics of the open source Mule ESB.

1.5.2 Taking a JBI dive with ServiceMix

The second open source ESB that will be discussed in this book is ServiceMix. We start this section by discussing the ServiceMix distribution before implementing the file polling example described in figure 1.17. The ServiceMix 3.2.1 distribution is already downloaded and installed with the execution of the Ant script as described in the previous Mule section.

Installing ServiceMix

The ServiceMix 3.2.1 distribution is available as zip and tar.gz, and a separate download is available as a web application archive. We use the binary distribution, whose contents should look like figure 1.19.

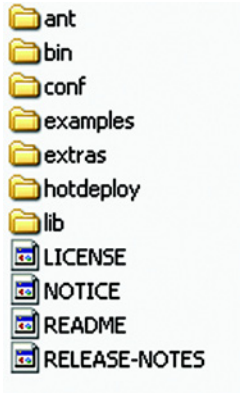

Figure 1.19
An overview of the directories within the ServiceMix distribution

Although we don't go into detail in this section about JBI and the functionality that ServiceMix implements, let's take a quick look at ServiceMix. The ant directory contains Ant scripts that you can use to, for example, install a JBI component or stop and start a JBI component. In other words, these Ant scripts are an easy way to administer the JBI container. The bin directory contains the start and stop scripts for ServiceMix.

The conf directory contains all kinds of configuration settings for ServiceMix, including the messaging platform and security parameters. The data directory is used by ServiceMix for the log files of the JBI container, the JBI components, ActiveMQ, and the ActiveMQ data files, and also serves as a repository for tracking installed JBI components and service assemblies. The data directory is created when you run ServiceMix for the first time. The examples directory contains a set of examples to demonstrate some of the ServiceMix features. The hotdeploy directory is used to install JBI components or JBI service assemblies, which we also talk about in chapter 2. The last directory, lib, contains the libraries necessary to run ServiceMix.

Testing the ServiceMix installation

Before we go further, you need to set an environment variable called `SERVICEMIX_HOME` that points to the ServiceMix installation root directory, esb/apache-servicemix-3.2.1. Next, go to ServiceMix's root directory in a console or command prompt and use the following command.

For Windows:

```
bin\servicemix.bat
```

For Unix:

```
bin/servicemix
```

The console or command prompt should now start ServiceMix, and the log messages should include this message:

```
ServiceMix JBI Container (ServiceMix) started
```

When this log message appears, you can consider the ServiceMix installation to be successful. Because ServiceMix lacks a simple example like Mule's echo example, we go ahead with the implementation of the file polling example as shown in figure 1.17. Before we can configure the file polling and writing logic within ServiceMix, we first have to deploy the file JBI component.

IMPLEMENTING A FILE POLLING EXAMPLE WITH SERVICEMIX

When you look in the hotdeploy directory where you installed ServiceMix, you see that it contains a JBI component named servicemix-file-3.2.1-installer.zip. Inside this installer file a JAR named servicemix-file-3.2.1.jar is available. Extract this library to the lib directory of your ServiceMix installation. "Wait," you might say, "didn't you just say that the hotdeploy directory is the place where JBI components can be deployed?" Well, you're absolutely right about that, but there's a small caveat to this.

To keep the example as simple as possible, we use a static configuration for our file polling implementation. This means that we don't make a JBI-compliant service assembly for this example, but we just use a XML configuration file. Making a JBI-compliant service assembly takes a few extra steps that we don't discuss here, but in later chapters we examine this process in detail. So for our static configuration file polling example, place the JAR file in the lib directory.

> **ServiceMix static configuration**
>
> ServiceMix supports the use of a static configuration in addition to the common service assembly deployment model. Note that the static configuration should not be used in projects, as this is not a JBI-compliant way to deploy a ServiceMix configuration. We only use the static configuration here to reduce the amount of knowledge that's necessary for our simple example.

All JBI components of ServiceMix also need a set of base classes available within the servicemix-shared-3.2.1-installer.zip file. All the JAR files inside this shared installer, including servicemix-common-3.2.1.jar and servicemix-shared-3.2.1.jar, should also be extracted to the lib directory. With the addition of these libraries, the file JBI component is ready to use.

To implement the file polling and writing logic as described in figure 1.17, we have to configure two components within ServiceMix: a file poller and a file sender. Like

Mule, ServiceMix needs an XML configuration for this. The big difference is that this XML configuration file follows the JBI specification. You can compare the differences between the Mule configuration and the ServiceMix configuration based on listing 1.2.

Listing 1.2 The file poller ServiceMix configuration: servicemix.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:sm="http://servicemix.apache.org/config/1.0"
    xmlns:file="http://servicemix.apache.org/file/1.0"
    xmlns:esb="http://esbinaction.com/helloworld">

  <bean id="jndi"
     class="org.apache.xbean.spring.jndi.SpringInitialContextFactory"
     factory-method="makeInitialContext" singleton="true" />

  <sm:container id="jbi" useMBeanServer="true"
      createMBeanServer="true">
    <sm:endpoints>
      <file:poller service="esb:poller"
                   endpoint="pollerEndpoint"
                   targetService="esb:sender"
                   file="file:inbox" />
      <file:sender service="esb:sender"
                   endpoint="senderEndpoint"
                   directory="file:outbox">
        <file:marshaler>
          <sm:defaultFileMarshaler>
            <sm:fileName>
              <bean class="org...ConstantExpression">
                <constructor-arg value="output.xml"/>
              </bean>
            </sm:fileName>
          </sm:defaultFileMarshaler>
        </file:marshaler>
      </file:sender>
    </sm:endpoints>
  </sm:container>
</beans>
```

Configure the ServiceMix container

Poll directory

Set file output directory

We don't cover the ServiceMix configuration in much detail, because we do this extensively in chapter 2. Note that JBI requires a basic structure that's pretty much the same for every JBI configuration. There's an element named `container` that belongs to a ServiceMix namespace. The `container` element represents the JBI container for which several endpoints can be configured. Every endpoint represents a component within your JBI implementation solution. In this case, we have two endpoints: a file poller and a file sender. Multiple endpoints can be configured, but for this example we have a file poller and a file sender endpoint. The `poller` element references a poller implementation within the file JBI component. Because we can set some additional attributes for the poller, the JBI container can be instructed to listen for new files in the inbox directory with the `file` attribute and knows that it must forward the file contents to the `esb:sender` service with the `targetService` attribute.

The `esb:sender` configured as the target service for the file poller must be able to write the file to the outbox directory with the static filename output.xml. To implement this, we have to configure a separate endpoint with the same semantics as the file poller except for some configuration details. For this file sender component, we can also configure the necessary attributes. The service name must be `esb:sender`, so that the JBI container can forward the file contents to this component. The `directory` attribute, which configures the destination of the file contents, is set to the outbox directory. Because we want the filename to be output.xml, we have to configure a `marshaler` with a `filename` element. We use a constant expression implementation class, because the filename is a static name.

TESTING THE FILE POLLING EXAMPLE WITH SERVICEMIX

Don't try to understand every detail of the configuration as we go into greater depth in upcoming chapters. With our configuration in place, we should now be able to test the file polling example. To execute the test, open a console or command prompt to the directory where you unpacked the source distribution of this book. There's a directory called workspace/workspace-servicemix/servicemix in this distribution with a resources/chapter1 directory inside it. The directory structure here is pretty much the same as for the Mule example, except for the servicemix directory (which holds the servicemix.xml configuration in listing 1.2). In the resources/chapter1 directory, execute the following command to start ServiceMix with the file polling implementation.

For Windows:

```
%SERVICEMIX_HOME%\bin\servicemix.bat servicemix\servicemix.xml
```

For Unix:

```
$SERVICEMIX_HOME/bin/servicemix servicemix/servicemix.xml
```

You can now copy test-file.xml, which is available in the resources/chapter1 directory, to the inbox directory. The file will be picked up by ServiceMix and the file contents will be written to the outbox directory in the file output.xml.

Well, you've done it! While reading just a few pages in this first chapter, you have seen a file polling implementation of Mule as well as ServiceMix.

1.6 Summary

With the need to increase the flexibility of the IT environments within organizations, applications have to be integrated and easily adaptable to new requirements. An ESB is an excellent product for integrating applications and helps you avoid having to write your own integration logic for every application over and over again. Using an ESB, you can overcome a number of the challenges seen in every integration project, such as location transparency, transport protocol conversion, message transformation, message routing, message enhancement, security, and monitoring and management.

Because this book is specifically about open source ESBs, we discussed some myths about open source ESBs in an integration market that is dominated by closed source ESBs. One of the main reasons why companies choose the common integration vendors

is the support and training. But we've shown that the open source ESBs discussed in section 1.3.2 all have support and training options provided by open source integration vendors such as MuleSource and IONA. The open source ESB projects have become competitive with the well-known integration products and have teams of full-time developers, a great QA, a release model, and an active community. And of course, the products are free, and the source and test code as well as the bug tracking status is open to everyone interested in the open source ESB product.

ENTERPRISE JAVA

OPEN SOURCE ESBs IN ACTION

Rademakers • Dirksen

Free ebook
SEE INSERT

An enterprise service bus—or ESB—is a piece of middleware that manages communication between enterprise services and applications. These plug into a software "bus" that distributes messages between them for a seamless integration. Commercial ESB software can be expensive, but outstanding open source alternatives are becoming popular.

Open Source ESBs in Action makes its subject accessible in a uniquely practical and example-rich way. It introduces two leading open source ESB implementations, Mule and ServiceMix, and uses them to illustrate how to apply ESBs in real life. It shows you how to implement well-known enterprise integration patterns such as transformation, routing, and message channels. With precisely crafted code the authors teach you structured techniques to solve important integration problems.

This book is written for Java developers and architects, and requires no previous exposure to ESBs.

What's Inside

- How to do pattern-based design
- How to use process engines
- How to implement Web Services, including security
- How to implement logic with Mule and SeviceMix
- How to monitor and manage your ESB

About the Authors

Tijs Rademakers is a Java architect with extensive practical experience using Mule, ServiceMix, jBPM, and WebSphere Process Server. **Jos Dirksen** is a software architect with a focus on security and quality. They speak regularly on ESB-related topics.

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

"A great book for any ESB practitioner."
—Rick Wagner
Acxiom Corporation

"A must-have ESB resource!"
—Craig Borysowich
Imagination Edge, Inc.

"The most comprehensive content that I've seen on open source ESBs."
—Rodney Biresch, Chariot Solutions

"The Bible for integration architects"
—Davide Piazza, Omnys s.r.l.

"... ample code samples and excellent descriptions."
—Jeff Davis, HireRight, Inc

"This book will take you to a new level."
—Christian Siegers
Stater International
Mortgage Services

ISBN-13: 978-1933988214
ISBN-10: 1933988215
54499
9 781933 988214

MANNING $44.99 / Can $44.99 [INCLUDING EBOOK]