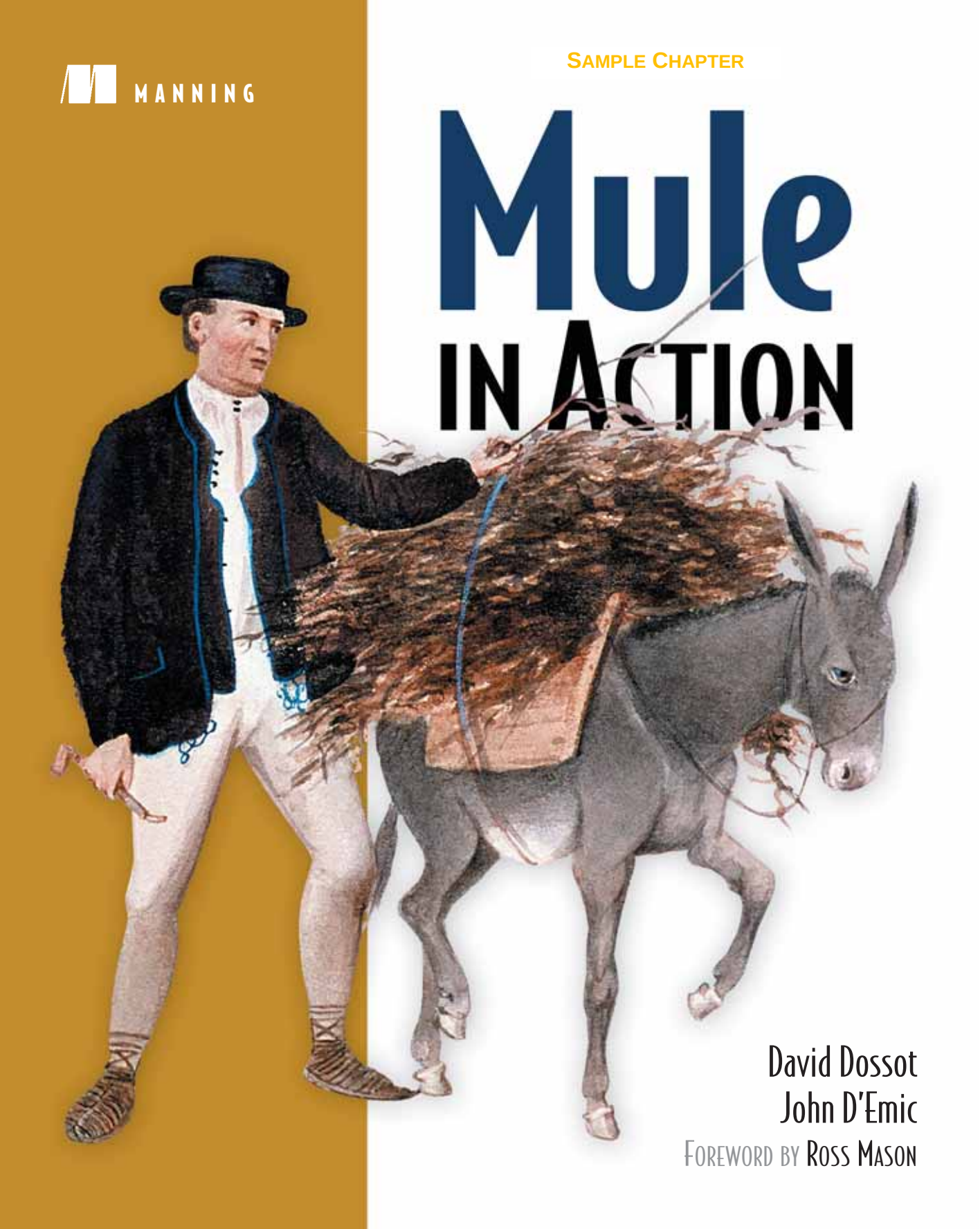
MANNING
SAMPLE CHAPTER
Mule
IN ACTION
David Dossot
John D'Emic
FOREWORD BY ROSS MASON

Mule in Action

David Dossot
John D'Emic

Chapter 1

brief contents

Part 1

Core Mule

Mule is a lightweight event-driven enterprise service bus (ESB) and an integration platform. As such, it more closely resembles a rich and diverse toolbox than a shrink-wrapped application. In the first chapter of this book, we'll introduce you to its history, overall architecture, and its terminology. Armed with this basic knowledge, you'll be able to further delve into the platform.

In chapter 2, you'll go through an extensive review of the principles involved in configuring Mule. You'll learn what it means and what it takes to configure Mule, including some tricks to get yourself organized.

Chapter 3 will be the first chapter dedicated to one of the major moving parts of Mule: the transports. You'll discover the main protocols that the platform supports in the context of actual configuration samples.

A second important feature of Mule is message routing. We'll explore the advanced capacities of Mule in this domain in chapter 4.

Message transformation is a crucial facet of enterprise service buses. Chapter 5 will allow you to learn how to take advantage of Mule transformers and how to create new ones.

Finally, we'll close this first part with chapter 6, which focuses on components, the place where message massage and business logic happens in Mule.

1 Discovering Mule

In this chapter

- Reviewing the challenges of enterprise integration
- Origins and history of the Mule project
- Architecture and terminology of Mule ESB

Integration happens.

All it takes is a simple requirement: connect to the inventory application, send this to the CRM, hook that to the accounting server. All of a sudden, your application, which was living a happy digital life in splendid isolation, has to connect to a system that's not only remote but also exotic. It speaks a different language, or a known language but uses a bizarre protocol, or it can only be spoken to at certain times during the night... in Asia. It goes up and down without notice. Soon, you start thinking in terms of messages, transformation, or adaptation. Welcome to the world of integration!

Nowadays, a standard corporate IT landscape has been shaped by years of software evolution and business mergers, which has usually led to a complex panorama of heterogeneous systems of all ages and natures. Strategic commercial decisions or critical reorganizations heavily rely on these systems working together as seamlessly

as possible. The need for application integration is thus a reality that all enterprise developers will have to deal with during the course of their career. As Michael Nygard puts it, "Real enterprises are always messier than the enterprise architecture would ever admit. New technologies never quite fully supplant old ones. A mishmash of integration technologies will be found, from flat-file transfer with batch processing to publish/subscribe messaging."[1]

Application integration encompasses all the difficulties that heterogeneity creates in the world of software, leading to diversity in all the aspects of system communications and interrelations:

- *Transport*—applications can accept input from a variety of means, from the file system to the network.
- *Data format*—speaking the right protocol is only part of the solution, as applications can use almost any form of representation for the data they exchange.
- *Invocation styles*—synchronous, asynchronous, or batch call semantics entail very different integration strategies.
- *Lifecycles*—applications of different origins that serve varied purposes tend to have disparate development, maintenance, and operational lifecycles.

This book is about Mule, a tool that can help you get over these difficulties. You'll learn how to make the most of it so your task of integrating applications will be focused on solving business problems and not on bothersome low-level concerns. Before getting down to the nitty-gritty of Mule, we'll first go through its origins and its general architecture. This is the main purpose of this first chapter. We'll also install Mule on your machine and make sure you can run the examples that accompany this book. By the end of this chapter, you'll have acquired the basic knowledge and lingo required to delve further into the platform.

Where does Mule come from? To answer this question, we have to travel just a few years back in time.

During the past decade, promising new standards such as SOAP[2] came to light. These new standards laid the foundations of interoperable applications by giving birth to the concept of web services. The normalization of the data model and service interface representations was a seminal event for the industry, as it began the search for platform-independent representation of all the characteristics of application communications. Web service technologies opened a lot of possibilities but also brought new challenges. One challenge is the proliferation of point-to-point communications across systems, as illustrated in figure 1.1. This proliferation often leads to a spaghetti plate integration model, with many-to-many relationships between the different applications. Though the interoperability problem was solved, maintenance was complicated and no governance existed.

---

[1] *Release It!*, Michael T. Nygard, Pragmatic Bookshelf, March 2007

[2] SOAP, originally defined as *Simple Object Access Protocol*, is a protocol specification for exchanging structured information in the implementation of web services in computer networks [Wikipedia].

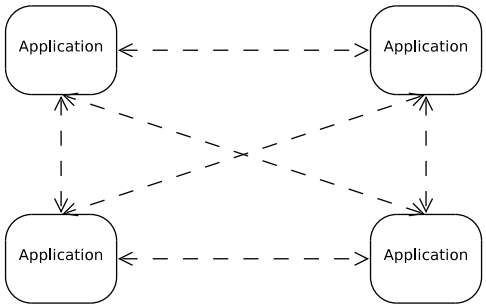

Figure 1.1 Point-to-point integration approach: everyone must speak the language of everyone.

When the industry realized that, despite the standardization of protocols, the challenges of integration were only growing, a new discipline came to life. Its goal was to foster the creation of tools, platforms, and practices to enable and facilitate application integration. The EAI (enterprise application integration) discipline was born, for better or worse. Indeed, as the industry was learning its way through these new concepts, many errors were made, leading to the impressive failure rate of 70 percent for all EAI projects in 2003.[3] While solving integration challenges, EAI brought a new wealth of complexities, including the need for heavy processes and extensive governance to attempt to manage the impact inherent in the invasiveness of most integration patterns. Figure 1.2 shows a typical broker-centric integration pattern.

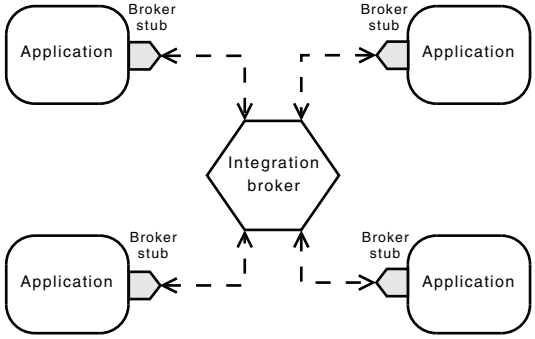

Figure 1.2 Application-invasive integration broker approach: everyone must speak the language of the broker.

Something better was needed. And it needed to be simpler and cleaner.

1.1 ESB, the EAI workhorse

Enterprise architects started to dream of a platform that would allow them to overcome the difficulties of application integration. This platform would foster better practices by encouraging loosely coupled integration and at the same time discouraging many-to-many connectivity. Moreover, this platform wouldn't require any change from existing applications. Like C3P0, it would be able to speak all languages and

[3] http://www.ebizq.net/topics/int_sbp/features/3463.html

make dissembling systems talk with each other through its capacity to translate from one form of communication to another. Acting like the bus at the core of computer architecture, this middleware would become the backbone of enterprise messaging, if not, according to some, the pillar of the service-oriented architecture (SOA) redesign that enterprises were going through. As illustrated in figure 1.3, the bus would play a central role without invading the privacy of all the applications around it.

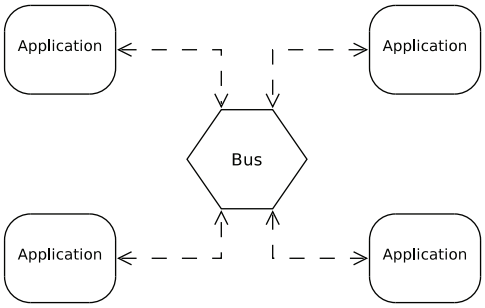

Figure 1.3 Enterprise service bus approach: the bus speaks the language of everyone.

To solve the issues caused by the centralized nature of previous integration platforms, this bus was designed to be distributable in several brokers deployed at strategic locations across a corporate network, as shown in figure 1.4. On top of solving the aforementioned integration issues, this bus would offer extra value-added services, like intelligent routing of messages between systems, centralized security enforcement, or quality of service measurement. The concepts defining the enterprise service bus (ESB) were established and vendors started to build their solutions, providing enterprise developers with a new breed of integration tooling.

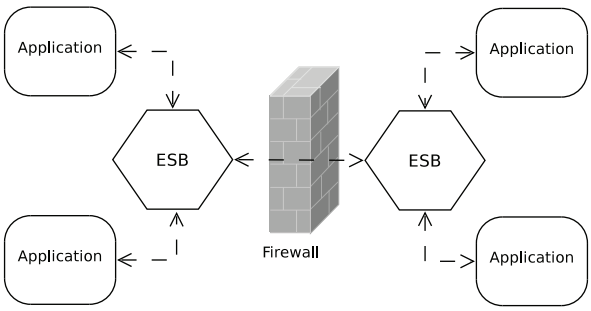

Figure 1.4 ESB nodes distributed at strategic network locations

This book widely assumes that you're knowledgeable of these concepts. Should you need more in-depth coverage of the subject, a lot of literature is available about ESBs and their use cases, benefits, and disadvantages. There's also a lively controversy about the role such tools should or shouldn't play in SOA. As always, there's no silver bullet: an ESB should be envisioned and deployed in a pragmatic and progressive manner. In his originative book *Enterprise Service Bus*, industry pundit David Chappell summarized

this necessity by saying, "An ESB provides a highly distributed approach to integration, with unique capabilities that allow individual departments or business units to build out their integration projects in incremental, digestible chunks."[4]

Vendors were prompt to deliver their first ESB solutions, mostly because they reused parts of their existing application servers and message-oriented middlewares. Hence, the solutions they came up with, though robust, feature-rich, and thoroughly documented, were often hard to deploy and seldom easy to employ. Developers had to jump through hoops before having their first example running, let alone the embryo of the integration solution they needed.

1.2 The Mule project

The Mule project was started with the motivation to make things simpler for programmers and give them a chance to hit the ground running. Pragmatism aside, another driver for the project was the need to build a lightweight and modular solution that could scale from an application-level messaging framework to an enterprise-wide highly distributable object broker.

Mule's core was designed as an event-driven framework combined with a unified representation of messages, expandable with pluggable modules. These modules would provide support for a wide range of transports or add extra features, such as distributed transactions, security, or management. Mule was also designed as a programmatic framework offering programmers the means to graft additional behavior such as specific message processing or custom data transformation.

This orientation toward software developers helped Mule to remain focused on its core goals and carefully avoid entering the philosophical debate about the role of an ESB in SOA. Moreover, Mule was conceived as an open source project, forcing it to stick to its mission to deliver a down-to-earth integration framework and not to digress to less practical or broader concerns. Finally, the strategic decision to develop Mule in the open allowed contributors to provide patches and improvements, turning this bus into a solid and proven platform.

What's in the name?

"After working on a couple of bespoke ESB systems, I found that there was a lot of infrastructure work to be done before you can really start thinking about implementing any logic. I regard this infrastructure work as "donkey work" as it needs doing for every project. I preferred Mule over Donkey and Ass just didn't seem right ;-). A Mule is also commonly referred to as a carrier of load, moving it from one place to another. The load we specialize in moving is your enterprise information."

—Ross Mason, CTO & Co-Founder, MuleSource Inc.

[4] *Enterprise Service Bus*, David A. Chappell, O'Reilly, June 2004

1.2.1 History

Mule publicly started as a SourceForge project in April 2003, and stayed there for 2 years, until it reached its first stable release and moved to CodeHaus. The architectural principles that would define the platform design for the coming years were laid during this maturation phase. At the core of this design was the Universal Message Object (UMO) API, a comprehensive yet compact abstraction of all the moving parts of the ESB. The key idea behind the UMO API was to unify the logic and model representations while keeping them isolated from the underlying transports. This paved the way for building all the required features of the ESB: message routing, data transformation, and protocol adaptation.

Version 1.0, which was released in April 2005, already contained numerous transports. This made it immediately appealing for many enterprise developers. Most of the following releases were focused on debugging, adding new features, and transports. Improving performance was also a subject of focus. From version 1.3, Mule started to be built with Maven (see section 12.1) and to be distributed as artifacts whose dependencies were strictly managed. This improvement simplified the usage of the platform, both at development and deployment times, as the total number of libraries a typical integration project will depend on is usually a few dozen.

A key milestone for the Mule project occurred when MuleSource got incorporated in 2006, in order "to help support and enable the rapidly growing community using Mule in mission-critical enterprise applications."

When Mule transitioned from version 1 to version 2, it went through a major overhaul that was mainly driven by the adoption of Spring 2 as its configuration and wiring framework. Until this major revision, a Mule instance was configured by an ad hoc parsing engine that was fully responsible for the different components' instantiation and lifecycle. Spring was an optional bean container, from which Mule was able to fetch objects. This was useful, for example, when the default configuration mechanism wasn't powerful enough to instantiate and configure a specific object.

It was also possible to fully configure Mule 1 from Spring, but this practice never really caught on, mainly because of the verbosity and the lack of expressiveness of the required XML configuration. The introduction of XML Schema-based configuration in Spring 2 solved both these issues. Specialized XML elements and attributes replaced generic elements and class references, leading to a more concise and communicative configuration syntax. The strong typing of attributes defined by XML Schema also provided type safety to the multiple configuration parameters. Moreover, unless one needs to configure custom implementations, there are no more class names in the configuration file.

This new configuration format is actually the most visible improvement of Mule 2. But there are other changes that happened, most only visible to advanced users or transport writers, who are both dependent of the internals of Mule. Though the original concepts remained, the previously ubiquitous UMO API has given way to a new set of restructured APIs. Thanks to the move to Spring, the main manager of Mule, which

was a monolithic singleton, has been replaced by a lighter context object that can be injected on demand. This new architecture puts Mule in position to leverage the wide range of runtime platforms supported by Spring, including OSGI.

1.2.2 Competition

All major JEE vendors (BEA, IBM, Oracle, Sun) have an ESB in their catalog. It's unremarkably based on their middleware technologies and is usually at the core of a much broader SOA product suite. There are also some commercial ESBs that have been built by vendors not in the field of JEE application servers, like the ones from Progress Software, IONA Technologies, and Software AG.

NOTE Commercial ESBs mainly distinguish themselves from Mule in the following aspects:

- Prescriptive deployment model, whereas Mule supports a wide variety of deployment strategies (presented in chapter 7).
- Prescriptive SOA methodology, whereas Mule can embrace the architectural style and SOA practices in place where it's deployed.
- Mainly focused on higher-level concerns, whereas Mule deals extensively with all the details of integration.
- Strict full-stack web service orientation, whereas Mule's capacities as an integration framework open it to all sorts of other protocols.
- Comprehensive documentation, a subject on which MuleSource has made huge progress recently.

Mule is, of course, not the only available open source ESB. To name a few, major OSS actors such as JBoss, Apache, and ObjectWeb provide their own solutions. SpringSource also provides an integration framework built on their dependency injection container. While most of these products use proprietary architecture and configuration, the integration products from the Apache Software Foundation are notably standard-focused: ServiceMix is based on the Java Business Integration (JBI) specification, Tuscany follows the standards defined by the Oasis Open Composite Services Architecture (SCA and SDO), and Synapse has an extensive support of WS-* standards.

It can be daunting to have to pick up a particular solution while considering all the available options. For some help with the selection process and a broader view of ESBs in general and open source ones in particular, you can turn to another book from Manning Publications Co. named *Open Source ESBs in Action* (Rademakers and Dirksen).

One way to decide whether a tool is good for you is to get familiar with it and see if you can wrap your mind around its concepts easily. So read on, as from now until the end of this chapter we'll proceed with a first review of Mule's different moving parts and how they relate to each other. We'll introduce Mule's core concepts only briefly in this first approach: all the concepts you will learn about in section 1.3 will be covered in great detail in subsequent chapters.

1.3 Mule's core concepts

Now that you're ready to begin your journey exploring Mule, let's discover the core concepts that sit at the heart of this ESB. Any domain has its own lingo. EAI in general and Mule in particular have one too. If you're familiar with the book *Enterprise Integration Patterns,*[5] you'll quickly realize that a significant part of Mule's terminology has been derived from this work. In fact, the core architecture of the platform itself has been influenced by the concepts laid by Gregor Hohpe and Bobby Woolf in these patterns. If you aren't familiar with these patterns, we recommend that you spend some time getting acquainted with them. The table of contents of the companion web site for the book (`http://www.enterpriseintegrationpatterns.com/toc.html`) can get you started with the terminology we're about to use.

To guide us in this discovery of Mule, we'll use a concrete integration case as a context of reference. Whereas we'll describe each moving part of the ESB, we'll also present what role it would come to play in this context of reference.

Let's suppose we have a publication application that accepts only XSL-FO[6] as its input. Moreover, this input should be sent as text messages to a JMS queue. The authors use an editing tool that generates DocBook[7] documents. This tool is only capable of sending these documents to an HTTP destination. We're now faced with the need to translate DocBook to XSL-FO and to adapt the HTTP protocol to JMS. To solve this, we'll leverage Mule to both translate the message format from one form to another and to take care of the protocol mismatch. As illustrated in figure 1.5, we'll deploy a Mule service between the authoring tool and the publication application. The ESB will act as a mediator between them.

What's happening inside the big gray box that represents a Mule instance? Let's discover how Mule's internals are coming into play to make this integration happen.

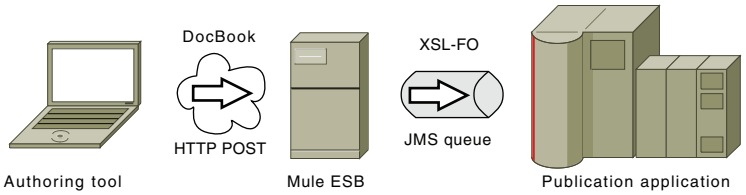

Figure 1.5 Mule ESB acting as a mediator between an authoring tool and a publication application

1.3.1 Model

The first logical layer we discover is the model layer. A Mule model represents the runtime environment that hosts services. It defines the behavior of Mule when processing

---

[5] *Enterprise Integration Patterns*, Gregor Hohpe, Bobby Woolf, Addison-Wesley Professional, October 2003

[6] XSL Formatting Objects, or XSL-FO, is a markup language for XML document formatting.

[7] DocBook is a semantic markup language for technical documentation.

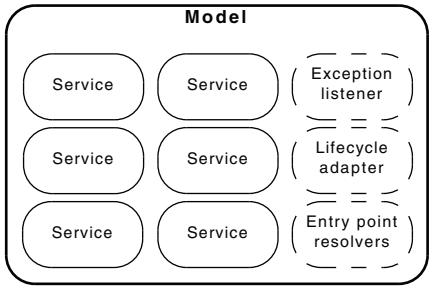

Figure 1.6 A Mule model is the runtime environment into which services are hosted.

requests handled by these services (see section 1.3.6). As represented in figure 1.6, the model provides services with supporting features, such as exception strategies. It also provides services with default values that simplify their configuration.

In our example, the authoring tool doesn't expect any message back from Mule (except the acknowledgement of its posting). As far as it's concerned, it'll send messages to Mule in a "fire and forget" mode. Mule itself will send messages to the JMS queue without expecting any reply from the publication application. Consequently, the model that the Mule instance would host would establish a runtime environment optimized for asynchronous processing of messages, as no synchronous reply is expected anywhere in the overall message processing chain.

1.3.2 Service

A Mule service is composed of all the Mule entities involved in processing particular requests in predefined manners, as shown in figure 1.7. To come to life, a service is defined by a specific configuration. This configuration determines the different elements, from the different layers of responsibility, that will be mobilized to process the requests that it'll be open to receive. Depending on the type of input channel it uses, a service may or may not be publicly accessible outside of the ESB.

For the time it gets handled by a service, a request is associated with a session object. As its name suggests, this object carries all the necessary context for the processing of a message while it transits through the service.

In our example, the service would need to act as a bridge between the incoming HTTP messages and the outgoing JMS messages. Interestingly, the default behavior of

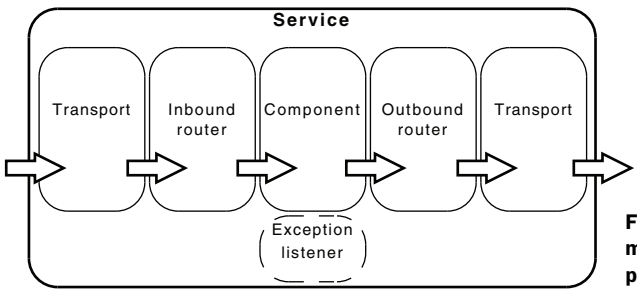

Figure 1.7 A Mule service mobilizes different moving parts to process requests.

a Mule service is to bridge its inbound router to its outbound one. Hence, you'd rely on this default behavior for the publication application.

1.3.3 Transports

As illustrated by figure 1.8, the transport layer is in charge of receiving or sending messages. This is why it's involved with both inbound and outbound communications.

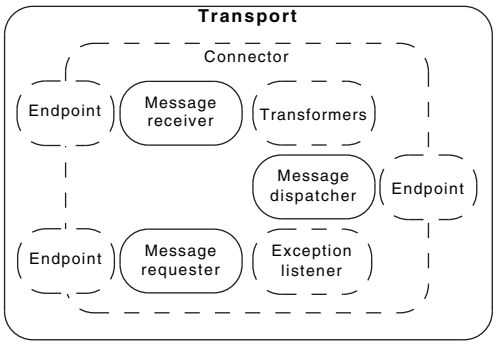

Figure 1.8 A Mule transport provides all the ESB elements required for receiving, sending, and transforming messages for a particular protocol.

A transport manifests itself in the configuration by the following elements: connectors, endpoints and transformers.

CONNECTOR

A connector is in charge of controlling the usage of a particular protocol. It's configured with parameters that are specific to this protocol and holds any state that can be shared with the underlying entities in charge of the actual communications. For example, a JMS connector is configured with a `Connection`, which is shared by the different entities in charge of the actual communication.

These communicating entities will differ depending on whether the connector is used for listening/polling, reading from, or writing to a particular destination: they would respectively be message receivers, message requesters, and message dispatchers. Though a connector isn't part of a service, it contributes these communication parts to it. Consequently, a service is dependent on one or more connectors for actually receiving or sending messages.

In our example, we'd need an HTTP connector and a JMS connector. The HTTP connector would provide the message receiving infrastructure in the form of a dedicated listener on a particular port. The JMS connector would provide the capacity to connect to the target JMS provider (through its specific connection factory) and to handle the dispatch of messages to the desired queue.

ENDPOINT

An endpoint represents the specific usage of a protocol, whether it's for listening/polling, reading from (*requesting* in Mule's terminology), or writing to a particular target destination. It hence controls what underlying entities will be used with the connector they depend on. The target destination itself is defined as a URI. Depending on

the connector, the URI will bear a different meaning; for example, it can represent a URL or a JMS destination.

Inbound and outbound endpoints exist in the context of a particular service and represent the expected entry and exit points for messages, respectively. These endpoints are defined in the inbound and outbound routers. It's also possible to define an inbound endpoint in a response router. In that case, the inbound endpoint acts as a response endpoint where asynchronous replies will be consolidated before the service returns its own response. Global endpoints can be considered abstract entities that get reified only when referenced in the context of a service: as such, they're a convenient way to share common configuration attributes.

In our example, we'd need an HTTP inbound endpoint and a JMS outbound endpoint. The HTTP endpoint would specify the port to open to the incoming HTTP traffic, whereas the JMS one would configure the message publisher for the desired queue.

TRANSFORMER

As its name suggests, a transformer takes care of translating the content of a message from one form to another. Mule ships with a wealth of general transformers that perform simple operations, such as `byte-array-to-string-transformer`, which builds a string out of an array of bytes using the relevant encoding. On top of that, each transport contributes its own set of specific transports, for example `object-to-jms-message-transformer`, which builds a `javax.jms.Message` out of any content. It's possible to chain transformers to cumulate their effects, as shown in figure 1.9.

Figure 1.9 Transformers can be chained to cumulate their effects.

Transformers can kick in at different stages while a message transits through a service. Essentially, inbound transformers come into play when a message enters a service, outbound transformers when it leaves, and response transformers when a message is returned to the initial caller of the service. Transformers are configured in different ways: globally or locally on endpoints by the user, and implicitly on connectors by the transport itself (it's possible to override these implicit transformers, as you'll see in section 3.3.1).

NOTE A transport also defines one message adapter. A message adapter is responsible for extracting all the information available in a particular request (data, meta information, attachments, and so on) and storing them in transport-agnostic fashion in a Mule message. See the note in section 5.1 for more on this.

In our example, we'd need a DocBook to XSL-FO transformer, which will be a generic `xslt-transformer` configured with a specific XSL-T stylesheet. This transformer could

be either inbound or outbound, depending of what message format we want to carry over inside the ESB. We'd also need a transformer to convert the XML message into JMS on the way out to the publication application. This transformer would in fact be applied implicitly by the JMS transport itself, without any particular configuration.

1.3.4 Routers

Routers play a crucial role in controlling the trajectory a message will follow when it transits in Mule. They're the gatekeepers of the endpoints of a service. In fact, they act like railroad switches, taking care of keeping messages on the right succession of tracks so they can reach their intended destinations. Some of these routers play a simple role and don't pay much attention to the messages that transit through them. Others are more advanced: depending on certain characteristics of a message, they can decide to switch it to another track. Certain routers go even further and act like the big classification yards you can see close to major train freight stations: they can split, sort, or regroup messages based on certain conditions.

These conditions are mainly enforced by special entities called *filters*. Filters are a powerful complement to the routers. Filters provide the brains routers need to make smart decisions about what to do with messages in transit. Like figure 1.10 suggests, they can base their filtering decisions on all the characteristics of a message and its properties. Some filters go as far as deeply analyzing the content of a message for a particular value on which their outcome will be based. And, of course, you can roll your own filters.

The location of a router in a service determines its nature (inbound, outbound, or response) and the possible roles it could decide to play (pass-through, aggregator, and so on). Inbound routers are traversed before a message reaches a component, while outbound ones are reached after a message leaves a component. Response routers (aka async-reply routers) take care of consolidating asynchronous replies from one or more endpoint as a unique service response to the inbound request. Mastering the art of message routing is key to taming this Mule. Throughout this book, you'll have numerous occasions to familiarize yourself with routers and filters.

Coming back to our example, where no particular routing is necessary, the inbound and outbound routers would be the simplest ones possible. They would be pass-throughs, giving way to any message that decides to transit through them.

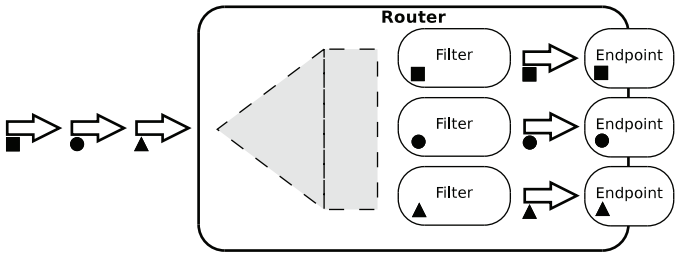

Figure 1.10 A router can leverage filters to dispatch messages based on their properties.

1.3.5 Components

Components are the centerpiece of Mule's services. Each service is organized with a component at its core and the inbound and outbound routers around it. Components are used to implement a specific behavior in a service. This behavior can be as simple as logging messages or can go as far as invoking other services. Components can also have no behavior at all; in that case they're *pass-through* and make the service act as a bridge between its inbound and outbound routers.

In essence, a component receives, processes, and returns messages. It's an object from which one method will be invoked when a message reaches it.

In our example, we don't need any specific behavior from the service component. As we said in section 1.3.2, we simply need to rely on the bridge that Mule establishes by default in a service. Hence, we won't need to define any explicit component: the implicit bridge that Mule will instantiate will solve our problem efficiently.

Figure 1.11 summarizes the different Mule elements that you've learned about so far and that we need to use for our example. We'll further detail this example in chapter 5.

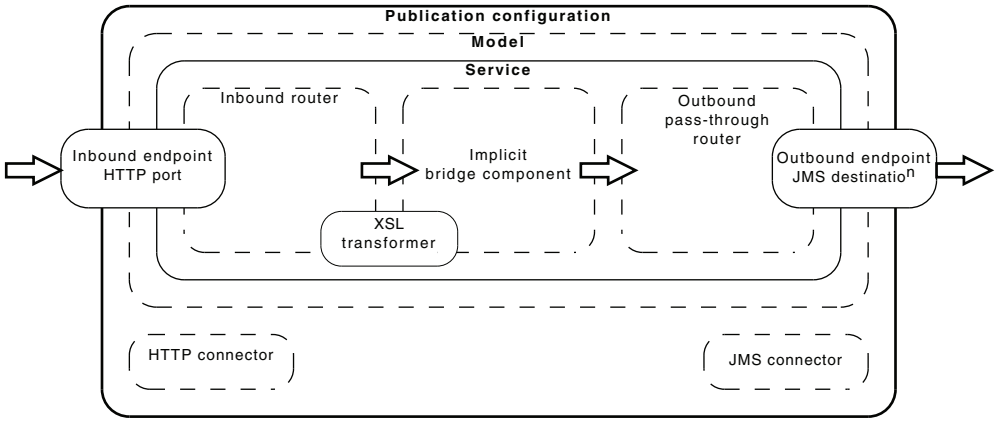

Figure 1.11 The different Mule moving parts involved in the Publication application configuration

So far, we've focused on discovering Mule's inners from a static point of view. Let's now see what's happening when a message flows through these different layers.

1.3.6 Request processing

While following the message flow in figure 1.5, you might have noticed that the arrows between the external applications and Mule are one-way only. This assumes that, notwithstanding any low-level transport acknowledgment mechanism that might exist, the caller on the left side of the arrow isn't interested in the immediate, or synchronous, response to its request. This doesn't preclude that it might well be interested in an ulterior, or asynchronous, response that would be coming via another channel.

MESSAGES AND EVENTS

To understand how Mule handles these different situations, it's important to grasp the notion of events. You might remember that Mule was introduced as an event-driven platform. Indeed, the default model (see section 1.3.1) used by Mule for processing requests is based on the work of Matt Welsh on the definition of a staged event-driven architecture (SEDA). The following are Welsh's core concepts of SEDA:[8]

> "SEDA is an acronym for staged event-driven architecture, and decomposes a complex, event-driven application into a set of stages connected by queues. This design avoids the high overhead associated with thread-based concurrency models, and decouples event and thread scheduling from application logic. By performing admission control on each event queue, the service can be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. SEDA employs dynamic control to automatically tune runtime parameters (such as the scheduling parameters of each stage), as well as to manage load, for example, by performing adaptive load shedding. Decomposing services into a set of stages also enables modularity and code reuse, as well as the development of debugging tools for complex event-driven applications."

When a message transits in Mule, it is in fact an event that's moved around. This event carries not only the actual content of the message but also the context in which this message is processed (see section 13.3). This event context is composed of references to different objects, including security credentials, if any, the session in which this request is processed, and the global Mule context, through which all the internals of the ESB are accessible (see section 13.2).

A Mule message is composed of different parts:

- The payload, which is the main data content carried by the message
- The properties, which contain the meta information much like the header of a SOAP envelope or the properties of a JMS message
- Optionally, multiple named attachments, to support the notion of multipart messages
- Optionally, an exception payload, which holds any error that occurred during the processing of the event

In our example, the initial payload of the Mule message will be the data the authoring tool would've HTTP posted, and the properties would contain any headers the client would've sent (most probably the usual HTTP suspects: `Content-Type` and `Content-Length`). Throughout the transformation chain, the payload would be altered from DocBook to XSL-FO, and finally, to become a JMS text message just before leaving Mule.

STANDARD PROCESSING

By default, an event gets routed in a service from an inbound endpoint to the component entry point, via any configured or implicit transformers. After that, the response

---

[8] SEDA, Matt Welsh, http://www.eecs.harvard.edu/~mdw/proj/seda

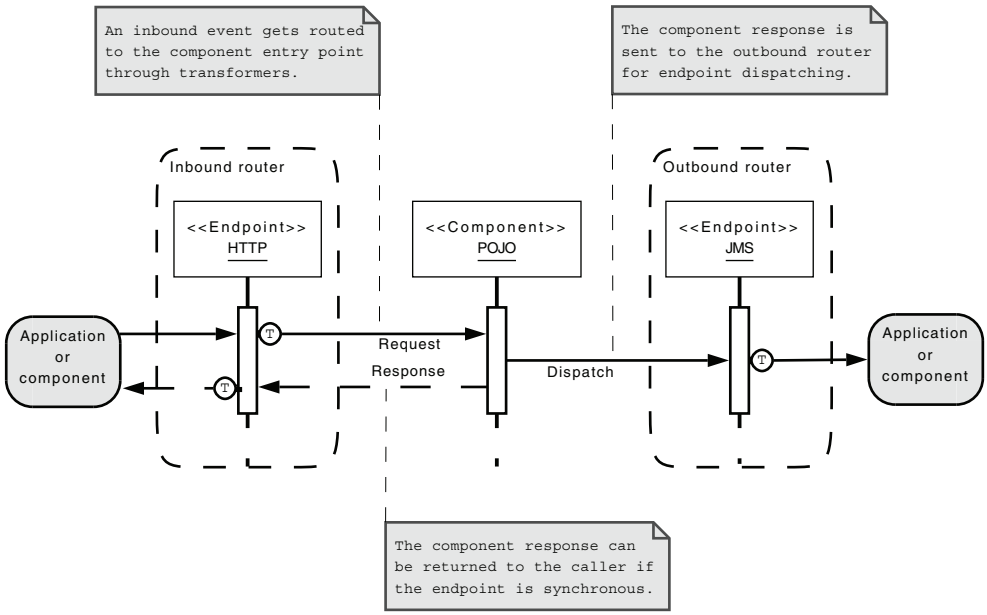

Figure 1.12 Standard event processing in a Mule service

of the component gets routed to an outbound endpoint, potentially via transformers as well. Figure 1.12 shows the standard processing of an event in a Mule service.
This default behavior can be altered in different ways:

- *No outbound router is defined*—the event won't travel further than the component.
- *Altered routing*—the component can programmatically decide not to let the event reach the configured outbound router and reroute it to another service (or discard it).
- *Nested routing*—the component can send the event to another service, wait for its response, and then let normal routing happen.

The way the response to the caller of the service will be routed can also vary greatly:

- *No response*—if the inbound endpoint is asynchronous, the caller of the service won't receive an immediate response when its request is accepted by Mule. In that scenario, the response of the component won't be returned to it.
- *Synchronous response*—to the contrary, if the inbound endpoint is configured to be synchronous, the response of the component will be immediately returned to the caller as a response to its request. If a response transformer is configured, it'll be used at this stage.
- *Asynchronous response*—in that case, the inbound endpoint is synchronous but the response that it'll return to the caller isn't the one coming out from the

component but one that'll be received on an asynchronous channel after the message has been sent through the outbound router. Response transformers would also apply in that case.

In our example of DocBook to XSL-FO bridging, we'd use a fully asynchronous configuration that would lead to a request processing schema similar to the one shown in figure 1.12. No response, except the standard HTTP acknowledgment, would be sent to the authoring tool in response to its request.

We're done with our quick tour of Mule. You've now discovered its general architecture, underlying principles, and principal moving parts. You should realize that the fundamental knowledge you need to grasp is limited. We believe that this is a key characteristic of Mule and a reason for its success.

1.4 Mule on your machine

Before going further, we need to make sure that you have Mule correctly installed on your machine. Bear in mind that, depending on the way you'll deploy Mule, you might not need to install it. For example, if you decide to embed Mule in a web application, you'll have nothing to install, as you'll deploy all the libraries within your WAR file. We'll cover the different deployment options in chapter 7.

Nevertheless, we'll install a standalone Mule server on your machine. This'll allow you to run the examples that can be downloaded from this book's companion web site. It'll also allow you to easily run your own experiments: as we'll introduce different Mule features, you'll certainly feel the urge to run a quick test to ensure you grok what you've just learned.

Here's an outline of the installation steps:[9]

- Download the latest stable full distribution of Mule Community Edition from MuleSource's web site (http://mulesource.org/display/MULE/Download). We need the full distribution because we'll install a standalone Mule server.
- Decompress the distribution archive in a directory of your choice.
- Add an environment variable named `MULE_HOME` that points to this directory. In Windows, this is done with a system property.
- Add the `MULE_HOME/bin` directory to your system's `PATH`.

That should be it.

Download the source code from the companion web site of this book (`http://muleinaction.com`) and decompress it in another folder of your choice. Ensure that your `MULE_HOME` points to a Mule version that matches the example source code requirements. Most examples will require Maven 2 to run: if you don't have this build tool already installed, it's a good time to do so. If you have several Mule instances deployed, be sure to check that `MULE_HOME` points to the version that matches the one used for the examples.

[9] The detailed installation instructions are available at http://mulesource.org/display/MULE2INTRO/Installing+Mule.

NOTE Though not necessary to execute some standalone examples, you can run `mvn clean install` in the root directory of the code samples. After a few minutes, you should get a successful build.

The first example we run isn't intended to demonstrate any feature of Mule; its goal is to ensure that both Mule standalone server and the code samples are correctly installed. This example is located in the chapter01/welcome directory. Start it by using the batch file appropriate for your operating system. You should see the following output in the console:[10]

```
Running in console (foreground) mode by default, use Ctrl-C to exit...
Running Mule...
--> Wrapper Started as Console
Launching a JVM...
Starting the Mule Server...
Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
 Copyright 1999-2006 Tanuki Software, Inc. All Rights Reserved.

  _    _      _                                _          ___  ___       _      _
 | |  | |    | |                              | |         |  \/  |      | |    | |
 | |  | | ___| | ___ ___  _ __ ___   ___  | |_ ___   | .  . |_   _| | ___| |
 | |/\| |/ _ \ |/ __/ _ \| '_ ` _ \ / _ \ | __/ _ \  | |\/| | | | | |/ _ \ |
 \  /\  /  __/ | (_| (_) | | | | | |  __/ | || (_) | | |  | | |_| | |  __/_|
  \/  \/ \___|_|\___\___/|_| |_| |_|\___|  \__\___/  \_|  |_/\__,_|_|\___(_)
```

When you've seen enough of this splendid ASCII art, press `control+C` in the console screen. Mule should stop with the following output:

```
INT trapped. Shutting down.
************************************************************************
* The server is shutting down due to normal shutdown request           *
* Server started: 02/06/08 4:00 PM                                     *
* Server shutdown: 02/06/08 4:00 PM                                    *
************************************************************************
<-- Wrapper Stopped
```

The sound of Mule's hooves has just started to echo in your machine! You'll get more of that in the coming chapters, but for now, this is a pretty good start.

1.5 Summary

In this chapter, you've discovered that Mule is a no-nonsense platform that has the potential to become the workhorse of your enterprise integration projects. The following, which is an excerpt from Mule's Architecture Guide, should now sound familiar to you. Here are all the concepts that you've just learned: Mule is an event-based architecture; actions within a Mule network are triggered by events occurring either in Mule or in external systems. Events always contain some sort of data—the payload—which will be used and/or manipulated by one or more components and a set of properties that are associated with the processing of the event. These properties are

[10] The first time you run Mule standalone, you'll be prompted to read and approve its license agreement.

arbitrary and can be set at any time from when the event is created. The data in the event can be accessed in its original state or in its transformed state. The event will use the transformer associated with the endpoint that received the event to transform its payload into a format that the receiving component understands.

The next chapter will give you the fundamental knowledge you need to configure Mule and understand the numerous configuration samples that will follow in subsequent chapters.

ENTERPRISE JAVA

Mule IN ACTION

David Dossot • John D'Emic
Foreword by Ross Mason, Creator of Mule

Mule is a widely used open source enterprise service bus. It is standards based, provides easy integration with Spring and JBoss, and fully supports the enterprise messaging patterns collected by Hohpe and Woolf. You can readily customize Mule without writing a lot of new code.

Mule in Action covers Mule fundamentals and best practices. It is a comprehensive tutorial that starts with a quick ESB overview and then gets Mule to work. It dives into core concepts like sending, receiving, routing, and transforming data. Next, it gives you a close look at Mule's standard components and how to roll out custom ones. You'll pick up techniques for testing, performance tuning, BPM orchestration, and even a touch of Groovy scripting.

Written for developers, architects, and IT managers, the book requires familiarity with Java but no previous exposure to Mule or other ESBs.

What's Inside

- Mule deployment, logging, monitoring
- Common transports, routers, and transformers
- Security, routing, orchestration, and transactions

Both authors are Java EE architects. **David Dossot** is the project "despot" of the JCR Transport and has worked with Mule since 2005. **John D'Emic** is Chief Integration Architect at OpSource Inc., where he has used Mule since 2006.

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

Free ebook
SEE INSERT

"A deep, anatomical view of Mule ESB."
—Ara Abrahamian, Architect, Coauthor of *Java Open Source Programming*

"A top-to-bottom example-driven guide I haven't found anywhere else."
—Ben Hall, Technical Lead, IBBS

"Outstanding examples show how to use Mule."
—Doug Warren, Software Architect, Java Web Services

"These guys know what they are talking about!"
—Fabrice Dewasmes, Java & Open Source Department Director, Pragma Consult

"Works better than a carrot to get the Mule going. Useful even for experts."
—Jeroen Benckhuijsen, Software Architect, Atos Origin

ISBN 13: 978-1-933988-96-2
ISBN 10: 1-933988-96-7
54499
9 781933 988962

MANNING $44.99 / Can $56.99 [INCLUDING eBOOK]