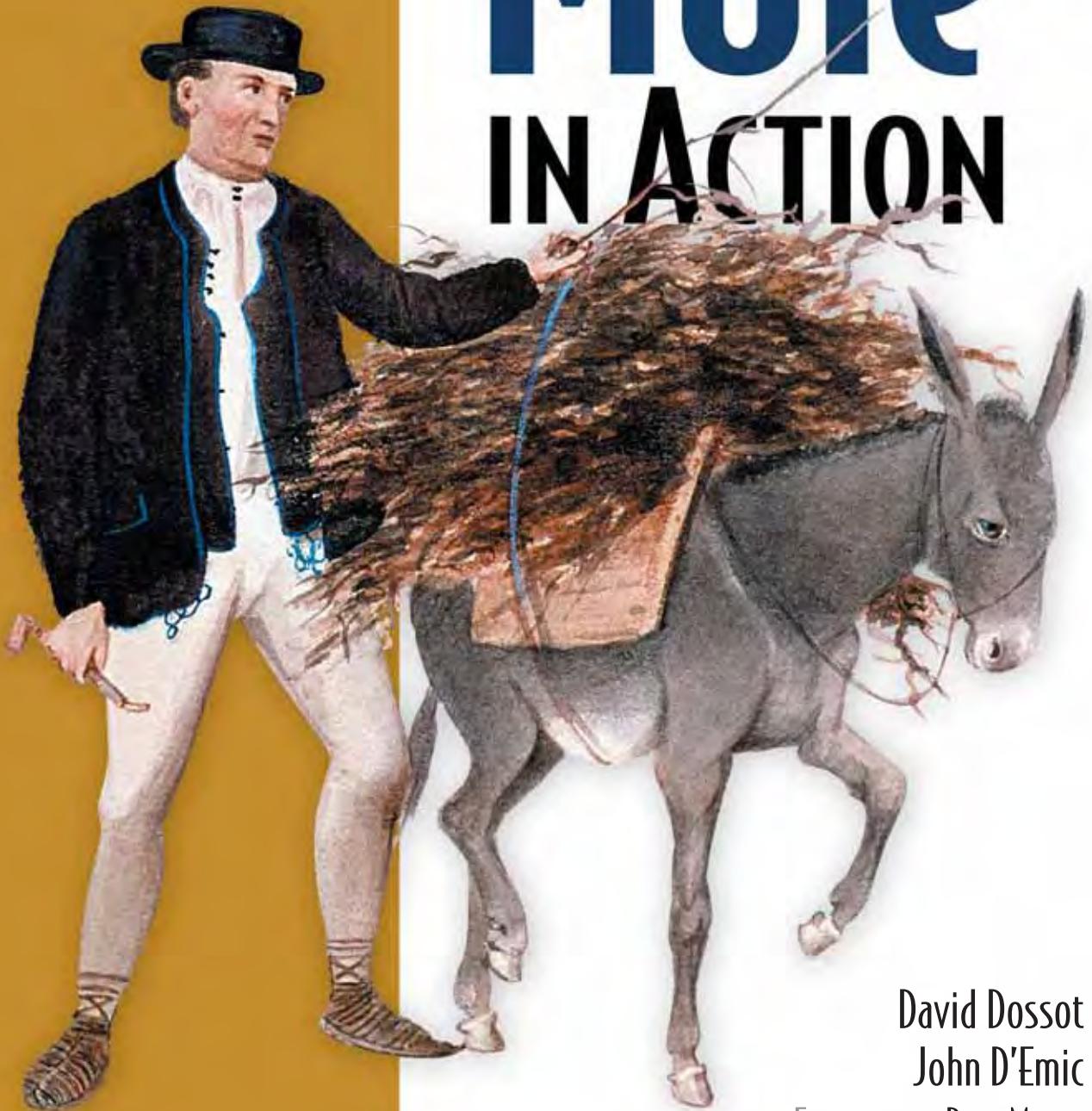


M MANNING

Mule IN ACTION



David Dossot
John D'Emic

FOREWORD BY ROSS MASON

Mule in Action

Mule in Action

DAVID DOSSOT
JOHN D'EMIC



MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit
www.manning.com. The publisher offers discounts on this book when ordered in quantity.
For more information, please contact

Special Sales Department
Manning Publications Co.
Sound View Court 3B fax: (609) 877-8256
Greenwich, CT 06830 email: orders@manning.com

©2010 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Development editor: Jeff Bleiel
Copyeditor: Benjamin Berg
Proofreader: Katie Tennant
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 978-1-933988-96-2
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 14 13 12 11 10 09

*We dedicate this book to those
who are tired of integration donkey work.
Your Mule has arrived!*

brief contents

| | |
|--|------------|
| PART 1 CORE MULE..... | 1 |
| 1 ■ Discovering Mule | 3 |
| 2 ■ Configuring Mule | 21 |
| 3 ■ Sending and receiving data with Mule | 39 |
| 4 ■ Routing data with Mule | 82 |
| 5 ■ Transforming data with Mule | 108 |
| 6 ■ Working with components | 139 |
| PART 2 RUNNING MULE..... | 165 |
| 7 ■ Deploying Mule | 167 |
| 8 ■ Exception handling and logging | 196 |
| 9 ■ Securing Mule | 218 |
| 10 ■ Using transactions with Mule | 233 |
| 11 ■ Monitoring with Mule | 251 |

PART 3 TRAVELING FURTHER WITH MULE 271

- 12 ■ Developing and testing with Mule 273
- 13 ■ Using the Mule API 299
- 14 ■ Scripting with Mule 327
- 15 ■ Business process management
and scheduling with Mule 342
- 16 ■ Tuning Mule 362

contents

foreword xvii
preface xix
acknowledgments xx
about this book xxi
about the authors xxv

PART 1 CORE MULE.....1

1 Discovering Mule 3

- 1.1 ESB, the EAI workhorse 5
- 1.2 The Mule project 7
 - History* 8 ■ *Competition* 9
- 1.3 Mule's core concepts 10
 - Model* 10 ■ *Service* 11 ■ *Transports* 12 ■ *Routers* 14
 - Components* 15 ■ *Request processing* 15
- 1.4 Mule on your machine 18
- 1.5 Summary 19

2 Configuring Mule 21

- 2.1 First ride 22
- 2.2 The Spring XML configuration 25
 - XML element families* 26 ■ *Configured values* 29 ■ *Schema locations* 32

| | | |
|-----|-------------------------------------|----|
| 2.3 | Configuration modularity | 34 |
| | <i>Independent configurations</i> | 34 |
| | <i>Inherited configurations</i> | 35 |
| | <i>Imported configurations</i> | 36 |
| | <i>Heterogeneous configurations</i> | 36 |
| 2.4 | Summary | 37 |

| | | |
|----------|--|-----------|
| 3 | Sending and receiving data with Mule | 39 |
| 3.1 | Understanding connectors and endpoints | 40 |
| | <i>Configuring connectors</i> | 41 |
| | <i>Configuring endpoints</i> | 43 |
| 3.2 | Working with files and directories using the file transport | 46 |
| | <i>Reading and writing files with file endpoints</i> | 47 |
| | <i>Using filters on inbound file endpoints</i> | 49 |
| | <i>Using STDIO endpoints</i> | 50 |
| 3.3 | Using email | 51 |
| | <i>Receiving email with the IMAP transport</i> | 51 |
| | <i>Sending mail using the SMTP transport</i> | 53 |
| 3.4 | Using web services | 55 |
| | <i>Consuming and exposing SOAP services with the CXF transport</i> | 55 |
| | <i>Sending and receiving data using the HTTP transport</i> | 61 |
| 3.5 | Using the JMS transport for asynchronous messaging | 64 |
| | <i>Sending JMS messages with the JMS outbound endpoint</i> | 65 |
| | <i>Receiving JMS messages with the JMS inbound endpoint</i> | 67 |
| | <i>Using selector filters on JMS endpoints</i> | 68 |
| | <i>Using JMS synchronously</i> | 68 |
| 3.6 | Receiving and sending files using the FTP transport | 70 |
| | <i>Receiving files with inbound FTP endpoints</i> | 71 |
| | <i>Sending files with outbound FTP endpoints</i> | 72 |
| 3.7 | Working with databases | 72 |
| | <i>Using a JDBC inbound endpoint to perform queries</i> | 73 |
| | <i>Using a JDBC outbound endpoint to perform insertions</i> | 75 |
| 3.8 | Using the XMPP transport | 76 |
| | <i>Sending Jabber messages on an outbound endpoint</i> | 77 |
| | <i>Receiving Jabber messages on an inbound endpoint</i> | 77 |
| 3.9 | The VM transport | 78 |
| | <i>Sending and receiving messages on VM endpoints</i> | 79 |
| | <i>Using persistent queues on VM endpoints</i> | 80 |
| 3.10 | Summary | 81 |

4 Routing data with Mule 82

- 4.1 Working with routers 83
 - Inbound routers* 84 ▪ *Outbound routers* 85
- 4.2 Using filters with routers 86
 - Filtering by type* 86 ▪ *Filtering by textual content* 87 ▪ *Filtering with expressions* 87 ▪ *Logical filtering* 89
- 4.3 Using inbound routers 89
 - Being picky with the selective-consumer router* 90 ▪ *Altering message flow with the forwarding-consumer router* 91 ▪ *Collecting data with the collection aggregator* 92 ▪ *Insuring atomic delivery with the idempotent receiver* 94 ▪ *Snooping messages with the wiretap router* 96
- 4.4 Outbound routing 97
 - Being picky with the filtering router* 97 ▪ *Sending to multiple endpoints with the static recipient list* 98 ▪ *Broadcasting messages with the multicasting router* 99 ▪ *Service composition with the chaining router* 100 ▪ *Chopping up messages with the message splitter* 102 ▪ *Using asynchronous-reply routers* 104
- 4.5 Summary 107

5 Transforming data with Mule 108

- 5.1 Working with transformers 109
- 5.2 Configuring transformers 111
- 5.3 Using core transformers 114
 - Dealing with bytes* 115 ▪ *Compressing data* 117 ▪ *Modifying properties* 118 ▪ *Leveraging expression evaluators* 120
- 5.4 Using XML transformers 122
 - Transforming format with XSL* 122 ▪ *XML object marshalling* 124
- 5.5 Using JMS transformers 125
 - Producing JMS messages* 125 ▪ *Consuming JMS messages* 126
- 5.6 Existing transformers in action 127
- 5.7 Writing custom transformers 131
 - Transforming payloads* 131 ▪ *Transforming messages* 134
- 5.8 Summary 138

| | |
|--|------------------------------------|
| 6 | Working with components 139 |
| 6.1 | Massaging messages 142 |
| <i>Building bridges 142 ▪ Echoing and logging data 144</i> | |
| <i>Building messages 145</i> | |
| 6.2 | Invoking remote logic 147 |
| <i>Feeling good with SOAP 148 ▪ Taking some REST 149</i> | |
| 6.3 | Executing business logic 151 |
| <i>Resolving the entry point 152 ▪ Configuring the component 156</i> | |
| <i>Handling workload with a pool 158 ▪ Reaching out with composition 160 ▪ Internal canonical data model 162</i> | |
| 6.4 | Summary 163 |

PART 2 RUNNING MULE.....165

| | |
|--|---|
| 7 | Deploying Mule 167 |
| 7.1 | Deployment strategies 168 |
| <i>Standalone server 169 ▪ NetBoot server 171 ▪ Embedded in a Java application 172 ▪ Embedded in a web application 174</i> | |
| <i>Embedded as a JCA resource 177</i> | |
| 7.2 | Deployment topologies 178 |
| <i>Satisfying functional needs 179 ▪ Dealing with the network 181</i> | |
| <i>Designing for high availability 184 ▪ Shooting for fault tolerance 187</i> | |
| 7.3 | Deployment management 189 |
| <i>Using development tools 190 ▪ Hitchhiking Galaxy 192</i> | |
| 7.4 | Summary 195 |
| 8 | Exception handling and logging 196 |
| 8.1 | Exception strategies 197 |
| <i>Positioning exception strategies 197 ▪ Exceptions and routing 202</i> | |
| 8.2 | Using retry policies 207 |
| <i>Implementing a retry policy 207 ▪ Using the SimpleRetryPolicy with JMS 209 ▪ Starting Mule with failed connectors using the Common Retry Policies 210</i> | |

8.3 Logging with Mule 212

Using log4j with Mule 212 ▪ Using Apache Chainsaw with log4j 213

8.4 Summary 217**9 Securing Mule 218****9.1 Demonstrating Mule security 219****9.2 Using security managers and understanding security providers 220**

Using Spring Security 221 ▪ Using JAAS 224

9.3 Securing endpoints with security filters 225

Securing an HTTP endpoint with Spring Security 225

Performing JMS header authentication with JAAS 226

Using password-based payload encryption 228 ▪ Decrypting message payloads with the PGP SecurityFilter 230

9.4 Summary 232**10 Using transactions with Mule 233****10.1 Using transactions with a single resource 235**

Using JDBC endpoints transactionally 235 ▪ Using JMS endpoints transactionally 237

10.2 Using multiple resource transactions 242

Spanning multiple resources with JBossTS 243 ▪ Using XA transactions in a container 246

10.3 Managing transactions with exception strategies 247

Handling component exceptions 248 ▪ Committing transactions with an exception strategy 249

10.4 Summary 250**11 Monitoring with Mule 251****11.1 Checking health 252**

Checking health at network level 253 ▪ Checking health at system and JVM levels 254 ▪ Checking health at JVM and Mule levels 256

11.2 Tracking activity 260

Using log files 261 ▪ Using notifications 264 ▪ Periodic data monitoring 266

11.3 Building dashboards 268

11.4 Summary 270

PART 3 TRAVELING FURTHER WITH MULE.....271

12 Developing and testing with Mule 273

12.1 Managing Mule projects with Maven 274

Setting up a Maven project 275 • *Using the Mule Maven dependencies* 278 • *Simplifying Maven projects with the Mule Maven archetypes* 282

12.2 Using Mule with an IDE 283

XML editing for Mule 283 • *Using Mule's IDE plug-in* 285

12.3 Testing with Mule 289

Functional testing 289 • *Mocking component behavior* 292
Load testing with JMeter 294

12.4 Summary 298

13 Using the Mule API 299

13.1 Piggybacking the Mule client 300

Reaching a local Mule 301 • *Reaching a remote Mule* 302
Reaching out with transports 306

13.2 Exploring the Mule context 307

Controlling a Mule instance 309 • *Reading the configuration* 309 • *Accessing statistics* 309 • *Looking up the registry* 310

13.3 Digging the Mule event context 311

Prospecting messages 312 • *Influencing message processing* 314

13.4 Keeping abreast with Mule 316

Leveraging lifecycle events 317 • *Intercepting messages* 319
Receiving notifications 322

13.5 Summary 325

14 Scripting with Mule 327

14.1 Using Rhino 328

Implementing component logic with Rhino 328 • *Using service interface binding in scripts* 331

| | | |
|------|--|-----|
| 14.2 | Using Groovy | 332 |
| | <i>Implementing transformers with Groovy</i> | 332 |
| | <i>Using the Groovy evaluator</i> | 335 |
| 14.3 | Using Spring | 336 |
| | <i>Implementing custom Mule functionality using Spring</i> | 336 |
| | <i>Auto-reloading scripts</i> | 339 |
| 14.4 | Summary | 341 |

15 *Business process management and scheduling with Mule* 342

| | | |
|------|--------------------------------------|-----|
| 15.1 | Orchestrating services with Mule | 343 |
| | <i>Introducing jBPM</i> | 346 |
| | <i>Using jBPM with Mule</i> | 348 |
| 15.2 | Job scheduling with Mule | 356 |
| | <i>Using Quartz to schedule jobs</i> | 357 |
| | <i>Polling endpoints</i> | 358 |
| | <i>Dispatching jobs</i> | 359 |
| 15.3 | Summary | 361 |

16 *Tuning Mule* 362

| | | |
|------------|--|-----|
| 16.1 | Understanding thread pools | 363 |
| | <i>Synchronicity aspects</i> | 365 |
| | <i>Transport peculiarities</i> | 368 |
| | <i>Configuration options</i> | 370 |
| 16.2 | Increasing performance | 372 |
| | <i>Profiler-based investigation</i> | 373 |
| | <i>Performance advice</i> | 377 |
| 16.3 | Summary | 379 |
| appendix A | <i>The expression evaluation framework</i> | 381 |
| appendix B | <i>The Mule community</i> | 387 |
| | <i>index</i> | 391 |

foreword

Secretly, my wife still harbors a little regret about the lost weekends I spent coding Mule, but without her, Mule would not have been created and the book you are reading would not exist.

Like thousands of developers before me, I was continually struggling with the complexities of systems integration. The problem was that the proprietary solutions of the day—there were no open source alternatives back then—set out to address integration by adding another piece of complexity to the problem. These products made too many assumptions about the environment and architecture, masking the ugliness with doodleware, slick demo applications, and suave salesmen. I used to work long hours trying to work around integration products rather than being able to leverage them. This resulted in me venting to the point where my wife firmly suggested that I stop complaining and do something about it. A Mule was born.

Six years on, and Mule represents a shift in the way we integrate applications. It provides a focus on service orientation and assembly instead of building monolithic application stacks. Integration and service orientation are becoming increasingly important parts of application developers' lives, since organizations never throw anything away. Couple this with the rise of SaaS, Web 2.0, and Cloud computing, and we have an evolution from traditional application development to an assembly model, where data is served in many forms from many sources inside and outside of our company firewalls.

This book provides the first thorough coverage of all aspects of Mule. It provides examples for everything you will need to do with Mule, from creating and consuming services to working with various technologies such as JMS, Web Services, and FTP. Importantly, it covers how to test, deploy, monitor, and tune Mule applications, topics that can trip up new users due to the flexibility of the Mule platform.

The great yet subtle element of this book is that the authors have captured the essence of pragmatism that is the founding principle of Mule. The notion that you can start small and build a complete ESB architecture over time is prevalent. Each chapter explains the tools provided by Mule for building service-oriented applications. The chapters cover everything, including configuration basics, message routing, data transformation, publishing services, and working with the Mule Galaxy registry.

This publication marks a significant milestone for the Mule project. It demonstrates that the ideals of open source and community building do work. The authors, David Dossot and John D'Emic, have been long-time community members and have made many other contributions to the project; this is a significant and lasting addition. I can see this book becoming the must-have guide for all current and prospective Mule users since it walks the reader through all aspects of Mule in the right amount of detail, focusing on the areas most important for building applications. Read on to learn how to unlock the power of Mule.

ROSS MASON
Creator of Mule

preface

Developers who've had to do application integration work know what a daunting endeavor it can be. While some applications provide facilities for integration, say with a rich API meant to be externally consumed, many were never designed to be accessed by other applications. Challenges also abound for applications that are designed for interoperability from the ground up. Developers will often spend a considerable amount of time dealing with the plumbing of a particular integration protocol, such as SOAP or JMS, to allow their applications to play nicely with the outside world.

Enterprise Integration Patterns, by Gregor Hohpe and Bobby Woolf, quantified these challenges, attached a language to them, and offered a catalog of solutions. Integration developers and architects could, if they chose, look at their integration problems through the lens of these patterns and implement them appropriately. Implementation, though, is where the work lies. The integration developer, in addition to dealing with JMS brokers, SOAP stacks, and legacy databases, now has to implement message routers, data transformers, and protocol adaptors to put the patterns into practice.

This is where Mule fits in. Mule provides a platform to develop and host your integration solutions. By implementing many of the patterns described in *Enterprise Integration Patterns*, Mule allows you to focus on solving the integration aspects specific to your domain. Mule frees you from the plumbing of integration, such as implementing an FTP client or transforming a Java object to XML—just as a web development framework frees you from dealing with the details of raw HTTP requests.

We have used Mule since 2005 in a variety of contexts, from high-performance message processing to enterprise integration and orchestration. In these contexts, we've used Mule either as a standalone ESB or as an application-level integration framework. This is the experience we want to share with you in this book.

acknowledgments

We'd like to thank our development editor at Manning, Jeff Bleiel, who has been a joy to work with. We'd also like to thank Nicholas Chase for his support with the DocBook writing and editing process. We want to extend further thanks to our reviewers, whose insights helped us build a better book: Ben Hall, John Griffin, Ara Abrahamican, Dimitar Dimitrov, Doug Warren, Fabrice Dewasmes, Jeroen Benckhuijsen, Josh Devins, Mikel Ayveis, Rick Wagner, Roberto Rojas, Tijs Rademakers, Prasad A. Chodavarapu, Rama Kanneganti, Davide Piazza, and Celso Gonzalez. Special thanks to Ross Mason for writing the foreword to our book and to our technical proofreader Andrew Perepelitsya.

DAVID I'd like to thank my wife and kids for their patience and constant support during the tough journey of writing this book. I also want to apologize to everybody who thought I was writing a book about eMule and was disappointed when they heard about ESB and integration.

JOHN I would like to thank my wife, Catherine, for supporting (and putting up with) me while I wrote this book. I also want to thank my family, colleagues, and friends. All have been continuously supportive throughout this process, through words of encouragement and sharing the occasional stiff drink.

about this book

Mule, as the preeminent open source integration platform, provides a framework for implementing integration solutions. This book will give you the tools for using Mule effectively. It's not a user guide: Mule's comprehensive user guide is available online already. Instead, it's a review of Mule's main moving parts and features put into action in real-world contexts. After a bit of history and some fundamentals about configuring Mule, we'll walk you through the different family of components that you'll use in your projects. We'll then review some runtime concerns such as exception handling, transactions, security, and monitoring. Then we'll delve into advanced subjects such as programming with Mule's API, business process orchestration, and tuning.

Who should read this book

This book is primarily targeted at Java developers who want to solve integration challenges using the Mule platform. It's also useful for architects and managers who're evaluating Mule as an integration platform or ESB solution. The majority of Mule's functionality can be leveraged without writing a line of Java code. As such, system administrators who find themselves repeatedly solving the same integration problems will also find this book useful. Additionally, system administrators tasked with supporting Mule instances will find this book, in particular part 2, of value.

How to use this book

The chapters in this book build upon each other. Readers new to Mule are encouraged to read the book in this manner. Readers familiar with Mule 1.x will find part 1 particularly useful, as it describes and provides examples of the new configuration syntax in depth. This book isn't intended as a reference manual—we deliberately chose to

provide examples of working Mule configurations over tables of XML Schema elements. More importantly, providing such a book would duplicate the content already available at www.mulesource.org, the Mule Javadocs, and the XSD documentation—where complete reference documentation is available. We’re hoping that reading *Mule in Action* gives you the skills to use these resources effectively.

Roadmap

Chapter 1 introduces the origins and history of the Mule project. You’ll also discover the architecture and terminology of Mule ESB.

Chapter 2 details the configuration of Mule with XML files and how you can organize them efficiently.

Chapter 3 shows you how to use Mule’s transport functionality to move data around using different mechanisms, like SOAP or JMS.

Chapter 4 covers Mule’s routing functionality. You’ll see how Mule’s routers let you control how data enters and leaves your services.

Chapter 5 describes message transformation within Mule. You’ll learn about commonly used transformers and will even create your own.

Chapter 6 covers service components and how you can leverage them to add custom behavior in your Mule services.

Chapter 7 details the different aspects of deploying Mule within your IT landscape, including deployment topologies.

Chapter 8 describes how to handle exceptions and logging with Mule. We’ll cover how exception strategies and logging let you identify and recover from errors.

Chapter 9 shows you how to secure your Mule applications. You’ll see how Mule’s security features let you authenticate, authorize, and encrypt messages that pass through your services.

Chapter 10 covers Mule’s transactions support. You’ll see how business expectations can be enforced by applying transactions on your endpoints.

Chapter 11 details your options in terms of monitoring Mule instances. You will also learn about best practices for auditing and discover how to build dashboards.

Chapter 12 shows you how to use Mule’s development tooling support and testing features to simplify your life.

Chapter 13 describes the principal members of the API you will use when writing custom code for your Mule projects.

Chapter 14 covers Mule’s support for scripting functionality. You’ll see how Rhino and Groovy can be used to accelerate the development of components, transformers, and custom routers.

Chapter 15 covers how the jBPM and Quartz transports can be used for service orchestration and scheduling.

Chapter 16 explains the threading model of Mule, how to configure it, and how to profile and tune the ESB.

Code conventions

The code examples in this book are abbreviated in the interest of space. In particular, namespace declarations in the XML configurations and package import in Java classes have been omitted. The reader is encouraged to use the source code of the book when working with the examples. The line length of some of the examples exceeds that of the page width. In cases like these, the ➔ marker is used to indicate a line has been wrapped for formatting.

All source code in listings or in text is in a fixed-width font like this to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets link to explanations that follow the listing.

Source code downloads

The source code of the book is available online from the publisher's website at <http://www.manning.com/MuleinAction>, as well as from this URL: <http://code.google.com/p/muleinaction/>.

Software requirements

The required configuration to run these examples follows:

- JDK 5 or better
- Maven 2.0.8 or better
- Mule 2.2.x Community Release

Mule Community Release can be downloaded from this page: <http://www.mule-source.org/display/MULE/Download>.

Author Online

The purchase of *Mule in Action* includes free access to a private web forum run by Manning Publications, where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to <http://www.manning.com/MuleinAction>. This page provides information about how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the book's forum remains voluntary (and unpaid). We suggest you try asking them some challenging questions, lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the cover illustration

The figure on the cover of *Mule in Action* is captioned “Man and his mule, from the village of Brgud in Istria, Croatia.” The illustration is taken from a collection of watercolors by Nikola Arsenovic, owned by the Ethnographic Museum of Belgrade, Serbia.

Arsenovic (1823–1885) was a professional tailor with a thriving business in the northeastern Croatian city of Vukovar. His ambition was to record traditional dress habits across the territories of Croatia and he set out to do this over a period of two years in the middle of the nineteenth century. Arsenovic had talent but no artistic training. His background as a tailor is visible in the quality and the attention to detail of decoration that these images exhibit. He worked at a time when traditional clothing still retained age-old forms, but in the decades following his work, the switch to more modern forms accelerated in these regions. In many cases his watercolors present the last and the only trace of certain traditional Croatian clothing styles. Manning obtained complimentary files of scanned images from the Ethnographic Museum of Split.

The rich variety of Arsenovic’s collection reminds us vividly of how culturally apart the world’s towns and regions were just 150 years ago. Isolated from each other, people spoke different dialects and languages. It was easy to identify where a person lived and his trade or station in life just by his clothes. Dress codes have changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness , the initiative, and the fun of the computer business with book covers based on the colorful diversity of regional dress customs of long ago, brought back to life by the pictures from this collection.

about the authors

David Dossot has worked as a software engineer and architect for more than 14 years. He's been using Mule since 2005 in a variety of different contexts and is the project "despot" of the JCR Transport. He's the project lead of NxBRE, an open source business rules engine for the .NET platform (selected for O'Reilly's Windows Developer Power Tools). He's also a judge for the Jolt Product Excellence Awards and has written several articles for SD magazine. He holds a production systems engineering diploma from ESSTIN.

John D'Emic has been working on integration projects in one form or another for the last 11 years. He's currently the chief architect at OpSource, Inc., and has been leveraging Mule in that capacity since 2006. He holds a degree in computer science from St. John's University.

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 SOAP² 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].

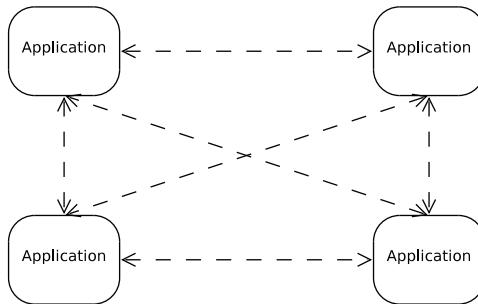


Figure 1.1 Point-to-point integration approach: everyone must speak the language of everyone.

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.

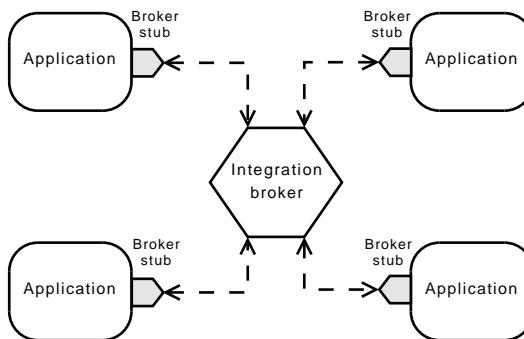


Figure 1.2 Application-invasive integration broker approach: everyone must speak the language of the broker.

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.

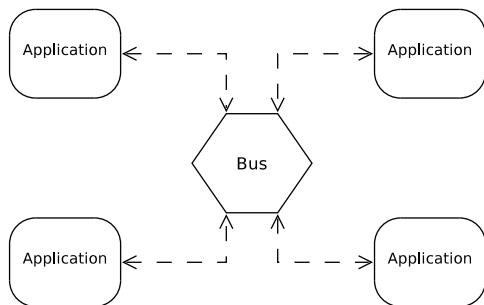


Figure 1.3 Enterprise service bus approach: the bus speaks the language of everyone.

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.

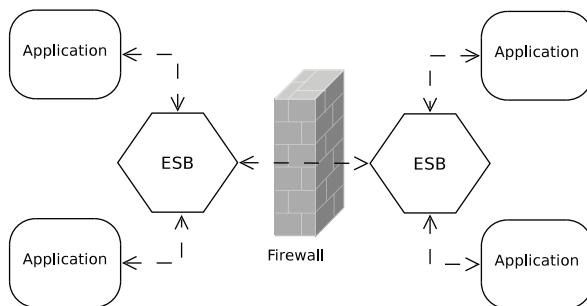


Figure 1.4 ESB nodes distributed at strategic network locations

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. Spring-Source 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-FO⁶ as its input. Moreover, this input should be sent as text messages to a JMS queue. The authors use an editing tool that generates DocBook⁷ 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.

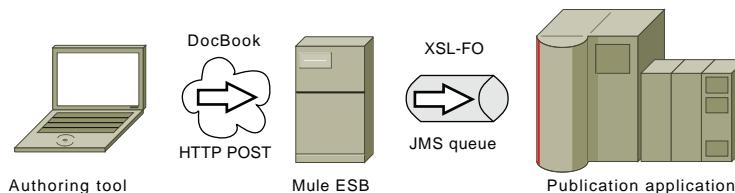


Figure 1.5 Mule ESB acting as a mediator between an authoring tool and a publication application

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.

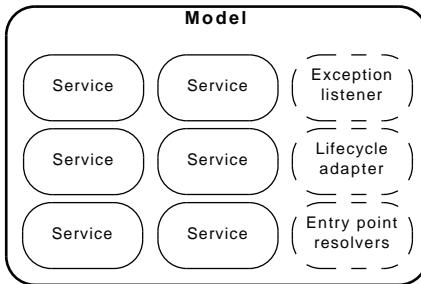


Figure 1.6 A Mule model is the runtime environment into which services are hosted.

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

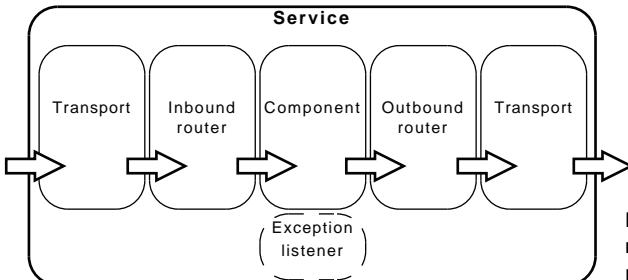


Figure 1.7 A Mule service mobilizes different moving parts to process requests.

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.

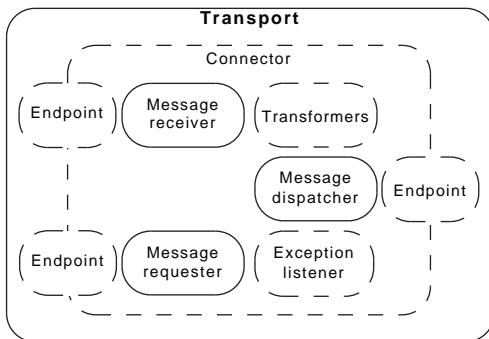


Figure 1.8 A Mule transport provides all the ESB elements required for receiving, sending, and transforming messages for a particular protocol.

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.

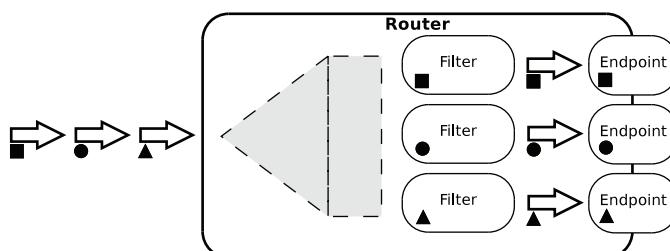


Figure 1.10 A router can leverage filters to dispatch messages based on their properties.

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.

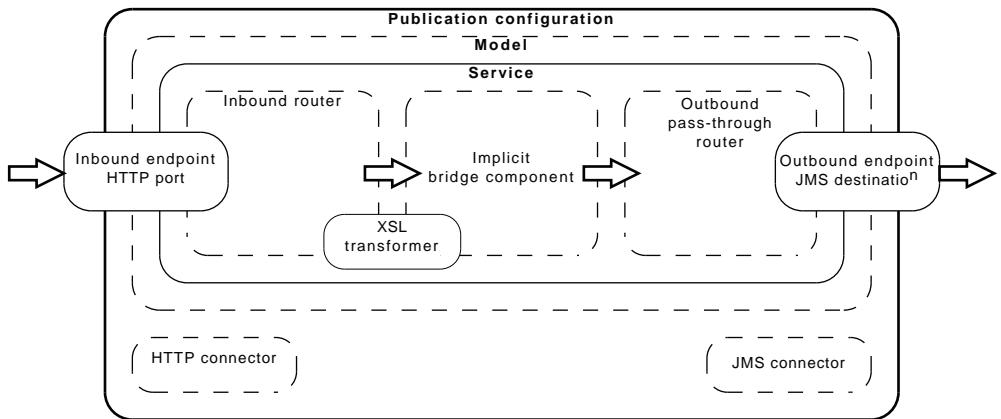


Figure 1.11 The different Mule moving parts involved in the Publication application configuration

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>

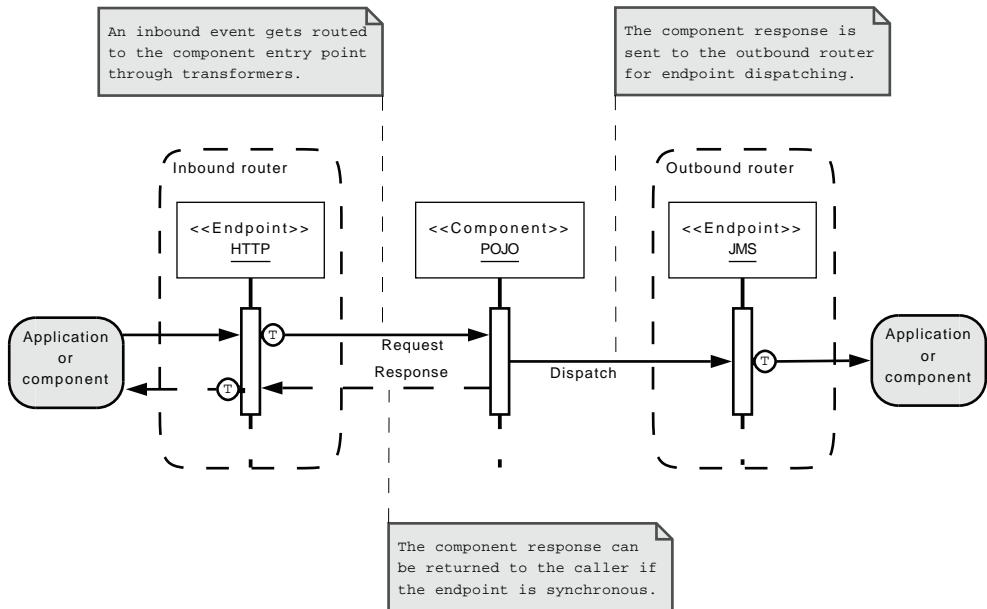


Figure 1.12 Standard event processing in a Mule service

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.

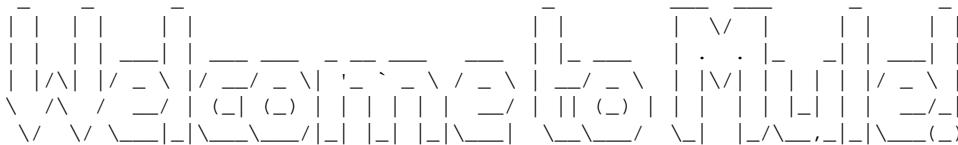
Download the source code from the companion web site of this book (<http://muleinaction.com>) and decompress it in another folder of your choice. Ensure that your `MULE_HOME` points to a Mule version that matches the example source code requirements. Most examples will require Maven 2 to run: if you 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.

Configuring Mule



In this chapter

- Configuring Mule with XML files
- Elements of a configuration file
- Configuration files organization strategies

In this chapter, you'll learn the fundamental principles of a Mule configuration file. Said differently, this chapter will give you the grammar of the configuration file, while the upcoming chapters will help you build your vocabulary. When you're done reading it, you'll be able to create new configuration files and set up the scene for your own services, which you'll learn to create in the coming chapters.

In essence, configuring Mule consists of defining the services you want to be active in a particular instance of the ESB. As seen in the previous chapter, these services are composed of and rely on many different moving parts, which also need to be configured. Some of these moving parts are intrinsically shared across several services, such as connectors. Others can be locally defined or globally configured and shared, such as endpoints. As you can guess, supporting this flexibility and richness in a configuration mechanism is pretty hairy.

To achieve this, Mule uses *configuration builders* that can translate a human-authored configuration file into the complex graph of objects that constitutes a running node of this ESB. The main builders are of two kinds: a Spring-driven builder, which works with XML files, and a script builder, which can accept scripting language files.

NOTE The scripting configuration builder uses a script in any language for which a JSR-223-compliant engine exists (such as Groovy, JRuby, or Rhino). By default, Mule supports Groovy configuration files, but other scripting languages can be added by installing the Mule Scripting Pack that can be downloaded from <http://mulesource.org/display/MULE/Download>. A scripted configuration is a low-level approach to configuring Mule. It's up to you to instantiate, configure, and wire all the necessary moving parts. This requires an expert knowledge of Mule internals. This explains why this approach is much less popular than XML configuration, as it's rare that anyone needs this level of control. This said, there can be circumstances where using Spring or XML isn't an option. In that situation, using a scripted configuration can save the day.

In this chapter we'll mainly focus on configuring the Spring XML builder, for several reasons:

- *It's the most popular*—you're more likely to find examples using this syntax.
- *It's the most user friendly*—Spring takes care of wiring together all the moving parts of the ESB, something you must do by hand with a scripted builder.
- *It's the most expressive*—dedicated XML schemas define the domain-specific language of Mule, allowing you to handle higher-level concepts than the scripting approach does.

We'll start by running a simple example, which will allow you to look at your first service configuration. We'll then review the overall structure, element families, and configurable items of the Spring XML file in general terms, just enough for you to get the gist of it and have the basic knowledge you'll need to grasp what you'll learn in the upcoming chapters. We'll also give some advice on how to organize your configuration files efficiently.

Are you ready for the ride? We believe the answer is yes, so let's start to look into our first Mule configuration file.

2.1 **First ride**

Though Mule comes complete with a “hello” example, we chose to get started with the “echo” example that's also bundled with the platform. We believe the echo example is the true “hello world” example for Mule, as it doesn't require any transformer, custom component, or specific router. That's why we decided to nickname it “Echo World.”

Let's now look into details of the Echo World example. As shown in figure 2.1, this application uses the `stdio` transport to receive messages from the console input (`stdin`) and send these messages unchanged directly to the console output (`stdout`).

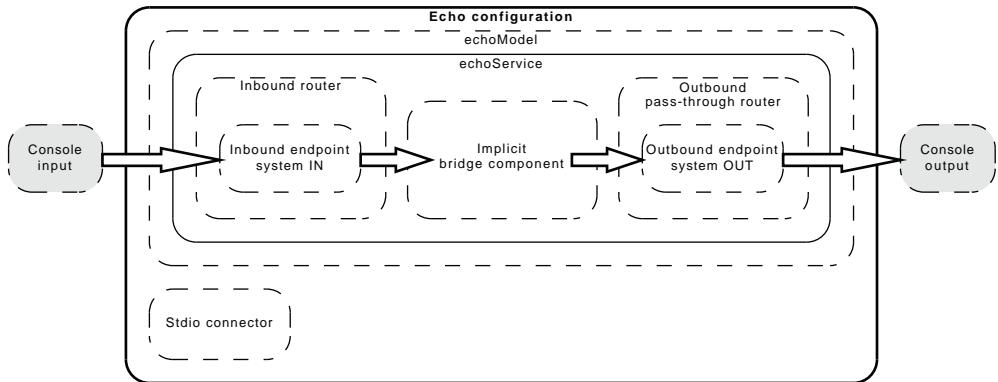


Figure 2.1 Moving parts and message flow of the echo example

A specific component, called a *bridge*, is used to pass the messages from the inbound router to the outbound one. A bridge is a neutral component: it doesn't perform any action or modify the messages that it processes. The outbound router is a pass-through one: it's the simplest router that exists. It dispatches the messages it receives to a unique endpoint.

Listing 2.1 provides the full configuration for this example. As an astute reader, you'll quickly notice that there's no bridge component. This is because a service uses a bridge component implicitly if none is configured. Except for this subtlety, the configuration file is pretty straightforward. Note how the specific attributes on the different elements help to make the configuration self-explanatory.

Listing 2.1 The echo example XML configuration

```

<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd">
    <stdio:connector name="SystemStreamConnector"
        promptMessage="Please enter something: "
        messageDelayTime="1000" />

    <model name="echoSample">
        <service name="echoService"> ← The echo service
            <inbound>
                <stdio:inbound-endpoint system="IN" />
            </inbound>
            <outbound>
                <pass-through-router>
                    <stdio:outbound-endpoint system="OUT" />
                </pass-through-router>
            </outbound>
        </service>
    </model>

```

Defines and configures
standard input
connector

```

        </outbound>
    </service>
</model>
</mule>
```

In chapter 1, we installed the Mule standalone server and downloaded the examples from this book's companion web site. So, at this point, you should be able to start this example. For this, go into the /chapter02/echo directory and run the echo-xml batch file that suits your OS. The output should be as shown here, with a version and a build number that match your Mule installation and with the host name of your machine:

```

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.

INFO 2008-06-28 14:19:26,035 [WrapperListener_start_runner]
org.mule.MuleServer: Mule Server initializing...
... (lots of lines) ...

INFO 2008-06-28 14:19:28,794 [WrapperListener_start_runner]
org.mule.DefaultMuleContext:
*****
* Mule ESB and Integration Platform *
* Version: 2.2.0 Build: 12377 *
* MuleSource, Inc. *
* For more information go to http://mule.mulesource.org *
*
* Server started: 28/02/09 14:19 *
* Server ID: b4b877b5-76e1-11dd-bbe5-e14763a30d37 *
* JDK: 1.5.0_07 (mixed mode) *
* OS: Linux (2.6.24-19-386, i386) *
* Host: emcee-mule (127.0.0.1) *
*
* Agents Running: None *
*****
Please enter something:
```

As prompted, enter something. Note that the first time a message gets dispatched to the outbound endpoint, Mule instantiates and connects the necessary dispatcher:

```

Please enter something:
Something
INFO 2008-06-28 15:52:46,697 [SystemStreamConnector.dispatcher.1]
org.mule.transport.stdio.StdoutMessageDispatcher: Connected:
StdioMessageDispatcher{this=15c97e4, endpoint=stdio://system.out,
disposed=false}

Something

Please enter something:
```

```

Something else
Something else

Please enter something:

```

When you're tired of playing with the echo, stop the execution of the application with control+C. All the log entries you'll see scrolling on your console will represent a normal and clean shutdown of the ESB. The very last lines shown in the console should be like this:

```

*****
* The server is shutting down due to normal shutdown request      *
* Server started: 28/02/09 15:52                                *
* Server shutdown: 28/02/09 15:53                               *
*****
<-- Wrapper Stopped

```

Note how control+C has been interpreted as a normal shutdown request. If the JVM were to exit abruptly, for example because of a hard crash, the wrapper script would restart the instance automatically.

Even if basic, this example has shown you what less than 20 lines of configuration can buy you in Mule. You also got the gist of the runtime environment of Mule, how it behaves at startup and shutdown times, and what it reports in the logs.

The Echo World example has given you some clues about the organization of the Spring XML configuration file.¹ Let's now broaden your view by exploring the structure of this configuration file.

2.2 The Spring XML configuration

The Spring configuration builder relies on XML to enforce the correct syntax (well-formedness) and on XML Schema to define the grammatical rules (validity). These rules define the usable elements² and where they can be used. These elements represent the different moving parts of the ESB and their configurable parameters.

Mule doesn't rely on a single monolithic schema to define all the configurable elements, but instead on several of them. One of these schemas defines the core elements and the general structure of the configuration file. Other elements are defined by optional additional schemas, such as the transport-specific ones. If you look at the first lines of listing 2.1, you'll see that the core and the VM transport specific schemas are imported.

BEST PRACTICE Always validate your XML configuration files before attempting to load them in Mule to simplify troubleshooting.

Figure 2.2 represents the elements defined by the core schema. Note how the services, which are the main actors of a configuration, are lost in the bottom-right corner of

¹ The same example is also provided as a scripted configuration in echo-world/conf/echo-config.groovy if you're curious to discover this configuration approach as well.

² Element as defined by the W3C DOM specification.

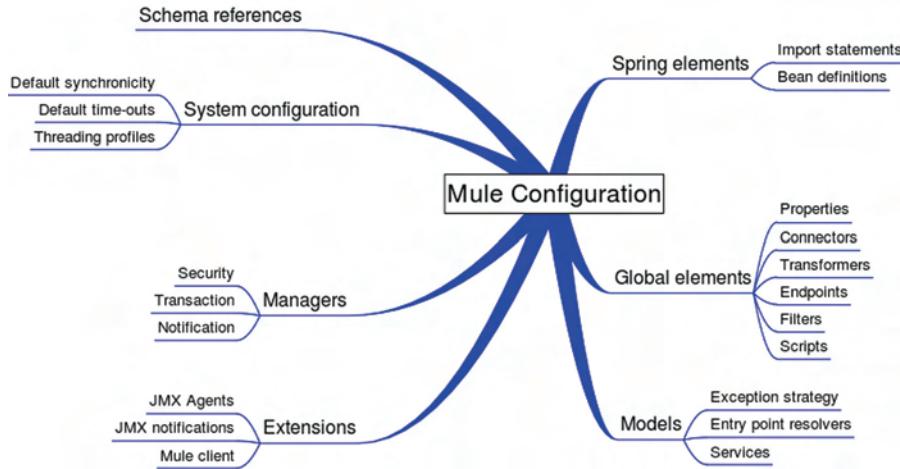


Figure 2.2 Configuration structure defined by Mule's XML core schema

the mind map. This gives you an idea of all the supporting common features and global elements you can declare in a configuration file. Throughout the book, you'll progressively learn more about each of these elements.

The Echo World configuration in listing 2.1 only has a few schema references and one global element (a connector). You can easily imagine that real-world configuration files declare many of these shared configuration artifacts to support the configuration of their services. In the examples coming in the next chapters, you'll often see such shared elements, like global endpoints or transformers. Some chapters will even be dedicated to some of these common elements, such as transaction managers or JMX agents.

For now, we'll consider these elements under several of their main characteristics. First we'll look at their main families, then how we can define configured values, and finally where to find the schemas that define them.

2.2.1 **XML element families**

Independent of their functions, we can conceptually distinguish three main families of elements in a Mule configuration:

- The specific elements
- The custom elements
- The Spring elements

It's important to understand what you can expect from each of these to build a configuration file in the most efficient manner.

SPECIFIC ELEMENTS

The specific elements are the most expressive configuration elements available, insofar as they constitute the domain-specific language of Mule. As such, these elements

and their attributes are characterized by highly specific names. Another characteristic is that they intrinsically refer to predefined concrete implementations: they're used to express all the configuration elements that aren't custom-made. This is why no class name comes into play when using this kind of configuration element. A benefit of this is the isolation your configuration gets from the internals of Mule's implementation: if a class is renamed, your configuration won't be affected.

The Mule core and each transport or module you include in your configuration will contribute such specific elements. Table 2.1 is a nonexhaustive list³ of these elements as defined by a few Mule XML schemas.

Table 2.1 Specific elements defined by some Mule schemas

| Schema Elements defined (nonexhaustive list) | Example |
|--|---|
| Mule Core Models Transformers Routers Filters Components Security manager Exception strategies Transaction manager | <pre data-bbox="456 614 1021 673"><object-to-string-transformer name="ObjectToString" /></pre> <p>Defines a global object-to-string transformer named ObjectToString.</p> <pre data-bbox="456 698 964 757"><global-property name="smtp.username" value="jqdoe" /></pre> <p>Defines a global property named smtp.username, whose value is jqdoe.</p> |
| HTTP Transport HTTP connector HTTP endpoint REST service component HTTP request and response transformers | <pre data-bbox="456 883 1034 942"><http:polling-connector name="pollingHttp" pollingFrequency="3000" /></pre> <p>Defines an HTTP connector named pollingHttp, which will be used for polling a URL every three seconds.</p> |
| XML Module XML transformers XML routers XML filters | <pre data-bbox="456 1060 978 1118"><xm:xslt-transformer name="DocBook2FO" xsl-file="db2fo.xsl" /></pre> <p>Defines an XSL-T transformer named DocBook2FO, which uses the templates defined in db2fo.xsl.</p> |

CUSTOM ELEMENTS

Custom elements constitute the Mule-oriented extension points of a configuration. They mainly allow you to use custom implementations or Mule classes for which no specific element has been created. These extension points are available for core Mule artifacts and also for some transports and modules. Custom elements usually rely on a fully qualified class name parameter to locate your custom code. They also provide a means to pass the property values your class needs to be properly configured. Table 2.2 shows a few examples of these custom elements.

³ There are more XML elements defined in these schemas; only the most notable ones are listed here.

Table 2.2 Custom elements defined by some Mule schemas

| Schema Elements defined (nonexhaustive list) | Example |
|--|--|
| Mule Core Custom transformer Custom router Custom filter Custom entry point resolver Custom security and encryption providers Custom exception strategy Custom transaction manager | <pre data-bbox="463 292 1136 366"><custom-transformer name="NameStringToChatString" class="org.mule.example.hello.</pre> <p data-bbox="542 366 885 388">NameStringToChatString"/></p> <p data-bbox="463 383 1110 426">Defines a custom transformer named NameStringToChatString as an instance of the specified class.</p> |
| TCP Transport Custom protocol | <pre data-bbox="463 561 1040 635"><tcp:custom-protocol class="org.mule.transport.tcp.integration.</pre> <p data-bbox="516 635 947 656"> ↳ CustomSerializationProtocol"/></p> <p data-bbox="463 651 1110 678">Defines a custom TCP protocol as an instance of the specified class.</p> |

Though it's possible to do what a specific XML element can do using a custom XML element, this should be avoided for two main reasons. The first is because you'll become coupled to a particular implementation in Mule itself: if a new class is created and used by the specific element, your custom one will keep referencing the old one. The second is because you'll lose the benefit of the strongly typed (schema validated) and highly expressive attributes defined on the specific elements.

BEST PRACTICE If possible, try to use a specific configuration element instead of custom one.

SPRING ELEMENTS

Thanks to the support of a few core Spring schemas, Mule can accept Spring elements in its configuration. Spring elements are used to instantiate and configure any object that'll be used elsewhere in the configuration, where they'll usually be injected in standard Mule elements. They're also used to easily construct configuration parameters such as lists or maps. Finally, they allow configuration modularity through the support of the import element (see section 2.3). Table 2.3 shows a few examples of these Spring elements. Note how the `spring:beans` element opens the door to the usage of any Spring schema (in this example we use the `util` one).

Besides configuration extension, Spring elements also grant access to the framework itself, making possible the usage of advanced features such as AOP. Though you can do a lot in Mule without deferring to Spring, getting acquainted with this framework will allow you to better grasp the core on which Mule is built. If you're unfamiliar with Spring or would like to learn more about it, we strongly recommend reading *Spring in Action* from Manning Publications Co. (Walls and Breidenbach).

Table 2.3 Elements defined by the supported Spring schemas

| Schema Elements defined (nonexhaustive list) | Example |
|---|--|
| Spring Beans Beans Bean Property | <pre data-bbox="380 294 983 982"><spring:bean name="cfAMQ" class="org.apache.activemq.spring. ↗ActiveMQConnectionFactory"> <spring:property name="brokerURL" value="tcp://localhost:61616"/> </spring:bean></pre> <p>Instantiates an ActiveMQ connection factory, configured with the specified broker URL parameter and named cfAMQ.</p> <pre data-bbox="380 529 1089 982"><spring:beans> <spring:import resource="classpath:applicationContext.xml" /> </spring:beans></pre> <p>Imports a Spring application context from the classpath.</p> <pre data-bbox="380 680 983 982"><inbound-endpoint ref="globalEndpoint"> <properties> <spring:entry key="valueList"> <util:list> <spring:value>value1</spring:value> <spring:value>value2</spring:value> </util:list> </spring:entry> </properties> </inbound-endpoint></pre> <p>Sets a list property on an endpoint.</p> |
| Spring Context Property placeholder resolver | <pre data-bbox="380 1033 1181 1117"><context:property-placeholder location="node.properties" /></pre> <p>Activates the replacement of \${...} placeholders, resolved against the specified properties file.</p> |

You've now learned to recognize the different families of XML elements you'll have to deal with and what you can expect from them. XML elements without values specific to your configuration are useless. Let's now see how you'll set these different values in your own configuration files.

2.2.2 Configured values

In a Mule configuration, values are seldom stored as text nodes; they're mainly stored as attributes of XML elements. The main exceptions are text-rich properties such as documentation or script elements. This makes the configuration easier to read, format, and parse, as attributes are less prone to being disrupted by unwanted

whitespaces. These attributes are strongly typed, with data types defined by the XML Schema standard.

We'll consider these configured values under their most notable traits:

- Default values
- Enumerated values
- Expressions
- Property placeholders
- Names and references

DEFAULT VALUES

Using attributes to hold values enables the definition of default values. Mule schemas take great care to define default values wherever possible. When you start writing your own configuration files, you'll quickly realize how valuable this is. Whenever you configure a Mule object, the question of its default configuration values will arise. What would usually take a visit to the JavaDoc page of the object will now require only a glance at the default values for the different attributes supported by the XML element that represent the object (provided you use a decent tool; see chapter 12).

ENUMERATED VALUES

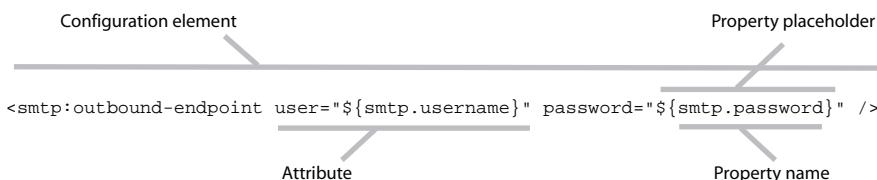
On top of being strongly typed, some attributes define an enumeration of possible configuration values. This greatly reduces the risk of introducing a bogus value that would only be caught later on at runtime. This also provides valuable information about the intent of the parameter and the abilities of the configured object.

EXPRESSIONS

Mule's expression evaluation framework offers the capacity to define configuration values that are evaluated dynamically at runtime. With simple attribute values of the form #[evaluator:expression], it is possible to access almost any data from the current message being processed or the Mule instance itself. Please refer to appendix A for more information about this powerful framework.

PROPERTY PLACEHOLDERS

An important aspect of any configuration file is the capacity to externalize certain values that can change at runtime. These values are called *properties*. They're generally used to define environment-specific parameters such as credentials, port numbers, or paths. It's possible to use such properties in lieu of fixed values in any attribute of a Mule configuration. The property is then referred to by its name using a special placeholder syntax. This is done using the classic Ant notation, as shown:



These properties are defined either as global ones in the Mule configuration, in a standard Java properties file or in JVM system properties. For the last two options, Spring's property placeholder resolver (shown in table 2.3) takes care of injecting the properties value in your configuration.

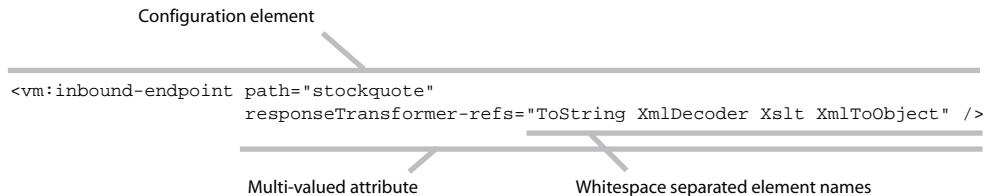
TIP *Environment properties* As with any other application, you'll have to externalize environment-specific values in properties files. These are usually passwords, remote service URLs, port numbers, time-outs, or cron expressions. With Spring XML configurations, a good practice is leveraging Spring's `PropertyPlaceholderConfigurer` to load properties from several files on the classpath. The idea is to define reasonable defaults for the development environment in a properties file embedded in the deployable itself, and to override these values with others defined in an environment-specific property file placed also on the classpath, but outside of the deployable. This is demonstrated here:

```
<context:property-placeholder  
    location="classpath:META-INF/default.props,  
    ↪classpath:override.props" />
```

This configuration element imports values from classpath files `META-INF/default.props` and from `override.props`. The latter can be left empty in development and tuned to use correct values in test or production environments. This construct also supports overriding with Java system properties: if a system property is defined, it'll take precedence over a property of the same name defined in one of these files.

NAMES AND REFERENCES

Being able to have elements that reference other elements is an essential aspect of a Mule configuration file. This is achieved by using name and reference attributes. Most of the elements can receive a name through this mechanism. References can sometimes be multivalued. In that case, whitespace is used to separate the different names that are referred to. The following demonstrates an inbound endpoint that refers to a chain of four transformers simply by listing their names:



You now know the secrets to setting values in your XML configurations. We'll now look at one thing that may still puzzle you: the location of all these different schemas.

2.2.3 Schema locations

You might be wondering where to get an up-to-date list of all the available schemas you can use in your configurations. This section will do better than give you this list: it'll allow you to figure out by yourself the right schema reference to use for any Mule library you decide to use.

When you look at the declaration in listing 2.1, you might be wondering if Mule will connect to the Internet to download the different schemas from the specified locations. Of course it doesn't. Each library (transport or module) embeds the schemas it needs. Mule leverages the resource resolver mechanism of Spring to "redirect" the public HTTP URIs into ones that are internal to the JAR file. How is this mechanism configured? To discover it, fire up your favorite archiving utility or IDE and open the library you want to use. There should be a directory named META-INF. After opening it, you should see something similar to the screen shot shown in figure 2.3.

The schema you're looking for is in the META-INF directory and is named after the transport or module name, prefixed with mule-. Figure 2.3 shows that in the VM transport, the schema is named mule-vm.xsd. Some transports have several schemas, one per variation of the main transport (such as HTTP and HTTPS). The target namespace of the schema you're looking for is declared on the root element. Listing 2.2 shows the root element of the VM transport schema.

Listing 2.2 Each Mule library defines a specific target namespace in its schema.

```
<xsd:schema xmlns="http://www.mulesource.org/schema/mule/vm/2.2"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:mule="http://www.mulesource.org/schema/mule/core/2.2"
  targetNamespace="http://www.mulesource.org/schema/mule/vm/2.2"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified">
```

By now you should be anxious to know where the schema location is defined. Using the right location is important in order for the "redirection" mechanism to kick in and allow Mule to validate and load your configuration file. This mechanism is configured by the file named spring.schemas, which is located alongside the library schema. You can see this file in figure 2.3. The content of this file is a simple mapping between the remote schema location and the archive file to use for it, as shown:

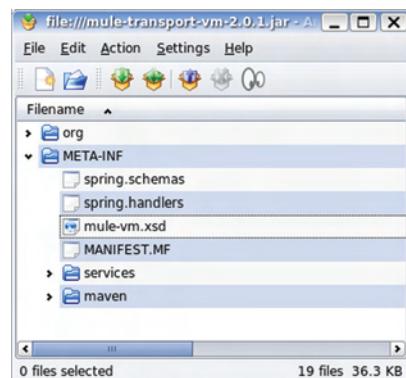
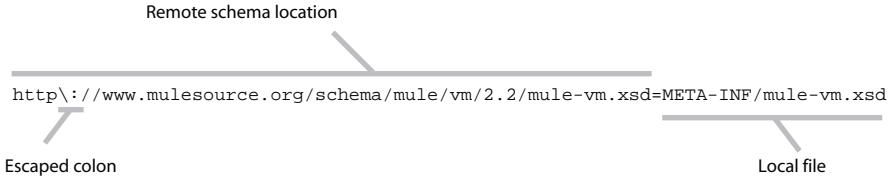


Figure 2.3 Locating the XML schema of the VM transport



The schema location you want to use is on the left of the equal sign. Bear in mind that because properties files require you to escape colons, the backslash in front of the equal sign should be omitted if you copy and paste the location directly into your configuration file. With the target namespace and the schema location in hand, you should now be able to add the VM transport schema to your configuration file. Listing 2.3 shows you what you should have come up with.

Listing 2.3 The root element of a Mule configuration that uses the VM transport

```
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"

      xsi:schemaLocation=" http://www.mulesource.org/schema/mule/core/2.2
                           http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
                           http://www.mulesource.org/schema/mule/vm/2.2
                           http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">
```

Defines VM transport namespace Location of VM transport schema

Throughout this chapter, we've referred to the Mule core schema. If you look for it in `mule-core.jar`, you'll be disappointed. It's in fact located in `mule-module-spring-config.jar`. Why there and not in the core JAR? Because using Spring XML is one possibility for building a Mule configuration. As we said in the introduction, there are other ways, such as scripting.

TIP *My valid configuration doesn't load!* It's possible to create a configuration that refers to all the right schemas and is well formed and valid, but still doesn't load. If this happens to you and you get a cryptic message like the following:

```
Configuration problem: Unable to locate
NamespaceHandler for namespace
[http://www.mulesource.org/schema/mule/xyz/2.2]
```

Then get ready for a "duh" moment. This means you've forgotten to add to your classpath the Mule library that defines the namespace handler required to load configuration elements in said namespace. Watch out for missing transport or module JARs.

By now, you're certainly eager to rush to your keyboard and start building your own configurations. Before that, we'd like to introduce the notion of modular configurations. This will help you tame the complexity that may arise in your configurations when your projects start expanding.

NOTE Consider also reading chapter 12, where we discuss the development tools you'll need, before starting a Mule project.

2.3 Configuration modularity

Have you ever had to wade through pages and pages of configuration, trying to sort things out and find your way around? As your integration projects grow and multiply, there's a risk that your configuration files may become bloated or redundant, hence hard to read, test, and maintain. Fortunately, the configuration mechanism of Mule allows you to relieve the ills of a monolithic configuration by modularizing it.

BEST PRACTICE Modularize your configuration files to make them easier to read and simplify their maintenance.

Cutting a configuration into several parts encourages the following:

- Reuse of common artifacts across several configurations
- Extraction of environment-dependent configuration artifacts
- Isolation of functional aspects that become testable in isolation

There are different approaches that can be used when modularizing a configuration. In the following sections, we'll detail these strategies:

- Independent configurations
- Inherited configurations
- Imported configurations
- Heterogeneous configurations

Of course, you can combine them together. At the end of the day, the objective of this discussion is to give you a hint of the possibilities and let you establish the configuration organization that best fits your project size and needs.

NOTE In 7.1.3, we'll discuss another handy option: using an existing Spring context as the parent context of your Mule configuration. Because this isn't done at configuration level, we won't detail this approach here.

2.3.1 *Independent configurations*

As shown in figure 2.4, a Mule instance can load several independent configuration files side by side. In this example, Mule will use two Spring XML configuration builders to instantiate, configure, and wire together the elements defined in both XML configuration files. As you can see, the environment-dependent properties have been exported in an external file, loaded from the instance-specific configuration file.

This approach is well suited for simple scenarios, where there's a loose coupling between the common and instance-specific elements. It allows a fair level of reuse, as global elements such as connectors, transformers, or endpoints can be shared with several instance-specific configurations.

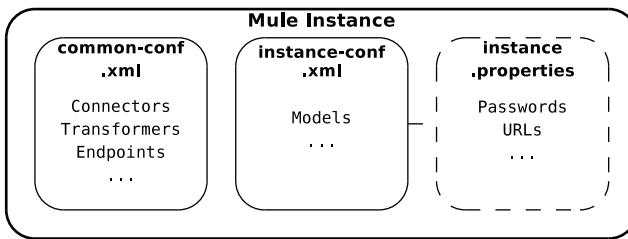


Figure 2.4 A Mule instance can load independent XML configuration files and properties files.

The main drawback of using independent configurations is that the instance-specific configuration doesn't strongly express its need to run alongside the common configuration. If the shared elements aren't used immediately, like transformers, you can start the Mule instance without the required common configuration and start having issues only later on at runtime.

Using inherited configurations alleviates this problem.

2.3.2 Inherited configurations

The concept of inherited configurations is illustrated in figure 2.5. The main idea is to express a formal parent-child dependency between two configurations. By strongly expressing this dependency, you'll have the guarantee at boot time that no configuration file has been omitted (unlike the behavior you get with side-by-side independent configurations as described in the previous section).

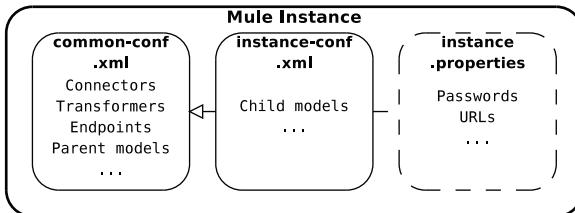


Figure 2.5 Models can be inherited, leading to enforced hierarchies of configuration files.

Inheritance can only be defined between models: this is why the parent and child models are represented. Because a model can be empty, this inheritance approach is suitable even if the common configuration doesn't have any model elements to share. It's also good to know that several different models can inherit from the same parent. The modularization can therefore span several configuration files.

How is this inheritance expressed? Simply by using the same name for the parent and child models and by flagging the child as being an heir, as shown here:

```
<model name="myConfig">
<model name="myConfig" inherit="true">
```

In this configuration sample, the parent model doesn't have an `inherit` attribute, which defaults to false when absent, and the child model has the attribute set to true.

2.3.3 Imported configurations

As briefly discussed in section 2.2.1, Spring can be leveraged to instantiate and configure any infrastructure or custom bean you need for your Mule instance. For example, Spring can perform JNDI lookups or wire your business logic beans together. After time, your Spring configuration may grow to the point that it starts to clutter your Mule configuration. Or you might decide that part of your Spring configuration is reusable. At this point, extracting your Spring beans to a dedicated application context file would be the right thing to do. Figure 2.6 shows a situation where such shared application contexts have been created and are used by a hierarchy of Mule configurations.

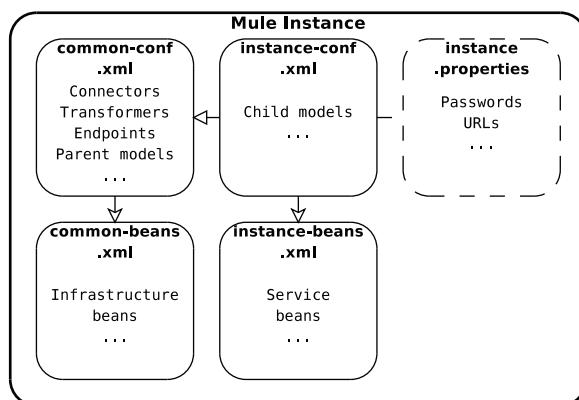


Figure 2.6 Imported Spring application context files can complement a hierarchy of Mule configurations.

You can easily import external Spring application context files from your Mule configuration files.⁴ The following illustrates how `instance-conf.xml`, shown in figure 2.6, would import its Spring context file:

```

<spring:beans>
  <spring:import resource="instance-beans.xml" />
</spring:beans>
  
```

Refer to Spring's documentation for more information on the usage of the `import` element.

2.3.4 Heterogeneous configurations

In the introduction to this chapter, we evoked the possibility of configuring Mule with a scripting language. The tedious work of setting up the core runtime environment and all the boilerplate code this requires is usually enough to put off the bravest script aficionado. The good news is that there's a solution. It's possible to mix several styles of Mule configuration in an instance. The example in figure 2.7 shows an instance that has been configured with a Groovy script and Spring XML configuration builders. By following this approach, you can declare all the global elements using the XML

⁴ Interactions between Mule and Spring configurations are further discussed in 7.1.3.

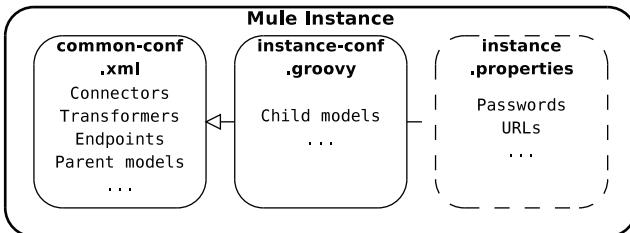


Figure 2.7 Mule can use heterogeneous builders to load a hierarchy of configuration files written in different syntax.

syntax and focus your scripted configuration on the services you need for your integration project.

Since scripted configuration works at the bare metal level, you express the dependency to a parent by looking up the model you want to depend on. If in figure 2.7, the common configuration defined a model named `commonModel`, the way the Groovy configuration would look it up and use it would be the following:

```
model = muleContext.registry.lookupModel("commonModel")
...
childService.model = model
```

As you can see, there's no child model *per se*. It's up to the service in the child model to actively register itself in the model defined in the common configuration.

We're sorry to report that after reading this section, you have no more excuses for building monolithic and kilometric configuration files. And if you've inherited such configurations, you should now have some pretty good ideas on how you could refactor them.

2.4 Summary

In this chapter, you've learned the general principles involved in configuring Mule. You've discovered the overall structure of the Spring XML configuration, its syntax, and the ways to define properties values. Several strategies for organizing your configuration files have been detailed: they'll allow you to grow your integration projects without getting lost in lengthy and monolithic XML files.

The scripted configuration mechanisms have been rapidly covered as well. It's up to you to decide on your syntax of choice for configuring Mule. By looking at the examples at the end of this chapter, you should have a fair idea of the efforts involved in the XML and scripted configuration options. In the rest of the book, all the examples will use the Spring XML syntax.

In many places, this chapter has referred you to other chapters. Without this crafty stratagem, this chapter would've had the size of the whole book. The coming chapters will look further into the details of the main Mule moving parts: endpoints, routers, transformers, and components. You'll learn about the main types of each and all the good they can do for you.

NOTE *Before we close...* We'd like to introduce you to Clood, Inc., our fictional but promising startup specializing in the domain of value-added services for businesses hosting applications in the cloud. Blessed with a double O, our startup is bound for a great future and a significant IPO. On the technical side of things, Clood, Inc., decided to leverage Mule as our integration platform to support all the services (monitoring, deployment controls, identity management, DNS, and so on) we intend to offer to our clients.

We're happy to inform you that, always looking for a challenge and some stock options, you've decided to join us on this cloudy adventure. Throughout the rest of the book, we'll review some of our activities at Clood, Inc., and how we're using Mule to accomplish them.

3

Sending and receiving data with Mule

In this chapter

- Understanding connectors and endpoints
- Common transports you'll use working with Mule
- Using web services with the CXF and HTTP transports

Can you remember the last trip you took? You probably used more than one means of transportation to get to where you were going. You may have taken a passenger jet somewhere first, then hopped in a taxi. Or you may have ridden a bike somewhere, locked it up, and then walked across the street. Perhaps you parachuted out of a plane, then drove home. In any case, you likely weren't teleported from one place to another.

Transporting data isn't much different than transporting people. While it would be nice if systems could magically move data between themselves, this isn't often

the case. One system might only support SOAP, while your system may only supports REST. Perhaps you need to get data from an FTP site into a database. Or maybe you need to receive JMS messages and save their payloads to a file.

We introduced Cloud, Inc., at the end of the last chapter. In this chapter, and throughout the rest of this book, we'll be using Cloud's integration activities as a reference point for some of our discussions and examples. Being a modern, cloud-based service provider, Cloud is forced to integrate with a variety of disparate systems and services as part of its daily business. Cloud also makes extensive internal use of open source technologies and tools. Let's start off by considering an integration scenario recently faced by Cloud.

Cloud, Inc., uses Nagios, a popular open source monitoring tool, to perform URL monitoring of its customers' cloud-hosted applications. When a URL check fails, Nagios sends an email to the Cloud call center where subsequent action is taken. You've been tasked to store these alerts in a database so management can run reports on the data—with the stipulation that you can't modify the Nagios installation. Since the reports will only be run once a day, you decide your best approach is to periodically download messages from the mail server, perform some analysis on them, and save the results into a relational database. Without using Mule, you might consider the following strategy:

- Implement an IMAP client using the Java Mail API.
- Schedule the client to download the messages at some interval.
- Write code to parse the email message.
- Use JDBC to insert the data into the database.

This approach will work, but an awful amount of effort is spent “plumbing”—writing things such as IMAP clients, JDBC clients, and schedulers. Wouldn't it be nice to worry about the business problem at hand—parsing the email data—and let Mule handle the details of moving the data around?

We'll see in this chapter how Mule reduces this plumbing work to a few lines of XML. We'll cover the major transports Mule supports and provide working configurations of them in action. In each section, we'll discuss a specific transport, such as SOAP or email, and examine how you can use it to send and receive data.

As moving data around is such an important part of using Mule, expect to make heavy use of the techniques in this chapter. After all, you'll need to get data into and out of Mule before you can do anything useful with it!

3.1 ***Understanding connectors and endpoints***

Mule connectors are a lot like the plane, car, bike, and parachute in our earlier examples—they enable you to get data from one place to another. And odds are you'll use more than one means of transport to get to where you're going. Connectors provide an abstraction layer over data transport mechanisms. Connectors exist for things such as files, email messages, databases, JMS, and even Jabber messages. A connector saves you the tedium of having to implement the details of a particular communication

mechanism yourself. This allows you to focus on solving your integration problem and not on the plumbing of a particular communications protocol.

The connectors bundled with Mule should cover the bulk of your data transport needs. If you're working with a protocol that isn't implemented in the core distribution, you can look at the MuleForge and see if there's a community contributed connector you can use. You also have the option of implementing your own transport using the Mule APIs. Let's take a closer look at connectors and endpoints, and then see how we can use them to simplify our IMAP integration project.

3.1.1 Configuring connectors

Your Mule configuration will often contain one or several connector configuration elements. Each transport that you use will contribute its own connector element with specific attributes. For example, here's a typical STDIO connector configuration:

```
<stdio:connector name="SystemStreamConnector"
                  promptMessage="Please enter something: "
                  messageDelayTime="1000" />
```

And here's a secure proxied HTTP connector configuration:

```
<http:connector name="HttpConnector"
                 proxyHostname="${proxyHostname}"
                 proxyPort="${proxyPort}"
                 proxyUsername="${proxyUsername}"
                 proxyPassword="${proxyPassword}" />
```

It's absolutely possible to have a configuration that doesn't contain any connector configuration. Why's that? If Mule figures out that one of your endpoints needs a particular connector, it'll automatically instantiate one for you, using all the default values for its different configuration parameters. This is a perfectly viable approach if you're satisfied with the behavior of the connector when it uses its default configuration. This is often the case for the VM or HTTP transports. Note that Mule will name these default connectors with monikers such as `connector.http.0`.

On top of that, a connector has a technical configuration also known as the *Transport Service Descriptor* (TSD). This hidden configuration is automatically used for each instance of the connector. It defines technical parameters such as what classes to use for the message receivers, requesters, and dispatchers; or the default transformers to use in inbound, outbound, and response routers. Knowing these default values is essential to grasping the behavior of a transport. Though documented on the Mule-Source web site, it's good to learn to locate the TSD yourself so you can check what's actually in the library you're using.

Let's now hunt for this hidden configuration file. For this, use your favorite archiving utility or IDE and open the JAR file of the transport you want to use (here, we picked the JMS transport JAR). Open the META-INF/services/org/mule/transport directory. After opening it, you should see something similar to the screen shot shown in figure 3.1.

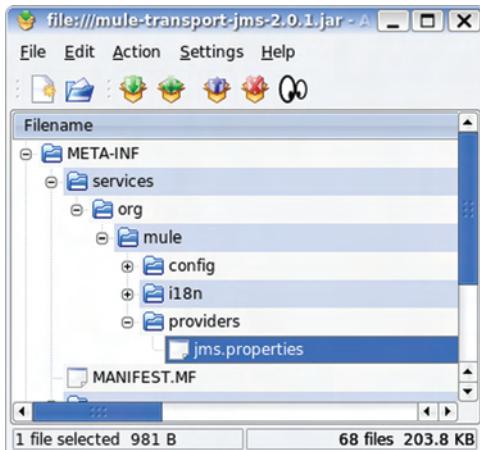


Figure 3.1 Finding the Transport Service Descriptor of the JMS transport

The TSD file is named after the protocol supported by the transport. In the case of JMS, it's then named `jms.properties`. Listing 3.1 shows you the content of this file for the JMS transport. Note the different factories and transformers used by the connector. There are often lines commented out with a hash mark (#), which lines often represent valid configuration alternatives.

Listing 3.1 The TSD for the JMS transport

```
connector=org.mule.transport.jms.JmsConnector
dispatcher.factory=\n    org.mule.transport.jms.JmsMessageDispatcherFactory
requester.factory=\n    org.mule.transport.jms.JmsMessageRequesterFactory
message.receiver=\n    org.mule.transport.jms.MultiConsumerJmsMessageReceiver
transacted.message.receiver=\n    org.mule.transport.jms.MultiConsumerJmsMessageReceiver
xa.transacted.message.receiver=\n    org.mule.transport.jms.XaTransactedJmsMessageReceiver
message.adapter=org.mule.transport.jms.JmsMessageAdapter
inbound.transformer=\n    org.mule.transport.jms.transformers.JMSMessageToObject
response.transformer=\n    org.mule.transport.jms.transformers.ObjectToJMSMessage
outbound.transformer=\n    org.mule.transport.jms.transformers.ObjectToJMSMessage
endpoint.builder=\n    org.mule.endpoint.ResourceNameEndpointURIBuilder
```

By default, as shown in listing 3.1, the JMS transport will apply transformers to all messages in order to extract their payload, ensuring that what travels inside of Mule is transport neutral. Suppose you now want to configure a Mule instance that deals with a lot of JMS destinations and whose components need to receive JMS messages and not

their payload. The best option is to override the Transport Service Descriptor and use a “no action” transformer (a typical “null object”). This is achieved as shown:

```
<jms:connector name="nativeJmsConnector">
    <service-overrides>
        inboundTransformer="org.mule.transformer.NoActionTransformer"
        outboundTransformer="org.mule.transformer.NoActionTransformer"
        responseTransformer="org.mule.transformer.NoActionTransformer"
    />
</jms:connector>
```

As you can see, the entries in the TSD file have their counterparts as attributes on the service-overrides element.

NOTE When you have only one connector for a particular protocol, whether it's a default connector automatically created by Mule or one that you've specifically configured, you don't need to add a reference to the connector name in your endpoint for that transport. But as soon as you have more than one connector for a particular protocol, any endpoint that uses this protocol will prevent Mule from loading your configuration and an exception containing the following message will be thrown: “There are at least 2 connectors matching protocol “xyz”, so the connector to use must be specified on the endpoint using the “connector property/attribute.” The message is self-explanatory and the remedy trivial: simply add a reference to the particular connector name on each endpoint that uses the concerned protocol. For example:

```
<vm:endpoint
    connector-ref="myVmConnector" name="LoanBrokerQuotes"
    path="loan.quotes"
/>
```

Now that we've examined connectors, let's see how endpoints are configured.

3.1.2 Configuring endpoints

An endpoint is a specific utilization of a connector. If a connector is something general like flying on a jet, then an endpoint is something concrete, like flying JetBlue on Flight 123 from JFK to SJC. Endpoints are the cornerstone of I/O in Mule; they're the tools you'll use to get data through your transformers, into your routers, and back and forth from your components.

Endpoints come in two different flavors: inbound and outbound. Inbound endpoints are used to accept data; outbound endpoints are used to send data. An inbound endpoint can do things such as receive SOAP messages, read file streams, and pull down email messages. An outbound endpoint is used to do things such as send SOAP messages, write to file streams, and send email messages. You'll use inbound and outbound endpoints to communicate between components and services inside Mule as well as with the outside world.

Whenever you have an endpoint destination that's shared by several routers, it's worth creating a global endpoint. A global endpoint is not typified for inbound or

outbound routing, making it usable in many different places in a configuration file. It must be named so it can actually be used in a service, which will reference the global endpoint by its name. Because it bears a name, a global endpoint can also help clarify the usage of a particular destination. This is illustrated by the following example, which demonstrates an in-memory (VM) endpoint whose name is more human-friendly than its actual path:

```
<vm:endpoint name="LoanBrokerQuotes" path="loan.quotes" />
```

Though pretty clear, the path attribute is less informative than the name used for this VM endpoint.

A global endpoint doesn't activate any connectivity. Think about it as a factory for creating "live" inbound or outbound endpoints in routers. For example, the preceding global endpoint would become an active inbound endpoint only if the following is used in an inbound router:

```
<inbound-endpoint ref="LoanBrokerQuotes" />
```

You'll quickly notice that all endpoints offer an address attribute. Why is that? This allows you to configure a generic endpoint using the Mule 1.x style of URL-based destination addresses instead of the dedicated attributes of the specific endpoint element. You'll seldom have to use this mechanism, but for your personal enlightenment, here are two strictly equivalent global endpoint definitions:

```
<file:endpoint name="tmpPoller"
    path="/tmp"
    fileAge="1000"
    pollingFrequency="2000"
/>
<endpoint name="tmpPoller"
    address="file:/mp?fileAge=1000&pollingFrequency=2000"
/>

```

You should prefer using the specific form for endpoints, as you'll gain in expressiveness and type safety.

TIP *Who cares about endpoint URIs?* Unless you use the scripted configuration builder, you may well never have to write an endpoint URI. But you'll be exposed to this syntax, and for different reasons. For example, Mule uses this representation in its log files, as it's compact and informative. For example, the following log entry tells you that a dispatcher for the STDIO transport has been connected:

```
INFO 2008-06-28 15:52:46,697 [SystemStreamConnector.dispatcher.1]
org.mule.transport.stdio.StdioMessageDispatcher:
Connected: StdioMessageDispatcher{this=15c97e4,
    endpoint=stdio://system.out, disposed=false}
```

Can you spot the URI? You're right; it's `stdio://system.out`. This syntax is also a convenient means to pass extra parameters to the element you configure in case no specific attribute exists. So if you're not familiar with

this notation, take a look at RFC-2396, which is titled “Uniform Resource Identifiers (URI): Generic Syntax”. If you’re having a hard time reading plain-text 80-column documents (shocking!), then you can read the JavaDoc of the `java.net.URI` class. You’ll get a good idea of the different parts that compose a URI.

Let’s look at an example of how connectors and endpoints work together. Listing 3.2 shows the IMAP-to-database problem from the beginning of this section implemented using Mule transports.

Listing 3.2 Parsing and sending email contents to a database

```

<spring:beans>
    <spring:import resource="spring-config.xml" /> ①
</spring:beans>

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="statsInsert"
        value="insert into alerts values
            (0,#[map-payload:HOST],
             #[map-payload:MESSAGE],
             #[map-payload:TIMESTAMP])"/> ②
</jdbc:connector>

<model name="URLAlertingModel">
    <service name="URLAlertingService">
        <inbound>
            <imap:inbound-endpoint host="mail.clood.com" user="mule" ③
                password="password">
                <email:email-to-string-transformer/> ④
            </imap:inbound-endpoint>
        </inbound>
        <component class="com.clood.monitoring.URLAlertComponent"/> ⑤
        <outbound>
            <pass-through-router>
                <jdbc:outbound-endpoint queryKey="statsInsert"/> ⑥
            </pass-through-router>
        </outbound>
    </service>
</model>
```

There we go; all the tedious work of writing an IMAP client, implementing a scheduler, and writing the JDBC client code has been reduced to a few lines of XML! We’ll be digging into the gory details of the IMAP and JDBC transports later on in this chapter. For now, let’s take a high-level look at what’s going on.

We first declare an external Spring configuration on ①. The Spring configuration is defining resources external to our Mule config—in this case the `dataSource` for our JDBC connector. The JDBC connector, along with the corresponding SQL insert we’ll use, is defined on ②. As you can probably infer, it’ll be used later on to insert a row of data into the `alerts` table. The data we’ll ultimately insert is coming from email messages on Clood’s IMAP server, which we’re connecting to using the IMAP inbound

endpoint configured on ③. When an email is received, it's converted to a String by the email-to-string transformer defined on ④. This transforms the email from a potentially tedious javax.mail.Message instance to something easier to deal with—a String. This String is passed to the component defined on ⑤, which in this case is a custom class you've implemented. The class, com.cloud.monitoring.URLAlertComponent, performs the business logic at hand: extracting relevant alert details from the mail message to persist in our database on ⑥.¹

As you can see, the level of effort for our integration has been substantially reduced. We've gone from having to work with some particularly hairy APIs to simply implementing our business logic and writing some XML. We'll see many more examples in this chapter where this is the case.

NOTE Endpoints will add headers, called *properties*, to a message as it's sent and received. Prefixed with MULE_, these properties contain metadata about each message. Some, such as MULE_MESSAGE_ID, are added to all messages processed by Mule. Other properties are specific to the underlying transport. If the underlying transport supports the notion of message headers, like JMS or SOAP, the MULE_ properties will be propagated to the protocol's header mechanism. You can see this if you look at the headers of JMS or SOAP messages once they leave Mule through an outbound endpoint.

In this section you learned how Mule abstracts I/O operations. We saw how connectors and endpoints work together to let you move data into and out of components. These are core concepts you'll need for the rest of your exploration into Mule's capabilities. Now let's dig in and look at the different transports Mule supports. As we've mentioned, Mule ships with a wealth of transports to ease your integration efforts. We'll start off by looking at how Mule works with file input and output.

3.2 **Working with files and directories using the file transport**

Performing input and output on files and directories is a core activity of most applications. Despite the availability of integration technologies such as SOAP and JMS, you'll often find yourself reading files from directories, parsing CSV files, and saving data to disk.

In this section we'll look at Mule's file transport and explore how you can use it to simplify your file and directory demands. We'll first look at how you can use the file transport to move files from one directory to another. We'll then look at how you can use filters to be selective about what sort of files you want to process. Finally we'll look at Mule's STDIO support, which allows you to interact with the input and output streams of your application.

¹ We're deliberately glossing over the details on transformations, components, and routing—all will be covered in depth in the next three chapters. For the remainder of this chapter we'll make sparing use of transformation and try to only use the implicit bridge component and pass-through routers, which simply pass data from inbound to outbound endpoints.

Configuring the file transport is fairly straightforward. Some of the configuration properties are listed in Table 3.1. You can see that they govern which directories files are read from, whether or not they’re deleted, how often a directory is polled, and the patterns to use when moving files.

Table 3.1 Configuring the file transport

| Property | Type | Target | Description |
|-------------------|---------|------------------------------|--|
| writeToDirectory | String | connector, outbound endpoint | The target directory for file output. |
| readFromDirectory | String | connector, inbound endpoint | The source directory for file input. |
| autoDelete | boolean | connector, inbound endpoint | Whether to delete the source file after it has been read. |
| outputAppend | boolean | connector, outbound endpoint | Specifies that the output is appended to a single file instead of being written to a new file. |
| pollingFrequency | long | connector, inbound endpoint | Specifies the interval in milliseconds that the source directory should be polled. |
| moveToDirectory | String | connector, inbound endpoint | The directory to move a file once it has been read. If this and autoDelete are unset then the file is deleted. |
| moveToPattern | String | connector, inbound endpoint | The pattern used to indicate what files should be moved to the <code>moveToDirectory</code> files not matching this pattern are deleted. |
| outputPattern | String | connector, outbound endpoint | The pattern used when creating new files on outbound endpoints. |
| streaming | boolean | connector | Specifies whether the file data should be streamed. |

3.2.1 Reading and writing files with file endpoints

Let’s consider another integration scenario encountered by Clood, Inc. Your boss and his cohorts are happily generating reports using the data we created in listing 3.2. A web-based business intelligence tool is being used to generate the reports. The BI tool is a bit flaky, though: occasionally it’ll crash and not save the state of the various reports that had been defined. After some research, you discover a switch to make the tool save a snapshot of its configuration to a specified directory. You enable this and realize either a glaring omission or bug with the software—the snapshots are all identically named, resulting in each snapshot being overwritten by each new snapshot! As you want to keep historical configuration snapshots, this is highly suboptimal. Additionally, you don’t have access to the source code of the BI tool, nor is your

support contract with the software provider up to date. Rather than face the prospect of being asked to revert to a configuration you no longer have, you decide to try to use Mule to get a backup of each file as it's written to disk.

Listing 3.3 demonstrates using the file transport to move files from one directory to another. In this case, we'll be polling a backup directory for new snapshot files every second, then immediately moving the snapshot file to an archive directory with a timestamp appended.

Listing 3.3 Using the file transport to move files from one directory to another

```
<file:connector name="FileConnector"
    streaming="false"
    pollingFrequency="1000">
    <file:expression-filename-parser/>
</file:connector>

<model name="fileModel">
    <service name="fileService">
        <inbound>
            <file:inbound-endpoint path=".//data/snapshot" />
        </inbound>
        <outbound>
            <pass-through-router>
                <file:outbound-endpoint path=".//data/archive"
                    outputPattern="SNAPSHOT-#[function:dateStamp]" />
            </pass-through-router>
        </outbound>
    </service>
</model>
```

The global properties for the file connector are set up first in ①, ②, and ③. Since we're concerned with the actual file itself, and not a stream of its data, we set the streaming property of the file connector to false on ①. We next need to set how often we want file inbound endpoints to poll their source directories for new content. This is accomplished by setting the pollingFrequency to 1000 milliseconds on ②. The filename parser is set on ③ to use the expression-filename-parser—we'll see how this is used in a moment. On ④ we define the file inbound endpoint. This will poll the .//data/snapshot directory every second for new files. When a new file is detected, it's sent to the file outbound endpoint defined on ⑤ and removed from the source directory.

A file inbound endpoint will, by default, remove the file from the source directory once its read by the inbound endpoint. You can override this behavior by setting the autoDelete property on the file connector to false. Be careful when doing this, though, as this will cause the file to be repeatedly read by a file inbound endpoint until it's removed from the source directory. You can control how aggressively Mule reads a file by setting the fileAge property on the file connector. The fileAge property specifies how long the endpoint should wait before reading the file again. For

instance, a `fileAge` of 60000 indicates Mule should wait a minute before processing the file again.

BEST PRACTICE Exercise caution when using the `autoDelete` property of the file connector.

Once the file has been sent to the pass-through router by the inbound endpoint, the file outbound endpoint will place the file in the `./data/archive` directory. The resulting file will be named using the supplied `outputPattern`, which is parsed according to the filename parser defined on ③. In this case, we're using the `expression-file-name-parser`. This allows you to use Mule expressions to control how files are named by the file transport.

NOTE In addition to setting `streaming` to `false`, you can also use a `FileAdaptor` to obtain a reference to the file object and not a byte stream. This is done by setting a service-override on the file connector as follows:

```
<file:connector name="FileObjectConnector">
    <service-overrides
        messageAdapter="org.mule.transport.file.FileMessageAdapter"
    />
</file:connector>
```

The Mule expression language is covered in depth in appendix A. The following list details some examples of using Mule expressions in filenames. As you can see, you can embed date stamps, UUIDs, and the original filename in the final filename:

- `function:dateStamp`—The date stamp of the current time
- `function:uuid`—A universally unique identifier
- `header:originalFilename`—The original name of the file

Now let's look at how we can control which files the inbound endpoint will process.

3.2.2 Using filters on inbound file endpoints

Sometimes you want to be selective about which files are processed by an inbound file endpoint. Perhaps you're interested in moving XML or JSP files from one directory to another while ignoring everything else. Mule allows endpoints to be particular about the data they send and receive using filters. A *filter* is a mechanism for controlling which data the endpoint is concerned with. Much like the filter in a coffee pot, it lets some things go through while blocking everything else.

Let's revisit our BI configuration backup solution. This has been working great so far, but inspecting the archive directory reveals some oddly named files. It seems that some temporary files are being written to the snapshot directory along with the snapshots. The only files we're interested in will match the pattern `SNAPSHOT*.xml`, so let's adjust listing 3.3 accordingly. Listing 3.4 accomplishes this.

Listing 3.4 Moving only certain files from one directory to another

```

<file:connector name="FileConnector"
    streaming="false"
    pollingFrequency="1000">
    <file:expression-filename-parser/>
</file:connector>

<model name="fileModel">
    <service name="fileService">
        <inbound>
            <file:inbound-endpoint path=".//data/snapshot">
                <file:filename-wildcard-filter
                    pattern="SNAPSHOT*.xml"/>
            </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <file:outbound-endpoint path=".//data/archive"
                    outputPattern=
                        "##[header:originalFilename]-#[function:dateStamp].xml"
                />
            </pass-through-router>
        </outbound>
    </service>
</model>

```

We've added a filename wildcard filter to the inbound endpoint at ①. As you can see from the pattern attribute, the filter will only pass files that start with *SNAPSHOT* and end with an *xml* extension. We've additionally modified the *outputPattern* at ② to preserve the original filename for the output files, while still appending the timestamp. In this case, the resulting files would be named like this: *SNAPSHOT-04-09-08_18-37-33.417.xml*.

TIP Remember that, by default, a file inbound endpoint will delete the source file after it has been read. If you want to keep the file, be sure to set *autoDelete* to false on your inbound endpoint or connector! Keep in mind, though, that the file will be read again at the next polling interval. This is useful in certain circumstances.

Closely related to file input and output is console input and output. While Mule is typically used in a noninteractive fashion from the standpoint of the console, for testing and debugging, it's often useful to be able to accept input from a keyboard or print information out on the screen. Let's look at the shortcuts Mule provides for performing this sort of I/O.

3.2.3 Using STDIO endpoints

In order to facilitate testing and debugging, Mule offers access to the standard I/O streams as inbound and outbound endpoints. This can be invaluable to quickly get data into and out of endpoints. Listing 3.5 copies input entered on the input stream to the output stream.

Listing 3.5 Echoing input from STDIN to output on STDOUT

```
<model name="stdioModel">
    <service name="stdioService">
        <inbound>
            <stdio:inbound-endpoint system="IN" />
        </inbound>
        <outbound>
            <pass-through-router>
                <stdio:outbound-endpoint system="OUT" />
            </pass-through-router>
        </outbound>
    </service>
</model>
```

We'll see further use of these endpoints in this chapter as we demonstrate the rest of Mule's I/O capabilities.

We've just taken a look at how you can use Mule's file transport to perform file and directory operations. We also saw how the STDIO transport enables you to get access to Mule's console to perform input and output. These are all important means of getting data into and out of Mule, but your integration demands will often be more complicated than working with files and keyboard input. Let's take a further look at Mule's transport options, starting with sending and receiving email.

3.3 Using email

It's hard to escape email messaging in the modern enterprise. While it's tempting to think of email as primarily for person-to-person communication, it's often used for more than friendly conversation. Email messages relay monitoring alerts, send order receipts, and coordinate scheduling. This is evident from the automated emails you receive when signing up to a web site, ordering a book from Amazon, or confirming a meeting request.

In this section, we'll investigate how to use the email transports to act on and generate these messages. First we'll take a look at receiving email with the IMAP transport. Then we'll look at sending email with the SMTP transport.

3.3.1 Receiving email with the IMAP transport

The IMAP connector allows you to receive email messages from a mail server using IMAP. IMAP, the Internet Message Access Protocol, is the prevailing format for email message retrieval—supported by most email servers and clients. IMAP can be a convenient means of interacting with applications that don't supply more traditional integration mechanisms. An example could be a legacy application that generates periodic status emails but doesn't offer any sort of programmatic API. Email can, at other times, be a preferred means of application interaction. You may need to programmatically react to email confirmations, for instance. In that case, using an IMAP endpoint is a natural fit.

Table 3.2 lists some of the more common configuration properties of the IMAP transport.

Table 3.2 Common configuration properties of the IMAP transport

| Property | Type | Target | Description |
|--------------------|---------|-----------------------------|--|
| mailboxFolder | String | connector | The IMAP folder to read messages from. |
| backupFolder | String | connector, inbound endpoint | If backupEnabled is true, the directory on the local disk to backup messages to. |
| backupEnabled | boolean | connector, inbound endpoint | Whether or not to store copies of read email messages. |
| deleteReadMessages | boolean | connector | Whether or not to delete messages from the mail server once they've been read. If false, the messages are marked as SEEN on the mail server. |
| checkFrequency | Integer | connector | Interval at which to poll the server for new messages. |

Let's reconsider the IMAP example we used in the beginning of this chapter. Clood's IMAP server is taking quite a beating. The call center and Mule are accessing the same IMAP folder every few minutes for new data—much to the chagrin of the mail administrator. To help remedy the issue, the mail administrator has set up a filter that moves only the URL alert messages to a folder for us to access. Since our data isn't terribly time sensitive, she additionally asked that we only pull down email once an hour. Listing 3.6 modifies our configuration to comply with these requests.

Listing 3.6 Explicitly configuring an IMAP endpoint's check frequency and folder

```

<spring:beans>
    <spring:import resource="spring-config.xml" />
</spring:beans>

<imap:connector name="imapConnector"
    checkFrequency="3600000"
    mailboxFolder="MULE_MESSAGES"
    deleteReadMessages="true" />
    
    1 Configures an IMAP inbound endpoint

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="statsInsert"
        value="insert into alerts values
            (0,
                #[map-payload:HOST],
                #[map-payload:MESSAGE],
                #[map-payload:TIMESTAMP]) />
</jdbc:connector>

<model name="URLAlertingModel">
    <service name="URLAlertingService">
        <inbound>
            <imap:inbound-endpoint host="mail.clood.com" user="mule"
                password="password">

```

```

<email:email-to-string-transformer/>
</imap:inbound-endpoint>
</inbound>
<component class="com.cloud.monitoring.URLAlertComponent"/>
<outbound>
    <pass-through-router>
        <jdbc:outbound-endpoint queryKey="statsInsert"/>
    </pass-through-router>
</outbound>
</service>
</model>

```

As the changes we've been requested to make are only allowed on an IMAP connector, we need to explicitly define one on ①. We're setting the `checkFrequency` to an hour (in milliseconds) and telling the connector to only poll the `MULE_MESSAGES` folder. Since Mule is the only reader of this folder, we're also setting the `deleteReadMessages` property to true. This will remove messages from the server as we read them, conserving disk space and making our mail administrator happy.

BEST PRACTICE Be cautious when using the `backupFolder` and `backupEnabled` properties. Mule will create a file for each email message it processes. This can quickly lead to filesystem issues such as inode exhaustion in mail-heavy environments.

Now that we've seen a few examples of how to read email messages, let's see how we can send them using the SMTP transport.

3.3.2 Sending mail using the SMTP transport

The SMTP transport lets you use an outbound endpoint to send email messages. This is useful in a variety of situations. We've already seen how an alerting system uses SMTP to send alert data. You've also no doubt received automated emails confirming purchases, forum subscriptions, and so forth. Sending email is also useful to perform notification of the completion of some long-running process—such as a notification that a backup was successful. Let's take a look at how to configure SMTP connectors and endpoints.

Table 3.3 lists the common properties for configuring the SMTP transport. In particular, the properties allow us to configure how the headers of the outbound emails are generated.

Table 3.3 Common configuration properties of the SMTP transport

| Property | Type | Target | Description |
|--------------------------|--------|---------------------|------------------------------|
| <code>to</code> | String | connector, endpoint | The recipient of the message |
| <code>subject</code> | String | connector, endpoint | The default message subject |
| <code>fromAddress</code> | String | connector | The from address |

Table 3.3 Common configuration properties of the SMTP transport (continued)

| Property | Type | Target | Description |
|------------------|--------|-----------|--|
| from | String | endpoint | The from address |
| replyToAddresses | String | connector | A comma-separated list of reply-to addresses |
| replyTo | String | endpoint | A comma-separated list of reply-to addresses |
| ccAddresses | String | connector | A comma-separated list of cc addresses |
| cc | String | endpoint | A comma-separated list of cc addresses |

Let's consider another example from our friends at Cloud, Inc. The accounting department occasionally receives invoices via FTP to an internal server. Accounting knows when these files arrive and it's usually the responsibility of you, or someone on your team, to get the invoice to them. Besides the fact that this is tedious (and error prone), Accounting has recently been less attentive to letting you know when to expect these files. They've also become increasingly irate when the invoices aren't delivered in a timely fashion. As such, automatically emailing the invoices as they arrive seems like the best bet. Listing 3.7 illustrates how to use an SMTP endpoint to accomplish this.

Listing 3.7 Using an SMTP endpoint to email an invoice

```
<model name="smtpModel">
    <service name="smtpService">
        <inbound>
            <file:inbound-endpoint path=".//data/invoice">
                <file:file-to-string-transformer/>
            </file:inbound-endpoint> </inbound>
        <outbound>
            <pass-through-router>
                <smtp:outbound-endpoint host="localhost"
                    from="mule@cloud.com"
                    subject="Accounting Invoice"
                    to="accounting@cloud.com">
                    <email:string-to-email-transformer/>
                </smtp:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

① **Configure file inbound endpoint**

② **Convert file to String**

③ **Configure SMTP outbound endpoint**

④ **Convert String to email message**

The file inbound endpoint configured on ① will pick up the invoices as they arrive at the target directory. This should seem familiar from listing 3.3, with the exception of the file-to-string transformer on ②. The file-to-string transformer is going to convert

the contents of the file invoice to a string. Once this occurs, the string is sent for delivery through the SMTP outbound endpoint. Before the email is sent, though, the string passes through the string-to-email transformer. Here the String becomes the body of an email message that's subsequently delivered to account@cloud.com.²

We just looked at how to receive email messages with the IMAP transport and send email messages with the SMTP transport. We saw how to augment the example from the beginning of this chapter to control how messages are downloaded from an IMAP server. We also saw how to use an SMTP endpoint in conjunction with a file endpoint to automatically send emails as new files are created in a directory.

Email usually isn't the best way to drive integration between applications. For one thing, it's hard to make inferences about the data in an email message. For instance, while two mail messages may be MIME encoded, one might contain XML while the other may contain a PDF attachment. Another complication with using email as a data transport is that the underlying protocols make no guarantee about the timelines or reliability of the message delivery. In the next two sections we'll be looking at Mule's web services and JMS connectors, which attempt to meet these needs.

3.4 Using web services

XML-based web services technologies provide an attractive means of consuming and exposing enterprise data. XML documents support a variety of schema languages to enforce document conformity. Most languages and platforms provide rich support for XML parsing, and various tools exist to map internal data models to XML schemas. HTTP, being a ubiquitous and firewall-friendly transport protocol, is a natural way to pass these documents around.

Mule has an assortment of functionalities for consuming, publishing, transforming, and parsing XML-based web services. In this section we'll start off by looking at Mule's support for web services using SOAP. Then we'll take a look at using Mule's HTTP transport, where you can use the language of the Web to exchange documents.

NOTE The examples in this chapter assume the use of XML web services. This doesn't imply that XML is the only means of defining web services data. Although SOAP is XML-centric, the HTTP transport can work with whatever data format you wish. JSON, in particular, is emerging as a practical alternative to XML; especially when working with rich, browser-based applications.

3.4.1 Consuming and exposing SOAP services with the CXF transport

The SOAP connector allows you to consume and expose SOAP services. To accomplish this, Mule employs CXF (<http://cxf.apache.org/>), a mature, document-driven web-services framework. As you'll see in the following sections, CXF provides a rich environment for working with SOAP services and messages.

² Just another reminder that we'll be covering transformation in detail in chapter 5.

Exposing and consuming SOAP services are sufficiently different enough that we're going to forego the usual discussion of common configuration steps and jump right into the endpoint details. First we'll look at how you can use the WSDL outbound endpoint to consume SOAP services. Then we'll look at how you can expose your service beans for SOAP consumption on an inbound endpoint.

If you've worked with SOAP before, you've probably encountered a WSDL. A WSDL is a description of the operations supported by a SOAP service. WSDLs are particularly useful for programmatically operating on web services. The WSDL outbound endpoint takes advantage of this fact.

Listing 3.8 demonstrates the WSDL endpoint by providing an interface to a web service that returns stock quotes. We'll be reading in a stock ticker name from the STDIO inbound endpoint and passing it to the GetQuote method of the stockquote service.

Listing 3.8 Invoking a remote web service using a WSDL outbound endpoint

```

<stdio:connector name="SystemStreamConnector"
    promptMessage="Please enter a stock ticker: "
    messageDelayTime="1000" />

<model name="cxftestModel">
    <service name="cxftestService">
        <inbound>
            <stdio:inbound-endpoint system="IN" synchronous="true" />
        </inbound>
        <outbound>
            <chaining-router>
                <outbound-endpoint
                    address=
"wsdl-cxf:http://www.webservicex.net/stockquote.asmx?\nWSDL&method=GetQuote"/>
                    <stdio:outbound-endpoint system="OUT" />
                </chaining-router>
            </outbound>
        </service>
    </model>

```

First off we'll configure an inbound endpoint on ①. This is where we'll be accepting the stock ticker symbol (such as IBM or JAVA) and using it as the payload for the GetQuote method. As you can see, we've marked the inbound endpoint as synchronous. An endpoint can support two types of message exchange: asynchronous and synchronous. Asynchronous message exchange assumes that your endpoint "fires and forgets" the message it sends. This is the behavior you want if the message you're sending either doesn't have a reply or the reply doesn't matter to you. An example of this might be a JMS message you place on a queue or a row you insert in a database. An endpoint is declared synchronous, as in ①, if the response is important. This could either be because you want to return the response to a client or you want the inbound endpoint to block while the current message is being processed. The latter is the case in this example: we want the STDIO inbound endpoint to block on new input until a response is received and output to the screen by the chaining router.

You might have noticed that we're using a different outbound router than the previous examples. The chaining router defined on ② indicates that the output of each endpoint will be used as the input into the subsequent endpoint (we'll discuss the chaining router in greater detail in the next chapter). In this case, we'll be taking the response from the web service and displaying it on the output stream. Once this is done, the inbound endpoint will prompt the user for a new ticker.

The web services acrobatics are performed by the outbound endpoint configured on ③. Mule will accept the quote provided by the inbound endpoint, marshal a SOAP request dynamically from the WSDL, and send it to the remote service. The XML response will then be passed to the STDIO endpoint and displayed on your console. The response, for now, will be raw XML as returned from the service. We'll see in chapter 5 how we can transform this data into a friendlier format.

As you can see, we're using the explicit endpoint URI syntax described in chapter 2 to specify the outbound endpoint. This is because at the time of writing, no schema definition for the `cxf-wsdl` endpoint exists. This will hopefully be rectified in the near future.

Let's look at how we can expose some business logic as a SOAP inbound endpoint. The CXF inbound endpoint works a bit differently than the previous endpoints we've seen; it'll use the service's component to dynamically generate a WSDL containing the operations of your SOAP service. We discuss components in depth in chapter 6, but you might recall from the first example in this chapter that a component is simply a POJO that implements some piece of business logic or message enrichment.

We'll discuss two options for exposing your component as a web service. The first option enables you to expose any POJO as a WSDL using sensible defaults. This is typically the easiest way to go, but sometimes you want to take greater control over how your WSDL and XSD are generated. If this is the case, you can leverage CXF's JAXB and JAX-WS bindings to obtain fine-grained control over the presentation of your web service. We'll look at using the simple front end first; then we'll investigate how to annotate our POJO and re-expose it using the JAX-WS front end.

Before we dig into the inbound endpoint configuration, we'll present the component we'll be working with for the remainder of this section: `GreetingServiceImpl`. The friendly class in listing 3.9 contains a single method called `getGreeting`. It will take a person's name and return a salutation.

Listing 3.9 `GreetingServiceImpl` is a Java class that returns a greeting

```
package com.muleinaction.cxf.simple;

public class GreetingServiceImpl {
    public String getGreeting(String name) {
        return "Hello, " + name;
    }
}
```

Although this example is a bit contrived, it serves to illustrate how to use the CXF inbound endpoint to build a web service. As you'll see in the next few chapters, components can be put to much better use than saying hello.

Now we're ready to use the simple front end to expose `GreetingServiceImpl`. Listing 3.10 will expose `GreetingServiceImpl` as a SOAP service.

Listing 3.10 Using a CXF inbound endpoint with the simple front end

```
<model name="cxf-simple">
    <service name="GreetingService">
        <inbound>
            <cxf:inbound-endpoint frontend="simple"
                address="http://localhost:9090/greeting" /> ① Configure CXF inbound endpoint
        </inbound>
        <component
            class="com.muleinaction.cxf.simple.GreetingServiceImpl"/> ② Specify component class to accept messages
        </service>
    </model>
```

The CXF inbound endpoint is configured on ①. We're defining the front end as simple and specifying the URL we want the WSDL to reside on. We next define our component on ②; as you can see, this is simply the `GreetingServiceImpl` class we implemented earlier. CXF will use reflection to examine the public methods of the `GreetingServiceImpl` class and expose them as SOAP operations in the WSDL. If you run this example, you'll be able to browse to the WSDL using this URL: `http://localhost:8080/greeting?WSDL`.

If you look at the WSDL Mule generates, you'll see it uses the hostname it's bound to as the default for the address location. This is often undesirable when you're using NAT or a load balancer and need to use the external address or name. You can override this explicitly by using the `wsdlLocation` attribute of the CXF inbound endpoint. In this scenario, you'd download the WSDL generated by Mule, modify the `soap:address` location, and then specify the location of the modified WSDL on the `wsdlLocation` attribute. The next time you restart Mule, the new location should take effect. The modified endpoint looks like this:

```
<cxf:inbound-endpoint
    frontend="simple"
    address="http://localhost:8080/greeting"
    wsdlLocation="/data/cxf/wsdl/greeting.wsdl"
/>
```

BEST PRACTICE Use the `wsdlLocation` attribute to explicitly define your WSDL when you need to override its address when working with NAT or load balancing.

Let's fire up a SOAP client and try the endpoint out. In the following example, we'll use SoapUI (<http://www.soapui.org/>) to inspect our web service. As you can see in figures 3.2, 3.3, and 3.4, we've imported our WSDL into SoapUI, generated a request, and received a response.

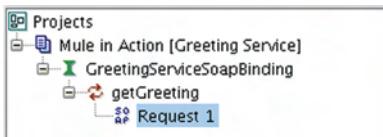


Figure 3.2 The imported WSDL



Figure 3.3 The SOAP request

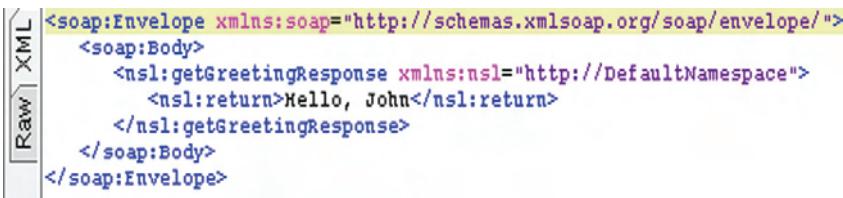


Figure 3.4 The SOAP response

This approach works fine for simple SOAP endpoints, but if your data model is complex or if you need specific control over how the WSDL is generated, your best bet is the JAX-WS front end.

JAX-WS is the Java API for XML Web Services. Along with JAXB, it allows you to use Java 5 annotations to define the particulars of your XML schema and WSDL. An in-depth discussion of JAXB and JAX-WS is beyond the scope of this book, but plenty of excellent resources exist online (including the CXF web site).

NOTE *Contract-first development with CXF* The examples in this section generate the XML artifacts for a web service, the schema, and WSDL from an object model. Often, though, it'll make sense to start with the WSDL or schema. This sort of web services development is called *contract-first* and is supported by CXF's wsdl2java tool. Wsdl2java will generate the JAX-WS annotated classes we'll see later from a preexisting XML schema and WSDL. Full documentation on using this tool is available from the CXF web site.

To illustrate how the JAX-WS front end works, we'll be modifying `GreetingServiceImpl`, which we defined earlier. We'll additionally be changing the operation name in the WSDL from `getGreeting` to `sayGreeting` using JAX-WS annotations.

The first step is to extract an interface from `GreetingServiceImpl`. As you can see in listing 3.11, we've created an interface called `GreetingService` that contains a `@WebService` annotation on the class declaration.

Listing 3.11 Extracting an interface from GreetingServiceImpl.java

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService
public interface GreetingService {
    @WebMethod(operationName = "sayGreeting")
    String getGreeting(String name);
}
```

The `@WebService` annotation on ① indicates we'll be exposing this interface as a web service. The `@WebMethod` annotation on the `getGreeting` method indicates that this method will become an operation in our WSDL, called `sayGreeting`. This will override the default name of `getGreeting` in the generated WSDL to `sayGreeting`. Now we need to refactor `GreetingServiceImpl`. Listing 3.12 modifies it to implement the `GreetingService` interface, along with the appropriate JAX-WS annotations.

Listing 3.12 Implementing GreetingService with JAXWS annotations

```
import javax.jws.WebService;
import javax.jws.WebMethod;

@WebService(endpointInterface =
    "GreetingService",
    serviceName = "GreetingService") ①
public class GreetingServiceImpl implements GreetingService { ②
    @WebMethod(operationName = "sayGreeting")
    public String getGreeting(String name) {
        return "Hello, " + name;
    }
}
```

The implementation is annotated similarly to the interface. We're using the `@WebService` annotation on ① to indicate the service name to use in the WSDL, which will be `GreetingService` in this case. We additionally annotate the `getGreeting` method on ② to indicate it should be exposed in the WSDL as `sayGreeting`. Now we just need to change the front end from simple to jaxws and we should be in business, as you can see in listing 3.13.

Listing 3.13 Using a CXF inbound endpoint with the jaxws front end

```
<model name="cxf-jaxws">
    <service name="GreetingService">
        <inbound>
            <cxf:inbound-endpoint
                frontend="jaxws"
                address="http://localhost:8081/greeting"
            />
        </inbound>
        <component class="GreetingServiceImpl"/>
    </service>
</model>
```

When we restart Mule, the new WSDL will be in place with the `getGreeting` operation renamed to `sayGreeting`.

Now that you're able to create SOAP-based web services using the `simple` and `jaxws` front ends, we can look at other methods of web services interoperability. Whereas SOAP is a good mechanism for exchanging XML documents, the verbosity of WSDL and XSD can sometimes be overkill. Let's look at an alternative way of using HTTP to exchange data.

3.4.2 **Sending and receiving data using the HTTP transport**

The HTTP transport allows you to send and receive data using the HTTP protocol. For instance, you can use HTTP's `POST` method to send data through an outbound endpoint or the `GET` method to return data from a synchronous inbound endpoint. XML is commonly used as the content of this data, although there's no reason why you can't send and receive plaintext, JSON, or CSV. The HTTP transport also features a polling connector. This'll let you repeatedly interrogate a remote URL and pull down its contents at a specified interval.

Let's look at how HTTP connectors and endpoints are configured. Table 3.4 shows you some of the options for configuring the HTTP transport. They should seem familiar to you if you've spent any time configuring a web browser.

Let's turn our attention back to Clood, Inc., and see how they're using an HTTP endpoint to receive data. Clood outsources some of its services, such as backup, to third-party providers. Clood's backup provider issues XML reports on the status of

Table 3.4 The HTTP transport lets you specify typical client side properties.

| Property | Type | Target | Description |
|----------------------------|---------|-------------------------------------|---|
| <code>proxyHostname</code> | String | connector | The proxy hostname. This lets you use a web proxy for requests. |
| <code>proxyPort</code> | String | connector | The proxy port. |
| <code>proxyUsername</code> | String | connector | The proxy username. |
| <code>proxyPassword</code> | String | connector | The proxy password. |
| <code>enableCookies</code> | boolean | connector | Enable cookies. |
| <code>user</code> | String | outbound endpoint | The username for the remote URL. |
| <code>password</code> | String | outbound endpoint | The password for the remote URL. |
| <code>host</code> | String | inbound endpoint, outbound endpoint | The host to either receive requests (inbound) or send requests (outbound). |
| <code>port</code> | int | inbound endpoint, outbound endpoint | The port to bind to for inbound endpoints and the port to send to for outbound endpoints. |
| <code>method</code> | String | inbound endpoint, outbound endpoint | HTTP method to use (GET, POST, PUT or DELETE). |

each night's backup run. They've offered to POST these reports to us, and as such we've been tasked to set up an HTTP endpoint to accept this data. Listing 3.14 shows how to accomplish this. It'll accept XML posted to the specified URL and save it to a file with an appended timestamp.

Listing 3.14 Accepting backup data on an HTTP inbound endpoint

```
<model name="httpInboundModel">
    <service name="httpInboundService">
        <inbound>
            <http:inbound-endpoint
                address="http://services.clood.com/backup-reporting"
                synchronous="true">
                <byte-array-to-string-transformer/> 1
            </http:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <file:outbound-endpoint path=".//data/reports"
                    outputPattern="backup-report-[function:dateStamp].xml"/>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

The diagram illustrates the flow of data in Listing 3.14. Step 1, labeled 'Accept XML reporting data' with a blue circle and number 1, shows the inbound endpoint accepting XML data. Step 2, labeled 'Transform posted byte array to String' with a blue circle and number 2, shows the byte array being converted to a string using a transformer. Step 3, labeled 'Save String to file' with a blue circle and number 3, shows the string being saved to a file with a timestamped filename.

The HTTP inbound endpoint is configured on ①. The HTTP connector will spin up a web server on the given address and accept documents on the “backup” URI. We have a byte-array-to-string transformer defined on ②. This converts the byte array posted by the client to a string. Finally, we have an outbound endpoint configured by ③ that will save each posted document as an XML file with a timestamp appended to the filename.

Using the HTTP inbound endpoint in this manner serves as a convenient integration point when a more formal transport like JMS isn't available. Also, like SOAP over HTTP, it's convenient when firewalls make other integration transports more difficult. Let's look at how the backup provider might use Mule to post these reports to the endpoint in listing 3.14. Listing 3.15 illustrates this.

Listing 3.15 Using an HTTP outbound endpoint to post data

```
<model name="httpOutboundModel">
    <service name="httpOutboundService">
        <inbound>
            <file:inbound-endpoint path=".//data/provider" /> 1
        </inbound>
        <outbound>
            <pass-through-router>
                <http:outbound-endpoint
                    address="http://services.clood.com/backup-reporting" /> 2
            </pass-through-router>
        </outbound>
    </service>
</model>
```

The file inbound endpoint specified on ① will wait for reports to appear in the specified directory. As the reports arrive, they'll be posted through the HTTP outbound endpoint defined on ② and ultimately arrive at the service we defined in listing 3.14.

NOTE SOAP versus REST, round 1. There's an ongoing debate in the web-services communities about the benefits of using SOAP versus using something called *REST*. REST, an abbreviation for *representational state transfer*, describes the architectural philosophies the WWW is based on. In the context of web-services debates, this usually boils down to RPC versus document exchange.

Occasionally you may have to repeatedly poll a remote URL and pull down the contents. This is an attractive option to pull contents from sites that don't offer RSS feeds, for instance. In listing 3.16 we set up an inbound endpoint to pull down the contents of www.cnn.com every 5 minutes.

Listing 3.16 Polling the contents of a web site

```
<http:polling-connector name="pollingHttp" pollingFrequency="300000" /> ←
<model name="httpPollingModel">
    <service name="httpPollingService">
        <inbound>
            <http:inbound-endpoint
                address="http://www.cnn.com"
                connector-ref="pollingHttp"
                synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint> ←
        </inbound> ←
        <outbound>
            <pass-through-router>
                <file:outbound-endpoint path=".out"
                    outputPattern="www.cnn.com-[DATE].html"/>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

Configures http:polling-connector ①

2 Poll www.cnn.com

3 Transform posted byte array to String

Saves web page as HTML file ④

The polling HTTP inbound endpoint configuration is a bit different than what we've seen. We first need to explicitly configure the connector at ①. You can see we're giving the connector the name `pollingHttp`. As you saw in the previous chapter, this is how the connector will be explicitly referenced in the rest of the configuration; it allows us to use regular HTTP inbound endpoints alongside polling endpoints. The inbound endpoint is configured on ②. We're explicitly declaring the `pollingHttp` connector by setting the `connector-ref` property on the inbound endpoint. We again encounter the `byte-array-to-string` transformer on ③. This performs the same function as previously: converting the byte array from the HTTP request into a string. Finally we have the file outbound endpoint on ④. This writes the web page contents out to a file appended with a date stamp.

In this section we learned how to use SOAP and HTTP to exchange data. We looked at using Mule’s CXF Transport to expose our data using SOAP. We saw a couple ways of using the transport to automatically expose your component’s method as SOAP endpoints. We also saw how to use WSDLs to consume SOAP services on endpoints. We examined how to use the HTTP endpoint, a lightweight alternative to using the CXF endpoint, as a method to exchange arbitrary data.

Whereas SOAP and HTTP are popular ways to do data exchange, there are more robust options available, especially when you have control over the applications in question. In the next section, we’ll consider how we can use JMS to do reliable message exchange.

3.5 **Using the JMS transport for asynchronous messaging**

The prevalence of HTTP makes web services an attractive alternative for integration outside of the firewall. Unfortunately HTTP wasn’t designed as an integration transport. As such, it fails to provide guarantees about delivery time, reliability, and security. The WS-* specifications are making strides in these areas, but their domain is limited to SOAP. Fortunately, the JMS protocol is an attractive alternative for integration “inside the firewall.” Let’s look at Mule’s JMS support.

JMS is an attractive option for application integration. If you’re working in a Java environment and have control over the network between your applications, using JMS makes a lot of sense—it’s asynchronous, secure, reliable, and often very fast. It also gives you the ability to work with arbitrary data payloads, in a purely Java environment you can even pass around serialized objects.

In this section we’ll investigate Mule’s support for JMS. We’ll start by seeing how we can send messages on queues and topics. Then we’ll see how we can use filters to be selective about the JMS messages we receive and send. Finally we’ll see how to use JMS messages, which are normally asynchronous, to perform synchronous operations on endpoints.

The JMS transport can be used to send and receive JMS messages on queues and topics, using either the 1.0.2b or 1.1 versions of the JMS spec. Mule doesn’t implement a JMS server, so you’ll use the JMS transport in conjunction with a JMS implementation such as ActiveMQ, OpenMQ, or Tibco EMS. Because of this, you’ll have to put the client JARs for your JMS provider in the `lib/user` directory in your Mule installation.

Configuring JMS with your broker can sometimes be a tricky proposition. As such, Mule provides a wealth of options for JMS connectors and endpoints to play nicely with the JMS implementation at hand. Table 3.5 lists some of these.

Table 3.5 Common configuration properties for the JMS transport

| Property | Type | Target | Description |
|---------------------|---------|---------------------------------------|--|
| persistentDelivery | boolean | connector | Toggle persistent delivery for messages |
| acknowledgementMode | String | connector | Set the acknowledgement mode to AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, or DUPS_OK_ACKNOWLEDGE |
| durable | boolean | connector | Toggle durability for topics |
| specification | String | connector | Specify with JMS specification to use, either 1.1 or 1.0.2b |
| honorQosHeaders | String | connector | Specifies whether individual should be allowed to override JMS QoS headers (persistentDelivery, and so on) |
| queue | String | inbound-endpoint, out-bound-endpoint | The queue to send to; can't be used in conjunction with |
| topic | String | inbound-end-point, out-bound-endpoint | The topic to send to; can't be used in conjunction with |

3.5.1 Sending JMS messages with the JMS outbound endpoint

Let's send some messages using a JMS outbound endpoint. In listing 3.17, we'll modify the backup reporting service used in listing 3.14 to send data to a JMS topic instead of a file. This'll allow all interested parties internal to Clood, Inc., to subscribe to the backup-reports topic and receive information about backup runs.

Listing 3.17 Sending backup reports to a JMS topic

```
<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="tcp://localhost:61616" />
    1 Configures activemq-connector

<model name="jmsModel">
    <service name="jmsService">
        <inbound>
            <http:inbound-endpoint
                address="http://services.clood.com/backup-reporting"
                synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint topic="backup-reports"/>
            3 Send JMS messages to topic
        </outbound>
    </service>
</model>
```

2 Configures HTTP inbound endpoint

```

        </pass-through-router>
    </outbound>
</service>
</model>

```

JMS brokers typically require slightly different configuration options. As such, you need to explicitly configure the connector for your broker. We're doing this for an external ActiveMQ instance on ①. The broker is listening on the loopback interface on port 61616. The JMS specification is declared here as well. If we wanted to use the 1.0.2b spec we'd simply change *1.1* to *1.0.2b*. An HTTP inbound endpoint is configured on ②, where the backup reports will be POSTed. Our JMS outbound endpoint is defined on ③. The string from the inbound endpoint will be sent to the backup-reports topic as a JMS TextMessage.

The JMS transport will create the specific type of JMS message based on the source data. A byte array will be instantiated as a BytesMessage, a Map will become a MapMessage, an InputStream into a StreamMessage and a String into a TextMessage, as we've just seen. If no better match is found, and the object implements Serializable, then an ObjectMessage will be created. Refer to section 5.5 for more information about JMS transformers.

Sending messages to a queue is just as easy. We simply change the `topic` attribute to the `queue` attribute in the JMS outbound endpoint configuration to look like this:

```
<jms:outbound-endpoint queue="backup-reports" />
```

Messages now sent through this endpoint will be placed on a queue called `backup-reports`. If you're using the 1.0.2b JMS specification, you'll need a separate connector for queues and topics, and then reference this connector on each endpoint. Listing 3.18 illustrates this.

Listing 3.18 Using the JMS 1.0.2b spec with queues and topics

```

<jms:activemq-connector
    name="jmsQueueConnector"
    specification="1.0.2b"
    brokerURL="tcp://localhost:61616" /> ①

<jms:activemq-connector
    name="jmsTopicConnector"
    specification="1.0.2b"
    brokerURL="tcp://localhost:61616" /> ②

<model name="jms102bModel">
    <service name="jms102bService">
        <inbound>
            <http:inbound-endpoint
                address="http://services.clood.com/backup-reporting"
                synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint>
        </inbound>
    </service>
</model>

```

```

<outbound>
    <pass-through-router>
        <jms:outbound-endpoint
            connector-ref="jmsTopicConnector" ③
            topic="backup-reports"/>
    </pass-through-router>
</outbound>
</service>
</model>

```

The two connectors, one for topics and one for queues, are configured on ① and ②. The outbound endpoint is configured on ③. We're explicitly telling the endpoint to use the `jmsTopicConnector` by setting the `connector-ref` attribute.

3.5.2 Receiving JMS messages with the JMS inbound endpoint

Let's look at receiving JMS messages on an inbound endpoint. We'll test that the `backup-reports` topic is working by subscribing to it and printing the reports to the console. Listing 3.19 shows how to implement this with a JMS inbound endpoint.

Listing 3.19 Receiving JMS messages and printing them on the console

```

<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="tcp://localhost:61616" />

<model name="Listing 3.19">
    <service name="Backup Reporting Service">
        <inbound>
            <jms:inbound-endpoint topic="backup-reports" /> ① Receive messages
        </inbound> on backup-
        <outbound> reports topic
            <pass-through-router>
                <file:outbound-endpoint path="OUT"/> ② Output
            </pass-through-router> payload to file
        </outbound>
    </service>
</model>

```

The inbound endpoint is configured on ①; it'll consume messages off the `backup-reports` topic. We're not explicitly defining a JMS message-to-object transformer here. Mule will implicitly use the JMS message-to-object transformer when no other transformers are present. The rules for this are the inverse of the JMS marshalling that occurs on the outbound endpoint. In this case, the `TextMessage` will be transformed into a string object. Finally, the string is saved to the file by ②.

JMS topics support durable subscribers. Messages for a durable subscriber will be queued on the JMS broker when the subscriber is unavailable. When the subscriber comes back online, the missed messages will be delivered. You have the ability to enable this behavior on an inbound endpoint. This is accomplished by configuring the connector for durability, as follows:

```
<jms:activemq-connector
    name="jmsConnector" specification="1.1"
    brokerURL="tcp://localhost:61616" durable="true" />
```

Now the JMS transport will treat all topic-based inbound endpoints as durable.

3.5.3 Using selector filters on JMS endpoints

Filters can be used on JMS endpoints to be selective about the messages they consume. This is analogous to how we used filters on the file inbound endpoint discussed previously. JMS inbound-endpoint filters use the JMS selector facility to accomplish this. Let's modify the JMS inbound endpoint from listing 3.19 to only accept reports created after midnight, January 1, 2008. You can see this in listing 3.20.

Listing 3.20 Filtering messages with a JMS selector

```
<jms:inbound-endpoint topic="backup-reports">
    <jms:selector expression="JMSTimestamp &gt;1199163600000" /> 
    <jms:jmsmessage-to-object-transformer/> Define JMS selector ①
</jms:inbound-endpoint>
```

You can use JMS selectors in this manner on any header property. We'll talk more about headers in chapter 6 when we discuss components in detail. You might be wondering what the `>` characters are all about on ①. This is the XML escape sequence for `>`, the greater-than character. Failing to escape characters such as `>` in your Mule configurations will produce XML parsing errors when Mule starts.

3.5.4 Using JMS synchronously

As JMS is inherently asynchronous in nature, you'll usually use JMS outbound endpoints in an asynchronous manner—sending messages and not waiting for an immediate response. Sometimes, though, you'll want to wait for a response from a message you're sending. You can accomplish this by setting the synchronous attribute to true on a JMS endpoint. This attribute tells the transport to wait for a response from the remote endpoint it's dispatching to. For some transports, such as SOAP, synchronous behavior is implied. In other cases, such as JMS, synchronous is false by default because the behavior is assumed to be asynchronous.

NOTE JMS outbound endpoints can't be used synchronously within a transaction. In a JMS transaction, messages are sent when the transaction is committed, which prohibits waiting for a response. We'll discuss using transactions with JMS in detail in chapter 10.

Let's look at an example of how this works. The following example illustrates an HTTP endpoint deployed by Clood to return the status of an order. Clients of this service supply an ID to the URL and a string is returned containing the status of the order. For instance, `http://services.clood.com/orders?id=1` will return the status for an order whose ID is 1. Behind the scenes, this service sends the order request to a JMS

queue and waits for a response. The result is returned to the user when the response is received. You can see this in listing 3.21.

Listing 3.21 Using JMS endpoints synchronously

```
<model name="jmsRemoteSyncModel">
    <service name="jmsRemoteSyncService">
        <inbound>
            <http:inbound-endpoint
                address="http://services.cloud.com/orders"
                synchronous="true"
                method="GET"/>           ↗ 1 Accept HTTP order
        </inbound>                                ↗ 2 Send JMS request to
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="order.status"
                    synchronous="true"/>      ↗ 3 Accept JMS request
            </pass-through-router>
        </outbound>                                ↗ 4 Define service component
    </service>                                     ↗ 5 Resolve entry
                                                points by method
    <service name="Order Service">
        <inbound>
            <jms:inbound-endpoint queue="order.status"   ↗ 6 Include given method
                synchronous="true"/>                   as entry point
        </inbound>
        <component
            class="com.cloud.OrderServiceImpl">
            <method-entry-point-resolver>
                <include-entry-point method="getOrderStatus"/>
            </method-entry-point-resolver>
        </component>
    </service>
</model>
```

First we set up the HTTP inbound endpoint on ① to accept an HTTP GET. The request is sent to the `order.status` queue on ②. The `synchronous` property on this endpoint indicates that a response is expected from the remote endpoint. To facilitate this, the JMS transport will create a temporary queue for the response data and set this as the `Reply-To` property of the JMS message. The request is received by the JMS inbound endpoint on ③, where it's processed by the `OrderServiceImpl` component defined on ④. The `method-entry-point-resolver` and `include-entry-point` on ⑤ and ⑥ tell Mule to invoke the `getOrderStatus` method of `OrderServiceImpl`.³ The return value of this method is sent as a JMS message to the temporary queue and is used as the response to the original HTTP GET on ①. The time to wait for a response on the temporary queue can be controlled by setting the `responseTimeout` option on the endpoint. Setting this option will have the endpoint wait the given number of milliseconds for a response before giving up. You can also disable creation of the temporary

³ We'll discuss the `method-entry-point-resolver` and `include-entry-point` properties in chapter 5.

queue by setting the `disableTemporaryReplyToDestinations` option on the connector or endpoint. This is useful if a JMS endpoint is being invoked synchronously but a response isn't provided or required. As we'll see in the next chapter, depending on the outbound routing used, a message can be sent to multiple endpoints. These will all be invoked synchronously if the inbound endpoint is synchronous.

There are scenarios where submitting data like this is particularly useful. One example is having a set of "competing consumers" that process data. In this case, you could have multiple "echo services" running in different Mule instances on separate servers. Each service will compete for messages of the messages queue, giving you transparent load sharing and redundancy.

BEST PRACTICE Use competing consumer behavior with JMS endpoints to load share messages across services.

Being able to use JMS with Mule is an important part of your integration toolbelt. You just learned how to send and receive messages with JMS endpoints using queues and topics. We saw how to use filters on JMS endpoints to be picky about the messages we send and receive. Finally, we looked at how to use Mule's synchronous support with JMS endpoints to perform synchronous operations over asynchronous queues.

NOTE An important facet of working with JMS is dealing with situations when the JMS servers are unavailable. In chapter 8 we'll examine how to use a reconnection strategy in concert with the JMS transport. This will allow your Mule instances to tolerate the failure of a JMS server without losing messages.

Now that you're comfortable using JMS messaging with Mule, let's turn our attention to a old but venerable transport protocol—FTP.

3.6 Receiving and sending files using the **FTP transport**

If you've been involved in anything Internet-related for a while, you surely remember the reign of FTP. Before HTTP and SSH, FTP was the de facto way to move files around between computers. While FTP's popularity has waned in recent years due to the rise of HTTP, SCP, and even BitTorrent, you'll occasionally encounter an application that necessitates its use.

In this section we'll look at using the FTP transport to send and receive data. First we'll look at how we can poll a remote FTP directory. We'll then see how we can send data to a remote FTP site using an outbound endpoint.

Configuring the FTP transport is similar to configuring an FTP client, as we see in table 3.6.

Table 3.6 The FTP transport's configuration is similar to that of a typical FTP client.

| Property | Type | Target | Description |
|------------------|---------|--|---|
| pollingFrequency | long | connector, inbound endpoint | For inbound endpoints, the frequency at which the remote directory should be read |
| outputPattern | String | connector, outbound endpoint | Specify the format of files written by outbound endpoints |
| binary | boolean | connector, inbound endpoint, outbound endpoint | Use binary mode when transferring files |
| passive | boolean | connector, inbound endpoint, outbound endpoint | Use passive mode when transferring files |
| user | String | connector, inbound endpoint, outbound endpoint | The username to use when connecting to the remote FTP server |
| password | String | inbound endpoint, outbound endpoint | The password to use when connecting to the remote FTP server |
| host | String | inbound endpoint, outbound endpoint | The host of the remote FTP server |
| port | int | inbound endpoint, outbound endpoint | The port of the remote FTP server |

3.6.1 Receiving files with inbound FTP endpoints

Let's look at how to configure an FTP inbound endpoint to poll a remote FTP directory every hour for new files. In listing 3.22, we'll pull down new files and persist them to disk, preserving the filename on the remote server.

Listing 3.22 Polling a remote FTP directory every hour for new files

```
<model name="ftpInboundModel">
    <service name="ftpInboundService">
        <inbound>
            <ftp:inbound-endpoint user="joe" password="123456"
                host="ftp.mycompany.com" port="123"
                path="/ftp/incoming"
                pollingFrequency="3600000" />
        </inbound>
        <outbound>
            <pass-through-router>
                <file:outbound-endpoint path=".out" />
            </pass-through-router>
        </outbound>
    </service>
</model>
```

The inbound endpoint is configured on ①. We're specifying the user, password, host, port, path, and polling frequency for the remote server. The FTP transport will establish a connection to this endpoint every hour and pass each new file over to the file

outbound endpoint defined on ②. The outbound endpoint will write the file to the `./out` directory using the same filename as on the server.

3.6.2 **Sending files with outbound FTP endpoints**

Sometimes you'll need to send a file to a remote FTP server. We saw this in listing 3.7, where Clood's accounting department was receiving some invoices via an FTP drop. The service in listing 3.23 illustrates how the third party might have used Mule to send the invoices to Clood, Inc.

Listing 3.23 Sending a file to a remote FTP server

```
<model name="ftpService">
    <service name="ftpOutboundService">
        <inbound>
            <file:inbound-endpoint path="./in"/> ①
        </inbound>
        <outbound>
            <pass-through-router>
                <ftp:outbound-endpoint user="joe"
                    password="123456"
                    host="ftp.clood.com"
                    port="123"
                    path="/data/invoice" /> ②
            </pass-through-router>
        </outbound>
    </service>
</model>
```

We have the file inbound endpoint configured on ①. As files are placed into the `./in` directory they'll be passed to the FTP outbound endpoint on ②. This'll place the file into the `/data/invoice` directory of `ftp.clood.com`.

We just saw how to configure Mule to poll and submit files via FTP. We'll now look at how we can use the JDBC transport to get data in and out of a database.

3.7 **Working with databases**

Databases are often the implied means of integration between applications. Every mainstream development platform provides rich support for database interaction. Because of this, it's not uncommon for databases to outlive the applications they were originally implemented to support. If you're working with an existing Java application, odds are you're using a database abstraction layer (perhaps implemented with Hibernate or the Spring JDBC template). In that case, it usually makes sense to leverage these libraries within your components to perform database access. If you're working with legacy databases or integrating with an application that doesn't provide native Java access, the JDBC transport is an attractive means of getting data into and out of the database.

In this section we'll look at using the JDBC transport. First you'll see how to use a JDBC inbound endpoint to perform queries. Then we'll look at using JDBC outbound

endpoints to perform insertions. Let's look at how to configure the JDBC transport. Table 3.7 shows us some common configuration properties. The `dataSource-ref` in particular is important; this is the reference to the configured datasource you'll use to access the database. This is typically configured as a Spring bean or a JNDI reference.

Table 3.7 Configuring the JDBC's transport dataSource reference, pollingFrequency, and queryKey

| Property | Type | Target | Description |
|-------------------------------|--------|--|---|
| <code>dataSource-ref</code> | String | connector | The JNDI or bean reference of the <code>dataSource</code> |
| <code>pollingFrequency</code> | long | connector, inbound endpoint | How often the query is executed |
| <code>queryKey</code> | String | connector, inbound endpoint, outbound endpoint | Specify the query to use |

For the remainder of this section we'll be revisiting Clood's monitoring database, discussed at the beginning of this chapter. This database, called `monitoring`, consists of a single table containing URL alerting data. The schema for this database is defined in listing 3.24.

Listing 3.24 A database schema for monitoring data

```
CREATE TABLE `alerts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `host` varchar(255) NOT NULL,
  `message` text NOT NULL,
  `timestamp` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Let's look at how we can use the JDBC transport to perform queries against this database, then revisit how to insert rows into it.

3.7.1 Using a JDBC inbound endpoint to perform queries

You'll use the JDBC inbound endpoint to perform queries against a database. This will generate data you can pass to components and outbound endpoints. Let's see how this works.

It turns out that other people in Clood, Inc., are interested in the data you're collecting. You've recently been tasked by the operations team to publish the URL alert data to a JMS topic. This'll allow various management applications to take action if a certain threshold of alerts occur in a given time frame. Listing 3.25 shows how to use a JDBC inbound endpoint to accomplish this. We'll query the `alerts` table every hour and publish each alert to a JMS topic that can be consumed by interested parties in operations.

Listing 3.25 Querying a table every hour and sending the results to a JMS topic

```

<spring:bean id="dataSource"
    class=
        "org.springframework.jdbc.datasource.DriverManagerDataSource">
    <spring:property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <spring:property name="url" value="jdbc:mysql://localhost/monitoring"/>
    <spring:property name="username" value="mule"/>
    <spring:property name="password" value="password"/>
</spring:bean>

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="alertQuery"
        value="select host,timestamp from alerts
            where DATE_SUB(NOW(),INTERVAL 1 HOUR) < timestamp;" />
</jdbc:connector>

<jms:activemq-connector
    name="jmsConnector"
    specification="1.1"
    brokerURL="tcp://localhost:61616" />

<model name="jdbcInboundModel">
    <service name="jdbcInboundService">
        <inbound>
            <jdbc:inbound-endpoint
                pollingFrequency="3600000"
                queryKey="alertQuery"/>
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint topic="alerts">
                    <transformers>
                        <xm:object-to-xml-transformer/>
                        <jms:object-to-jmsmessage-transformer/>
                    </transformers>
                </jms:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>
</model>

```

Configure JDBC datasource ①

Configure JDBC connector ②

Define SQL query ③

Configure JMS connector

Execute query once an hour

Transform and send each row result as a JMS message ④

We'll start by examining ①. Here we're configuring a Spring datasource with the details of our database (which is MySQL in this example.) We reference this datasource on the JDBC connector defined on ②. The actual query we'll use is defined on ③. If you look closely, you'll notice that we're escaping out the less-than sign in the query, so as to avoid XML parsing issues when Mule starts up. This query will return a row for every alert inserted into the database in the last hour. Each member of the result set will be sent as an individual Mule message to the outbound router. For instance, if 10 alerts were returned by alertQuery then 10 messages would be published by the JMS outbound endpoint on ④.

The payloads of the messages themselves are a `java.util.Map` instance. The map's keys are the column names of the result set. The key values are the row value for each result. In the previous example, each message would contain a map with two keys:

host and timestamp. The value of the host key will be the host of the alert and the value of the timestamp key will be the timestamp of when the alert occurred. You'll see in chapters 5 and 6 how Mule's transformation and component facilities will allow you to put this data in a meaningful format for your project. For now, we're using object-to-xml-transformer to serialize the map into XML and send it to the outbound endpoint.

Let's now revisit how to perform insertions using a JDBC outbound endpoint.

3.7.2 Using a JDBC outbound endpoint to perform insertions

The JDBC transport allows you to insert rows into a table using an outbound endpoint. We saw at the beginning of this chapter how we can use a JDBC outbound endpoint to insert data arriving from an IMAP mailbox. JDBC outbound endpoints can also be used in a variety of other scenarios, from loading CSV files into a database to saving log events to a database table. Let's take a closer look at the example from the beginning of this chapter to see how the JDBC inbound endpoint works.

Listing 3.26 Using a JDBC outbound endpoint to perform insertions

```
<spring:beans>
    <spring:import resource="spring-config.xml"/>
</spring:beans>

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="statsInsert"
        value="insert into alerts values
(0 ,#[map-payload:HOST],#[map-payload:MESSAGE],\
#[map-payload:TIMESTAMP])"/>
</jdbc:connector>
```

① Reference external Spring configuration


```
<model name="URLAlertingModel">
    <service name="URLAlertingService">
        <inbound>
            <imap:inbound-endpoint host="mail.cloud.com" user="mule"
                password="password">
                <email:email-to-string-transformer/>
            </imap:inbound-endpoint>
        </inbound>
        <component class="com.cloud.monitoring.URLAlertComponent" />
        <outbound>
            <pass-through-router>
                <jdbc:outbound-endpoint queryKey="statsInsert"/>
            </pass-through-
router>
        </outbound>
    </service>
</model>
```

② Define SQL insert

③ Use component to generate value map

④ Perform insertion

The first thing you'll notice is the external Spring configuration reference declared on ①. Instead of declaring our JDBC datasource directly, as we did in listing 3.24, we're pulling in an external Spring configuration file. This is often useful if you're sharing the same Spring configuration between multiple applications or if your

Spring config is particularly long and you don't want to clutter up your Mule configuration. Our SQL insert statement is defined on ②. The SQL insert is populated by a map supplied to the endpoint. We're using the #[map-payload:KEY] placeholders to extract values from the map and use these in the `INSERT` statement. In this case, we're extracting the map's values for keys `HOST`, `MESSAGE`, and `TIMESTAMP`. The map itself is created by the `URLAlertComponent` class declared on ③. This class populates the map based on the contents of the email supplied to it. When the map is returned from the component, it's passed to the JDBC outbound endpoint defined on ④ and the data is inserted into the alerts table.

Databases are often the implied integration means between applications. Through the examples in this section, you should now know how to use Mule to leverage this fact. We saw that we can use a select statement as a source of data for an inbound endpoint. We then looked at how we can insert data into tables using outbound endpoints.

Now let's take a look at how we can use the XMPP transport to communicate with Jabber. You'll see how this will let you use instant messaging as an integration mechanism.

3.8 Using the XMPP transport

Instant messaging is a standard means of communication in many enterprises. While public IM implementations such as AIM or Yahoo! Messaging are common, Jabber is emerging as a secure, private, and open alternative.

In this section we'll look at how we can use the XMPP protocol, which underlies Jabber, to send and receive messages. This is useful in a variety of scenarios, from sending out instant messages when an automated software build fails, to using instant messages as a mechanism to remotely signal applications.

As we've seen with the other transports in this chapter, the configuration properties in table 3.8 for the XMPP transport mimic the properties you'd set on a client for the transport (in this case a Jabber client).

Table 3.8 Configuring the XMPP transport to connect to a Jabber server

| Property | Type | Target | Description |
|------------------------|--------|--|--|
| <code>user</code> | String | connector, inbound endpoint, outbound endpoint | The username to use when connecting to the Jabber server |
| <code>password</code> | String | connector, inbound endpoint, outbound endpoint | The password to use when connecting to the Jabber server |
| <code>host</code> | String | connector, inbound endpoint, outbound endpoint | The hostname of the Jabber server |
| <code>port</code> | int | connector, inbound endpoint, outbound endpoint | The port the Jabber server is running on |
| <code>recipient</code> | String | connector, inbound endpoint, outbound endpoint | The default recipient of all messages |

3.8.1 Sending Jabber messages on an outbound endpoint

Let's look at how to send a Jabber message on an outbound endpoint. Listing 3.27 defines a service that accepts a JMS message off a queue, then sends the JMS message payload to a Jabber user.

Listing 3.27 Sending Jabber messages with the XMPP transport

```
<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="tcp://localhost:61616" /> ① Configure ActiveMQ connection

<model name="xmpp-outbound">
    <service name="XMPPInbound">
        <inbound>
            <jms:inbound-endpoint queue="messages" /> ② Configure inbound JMS queue
        </inbound>
        <outbound>
            <pass-through-router>
                <xmpp:outbound-endpoint
                    user="mule"
                    password="mule"
                    host="jabber.cloud.com"
                    port="5222"
                    recipient="john" /> ③ Configure outbound XMPP endpoint
            </pass-through-router>
        </outbound>
    </service>
</model>
```

① and ② should look familiar to you by now; this is simply configuring our ActiveMQ JMS connector and endpoint. We're using an inbound endpoint to receive messages off the `messages` queue. The XMPP outbound endpoint is configured on ③. It'll take the payload of the JMS message and send it to `john`, a jabber account that lives on `jabber.cloud.com`.

Sending messages like this could be useful in a variety of scenarios. In particular, you might use an endpoint like this in conjunction with an error channel. This'll let you use Mule to alert you to various events as they occur. We'll talk more about that in chapter 8. Now that we've seen how to send messages, let's look at how we can react to them.

3.8.2 Receiving Jabber messages on an inbound endpoint

Receiving IMs is just as easy as sending them. The service in listing 3.28 will accept an XMPP message and send the contents to an email address.

Listing 3.28 Receiving Jabber messages via the XMPP transport

```
<model name="xmppInboundModel">
    <service name="xmppInboundService">
        <inbound>
            <xmpp:inbound-endpoint
                user="mule"
```

```

        password="mule"
        host="jabber.cloud.com"
        port="5222"
        recipient="mule"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <smtp:outbound-endpoint
                host="localhost"
                from="mule"
                subject="Test Message"
                to="mule@company.com"/>
        </pass-through-router>
    </outbound>
</service>
</model>
</mule>
```

The inbound endpoint is configured on ①; the configuration is identical to that of the outbound endpoint. The outbound endpoint is configured on ②; this will send the contents of the instant message to the specified email address.

We've just seen how we can use Mule's XMPP transport to send and receive messages using Jabber. This can provide a friendly face for your Jabber services to communicate with your end users. Let's conclude our discussion on transports by looking at Mule's in-memory, or VM transport.

3.9 The VM transport

The VM transport is a special kind of transport that you'll use to send messages via memory. These messages never leave the JVM the Mule instance is running in. The benefits of this might not seem immediately obvious, but you'll soon see how the VM transport allows you to leverage the virtues of an asynchronous messaging framework such as JMS without the overhead of a broker. You'll see in the next section how we can use the VM transport, in conjunction with Mule's routing functionality, to dispatch data across multiple components and services.

Table 3.9 shows some of the properties we can set for the VM transport. The `queueEvents` property is of particular interest. The VM transport will, by default, deliver messages directly to each component. When `queueEvents` is set to `true`, Mule will put these messages in an in-memory queue. As we'll see, when coupled with persistent delivery, this lets you use the VM transport as a reliable queue for in-VM messaging.

Table 3.9 Configuring the VM transport

| Property | Type | Target | Description |
|---------------------------|---------|-----------|--|
| <code>queueEvents</code> | boolean | connector | Specifies whether events should be queued on the endpoint. If false, the default, then events are simply delivered directly to the component. If true, then events are placed on an in-memory queue. |
| <code>queueTimeout</code> | long | connector | How often queued messages are expunged. |

3.9.1 Sending and receiving messages on VM endpoints

The VM transport supports the sending and receiving of messages. It's functionally similar to the JMS transport. The VM transport is typically used to send a message through services internal to a Mule instance before routing it to a final endpoint or service. Cloood, for instance, uses the VM transport to pass messages received from both JMS and HTTP endpoints to a common validation service. Once validated, the message is sent to a dispatching service, where it's appropriately routed. Listing 3.29 demonstrates this.

Listing 3.29 Messaging with the VM transport

```

<model name="vmModel">
    <service name="Order Input">
        <inbound>
            <http:inbound-endpoint
                address="http://localhost:9756/orders"
                synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint>
            <jms:inbound-endpoint queue="orders.in">
                <transformers>
                    <jms:jmsmessage-to-object-transformer/>
                </transformers>
            </jms:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint path="order.validation"/>
            </pass-through-router>
        </outbound>
    </service>
    <service name="Order Validation">
        <inbound>
            <vm:inbound-endpoint path="order.validation"/>
        </inbound>
        <component class="com.clood.order.OrderValidationService">
            <method-entry-point-resolver>
                <include-entry-point method="validateOrder"/>
            </method-entry-point-resolver>
        </component>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint path="order.dispatch"/>
            </pass-through-router>
        </outbound>
    </service>
    <service name="Order Dispatch">
        <inbound>
            <vm:inbound-endpoint path="order.dispatch"/>
        </inbound>
        <outbound>
            <static-recipient-list-router>

```

The diagram illustrates the flow of data between five numbered steps:

- Step 1: Accept order data** (VM inbound endpoint)
- Step 2: Pass order data out to validation VM queue** (VM outbound endpoint)
- Step 3: Accept order data on validation VM queue** (VM inbound endpoint)
- Step 4: Send data to dispatch VM queue** (VM outbound endpoint)
- Step 5: Route order** (Static recipient list router)

```

<recipients>
    <spring:value>
        jms://order.submission.ops
    </spring:value>
    <spring:value>
        jms://order.submission.sales
    </spring:value>
</recipients>
</static-recipient-list-router>
</outbound>
</service>
</model>

```

Order data is accepted on the HTTP and JMS endpoints configured on ①. An order must be validated and then dispatched to a set of JMS queues where it can be provisioned. The validation and dispatch could be performed using a component and outbound router in the Order Input service. Alternatively, the validation code and dispatch could be hosted in their own services. The latter approach decouples order validation and dispatch from order submission, allowing each stage of the order process to be managed separately. This is implemented by the VM endpoints on ②, ③, ④, and ⑤, which breaks down the order processing into stages. Each stage—order input, order validation, and order dispatch—is handled by a dedicated service that can be independently managed. Service decomposition in this manner allows you to apply concerns such as exception handling, tuning, and security separately for each stage.

BEST PRACTICE Use the VM transport to implement service decomposition without using an external messaging system such as JMS.

A limitation to this approach becomes apparent if the Mule instance dies while a message is being passed through the queues. With the default behavior, such messages will be stored in memory and be lost. Let's see how we can use persistent queues to overcome this limitation.

3.9.2 Using persistent queues on VM endpoints

In order to save our VM messages to disk, we need to set the queueEvents property on the VM connector. We then need to define the queueing profile—this is configured after the service definitions, as you can see in listing 3.30.

Listing 3.30 Using persistent VM queues

```

<vm:connector name="vmConnector" queueEvents="true"/>

<service name="Order Dispatch">
    <inbound>
        <vm:inbound-endpoint path="order.dispatch"/>
    </inbound>
    <outbound>
        <static-recipient-list-router>
            <recipients>
                <spring:value>

```

```
jms://order.submission.ops  
</spring:value>  
<spring:value>  
    jms://order.submission.sales  
</spring:value>  
</recipients>  
</static-recipient-list-router>  
</outbound>  
</service>  
  
<queue-profile persistent="true" maxOutstandingMessages="1000" />  
</model>
```

Now your VM messages will be persisted to disk as they're delivered. The next time Mule starts up, it'll attempt to redeliver all the messages saved in the persistent queue.

NOTE The VM transport uses a `QueuePersistenceStrategy` implementation to serialize the contents of the VM queue. When queue persistence is enabled, Mule will use the `FilePersistenceStrategy` implementation to serialize the contents of your queues. For this to work, the payloads of your messages must implement the `Serializable` interface. Bear this in mind if you're passing around objects in your message payloads.

We saw in this section how to send and receive messages using the VM transport, Mule's in-memory message-passing facility. We even saw how we can use the in-memory transport in conjunction with persistent queues to perform reliable messaging. We'll see more facets of the VM transport in the next chapter, where we'll use it to route data between Mule services. You'll see how it becomes an important way to organize the flow of data between your services.

3.10 Summary

In this chapter you've seen how you (and Clood, Inc.) can leverage Mule's transports to move data between applications. You've become familiar with connectors and endpoints, which provide a common abstraction for working with disparate communication protocols. You've seen how they allow you to focus on solving integration problems and let Mule handle the underlying plumbing.

While transport functionality is critical, moving data into and out of Mule is only one piece of the integration puzzle. We'll see in the next chapter how we can leverage Mule's routing functionality to intelligently transport data between endpoints.

Routing data with Mule



In this chapter

- Understanding inbound and outbound routing
- Filtering with routers
- Using inbound routers
- Using outbound routers

You've probably been exposed to a router at some point in your career. Usually these are of the network variety—like the DSL router in your bedroom or the core router in your data center. In either case, the router's function is the same: to selectively move around data. Not surprisingly, many of the concepts that underlie network routing are also applicable to routing data between applications. We'll see in this chapter how multicasting, static routing, forwarding, and filtering are just as applicable to enterprise application integration as they are to networking devices.

We've already seen some examples of routing. In the previous chapter, we used pass-through routing to move data from inbound endpoints to outbound endpoints. We also used a chaining router to pass the response of one endpoint to the input of another endpoint. Let's consider an integration scenario and see how Mule's routing support helps us out.

Recall Clood, Inc.'s monitoring database from the last chapter? Let's see how we can modify the outbound routing to send the message to two destinations simultaneously. We'll insert a row into the database and send an instant message to Clood's Jabber server. Listing 4.1 shows the modified outbound routing configuration.

Listing 4.1 Multicast data to a JDBC and XMPP endpoint

```
<outbound>
    <multicasting-router>
        <jdbc:outbound-endpoint queryKey="statsInsert" />
        <xmpp:outbound-endpoint user="mule" password="mule"
            host="jabber.company.com"
            port="5222"
            recipient="yourboss"
            transformer-refs="StatToInstantMessageText" />
    </multicasting-router>
</outbound>
```

① Define multicasting router

The multicasting router defined on ① will send the message to each of the listed outbound endpoints at the same time. By changing a few lines of XML, we're able to make a database insert and send an instant message in parallel. Using routers enables you to loosely couple the components and messages. When implemented correctly, neither have any knowledge of or dependencies on the routing strategies employed.¹

In this chapter we'll investigate Mule's data routing capabilities. We'll start off by looking at the different types of routers supported by Mule. You'll learn the differences between inbound and outbound routing. We'll discuss Mule's filtering capabilities, then we'll dive in and examine each of Mule's supplied routers. We'll look at how Clood, Inc., uses Mule's routing features to manage its order entry and provisioning. We'll close out the chapter by considering a special type of outbound router, the `async-request-reply` router, which will allow us to aggregate data from multiple sources. By the end of this chapter, you'll hopefully be able to look at your integration challenges in terms of Mule's routing concepts. This will make it easier for you to identify and solve integration problems.

4.1 Working with routers

Mule provides a rich set of routing abstractions to help you organize the flow of your data. In this section we'll investigate the different types of routers and see where we can use them to direct message flow. You'll see how Mule's routers allow you to approach your integration efforts using a common, pattern-based language. This'll help you identify and reuse routing approaches across different problem domains. We'll start off by introducing inbound routing and seeing how to control what data enters our services. We'll then look at outbound routers, which control how data leaves a service.

¹ You may have noticed we have something called `transformer-refs` configured on the XMPP outbound endpoint. You'll see later in this chapter when we discuss the multicasting router how this lets us tailor the statistical data for something appropriate in an instant message.

4.1.1 Inbound routers

The first type of router we'll consider is an inbound router. Inbound routers work in tandem with inbound endpoints to control how messages are sent to a service's component. You'll typically use inbound routers to do things such as selectively process messages, enforce atomic message consumption, and deal with aggregating messages. Figure 4.1 shows where the inbound router sits in the event-processing chain.



Figure 4.1 Using an inbound router to route data from an inbound endpoint to a component

As the figure illustrates, messages are first received by endpoints and then passed to the inbound router. This is echoed in the configuration file; your inbound router definitions will follow your inbound endpoint definitions. When the inbound endpoint receives the message, one of three things will happen: the message will be sent to the component, the message will be forwarded to the catch-all strategy, or if there's no catch-all strategy configured on the service, the message will be dropped.

The catch-all strategy defines the service's behavior in the event that a message isn't sent to the component. You'll typically use catch-all strategies to implement error handling and logging for unroutable messages. The following catch-all strategies are available:

- *forwarding-catch-all-strategy*—Forwards messages to the specified endpoint
- *logging-catch-all-strategy*—Logs non-matching messages
- *custom-forwarding-catch-all-strategy/custom-catch-all-strategy*—Custom forwarding and catch-all strategies

We'll see the forwarding catch-all strategy and the logging catch-all strategy in this chapter. Consult the online Mule documentation for more information about implementing your own forwarding and catch-all strategies.

Let's look at how inbound routers are typically configured. Listing 4.2 implements a service that defines two inbound endpoints, one for JMS messages and another for the VM messages.

Listing 4.2 Selectively consuming messages using the selective-consumer router

```

<service name="selectiveConsumerService">
  <inbound>
    <jms:inbound-endpoint queue="messages" />
    <vm:inbound-endpoint path="messages" />
    <selective-consumer-router>
      <regex-filter pattern="^STATUS: (OK|SUCCESS)$" />
    </selective-consumer-router>
  </inbound>
</service>
  
```



Consume messages that contain specified regex pattern

```

<forwarding-catch-all-strategy>
    <jms:outbound-endpoint queue="errors" />
</forwarding-catch-all-strategy>
</inbound>
<outbound>
    <pass-through-router>
        <stdio:outbound-endpoint system="OUT" />
    </pass-through-router>
</outbound>
</service>

```

- 2 Forward unroutable messages to error queue
- 3 Print message to screen

We have a selective consumer defined on ① that only consumes messages whose payloads match the specified regular expression pattern. Any messages that don't pass this regular expression filter will be handled by the forwarding catch-all strategy defined on ②. The forwarding catch-all strategy will send all unroutable messages to the specified JMS queue. Messages that make it through the filter will move on to the pass-through router. Finally, the message will be printed on the screen by the STDIO outbound endpoint defined on ③.

You may be wondering why we didn't just use an endpoint filter on each endpoint. As you'll see in the next chapter, routing can occur after the transformation process. This gives you the opportunity to transform your inbound messages before inbound routing takes place, letting you share the same routing logic across all inbound data. You could, for instance, transform all your data payloads into XML, then use a JXPath filter on a selective-consumer router to only consume messages whose payloads match the specified expression. In this scenario, if you need to change the XPath expression, you can do it once on the selective-consumer router and not across all your inbound endpoints.

Now that we've seen how inbound routers control the message flow into a service, let's look at controlling how messages leave a service.

4.1.2 Outbound routers

You've already seen quite a bit of outbound routing. In the previous chapter, all of our examples used either the pass-through router or the chaining router. You'll use outbound routers to control how messages are distributed to outbound endpoints. Figure 4.2 illustrates where outbound routing fits in the event chain.

As you can see, data from the component is sent to the outbound router. The outbound router is responsible for sending the message to the appropriate outbound endpoints. In listing 4.1, the outbound router is required before data can be sent to an outbound endpoint. This makes it a little different than inbound routing, where an inbound router definition isn't required. Once again you have the option of

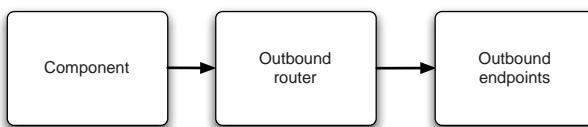


Figure 4.2 Using an outbound router to route data from a component to outbound endpoints

defining a catch-all strategy for outbound routing. The strategies are the same as for both inbound and outbound routers; this gives you the flexibility to have different logging or error handling for each.

Let's look more closely at configuring outbound routing. We'll modify listing 4.2 from earlier. Instead of controlling the messages entering the service, we'll control what messages leave. This is accomplished by removing the selective-consumer router from the inbound endpoint and adding a filtering router to the outbound configuration. Listing 4.3 demonstrates this.

Listing 4.3 Replacing the selective-consumer router with the filtering router

```
<service name="selectiveConsumerService">
    <inbound>
        <jms:inbound-endpoint queue="messages" />
        <vm:inbound-endpoint path="messages" />
    </inbound>
    <outbound>
        <filtering-router>
            <stdio:outbound-endpoint system="OUT" />
            <regex-filter pattern="^STATUS: (OK)$" />
        </filtering-router>
        <forwarding-catch-all-strategy>
            <jms:outbound-endpoint queue="errors" />
        </forwarding-catch-all-strategy>
    </outbound>
</service>
```

We're now accepting all messages on both inbound endpoints, but only passing those whose payloads match the regex filter.

Outbound routing is useful for more than message filtering. As we'll see later in this chapter, Mule's outbound routers enable you to completely control how data leaves your services. You'll see how this enables you to route data to multiple endpoints, compose the responses from those endpoints, and chain endpoints together. Before we dig into some of Mule's supplied routers, let's talk about how filters work in conjunction with routing.

4.2 Using filters with routers

Filtering is an important facet of routing with Mule. Many routers take a filter as an argument that defines or augments their behavior. We've already seen how to configure a filter in listing 4.2; there we configured a regex filter to selectively consume messages whose payloads matched the supplied regular expression. Now we'll look at some other filtering options. For the rest of the examples in this section, we'll see how Clood, Inc., uses the selective-consumer router defined in listing 4.2 to selectively consume messages representing an order.

4.2.1 Filtering by type

One of the simplest types of filters you'll work with is the payload-type filter. It filters messages based on the type of their payload. In the following example we have a

payload-type filter configured to accept messages whose payloads are instances of com.clood.model.Order. This sort of filtering is useful when different types of objects share the same inbound endpoints and you’re only interested in consuming a particular type. In our case, perhaps the endpoint in question is consuming off a JMS queue that contains order and receipt messages. This example shows how to restrict consumption to messages with a com.clood.model.Order payload:

```
<selective-consumer-router>
    <payload-type-filter expectedType="com.clood.model.Order"/>
</selective-consumer-router>
```

The payload-type filter is useful to ensure only a certain data type reaches your components, but sometimes you’ll want to go deeper into the message payload for message selection. Let’s see different ways in which we can consume order messages that are fulfilled.

4.2.2 Filtering by textual content

Regular expression and wildcard filters enable you to filter messages based on their payload’s textual content. These filters are generally applicable when your payload is a String. As we saw in listing 4.2, this allows us to select messages whose payloads match the supplied regular expression. A wildcard filter is useful when you don’t need the power of a full-blown regular expression; it uses shell-like pattern matching to pass messages through. The following example shows a selective-consumer router that consumes any message that contains the string “FULFILLED” in the payload:

```
<selective-consumer-router>
    <wildcard-filter pattern="*FULFILLED*"/>
</selective-consumer-router>
```

Assuming Clood’s “order” messages from earlier were textual, this selective-consumer router would only pass messages containing the string “FULFILLED” somewhere in the message body. Selection based on textual content is useful for unstructured string payloads. For structured payloads such as XML, though, there are better ways to apply filtering. Let’s look at how we can use expression filtering on XML and object payloads.

4.2.3 Filtering with expressions

One of the benefits of working with structured data, such as XML or object graphs, is the availability of structurally aware query tools. Mule provides the JXPath filter to perform filtering on these payloads. The JXPath filter leverages the Apache JXPath library (<http://commons.apache.org/jxpath/>) to perform filtering on XML and objects. Let’s assume Clood, Inc., is passing order data using an XML payload that looks like listing 4.4.

Listing 4.4 XML representation of an order

```
<order>
    <purchaserId>409</purchaserId>
    <productId>1234</productId>
```

```
<status>FULFILLED</status>
</order>
```

As in the regex filter example, we want to filter based on whether the result is set to “FULFILLED”. The difference is that now our payload is XML instead of colon-delimited text. The JXPath filter in the following example will accomplish this for us:

```
<selective-consumer-router>
    <expression-filter evaluator="xpath"
        expression="(order/status)='FULFILLED'"/>
</selective-consumer-router>
```

We specify the evaluator on expression-filter to let it know how to evaluate the expression. In this case we’re using the JXPath evaluator, but other evaluators are available.

The JXPath evaluator can also be used against Java object graphs. Let’s assume that instead of an XML document, our payload was an instance of the JavaBean in listing 4.5.

Listing 4.5 Java class for an order

```
public class Order {
    private long purchaserId;
    private long productId;
    private String status;

    public String getPurchaserId() {
        return purchaserId;
    }

    public void setPurchaserId(long purchaserId) {
        this.purchaserId = purchaserId;
    }

    public long getProductId() {
        return productId;
    }

    public void setProductId(long productId) {
        this.productId = productId;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
}
```

The same expression would evaluate against instances of this Java object as it would against the XML document.

Now we’ll look at logical filters. They let you perform complex filter evaluations by using multiple filters together.

4.2.4 Logical filtering

Logical filters let you perform boolean operations using two or more filters. Mule supplies and, or, and not filters for boolean evaluation. These are equivalent to the `&&`, `||`, and `!` operators in the Java language. Let's see how to combine two filters using an and-filter. Listing 4.6 contains the configuration.

Listing 4.6 Using a logical filter

```
<selective-consumer-router>
    <and-filter>
        <payload-type-filter expectedType="java.lang.String"/>
        <expression-filter evaluator="xpath"
            expression="(order/status)='FULFILLED' "/>
    </and-filter>
</selective-consumer-router>
```

As you probably already guessed, this filter will consume messages that are both of type String and that the supplied XPath expression successfully evaluates. You can also nest logical filters. Listing 4.7 demonstrates this.

Listing 4.7 Nesting logical filters

```
<selective-consumer-router>
    <and-filter>
        <payload-type-filter expectedType="java.lang.String"/>
        <or-filter>
            <expression-filter evaluator="xpath"
                expression="(order/status)='FULFILLED' "/>
            <expression-filter evaluator="xpath"
                expression="/order/productId = '1234' "/>
        </or-filter>
    </and-filter>
</selective-consumer-router>
```

In this example, we're requiring the payload to be a String and to only accept messages whose result is "FULFILLED" or where the productId is 1234.

Filtering is an important aspect of using routing with Mule. It's so important, in fact, that most of the routers we discuss in this chapter are extensions of either the selective-consumer router (on the inbound side) or the filtering router (on the outbound side). Now that you understand inbound routing, outbound routing, and using filters in conjunction with both, we're ready to start our discussion of some of Mule's more important supplied routers.

4.3 Using inbound routers

In this section we'll examine Mule's capabilities to route messages received on inbound endpoints. Let's start our discussion with a router that should be familiar by now: the selective consumer.

4.3.1 Being picky with the selective-consumer router

The selective-consumer router allows you to selectively consume messages from inbound endpoints. A filter is typically provided to the selective-consumer router in order to specify which messages get consumed. Messages that aren't consumed are forwarded to the catch-all strategy defined for the service. If no catch-all strategy is defined then the message is dropped. Figure 4.3 illustrates the selective-consumer router.

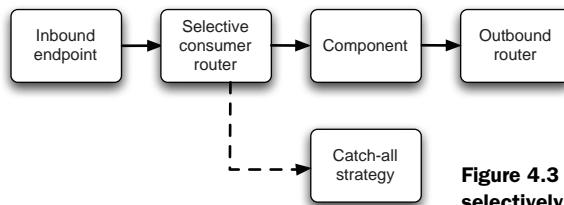


Figure 4.3 The selective-consumer router can selectively route messages.

In listing 4.2 you saw the selective-consumer router working in conjunction with a regex filter to pass messages that matched the supplied regular expression. In that case we were looking for a success string to appear somewhere in the payload of the message. Let's modify this example to check the header of the message. Listing 4.8 demonstrates using the selective-consumer router to only pass messages that have a STATUS header of OK.

Listing 4.8 Using the selective-consumer router to pass messages

```

<service name="selectiveConsumerService">
    <inbound>
        <jms:inbound-endpoint queue="messages" />
        <vm:inbound-endpoint address="vm://messages" />
        <selective-consumer-router> ①
            <expression-filter evaluator="header" expression="STATUS=OK" />
        </selective-consumer-router>
        <forwarding-catch-all-strategy>
            <jms:outbound-endpoint queue="errors" />
        </forwarding-catch-all-strategy>
    </inbound>
    <outbound>
        <pass-through-router>
            <stdio:outbound-endpoint system="OUT" />
        </pass-through-router>
    </outbound>
</service>
  
```

The selective-consumer router on ① will only pass messages that have a header field of STATUS with a value of OK. We've seen quite a bit of the selective-consumer router, so let's turn our attention to something more exciting—the forwarding-consumer router.

4.3.2 Altering message flow with the forwarding-consumer router

You'll occasionally want certain messages to bypass being processed by a component. As we'll see in chapter 6, component processing can be used to enrich a message. This enables you to alter a message as it flows through a service. But you might have a situation where a message doesn't need enrichment. In cases like this, the forwarding router comes in handy. A forwarding router enables you to route selected messages "around" a component. These messages are subsequently sent to the outbound router. Figure 4.4 illustrates this.

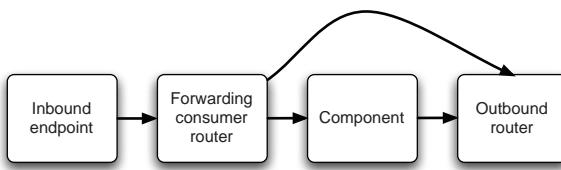


Figure 4.4 The forwarding-consumer router can selectively bypass component processing

The forwarding-consumer router is an extension of the selective-consumer router. As such, it also works in conjunction with a filter to determine what messages to forward. You can configure a forwarding-consumer router without a filter and it'll essentially act like the bridge component, sending all messages to the outbound endpoint. Let's consider an example and see the forwarding-consumer router at work. Listing 4.9 is a modification of listing 4.2, which we saw earlier in this chapter. Instead of dropping messages that don't contain a success string, we'll instead pass them to a component that'll rectify the messages' nonsuccess state, mark them successful, and pass them out to the outbound router. Messages that contain the success string will bypass the component and go directly to the outbound router.

Listing 4.9 Bypassing a component with a forwarding-consumer router

```

<service name="forwardingConsumerService">
    <inbound>
        <jms:inbound-endpoint queue="messages.in"/>
        <forwarding-router>
            <regex-filter pattern="^STATUS: (OK|SUCCESS)$"/>
        </forwarding-router>
        <selective-consumer-router>
            <regex-filter pattern="^STATUS: (CRITICAL)$"/>
        </selective-consumer-router>
    </inbound>
    <component>
        <spring-object bean="messageEnricher" />
    </component>
    <outbound>
        <pass-through-router>
            <jms:outbound-endpoint queue="messages.out" />
        </pass-through-router>
    </outbound>
</service>
  
```

1 **Forward messages matching filter**

2 **Configure the selective-consumer router**

3 **Fix messages in nonsuccess state**

4 **Send messages to queue**

We have a filter enclosed within the forwarding-consumer router. Messages that match this filter will bypass the messageEnricher bean defined on ①. We have a second router, a selective consumer-router, defined on ②. The selective-consumer-router will route messages that haven't been forwarded by the forwarding router. If the message is accepted by the regex-filter defined on the selective-consumer-router it will be passed to the messageEnricher on ③ and ultimately dispatched for outbound routing. Messages not accepted by ④ are handled by the catch-all strategy for the service.

Now that you know how to route around a component, let's look at some inbound routers that deal with organizing data to route into a component.

4.3.3 **Collecting data with the collection aggregator**

Sometimes you need to collect messages from several sources before passing the data to a component for processing. This is useful when performing continuous evaluation of some source of inbound data. A collection aggregator is used to hold this data, then submit it to the component after it's received all the data it needs or a timeout occurs. The collection aggregator in figure 4.5 illustrates this.

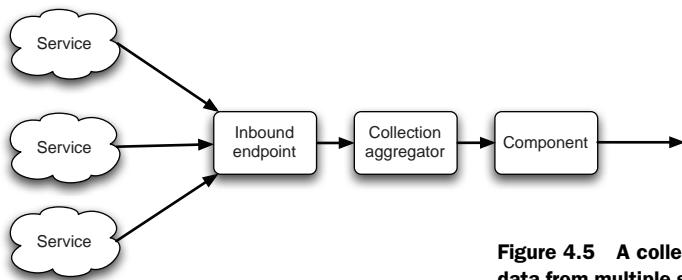


Figure 4.5 A collection aggregator can accept data from multiple sources in a given time frame.

Clood uses a collection aggregator to perform analysis on application response time metrics accumulated by their monitoring systems. Nagios instances deployed across Clood's environment perform periodic HTTP checks against their client's web sites. The response times of these checks are published to a JMS topic. Subscribers to this topic can then process and handle the data appropriately. One such subscriber consumes these metrics for 10 minutes and republishes an average of the group.

Listing 4.10 illustrates how this is configured.

Listing 4.10 Aggregating messages with the collection aggregator router

```

<service name="collectionAggregatorService">
    <inbound>
        <jms:inbound-endpoint topic="metrics.responsetimes">
            <jms:jmsmessage-to-object-transformer
                returnClass="com.muleinaction.ResponseTime"/>
        </jms:inbound-endpoint>
        <collection-aggregator-router timeout="600000" /> ←
    </inbound>
    <component>
  
```

① Define collection aggregator router

```

<method-entry-point-resolver>
    <include-entry-point method="averageResponseTimes" />
</method-entry-point-resolver>
<spring-object bean="metricService"/>
</component>
<outbound>
    <pass-through-router>
        <jms:outbound-endpoint topic="metrics.avg.responsetimes" />
    </pass-through-router>
</outbound>
</service>

```

The collection aggregator router defined on ① will accept response time metrics for a 10-minute interval. When this interval is up, the collection of messages will be sent to the metricService component configured on ②. This will average the response time of the metrics in the collection and return the result, which will then be sent to the JMS topic metrics.avg.responsetimes. You might've noticed the method-entry-point-resolver configured on ③. This explicitly tells Mule what method to invoke on the metricService defined on ③. We'll discuss entry point resolution in detail when we discuss components in chapter 6.

Grouping of inbound messages is accomplished by looking at the correlationId on each message. The correlationId is set by the outbound routers of each external service. The collection aggregator will wait for messages that share the same correlationId and group them together in a list, which is passed to the component for evaluation. The correlationId is usually something that makes sense in the given business scenario. In this example it might be the ID of the monitoring job being run. This will let the service accept monitoring results from different Nagios instances in parallel.

In addition to the correlationId, a correlationGroupSize can also be set. This determines the number of messages that are needed before the list can be sent to the component for processing. By default, if the correct number of messages needed to meet the correlationGroupSize isn't reached by the timeout interval, Mule logs an error and the data isn't sent to the component. You can override this behavior by setting the failOnTimeout property on the collection aggregator router to true. This will allow incomplete groups to be sent to the component.

Remember that the collection aggregator router is an extension of the selective-consumer router, so you can make full use of filtering when processing messages. Listing 4.11 will use a JXPath expression filter to only accept metrics for a particular client.

BEST PRACTICE Use filters in conjunction with inbound routers to harness selective-routing capabilities.

Listing 4.11 Using an expression-filter with the collection aggregator

```

<service name="collectionAggregatorService">
    <inbound>
        <jms:inbound-endpoint topic="metrics.responsetimes">
            <jms:jmsmessage-to-object-transformer

```

```

        returnClass="com.muleinaction.ResponseTime" />
    </jms:inbound-endpoint>
    <collection-aggregator-router timeout="60000">
        <expression-filter evaluator="xpath"
                           expression="(metric/clientId)='client1'">           ←
        </collection-aggregator-router>
        <collection-aggregator-router timeout="600000" />
    </inbound>
    <component>
        <method-entry-point-resolver>
            <include-entry-point method="averageResponseTimes" />
        </method-entry-point-resolver>
        <spring-object bean="metricService" />
    </component>
    <outbound>
        <pass-through-router>
            <jms:outbound-endpoint topic="metrics.avg.responsetimes" />
        </pass-through-router>
    </outbound>
</service>

```

Consume messages selectively

Now that we can process multiple messages as a group, let's look at how we can guarantee that we only process a message once.

4.3.4 Insuring atomic delivery with the idempotent receiver

It can be important to guarantee that a message is only processed once. Obvious examples abound in the banking industry. You want to be sure you're not processing the same withdrawal or debit twice, for instance. It's conceivable, though, for a message to be delivered or sent more than one time. Someone might hit the Submit button twice on an online banking form or a malicious user may be deliberately injecting duplicate messages into your system.

Mule's idempotent-receiver router allows you to ensure that a message is consumed atomically. The idempotent-receiver router generates an ID for each message it consumes. The IDs are subsequently stored and scanned as new messages arrive. Duplicate IDs are handled by the catch-all strategy for the router, if one is defined. If no strategy exists, the message is discarded and some information is logged. Figure 4.6 illustrates how this works.

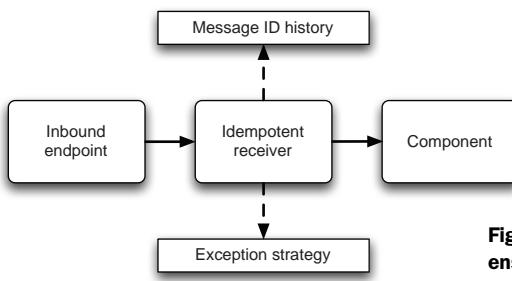


Figure 4.6 The idempotent-receiver router ensures that only one copy of a message is routed to a component.

Let's return to Clood, Inc.'s order provisioning system. After some time in production, it became apparent that Clood's order endpoints were receiving duplicate order messages, ultimately resulting in clients getting billed twice for their provisioning fees. This turned out to be the result of a bug in Clood's JMS provider that has subsequently been patched, but Clood's management is rightfully worried about this issue occurring again. Since every order is assigned a unique ID as it enters the system, it's possible to use an idempotent receiver to ensure that an endpoint only processes the same order once. Listing 4.12 shows how to implement this.

Listing 4.12 Insuring atomic delivery with the idempotent-receiver router

```
<service name="idempotentReceiverService">
    <inbound>
        <jms:inbound-endpoint queue="orders" />
        <idempotent-receiver-router idExpression="#[header:orderId]">
            <simple-text-file-store directory=".order-ids"/>
        </idempotent-receiver-router>
        <forwarding-catch-all-strategy>
            <jms:outbound-endpoint queue="duplicate.orders" />
        </forwarding-catch-all-strategy>
    </inbound>
    <component>
        <method-entry-point-resolver>
            <include-entry-point method="process" />
        </method-entry-point-resolver>
        <spring-object bean="orderService" />
    </component>
</service>
```

The diagram illustrates the flow of the idempotent-receiver router. Step 1, labeled with a blue circle containing a number 1, shows an arrow pointing from the 'Evaluate expression to extract ID' text to the 'idExpression="#[header:orderId]"' attribute in the configuration. Step 2, labeled with a blue circle containing a number 2, shows an arrow pointing from the 'Store IDs in given directory' text to the 'simple-text-file-store' component. Step 3, labeled with a blue circle containing a number 3, shows an arrow pointing from the 'Send duplicate orders to queue' text to the 'forwarding-catch-all-strategy' component.

The idempotent-receiver router is defined on ①. The first thing of interest is the `idExpression`; this is the value we'll be using as the idempotent identifier. We're specifying the `idExpression` using Mule's expression evaluator syntax. Expression evaluators allow you to dynamically extract data from a message using various mechanisms, such as taking the value of the message's headers or extracting the message's payload.² In this case, we're extracting an `orderId` from the header of the message and using this as the idempotent ID. We're declaring how we want to store and reference the IDs in ②. In this case, we're using `simple-text-file-store` to save messages to a text file (you can implement more complex schemes to save messages if the simple text file store isn't suitable). If you choose not to specify an `idExpression`, Mule will use the message's ID as the default. Duplicate messages are handled by the exception strategy for the model. In this case, the forwarding catch-all strategy on ③ will send duplicate orders to the `duplicate.orders` queue where they can be dealt with appropriately.

Let's wrap up our discussion of inbound routing by examining the wiretap router, which enables us to watch the messages coming into a service.

² Full information regarding the available evaluators is available in appendix A.

4.3.5 Snooping messages with the wiretap router

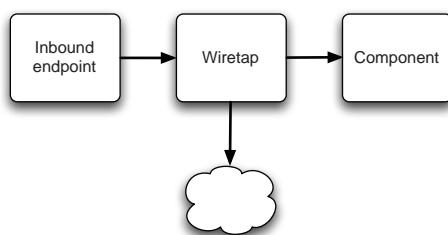
The wiretap router lets you inspect traffic going into a service. Analogous to a wiretap on a phone, Mule's wiretap enables you to copy messages to a remote endpoint without the knowledge of other inbound routers or the component. This is useful in a variety of scenarios, from message logging to quality-of-service and billing applications. Figure 4.7 illustrates the wiretap router in use.

The wiretap will send a copy of each message that arrives for the component to an outbound endpoint. Let's consider an example. Clood, Inc., is interested in getting real-time statistics for the orders it's processing. Clood's order processing is handled by a Spring-configured bean called `orderProcessingService`. This object contains all the logic for processing orders as they arrive off a JMS queue called `orders`. Clood could modify `orderProcessingService` to collect statistics on orders as they come in, but this would involve modifying already well-proven order processing code as well as doing a redeployment of the code to all Clood's Mule instances. It perhaps most importantly introduces coupling between the statistics collection and the order processing. Let's see how Clood uses the wiretap router to "listen" as order messages pass into the `orderProcessingService` and send a copy to a separate endpoint for statistics generation. Listing 4.13 illustrates this.

Listing 4.13 Sniffing traffic with a wiretap router

```
<service name="wiretapService">
    <inbound>
        <jms:inbound-endpoint queue="orders" />
        <wire-tap-router>
            <vm:outbound-endpoint path="statisticsService" />
        </wire-tap-router>
    </inbound>
    <component>
        <spring-object bean="orderProcessingService" />
    </component>
</service>

<service name="wiretapReceiver">
    <inbound>
        <vm:inbound-endpoint path="statisticsService">
            <jms:jmsmessage-to-object-transformer
                name="JmsMessageToString"
                returnClass="java.lang.String" />
        </vm:inbound-endpoint>
    </inbound>
    <component>
```



Define wiretap router

Accept copy of each order message

```

<spring-object bean="statisticsService" />
</component>
</service>

```

The wiretap router configured on ① will send a copy of each message received by the orderProcessingService to the statisticsService endpoint. Like the forwarding consuming router, the wiretap router is an extension of the selective consuming router, so you can use filtering to tap only the messages you're interested in. Remember that the wiretap router simply copies and forwards the message; it doesn't modify messages that pass through it.

In this section we've nose-dived into Mule's inbound routing capabilities. You've seen how selective consuming routers form the basis for Mule's inbound routers. We looked at how we can use this in conjunction with forwarding-consumer routers to alter message flow around our components. You then saw how you can aggregate collections of inbound messages using a collection aggregator. Finally, you saw how you can use the idempotent receiver to guarantee message atomicity as well as use the wiretap router to snoop on traffic. Now that we've looked at Mule's inbound routers in depth, we can turn our attention to outbound routers.

4.4 Outbound routing

Outbound routers dictate how messages leave a service. In the last chapter, you saw how to use the pass-through router to send messages to a single endpoint. You also saw how to apply the chaining router to use the response of one endpoint as the input into another. In this section we'll investigate more of Mule's outbound routing options.

4.4.1 Being picky with the filtering router

The filtering router lets you control what messages leave your component. Much like the inbound selective-consumer router, the filtering router uses a filter to determine what to let through. The routers we describe in this section are extensions of the filtering-outbound router, so it's important to be familiar with it. The filtering router is illustrated in figure 4.8.

In listing 4.14 we're using a payload-type filter in conjunction with the filtering router to only allow messages with Double payloads to pass to the outbound JMS endpoint.

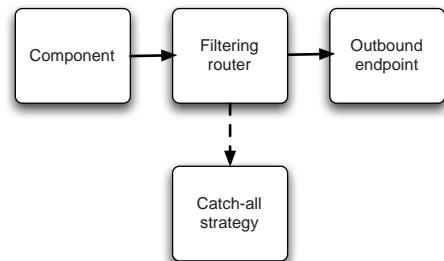


Figure 4.8 A filtering router stops select messages from leaving a component.

Listing 4.14 Using the filtering router to filter outbound messages

```

<service name="filteringRouterService">
<inbound>
    <jms:inbound-endpoint queue="metrics.responsetimes"/>
    <collection-aggregator-router timeout="600000" />

```

```

</inbound>
<component>
    <method-entry-point-resolver>
        <include-entry-point method="averageResponseTimes" />
    </method-entry-point-resolver>
    <spring-object bean="metricService" />
</component>
<outbound>
    <filtering-router>
        <jms:outbound-endpoint queue="metrics.avg.responsetimes" />
        <payload-type-filter expectedType="java.lang.Double" />
    </filtering-router>
    <logging-catch-all-strategy/>
</outbound>
</service>

```

Only accept messages with Double payloads ①

Log messages ② that don't pass

The configuration, as you can see, is almost identical to that of the selective-consumer router. We're specifying a filter on ① that'll block any message that doesn't have a payload of Double. You might've noticed that we've configured a logging catch-all strategy on ②. Since we should never receive messages that have a payload other than Double, we want the filtering router to log these messages to the logging catch-all strategy. We can then act on the information and debug why we're receiving data that isn't a Double.

Using filtering with routers should be old hat by now, so let's look at some other more creative routing options. We'll start with the static recipient list.

4.4.2 Sending to multiple endpoints with the static recipient list

The static-recipient-list router lets you simultaneously send the same message to multiple endpoints. You'll usually use a static recipient list when each endpoint is using the same transport. This is often the case with VM and JMS endpoints. Figure 4.9 illustrates the static recipient list.

Clood uses *error channels* to manage the reporting of exceptions in their infrastructure. When errors occur during message processing, Clood sends the message to multiple recipients for processing. The static recipient list comes in handy in situations like this. Listing 4.15 shows how Clood uses the static recipient list to send a message that arrives on an inbound endpoint to three JMS queues.³

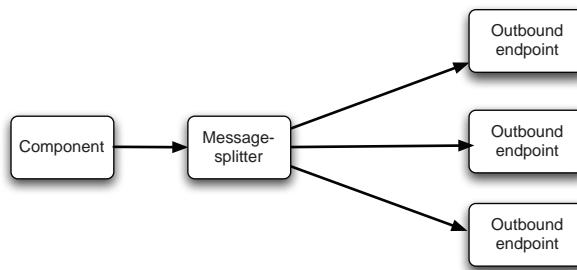


Figure 4.9 A static recipient list can route data to multiple endpoints.

³ We'll talk about exception handling extensively in chapter 8.

Listing 4.15 Send outbound messages to list of endpoints

```

<service name="staticRecipientListService">
    <inbound>
        <vm:inbound-endpoint address="vm://errors"/>
    </inbound>
    <outbound>
        <static-recipient-list-router>
            <recipients>
                <spring:value>jms://errors.ops</spring:value>
                <spring:value>jms://errors.enqr</spring:value>
                <spring:value>jms://errors.reports</spring:value>
            </recipients>
        </static-recipient-list-router>
    </outbound>
</service>

```

The configuration is fairly straightforward. We define a static-recipient-list router on ① and enumerate our outbound endpoints on ②. The `spring:value` element might seem strange to you. This is simply Mule reusing the Spring schema's `value` element.

BEST PRACTICE Use static recipient lists when sending the same message to endpoints using identical transports.

The static-recipient-list router is useful for sending data across homogeneous endpoints, but it's often useful to broadcast a message across different transports. Let's look at the multicasting router, which we saw in the beginning of this chapter, to see how to accomplish this.

4.4.3 Broadcasting messages with the multicasting router

The multicasting router is similar to the static recipient list in that it simultaneously sends the same message across a set of outbound endpoints. The difference is that the multicasting router is used when the endpoint list contains different types of transports. We saw this situation in the first example of this chapter. If you recall, we had a multicasting router receiving statistical data from a component. Our requirement was to let our boss know when new data was being inserted into the database. The data from the component was numeric and as such was suitable for database insertion. You probably want to send your boss something more informative than a set of numbers in an IM, though. This is where the multicasting router comes in handy; it lets you tailor the data for each outbound endpoint. Figure 4.10 illustrates the multicasting router.

We saw how Clood, Inc., uses multicast routing at the beginning of this chapter. Let's add another endpoint to this multicast router now to further demonstrate how it works. Listing 4.16 shows how to do this.

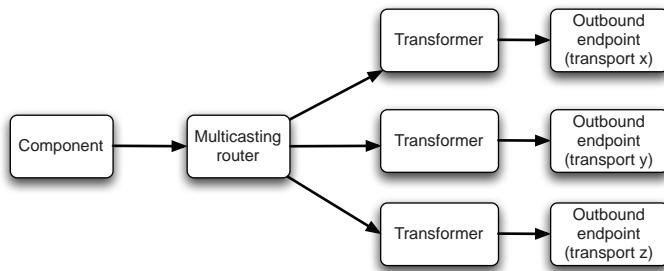


Figure 4.10 The multicasting router can send messages to multiple endpoints over different transports.

Listing 4.16 Using the multicasting router to send data to multiple endpoints

```

<outbound>
    <multicasting-router>
        <jdbc:outbound-endpoint queryKey="statsInsert" />
        <xmpp:outbound-endpoint user="mule"
            password="mule"
            host="jabber.clood.com"
            port="5222"
            recipient="joe"
            transformer-refs="StatToInstantMessageText" />
        jms:outbound-endpoint topic="alerts" />
    </multicasting-router>
</outbound>
  
```



We simply need to add the additional endpoint, in this case a JMS outbound endpoint publishing to a topic on ①, and now data leaving the component will be inserted into the database, sent to Clood, Inc.'s Jabber server, and published to a JMS topic.

BEST PRACTICE Use the multicasting router when sending the same message to endpoints using different transports.

Transformers can be used on each endpoint to put the contents of the message in a suitable format (as we mentioned previously, we'll be talking more about transformation in the next chapter).

The multicasting router is an important piece of functionality in the integration world. It's a rare occasion when every party interested in messages from a service is "speaking the same language" from a transport perspective. The multicasting router allows you to easily move the same messages across these different endpoints. Let's turn our attention now to the chaining router. As you recall from the last chapter, this lets you take advantage of the response from synchronous endpoints in your service's outbound configuration.

4.4.4 Service composition with the chaining router

Service composition is an important part of enterprise integration. Often, you'll find yourself needing to feed the output of some service into the input of another. While it's possible to model this using a series of VM queues, it would be convenient to have

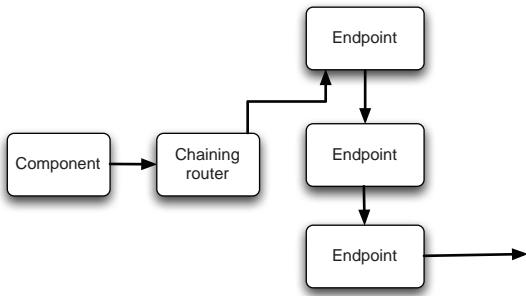


Figure 4.11 The chaining router can sequentially and synchronously send data to multiple endpoints.

an abstraction in place for this sort of operation. Luckily, Mule provides a chaining router for this situation. As illustrated in figure 4.11, the chaining router allows you to feed the output of one endpoint into the input of another.

The examples in the previous chapter used the chaining router in conjunction with an STDIO outbound endpoint. Let's look at another example. Listing 4.17 shows a chaining router that accepts a ZIP code from a STDIO inbound endpoint. The ZIP code is bridged to the chaining router, which is configured with two endpoints. It submits the ZIP code to the first endpoint, a SOAP service that returns the weather information for the corresponding area. When the SOAP result is returned, it's passed to the outbound JMS endpoint, which places the data on a queue.

Listing 4.17 Using the response of an endpoint as the input for another endpoint

```

<service name="chainingRouterService">
    <inbound>
        <stdio:inbound-endpoint system="IN" />
    </inbound>
    <outbound>
        <chaining-router>
            <outbound-endpoint address=
"http://www.webservicex.net/usweather.asmx?WSDL&method=\nGetWeatherReport"/>
            <jms:outbound-endpoint queue="weather" />
        </chaining-router>
    </outbound>
</service>
    
```

The chaining router is a good choice for small and ad hoc service composition. You'll see an alternative way to perform service composition when we look at using BPM engines with Mule in part 3 of this book.

We've now looked at a few ways to send a single message across multiple endpoints. The static-recipient-list router lets you send a message to a set of homogeneous endpoints. The multicasting router lets you send a message across a set of heterogeneous endpoints. Finally, the chaining router lets you feed the result of one endpoint into the input of another. This is all great for a single message, but how do we deal with sending different messages to individual endpoints? The message splitter router we

consider next will show us how to split a message up and route its pieces separately to different endpoints.

4.4.5 Chopping up messages with the message splitter

Dealing with collections of data sometimes calls for handling independent elements of the collection differently. This can arise when your components are supplying collections or XML documents to outbound endpoints. In these situations, you might want to split the collection or XML document into its constituent elements and deal with each element independently. Mule provides the message-splitter router for these cases. Figure 4.12 shows how message-splitter routers work.

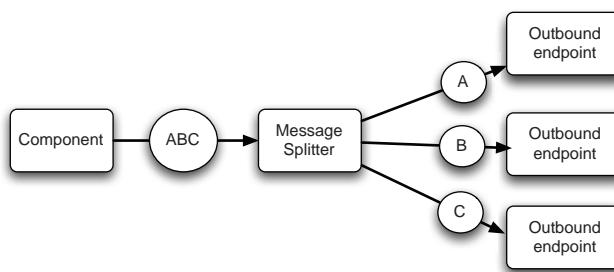


Figure 4.12 A message splitter can break up a message and send its parts to multiple endpoints.

In this section, we'll start by looking at using the list message splitter in conjunction with a JXPath filter. We'll route a collection of objects to different endpoints based on their properties. Then we'll examine using the XML message splitter to splice up an XML file and route each element separately.

Let's consider the Order class from listing 4.5. In listing 4.18 we've implemented an inbound endpoint to accept a list of Order objects off a JMS queue. The list is bridged to an outbound message splitter. The list-message-splitter-router is configured to use a JXPath filter to route each message differently. The JXPath filter will select the route based on the value of the status property of each Order object.

Listing 4.18 Using JXPath expressions to split up a collection of orders

```

<service name="listMessageSplitterService">
    <inbound>
        <vm:inbound-endpoint path="orders" />
    </inbound>
    <outbound>
        <list-message-splitter-router>
            <vm:outbound-endpoint path="orders.fulfilled">
                <expression-filter evaluator="xpath"
                    expression="status = 'FULFILLED'" />
            </vm:outbound-endpoint>
            <vm:outbound-endpoint path="orders.pending">
                <expression-filter evaluator="xpath"
                    expression="status = 'PENDING'" />
            </vm:outbound-endpoint>
        </list-message-splitter-router>
    </outbound>
</service>
  
```

1 Define list message-splitter router

```

<vm:outbound-endpoint path="orders.unknown" /> ←
  </list-message-splitter-router>
</outbound>
</service>

```

Define “fall-through”
outbound endpoint ②

The list-message-splitter router defined on ① sends messages to different queues depending on the value of the status property. This allows us to send completed orders to one queue and pending orders to another. Note that we have an outbound endpoint defined on ② without a filter. This acts as a “fall-through” endpoint; if none of the other endpoints’ filters match, the message will be routed through this endpoint.

Let’s assume that instead of passing around a list of Order objects, we instead received XML representing orders. This document looks like listing 4.19.

Listing 4.19 An XML document with orders

```

<orders>
  <order>
    <id>1234</id>
    <status>FULFILLED</status>
  </order>
  <order>
    <id>1235</id>
    <status>PENDING</status>
  </order>
  <order>
    <id>1236</id>
    <status>PENDING</status>
  </order>
</orders>

```

We want to split up the document into individual documents, with each containing the XML for one order. We’ll then route each order to the appropriate queue as before. Listing 4.20 uses the filtering XML message splitter to take care of this.

Listing 4.20 Splitting up an XML payload

```

<service name="xmlMessageSplitterService">
  <inbound>
    <vm:inbound-endpoint path="orders" />
  </inbound>
  <outbound>
    <mulexml:message-splitter splitExpression="/orders/order" > ←
      <vm:outbound-endpoint path="orders.fulfilled">
        <expression-filter evaluator="xpath"
          expression="/order/status = 'FULFILLED' "/>
      </vm:outbound-endpoint>
      <vm:outbound-endpoint path="orders.pending">
        <expression-filter evaluator="xpath"
          expression="/order/status = 'PENDING' "/>
      </vm:outbound-endpoint>
      <vm:outbound-endpoint path="orders.unknown" />
    
```

Split XML message ①

```

</mulexml:message-splitter>
</outbound>
</service>

```

As you can see, in ① we've changed `list-message-splitter-router` to `xml-message-splitter`, defined in the `mulexml` namespace. We've supplied a `splitExpression` to the router to instruct it how to slice up the XML. In this case we want to extract each order element from the document. We then use an XPath expression to evaluate the status of each order document and route it accordingly.

NOTE The payload of messages leaving the message-splitter router will be instances of `org.dom4j.Document`.

We've just seen how we can split up a message and route its constituent parts. What if we want to go a step further and elicit a response from each endpoint, then make a decision based on the response as a whole? We'll see how as we consider asynchronous-reply routers.

4.4.6 Using asynchronous-reply routers

The chaining router enabled us to set up a sequence of outbound endpoints where we could feed the output of one endpoint as input to the next. Sometimes, though, we need to send a message to a set of endpoints in parallel, wait for their output, and send an atomic response based on the aggregate data. An asynchronous-reply router makes this possible. As you can see in figure 4.13, the async reply routing occurs after all the outbound endpoints have returned data or have timed out.

You'll typically use VM queues to aggregate the responses and trigger the asynchronous-reply router, as we'll see shortly. Like the other routers, the asynchronous-reply router's place in the configuration file is in line with its location in the event context. The `async-reply` element signals asynchronous reply routing and occurs after the outbound configuration.

Let's consider an example of where an asynchronous-reply router is useful. Clood, Inc., has a series of virtual machine (VM) farms deployed across their organization. Customers use a web services API to provision and deprovision VMs on this environment. The customer doesn't care (or know) what VM farm their VM is spun up on. Clood, though, wants to distribute VM provisioning more or less evenly across each

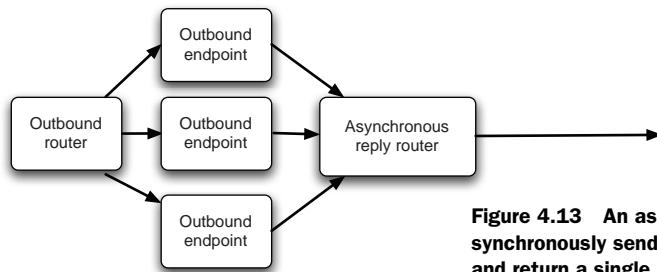


Figure 4.13 An asynchronous-reply router can synchronously send messages to multiple endpoints and return a single response

farm. To accomplish this, Clood is using an async-reply router to query each VM farm, collect the results, and return the farm with the lowest number of running VMs. Listing 4.21 lists the Mule configuration to accomplish this.

Listing 4.21 Using the async-reply router to select an appropriate VM farm

```
<model name="asyncRequestReplyModel">

    <service name="asyncRequestReplyService">
        <inbound>
            <vm:inbound-endpoint
                path="farmRequests"
                synchronous="true" />
        </inbound>
        <outbound>
            <multicasting-router>
                <vm:outbound-endpoint
                    path="farm1"
                    />
                <vm:outbound-endpoint
                    path="farm2"
                    />
                <vm:outbound-endpoint
                    path="farm1"
                    />
                <reply-to
                    address="vm://farmResponses" />
            </multicasting-router>
        </outbound>
        <async-reply>
            <vm:inbound-endpoint
                path="farmResponses" />
            <custom-async-reply-router
                class="FarmResponseAggregator" />
        </async-reply>
    </service>

    <service name="farm1">
        <inbound>
            <vm:inbound-endpoint
                path="farm1"
                synchronous="true" />
        </inbound>
        <component>
            <method-entry-point-resolver>
                <include-entry-point method="getStatus" />
            </method-entry-point-resolver>
            <spring-object bean="farm1"/>
        </component>
    </service>

    <service name="farm2">
        <inbound>
            <vm:inbound-endpoint
                path="farm2"
                synchronous="true" />
        </inbound>
        <component>
            <method-entry-point-resolver>
                <include-entry-point method="getStatus" />
            </method-entry-point-resolver>
            <spring-object bean="farm2"/>
        </component>
    </service>

```

The diagram illustrates the flow of requests and responses in the Mule configuration:

- 1 Receives farm status requests**: An arrow points from the `farmRequests` endpoint to the `multicasting-router`.
- 2 Multicast request to each farm**: An arrow points from the `multicasting-router` to the `farm1` and `farm2` endpoints.
- 3 Send each response to reply-to address**: An arrow points from the `custom-async-reply-router` back to the `reply-to` address.
- 4 Configure async-reply router**: A callout points to the `vm:inbound-endpoint` within the `async-reply` block.
- 5 Define custom router**: A callout points to the `spring-object` bean within the `component` block.
- 6 Farm endpoint**: Callouts point to the `method-entry-point-resolver` and `spring-object` within the `farm1` service component.
- 7 Farm endpoint**: Callouts point to the `method-entry-point-resolver` and `spring-object` within the `farm2` service component.

```

        path="farm2"
        synchronous="true" />
    </inbound>
    <component>
        <method-entry-point-resolver>
            <include-entry-point method="getStatus" />
        </method-entry-point-resolver>
        <spring-object bean="farm2" />
    </component>
</service>

<service name="farm3" > ← ⑧ Farm endpoint
    <inbound>
        <vm:inbound-endpoint
            path="farm3"
            synchronous="true" />
    </inbound>
    <component>
        <method-entry-point-resolver>
            <include-entry-point method="getStatus" />
        </method-entry-point-resolver>
        <spring-object bean="farm3" />
    </component>
</service>
</model>
```

The farm status request is first accepted by ①. On ② we multicast this request out to each farm for a response. For the purposes of this example, these are just stubs defined on ⑥, ⑦, and ⑧ that randomly generate VM farm data. As the farm metrics are returned, they're sent to the endpoint defined by the reply-to definition on ③. Our async-reply configuration begins on ④. The responses from each VM farm are received by the VM inbound endpoint. Once all the responses are received they're passed to the custom-async-reply router defined on ⑤. The defined class, `FarmResponseAggregator`, will choose the farm with the lowest number of provisioned VMs and return it as the result of the synchronous inbound endpoint defined on ①. This result is then returned by the VM endpoint we started with on ①.

If a response isn't received from each farm then, by default, the responses aren't aggregated and the request fails. This often isn't the desired behavior. Consider if one of the VM farms is under heavy load and can't respond to the status request in a timely fashion. We still want to provision the VM to another farm, not abandon the request. The `async-reply` router, much like the collection aggregator, allows you to set the `failOnTimeout` value to `true` in these scenarios. This will cause the partial result to be aggregated to evaluate the response.

NOTE The implementation for `FarmSelectionService` is included with the source code for the examples in this book. Mule also ships with some stock reply routers: you can replace the custom-async-reply router with either a single-async-reply router or a collection-async-reply router to try them out. The single-async-reply router will return the first message that's received on its endpoint. Conversely, the collection-async-reply router will return all the messages received on the endpoint.

By now you've seen quite a few ways to control message flow out of your components. Just as you used the selective consumer routing to control inbound message flow, you saw how you can use the filtering router to control what messages leave a component. You saw how to dispatch messages to a list of receivers with the static recipient list as well as broadcast a message through different transports with the multicasting router. You saw how to do rudimentary service composition using the chaining router as well as message splitting with the message-splitting router. Finally we looked at using the async-reply router to aggregate data from a multitude of sources to return a single result.

4.5 **Summary**

In this chapter we explored Mule's routing capabilities. You saw how inbound and outbound routers control how data moves through your services. We took a look at Mule's filtering capabilities and how they apply to routing. We examined each of Mule's core routers in depth and saw examples of each at work. Perhaps most importantly, you may have noticed that services, endpoints, and components can be ignorant of routing changes. This loose coupling allows you to drastically change the routing behavior of your services without modifying your component's code, endpoint definitions, or external services.

You're now able to configure Mule, work with its transports, and route data between services. The next two chapters will round out your first steps with Mule. We'll first look at transformation, which allows you to adapt messages based on their context. We'll then look at components, which allow you to embed your own business logic into Mule.



Transforming data with Mule

In this chapter

- Message transformation principles
- Common transformers from Mule Core
- JMS and XML transformers
- Writing custom transformers

Nowadays, every application understands XML and uses interoperable data structures, right? If you replied yes, be informed that you live in Wonderland, and sooner or later, you'll awake to a harsh reality! If, like most of us, you answered no, then you know why data transformation is such a key feature of an ESB.

We're still far away from a world of unified data representation, if we ever reach that point. Unifying data is a tremendous effort. For public data models, it takes years of work by international committees to give birth to complete and complex standards. In large corporations, internal working groups or governance bodies also struggle to establish custom unified data representations. In the meantime, the everyday life of a software developer working on integration projects is fraught with data transformation challenges.

In this chapter, you'll learn how Mule can help you with your data transformation needs. We'll first discuss the way transformers behave and how you can work with them. We'll then review the usage of a few existing transformers that illustrate the diversity and capacities of Mule:

- *Core transformers*—General-purpose transformers from the core library of Mule
- *XML transformers*—Specialized transformers from the XML module
- *JMS transformers*—Transport-specific transformers for JMS

While discovering the existing transformers, we'll look at how Clood, Inc., uses them to satisfy their own needs but also the needs of one of their clients. We'll then look at a custom transformer Clood has rolled out to support client-facing email generation.

When you're done with this chapter, you'll have a clear picture of how Mule will remove the data transformation millstone from your integration projects.

5.1 Working with transformers

A Mule transformer has simple behavior, as illustrated by the flow chart shown in figure 5.1. As this diagram suggests, a transformer strictly enforces the types of data it receives and outputs. This can be relaxed by configuration: in that case, a transformer won't report an exception for bad input, but will return the original message unchanged, *without enforcing the expected result type (return class)*. Therefore use this option sparingly.

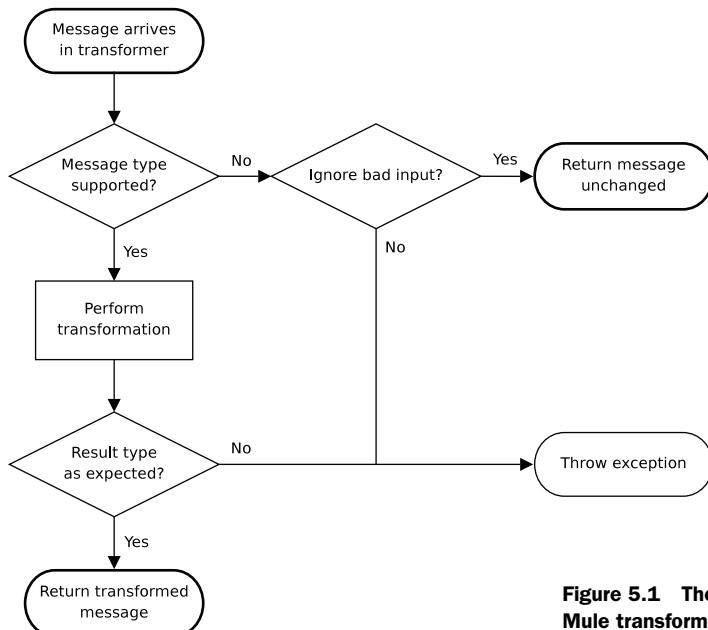


Figure 5.1 The default behavior of a Mule transformer

NOTE When a message transits in Mule, it's an event, more specifically an instance of `org.mule.api.MuleEvent`, that's moved around. This object carries not only the actual content of the message but also the context of the event. This context is composed of references to different objects, including security credentials, if any, the session in which this request is processed, and the Mule context, through which all the internals of the ESB are accessible. The content of a message, also known as *payload*, is itself wrapped in an instance of `org.mule.api.MuleMessage`, which provides different means of accessing the payload under different forms. A `MuleMessage` also contains properties, much like the header of a SOAP envelope or the properties of a JMS message, and can also have multiple named attachments (read more about this in section 13.3.1).

A transformer can alter a message in different ways:

- *Payload type transformation*—The data type of the message payload is transformed from one form to another. For example, a `java.util.Map` is transformed into a `javax.jms.MapMessage`.
- *Payload format transformation*—The data format of the message payload is transformed from one form to another. For example, a DocBook XML instance is transformed into an XSL-FO instance.
- *Properties transformation*—The properties of the message are modified, whether by adding new properties or by removing, renaming, or changing the values of existing properties. For example, a message needs a particular property to be set before being sent to a JMS destination.

Transformers often come in pairs, with the second transformer able to undo what the first one has done. For example, compression and decompression, which are payload format transformations, are handled by two different transformers. A transformer that can reverse the action of another is called a *round-trip transformer*. Making a message go through a transformer and its round-trip should restore the original message.

NOTE *Transformation versus adaptation* Wouldn't this be a great title for a controversial book? In Mule, message adaptation and transformation are two different and complementary notions. Mule uses the concept of message adaptation to standardize the handling of messages whose natures are specific to each transport. Mule's moving parts need to access the message properties and the byte payload in a unified manner. Mule message adapters, following the adapter design pattern,¹ do this by wrapping the transport-specific message payload with an adapter. This adapter is stateful: it contains not only the original message that it wraps, but also significant information extracted from it (such as encoding, properties, or attachments) and some extra context (such as the current exception). Transformation allows you to alter the content of the message adapter:

¹ See http://en.wikipedia.org/wiki/Adapter_pattern

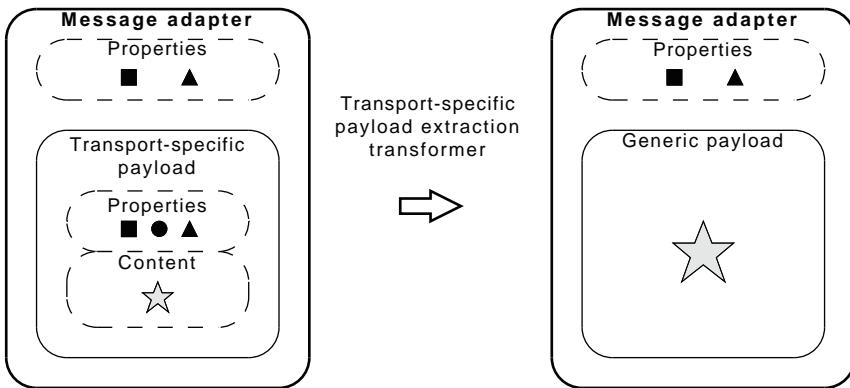


Figure 5.2 Message adaptation and transformation have different but complementary goals.

either its payload or its properties, as discussed previously. Figure 5.2 illustrate these two concepts: a message adapter that wraps a transport-specific message, and that has extracted some of its properties, has its payload transformed into a transport-agnostic one.

Understanding how adaptation complements transformation will help you understand the behavior of the transport-specific transformers, which will be discussed in the next section.

Mule is extremely rich in terms of available transformers: each Mule library you'll use in your project can potentially contain transformers:

- The Mule core contains a wealth of general-purpose transformers: we'll detail a few of them in section 5.3.
- Modules can also contain transformers: in section 5.4, we'll look at some of those that come with the XML module.
- Transports may provide transformers as well: we'll review the ones that are bundled with the JMS transport in section 5.5.

You've just discovered the basics of message transformation in Mule. You now need to learn the fundamentals and the subtleties of transformers' configuration in order to use them efficiently in your own projects.

5.2 Configuring transformers

To use a transformer in your Mule instance, you need to first declare it. This is usually done globally in your configuration file, as we discovered in chapter 2. The following configuration fragment shows the declaration of an object-to-byte array transformer (discussed in 5.3.1) named `ObjectToByteArray`:

```
<object-to-byte-array-transformer name="ObjectToByteArray" />
```

A transformer element supports two common configuration attributes, in addition to its name:

- *ignoreBadInput*—The purpose of this boolean attribute has been demonstrated in figure 5.1. As we said earlier, this instructs the transformer to perform no action and return the message unchanged in case its type isn't supported.
- *returnClass*—This attribute allows you to configure the fully qualified name of the type of class that the transformer is expected to return. This is useful if you want to strictly enforce a stricter type than the transformer's default (for example, a transformer might target `java.lang.Object` whereas you want it to produce only `java.util.Map` objects).

After it's been declared, and thus named, you can refer to the transformer from your endpoint. Whether they're global or not, endpoints support two styles of transformer references—either whitespace-separated names listed in an attribute or defined by specific child elements. This is illustrated in listing 5.1, where you can see a chain of three inbound transformers defined on an endpoint in two different ways.

Listing 5.1 Two different ways to declare a similar chain of transformers on an endpoint

```
<http:inbound-endpoint host="localhost" port="8080"
    transformer-refs="ObjectToByteArray
                      DocbookToFO
                      ObjectToString" />
    ↗
    ↗ List of inbound transformer references

<http:inbound-endpoint host="localhost" port="8080">
    <transformers>
        <transformer ref="ObjectToByteArray" />
        <transformer ref="DocbookToFO" />
        <transformer ref="ObjectToString" />
    </transformers>
</http:inbound-endpoint>
    ↗
    ↗ List of inbound transformer reference elements
```

The second approach is more foolproof, because individual transformer references can be checked by advanced XML editors, whereas a list of references will only be checked at runtime when Mule tries to load the configuration. Note that the `transformers` parent element can be omitted if you don't also have response transformers. This said, using it is better for clarity if your endpoint contains other child elements (such as filters).

It's also possible to declare an anonymous transformer locally inside an endpoint. For example, the following defines a global STDIO endpoint that'll encode its input in Base64:

```
<stdio:endpoint name="sysin" system="IN">
    <base64-encoder-transformer />
</stdio:endpoint>
```

This local transformer is deemed anonymous because it doesn't have a `name` attribute; thus, it can't be referenced. This kind of declaration can be a viable option for extremely short configurations. As soon as your configuration won't fully fit on one or

two screens, you might risk losing track of the different transformers you use and might miss opportunities for reusing them in different places. Hence, when your configuration starts to grow, you should prefer using global transformers and reference them from your endpoints. For the sake of brevity, we'll often use anonymous transformers in the upcoming examples.

There's a caveat with endpoint transformers: as soon as you declare a transformer on an endpoint, *you override the default transformer* that the transport could've set for this endpoint. This can be extremely surprising at first, as you might not realize that by adding the extra transformation behavior you cancel the default one. This implies that *if you want to add your transformation logic to an implicit one, you first have to declare the default transformation explicitly*. We'll come back to this fact later on, in section 5.3.2.

WARNING Local trumps global If you declare one or several transformers on a global endpoint and add other transformers to the inbound or outbound endpoint that references the global one, the locally declared transformers will override the globally declared ones. For example, consider the global endpoint declaration and its reference shown in listing 5.2.

Listing 5.2 Local endpoint transformer declarations override global ones.

```
<stdio:endpoint name="sysin" system="IN"
                 transformer-refs="Base64Encoder" />           ↪ Global endpoint with
<inbound-endpoint ref="sysin"
                    transformer-refs="Base64Decoder">           ↪ Inbound endpoint with
                                                               a transformer override
```

In this example, the inbound endpoint built from referencing the global one named sysin will have only the Base64Decoder transformer applied to it. The local declaration of this transformer has overridden the declaration of the Base64Encoder on the global endpoint definition.

If you look at the available attributes on an endpoint definition element, you'll see that two of them are related to transformers. One is named `transformer-refs` and the other is named `responseTransformer-refs`. Similar to what's shown in listing 5.2, there's also a child element named `response-transformers` that can be used inside an endpoint. Why two types of transformers?

The answer to this question is given in figure 5.3.

Regular transformers (inbound and outbound) kick in when a message traverses an endpoint in the direction that's natural for the router it sits in, which is toward the component for inbound routers and away from it for outbound ones. In contrast, response transformers kick in when a message flows back as a response to the incoming request, when a response is expected in a synchronous manner. This synchronous response can be expected from a component in an inbound router or from the remote destination in an outbound router.

There's another important fact about inbound transformers you must be aware of. *It's up to the component to decide whether to apply the configured inbound transformer.* This can

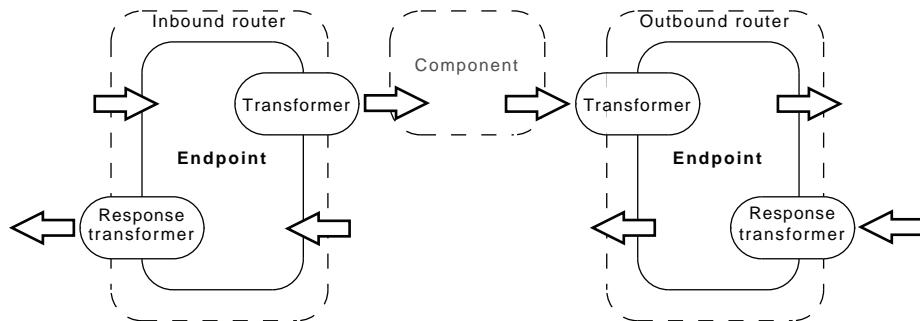


Figure 5.3 Endpoint transformers and response transformers can be used in inbound and outbound routers.

be confusing! We'll discuss this further in the next chapter. In the meantime, our examples will focus only on bridge services built on the implicit bridge-component, which honors inbound transformers.

TIP *My transformers are ignored!* If you've configured transformers on your endpoint and they aren't applied as you expect them to be, ask yourself the following questions:

- Are the transformers configured for the right part of the message flow? You may need to apply the transformers on the response.
- Is the service component honoring transformers? The component you're using might not let transformers kick in.

Most of the time, this'll help you figure out the source of the problem.

As you've noticed, configuring transformers requires a good understanding of the default behavior of transports and the message flows around service components. We'll come back to these two aspects in the coming sections of this chapter and in the next chapter about components.

You've now acquired enough knowledge about using and configuring transformers. In the next sections, we'll take a deeper look at some notable transformers picked from the core library of Mule and from the XML module and the JMS transport.

BEST PRACTICE Look for implementations of `org.mule.api.transformer.Transformer` in Mule's API to discover all the available transformers.

5.3 Using core transformers

There are almost 30 transformers in the core library of Mule. They all provide transport-independent transformation features such as compression, encryption, or payload value extraction. In this section we'll look at four of them (six if we count the round-trip ones) that illustrate common payload type and properties transformations:

- *Dealing with bytes*—Using byte transformers to perform payload type transformation with bytes and streams
- *Compressing data*—Applying GZip transformation to compress or decompress payload
- *Modifying properties*—Working with the transformer that fiddles with the properties of a message
- *Leveraging expression evaluators*—Creating new payloads with a transformer that can evaluate expressions of all sorts

TIP *A lazy but handy transformer* Mule's core contains a lazy transformer named no-action-transformer. What's the point of a transformer that does nothing? Its main intention is to override an implicit active transformer to prevent its action. If you remember the discussion about service overrides in section 3.1.1, you should recall how we leveraged this transformer to alter the default transformation behavior of a connector. The no-action transformer can be used at endpoint level too, if you don't want to override the connector default behavior. Here you can disable the default transformer that extracts the payload of JMS messages at the inbound endpoint level:

```
<jms:inbound-endpoint queue="theQueue">
    <no-action-transformer/>
</jms:inbound-endpoint>
```

With this configuration, JMS messages consumed by the endpoint will be passed untransformed to the component. As you can see, far from being prone to inaction, the `NoActionTransformer` actively resists working, which allows you to disable default transformation behaviors.

For our first bite at core transformers, we'll appropriately look at a pair of core transformers that deal with bytes.

5.3.1 Dealing with bytes

Bytes are a finer grain of data unit exchanged between systems. In these systems, bytes are usually handled as streams or as arrays, with the former suited for large volumes of data and the latter for smaller chunks of information.

In Mule Core, the `ByteArrayTo` transformers can handle both arrays *and* streams of bytes as their input. This means that if the endpoint you're declaring a transformer on is capable of streaming, the transformer will be able to act on the flow of data without the need to first store it in an array.

Let's consider first the versatile byte-array-to-object transformer. If this transformer receives a byte payload that represents a serialized Java object, the result will be an object deserialized from these bytes. If the payload isn't a serialized object, the transformer will simply build a String from the bytes. The following demonstrates this transformer configured to deserialize byte arrays or streams into instances of `java.util.Map` only:

```
<byte-array-to-object-transformer
    name="ByteArrayToMap" returnClass="java.util.Map" />
```

If this transformer is used in an endpoint that receives anything other than serialized map objects, a `TransformerException` will be thrown and the processing of the message will be stopped.

WARNING *I U+2764 Unicode* Ever entered your accented first, last, or street name in a web site and got a back a series of question marks in the confirmation screen? If yes, you've been victim of a developer's assumption that there are no characters beyond the 128 defined in the US-ASCII set. Whenever you transform bytes into Strings or vice versa, you must consider the encoding that'll be used during the conversion. This is because the representation of a single character doesn't always translate to a single byte: the byte representation of a character is dictated by encoding. Note that UTF-8 is nowadays a common encoding, as it's backward-compatible with ASCII and efficient enough for most non-Asian character sets.

In Mule, the encoding used by a transformer is determined by looking first at the encoding of the message. If none is specified, the encoding defined on the endpoint where the transformation happens will be used. If the endpoint has no encoding set, then the default encoding of the platform will be used. In Mule 2, the default encoding is UTF-8. Your best option is to have the encoding specified in the message. Some transports do this automatically for you, such as the HTTP transport, which recognizes the `Content-Encoding` header and sets the value on the message accordingly. If this isn't possible, then you'll have to ensure that clients use a predefined encoding and stick to it.

The alter ego of this transformer is the object-to-byte-array transformer. As expected, it works the opposite way: it transforms Strings and streams into byte arrays and marshals serializable payloads into bytes using the standard Java serialization mechanism. Note that this transformer chokes on nonserializable payloads, as it can't possibly perform any transformation on them. The following shows the simple declaration of the transformer we use in the publication example detailed in section 5.6 to copy all the bytes of an incoming input stream into an array:

```
<object-to-byte-array-transformer name="ObjectToByteArray" />
```

Copying a complete input stream to a byte array can have a serious, if not fatal, impact on the memory of your Mule instance. This should never be taken lightly. So why would we do something that foolish? When processing an event asynchronously, you have no guarantee that the input stream will stay open for the duration of the message processing. If the inbound endpoint decides it has received its response, it'll close its connection, taking down all the open streams. Using this transformer fills the message with a payload that can be processed safely anytime after it has been received: this is a way to "detach" the message from its transport.

BEST PRACTICE Always consider the memory impact of the transformation you intend to use.

If you list all the byte-related transformers in Mule's core, you'll find a few others. We'll quickly detail them here:

- *byte-array-to-serializable-transformer and serializable-to-byte-array-transformer*—These are specialized versions of the byte-array-to-object transformer and the object-to-byte-array transformer, which only transform to and from serialized Java objects.
- *byte-array-to-hex-string-transformer and hex-string-to-byte-array-transformer*—This pair of transformers isn't related to the others. As their names suggest, they transform from and to hexadecimal representations of bytes.
- *byte-array-to-string-transformer and string-to-byte-array-transformer*—These behave like their byte counterparts except that they rely on the current encoding to transform bytes to and from strings.

NOTE Some less-popular core transformers don't have a matching XML configuration element. If you intend to use one of them, you'll need to declare it as a custom one (see section 5.7.1). That's the case for the `org.mule.transformer.simple.ByteArrayToMuleMessage` and `org.mule.transformer.simple.MuleMessageToByteArray` transformers. These are specialized versions of the serializable-byte array transformers that transform not only the payload of the message but the whole `MuleMessage` object. This can be useful for exchanging messages over the wire between Mule instances in a form that's directly usable by Mule and that carries all the meta information and specific context with it.

You now have a good idea of the transformers you can leverage to transform your data from bytes to other forms. Let's discover how to deflate message payloads.

5.3.2 Compressing data

Under their byte representations, messages can become big to the point that they're not practical to send over the network. For example, JMS providers often discourage publishing messages with heavy payloads: when you start going beyond a hundred kilobytes, it's usually a good time to consider compression. With XML being a common payload in messaging systems nowadays, you can expect drastic reductions of data volume, as XML is a good candidate for compression.

So how do we compress data in Mule? Let's suppose that you have to publish large strings to a JMS queue. The receiving consumer, which listens on this queue, expects you to compress the data before you send it. If you're in such a situation, the `gzip-compress` transformer is the one you're looking for. Here's how you'd use it:

```
<jms:outbound-endpoint queue="compressedDataQueue"
    connector-ref="dataJmsConnector">
    <transformers>
```

```

<string-to-byte-array-transformer />
<gzip-compress-transformer />
<jms:object-to-jmsmessage-transformer />
</transformers>
</jms:outbound-endpoint>

```

Why on earth do we use three transformers? Why can't we just apply the GZipCompressor to the JMS outbound endpoint? Here's the explanation:

- The reason why we use a string-to-byte array transformer before the GZipCompressor is subtle. Because the endpoint receives a string payload and because string is serializable, the natural behavior of the compressor would be to serialize the string first, and then compress it. But what we actually want to send to the JMS queue are the bytes that constitute the string in a compressed manner. This is why we use the string-to-byte array transformer first.
- The reason why we need the object-to-JMS message transformer is also interesting. If you remember the discussion of section 5.1, as soon as you declare a transformer on an endpoint, you cancel the default see one it could have, forcing you to declare it explicitly. While you don't need to use the ObjectToJmsMessage transformer on JMS outbound endpoints that don't declare a transformer, you need to do it in this case, as we already had two transformers on the endpoint.

Conversely, if the receiving consumer were a Mule JMS inbound endpoint, we'd have to use several transformers; in fact we'd use the round-trip twins of the ones on the outbound endpoint demonstrated earlier, but in reverse order. This is demonstrated here:

```

<jms:inbound-endpoint queue="compressedDataQueue"
                      connector-ref="dataJmsConnector">
    <transformers>
        <jms:jmsmessage-to-object-transformer />
        <gzip-uncompress-transformer />
        <byte-array-to-string-transformer />
    </transformers>
</jms:inbound-endpoint>

```

The transformers we've seen so far were all performing payload type transformations. Let's now look at a transformer that can modify message properties.

5.3.3 ***Modifying properties***

Whether they were called properties, headers or metadata, you should've already been exposed to the notion of extra chunks of data carried alongside the main payload of a message. Properties are named values that establish context so a message can be properly processed or interpreted. For example, the headers that you send with data in an HTTP POST action can be considered to be the properties of a message whose payload would be the body of the HTTP operation.

In Mule, all properties are stored together in a map that's part of the message context. These properties are of three kinds:

- *Mule-specific properties*—These properties carry contextual information with the message so Mule’s different moving parts can work properly. Encoding, session, and correlation identifiers are examples of this kind. You can get a list of these properties by looking at the source code of org.mule.api.config.MuleProperties. If you do so, you’ll notice that their names are in uppercase with underscores instead of spaces (like Java constants). You usually don’t want to mess with these properties.
- *Transport-specific properties*—If a transport supports the notion of properties, the connector will store these values with the actual payload in the message context. The connector will also convert message properties into transport properties when sending messages. This is why you find all the HTTP headers in message properties after sending a message to Mule over this protocol.
- *User properties*—Pretty much like protocols allow you to roll your own properties in addition to the specific ones, Mule allows you to add any number of typed properties to a message. This can be useful for many things, as these properties will be kept and moved around the Mule instance (or instances if you have several connected nodes). Consider, for example, a complex routing scenario, where messages are analyzed by a business rules engine, get tagged with some properties, and must then be sorted or dispatched accordingly.

Adding, modifying, or removing properties is therefore an important aspect of dealing with messages in Mule. This is where the message-properties transformer comes in handy. Let’s consider its different operations while looking at a few examples.

Removing transport-specific properties that you don’t want to carry over is a common message sanitizing operation. The following sample shows an HTTP inbound endpoint that removes some of the standard browser headers from incoming messages:

```
<http:inbound-endpoint address="http://localhost:8080/log">
  <message-properties-transformer>
    <delete-message-property key="Accept" />
    <delete-message-property key="Accept-Encoding" />
    <delete-message-property key="Accept-Charset" />
    <delete-message-property key="Accept-Language" />
    <delete-message-property key="Cache-Control" />
    <delete-message-property key="User-Agent" />
  </message-properties-transformer>
</http:inbound-endpoint>
```

As we said before, adding properties helps tag messages so they can be routed smartly. The following defines a global transformer that sets a custom property used for tracking error messages to its default value, but only if it hasn’t been already set:

```
<message-properties-transformer name="AddDefaultErrorIfAbsent"
  overwrite="false">
  <add-message-property key="ErrorFlag" value="DefaultError" />
</message-properties-transformer>
```

If you need to set several properties at once, you have two options. The first consists in using several `add-message-property` elements, as shown here:

```
<message-properties-transformer
    name="AddStatisticsSpreadsheetResponseHeaders" >
    <add-message-property
        key="Content-Type" value="application/vnd.ms-excel" />
    <add-message-property
        key="Content-Disposition" value="attachment;
filename=stats.csv" />
</message-properties-transformer>
```

Alternatively, you can leverage a dedicated child element that accepts Spring's map entry elements. Here's the declaration of a global transformer that Clood, Inc., uses to add meta information about their customers to a Mule message:

```
<message-properties-transformer name="CustomerPropertiesSetter">
    <add-message-properties>
        <spring:entry key="CustomerId" value="${customer.id}" />
        <spring:entry key="AccountId" value="${account.id}" />
    </add-message-properties>
</message-properties-transformer>
```

Notice how they leveraged properties placeholders to avoid hard-coding the values that are likely to change between environments. It's also possible to rename an existing property. The following demonstrates a transformer that renames the transport property that holds the incoming HTTP method used from `http.method` to `inbound.http.method`:

```
<message-properties-transformer name="HttpMethodHeaderRenamer">
    <rename-message-property key="http.method" value="inbound.http.method" />
</message-properties-transformer>
```

BEST PRACTICE Use property transformers to deal with transport- or routing-related message metadata.

You're now able to modify the properties of your messages, whether to satisfy an internal need in your Mule instance or for an external transport or remote system. Let's now discover a last core transformer that's extremely resourceful.

5.3.4 Leveraging expression evaluators

In section 4.2.3, you learned about the support for expression evaluation that exists in Mule. A wealth of expression evaluators are available, and each supports a different expression syntax: turn to appendix A if you want to know more about this subject. The *expression transformer* can leverage these evaluators to transform the payload of the message it processes. This transformer can be configured to evaluate one or several expressions. Depending on this configuration, the resulting message payload will be an object (single expression) or an array of objects (multiple expressions).

Internally, Clood, Inc., has to deal with a lot of Internet addresses, for monitoring the activity of their clients. They use instances of `java.net.InetAddress` as the

payload of administrative messages that run around in their Mule instances. Unfortunately, one of their monitoring applications needs to receive only basic information about an Internet address: its host IP, and whether it's multicast. To feed it with the right information, Cloud uses an expression transformer in order to extract the relevant bits:

```
<expression-transformer>
  <return-argument evaluator="bean" expression="hostAddress" />
  <return-argument evaluator="bean" expression="isMulticastAddress" />
</expression-transformer>
```

The output of this transformer is an array of objects; the first is a string representing the host address and the second is a boolean specifying whether it's multicast or not. Note that we used the bean evaluator and that the expression is a Jaxen XPath bean property name (similar to the standard JavaBean one, except for boolean fields). Because we haven't set the optional argument to true—that is, the optional boolean argument named `optional`—if any of these two expressions would return null, the whole transformation and message-processing chain would be aborted with an exception being thrown.

The expression transformer is so powerful that it can most of the time replace a custom transformer. Therefore, before writing any code, check first whether you can achieve your goal with the expression transformer.

NOTE *Automagic transformation* Before we close this section on core transformers, let's look at the *auto transformer*. As its name suggests, this transformer can apply a desired transformation automatically. How does it do that? It selects the most appropriate transformer based on the return class that you specify on its declaration:

```
<auto-transformer returnClass="com.cloud.statistic.ActivityReport" />
```

The auto transformer can only select globally configured transformers that are *discoverable* (more on this notion later). It works better with custom objects, as shown in the example, instead of generic ones such as strings or byte arrays. For the latter, there are way too many choices available for the auto transformer to pick up the right one.

When should this transformer be recommended? Mainly when a single endpoint receives a variety of different payloads and needs to transform them to a particular custom object type.

We're now done with our quick tour of a few core transformers. You've learned to deal with bytes, compress them, alter message properties, and leverage the power of expressions. Your bag of transformation tricks already allows you to perform all sorts of message manipulations. This said, none of the transformers we've looked at perform data format transformation. Because XML is particularly well suited for data format transformations, we'll now look at some of the transformers you can find in the Mule XML module.

5.4 Using XML transformers

The XML module provides several transformers. We'll look at the two most significant:

- *XSL transformer*—Transforms an XML payload into another format, thanks to XSLT.
- *XML marshaling transformers*—Marshals and unmarshals objects to and from XML.

The other transformers of the XML module, which we don't cover here, provide extra features such as transforming from and to a DOM tree or generating pretty-printing XML output.

Before we look at the XSL transformer, a quick remark about the namespace prefix used for the XML module configuration elements. You'll notice that it's `xm:`. Why not `xml:` you might ask? Because `xml:` is a reserved prefix that always binds to <http://www.w3.org/XML/1998/namespace>. Now that we're sure you won't be surprised by this detail, let's proceed.

5.4.1 Transforming format with XSL

XSL transformation, aka XSLT (<http://www.w3.org/TR/xslt>), is a powerful and versatile means to transform an XML payload into another format. This other format is usually XML too (with a different DTD or schema), but it can also be HTML or even plain text. So what does it take to use this transformer? Here's the definition of the XSL transformer that performs the DocBook-to-FO transformation in section 5.6:

```
<xm:xslt-transformer name="DocbookToFO"
    xsl-file="docbook-xsl/fo/docbook.xsl" />
```

So far, nothing exciting: this creates a global transformer named `DocbookToFO` and loads the specified file from the classpath. If you know DocBook XSLs, you're aware that they're extremely modularized and rely on a lot of import statements. This is pretty common. An XSL template document often has external resources, such as other XSL files or even external XML documents. How does the XSL transformer load its main XSL and its external resources? The XSL transformer uses a file lookup fallback strategy that consists of looking first in the file system and then in the classpath, and finally trying a regular URL lookup. With this in mind, you should be able to write XSL templates that work even if your Mule instance doesn't have access to the Internet.

TIP The XSL transformer is extremely versatile as far as source and return types are concerned. This transformer goes to great lengths to accept a wide variety of input types (bytes, string, W3C, and dom4j elements), so you seldom need to perform any pretransformation prior to calling it. This transformer also infers the best matching return type based on the input source (or the `returnClass` attribute if it's been set). For example, if a W3C Element is used as a source, the transformer will build a W3C Node as a result.

XSL templates can receive parameters when they start processing an XML source. How do we do this in Mule? Going back to our previous example, let's pretend that we want to change the default alignment from justified to centered. This is achieved by passing the appropriate parameter to the DocBook XSL, as shown here:

```
<xm:xslt-transformer name="DocbookToFO"
  xsl-file="docbook-xsl/fo/docbook.xsl">
  <xm:context-property key="alignment" value="center" />
</xm:xslt-transformer>
```

Note how we used the repeatable `xm:context-property` element. It's in fact a Spring XML list entry, which means that you can also use the `key-ref` and `value-ref` attributes if you want to refer to beans in your configuration.

You might wonder if this is the way to inject dynamic values in the XSL transformer. Not really; there's a much better solution. Suppose the alignment parameter you need to pass to the XSL is defined in a property of the current message named `fo-alignment`. The best way to pass this value to the XSL is to leverage the expression evaluation framework that we talked about in section 5.3.4. This is achieved this way:

```
<xm:xslt-transformer name="DocbookToFO"
  xsl-file="docbook-xsl/fo/docbook.xsl">
  <xm:context-property key="alignment" value="#{header:fo-alignment}" />
</xm:xslt-transformer>
```

Now we're talking! Not only will the XSL transformation work on the current message payload, but it can also work on all sorts of dynamic values that you'll grab thanks to the expression evaluators.

If you've used XSLT before, you should know that it's a pretty intense process that pounds on CPU and memory. The XSL transformer supports performance optimization parameters that allow you to fine-tune the maximum load and throughput of your transformation operations. The following shows the original example configured to have a maximum of five concurrent transformations happening at the same time:

```
<xm:xslt-transformer maxActiveTransformers="5" maxIdleTransformers="5"
  name="DocbookToFO" xsl-file="docbook-xsl/fo/docbook.xsl" />
```

Note that because this particular XSL is expensive to load in memory, we don't want to dereference any transformer once it's been created. This is why we've set the number of idle transformers to be the same as the maximum number of transformers. A lower number would've implied the potential destruction and recreation of transformers.

NOTE *Workers in the pool* This transformer uses a pool of `javax.xml.transform.Transformer` workers to manage the load. Without erring on the side of premature optimization, always take a little time to consider the expected concurrent load this transformer will have to deal with. If a message arrives when this transformer has exhausted its pool, it'll wait indefinitely until a transformation is done and a worker is returned to the pool (where it won't have time to chill out, unfortunately). Therefore, if you estimate that the amount of messages you'll need to transform is likely to exceed the maximum number of active transformers, or if you simply

don't want to reach that limit, you might want to locate this transformer behind an asynchronous delivery mechanism. This way, you'll refrain from blocking threads in a chain of synchronous calls waiting for the workers to pick up a transformation task.

You've just learned how to transform payload format with XSL, so let's see another transformer from the XML module that can deal with the payload type itself.

5.4.2 XML object marshalling

If you've done more than trivialities with Java serialization, you've realized that it's challenging at best... and challenged at worst. Alternative marshalling techniques have been developed, including creating XML representations of objects. The XML marshaller and unmarshaller from the XML module leverage ThoughtWorks' XStream, for that matter (<http://xstream.codehaus.org/>).

Thanks to XStream, these transformers don't require a lot of configuration. For example, you don't need to provide an XML schema, as is often the case with other XML data binders. Note also how this transformer can marshal any object, unlike the byte array ones we've seen before, which rely on Java's serialization mechanism. The following declare an XML object marshaller:

```
<xm:object-to-xml-transformer />
```

Nothing spectacular, but a lot of capability behind the scene. As a variant of the previous example, the following declares a marshaller that serializes the full `MuleMessage` instead of just the payload:

```
<xm:object-to-xml-transformer acceptUMOMessage="true" />
```

This transformer would be well suited for persisting detailed error messages in an exception strategy, as it would marshal the whole `MuleMessage` in XML, a format an expert user can analyze and from which you can easily extract parts for later reprocessing. It can also be useful for scenarios where you want to send a `MuleMessage` over the wire without resorting to standard Java serialization.

Similarly, the declaration of the round-trip XML object unmarshaller this trivial:

```
<xm:xml-to-object-transformer />
```

NOTE At the time of this writing, the XML module configuration schema of Mule 2.2 doesn't allow you to take full advantage of the transformer class behind the scene, which supports XStream's concepts of custom aliases, converters, and drivers. Until this is corrected, if you need one of these features, you'll have to declare these transformers as if they were custom ones (see section 2.2.1) so you can access their full configuration properties. The following demonstrates an XML marshaller that aliases `org.mule.DefaultMuleMessage` classes into `MuleMessage`:

```
<custom-transformer
    class="org.mule.module.xml.transformer.ObjectToXml">
    <spring:property name="acceptUMOMessage" value="true" />
```

```
<spring:property name="aliases">
<spring:map>
    <spring:entry key="MuleMessage"
        value="org.mule.DefaultMuleMessage" />
    </spring:map>
</spring:property>
</custom-transformer>
```

When marshalling `org.mule.DefaultMuleMessage`, this transformer will output an XML document whose root element will be named `MuleMessage`.

As we've seen, the XML module contains transformers that can be useful in different scenarios, even if you don't use XML extensively². Now that we've explored some core and module transformers, let's look at a few transport transformers.

5.5 Using JMS transformers

Most of the two dozen Mule transport libraries contain transformers. These transformers usually transform the messages from a type that's specific to the transport to one that's independent from it, and vice versa. They're usually automatically applied for you by the transport itself, whenever a message is received or sent. So why bother mentioning them? If you remember the discussion from the beginning of section 5.1, specifying transformers on endpoints disables their default transformers. Hence, you'd better know that they exist and how to configure them.

This section will focus on the transformers that come with the JMS transport, as this messaging API is popular and often involved in integration projects. These transformers are useful for the two main messaging actions, which are

- *Producing JMS messages*—Transforming a message payload and its properties into a `javax.jms.Message` ready to be sent
- *Consuming JMS messages*—Extracting the body of an incoming `javax.jms.Message` into a JMS-agnostic payload

Note that, because of the asymmetric way they handle JMS properties, these transformers can't be considered true round-trip transformers.

5.5.1 Producing JMS messages

The JMS transport uses a `javax.jms.MessageProducer` to send messages to a particular destination (queue or topic). JMS can only send messages that comply with its API. Therefore, the transport needs to transform the current Mule message into a JMS message. For this, it uses the object-to-JMS message transformer, which transforms the payload to the specific JMS message whose body data type is best fitted.

How does Mule select the JMS message type? You may remember that JMS 1.1 defines five types of messages, which all implement `javax.jms.Message` and differ

² The Extended XML Module is a MuleForge project that offers extra XML to object marshallers. See <http://www.mulesource.org/display/EXTENDEDXML/Home> for more information.

only by the type of data they carry in their body. Table 5.1 shows the strategy used by Mule to select a JMS message based on the source type.

The `jms:object-to-jmsmessage-transformer` also takes care of copying all the current message properties into JMS message properties, unless their name is a reserved JMS header name. Note that the Mule correlation ID message property is copied into the JMS correlation ID header. This propagation of correlation IDs across systems is an essential messaging pattern.

Configuring this transformer is trivial, as shown in the following example:

```
<jms:object-to-jmsmessage-transformer name="ObjectToJmsMessage" />
```

This declares a global object-to-JMS message transformer named `ObjectToJmsMessage`. Nothing more is needed, as the transformer is smart enough to just do the right thing with your messages.

NOTE If you're somewhat versed in the art of JMS, you should know that an active JMS session is needed to actually create any type of JMS message. Consequently, the object-to-JMS message transformer needs access to a valid session to do its work. This is achieved by either retrieving the session from the current transaction (if any) or by getting one, possibly cached, from the JMS connector itself.

Let's now look at the alter ego of this transformer, which is used when receiving messages.

5.5.2 Consuming JMS messages

When the JMS transport reads (or to speak the correct lingo, *consumes*) a message, its payload is a `javax.jms.Message` object. The actual payload of the JMS message resides in its body, whose type is one of the five we've seen previously. It's when the body payload needs to be extracted that the JMS message-to-object transformer comes into play. This transformer extracts the body payload from the JMS message source but doesn't alter the message properties (the JMS message adapter already took care of extracting the notable JMS message headers and all the JMS message properties).

How does this transformer decide what type of data it will return? This decision is based on the rules shown in table 5.2.

Table 5.1 Return type decision table of the object-to-JMS message transformer

| Source type | JMS message type |
|-----------------------------------|------------------|
| Input stream List ^a | Stream message |
| Map | Map message |
| String | Text message |
| Serializable object | Object message |
| Byte array | Bytes message |
| JMS message | Itself |

a. Mule restricts the types of objects that can go in this list based on the supported types defined in the JMS specification for StreamMessages.

Table 5.2 Return type decision table of the JMS message-to-object transformer

| Source type | Return type |
|----------------|---------------------|
| Stream message | List of objects |
| Map message | Map |
| Text message | String |
| Object message | Serializable object |
| Bytes message | Byte array |

Why do we have a sixth type of JMS message? This is because JMS supports the notion of a *bodyless message*, which is a message with no payload and only headers and properties. In that case, the payload itself is of no interest, and you should only focus on the headers and properties that the JMS transport message adapter extracted for you.

Like its alter ego, configuring this transformer is straightforward, as shown:

```
<jms:jmsmessage-to-object-transformer name="JmsMessageToObject" />
```

This declares a global JMS message-to-object transformer named `JmsMessageToObject`. As a bonus, this transformer can do a little more for you. It can convert the extracted string to a byte array (or the extracted byte array into a string) if you specify the desired return class:

```
<jms:jmsmessage-to-object-transformer name="JmsMessageToString"
    returnClass="java.lang.String" />
```

This transformer will only return a String if it has extracted a byte array from the JMS message. Otherwise, the return value will stay unchanged, as it was after extraction from the JMS message.

As you've observed, the transformers from the JMS transport are designed to allow you to leverage the powerful messaging infrastructure that is JMS without having to deal with the low-level details. Integrating JMS providers will now be bliss!

We're done with our exploration of Mule transformers. We've learned how and when to use them and have discovered some of the most notable ones from the core, module, and transport libraries. It's now time to look at an example that puts several of them into action.

5.6 Existing transformers in action

Clood, Inc., has many prestigious clients, including a famous publishing company. When they turned to Clood for advice on how to implement their authoring platform (presented in section 1.3), Clood naturally oriented them toward Mule. After all, nothing prevents our clients from using the same tools we do. Figure 5.4 shows a conceptual view of the configuration that we've built to support the publication application scenario.

Let's detail the different transformers we use:

- *Object-to-byte-array transformer*—The inbound HTTP endpoint creates a message with a byte stream payload. Knowing that the XSLT transformer is perfectly able to consume this kind of input, why do we transform the stream into a byte array first? The reason lies in the way the XSLT transformer consumes the input stream, which prevents it from being fully drained before the HTTP connection (and the related input stream) gets closed. By using this transformer first, we fully drain the input stream before the HTTP connection gets closed.
- *XSL transformer*—This transformer performs the bulk of the work of this service and transforms the incoming DocBook XML data into XSL-FO.³ This

³ The DocBook to XSL-FO transformation stylesheets are available from the Docbook Project: <http://docbook.sourceforge.net>.

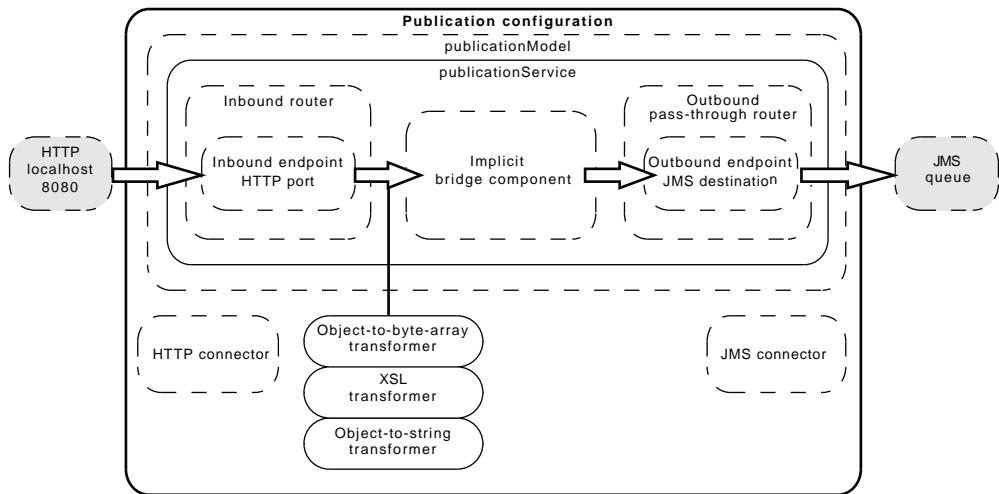


Figure 5.4 Configuration overview of the Publication Application integration example

transformation is slow and resource-greedy, but will be throttled thanks to the pool of workers embedded in the XSL transformer. The client is already disconnected when the XSL transformation happens, which fits well the overall asynchronous design of the solution.

- *Object-to-string transformer*—One of the requirements was to deliver the XSL-FO document as a string to the targeted JMS queue and not as a byte array. This is why we have this third transformer kicking in to finalize the transformation chain.
- *Message-properties transformer*—We have to remove all the message properties added by the HTTP transport on the Mule message, as their names aren't compatible with the JMS specification. If we don't remove them, they'll be carried up to the JMS outbound endpoint, where they'll be transformed into JMS message properties and warnings will be logged.

As you can see, we apply all the transformers on the inbound endpoint, while we could've put some or all of them on the outbound endpoint. Why is that? If we would've declared a transformer on the outbound JMS endpoint, this would've removed the implicit object-to-JMS transformer that the transport uses. We would then have had to explicitly configure this JMS transformer on the outbound endpoint. By declaring the transformers on the inbound endpoint, we avoided this.

Concretely, this amounts to a limited Mule configuration. If you look at the sample provided at the companion site of this book, you'll see three configuration files:

- *publication-jms-config.xml*—This is the JMS-specific configuration of the application. Following our advice to avoid monolithic configurations, we've grouped in this file all the configuration artifacts that could change if we used another JMS provider (for example, one for integration tests and another for production).

- *publication-config.xml*—This is the main configuration file where the service and the transformers are defined.
- *server-stub-config.xml*—This configuration file isn't really part of the integration solution. It simulates the application that will consume the JMS queue and process the XSL-FO document. For this example, it simply dumps the XSL-FO file in the file system.

Listing 5.3 shows the content of publication-config.xml, less the mule root element.

Listing 5.3 The configuration of the publication application

```

<spring:beans>
    <spring:import resource="publication-jms-config.xml" /> Imports JMS  
configuration file
</spring:beans>

<object-to-byte-array-transformer name="ObjectToByteArray" />

<object-to-string-transformer name="ObjectToString" />

<message-properties-transformer
    name="MessagePropertiesSanitizer">
    <delete-message-property key="Content-Length" />
    <delete-message-property key="Content-Type" />
    <delete-message-property key="User-Agent" />
    <delete-message-property key="http.context.path" />
    <delete-message-property key="http.method" />
    <delete-message-property key="http.request" />
    <delete-message-property key="http.request.path" />
    <delete-message-property key="http.version" />
</message-properties-transformer>

<xm:xslt-transformer name="DocbookToFO"
    xsl-file="docbook-xsl/fo/docbook.xsl" />

<model name="publicationModel">
    <service name="publicationService"> Declares publication  
service as a bridge
        <inbound>
            <http:inbound-endpoint host="localhost" port="8080">
                <transformers>
                    <transformer ref="ObjectToByteArray" />
                    <transformer ref="DocbookToFO" />
                    <transformer ref="ObjectToString" />
                    <transformer ref="MessagePropertiesSanitizer" />
                </transformers>
            </http:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="publicationQueue"
                    connector-ref="publicationJmsConnector" />
            </pass-through-router>
        </outbound>
    </service>
</model>
```

References all
global transformers
by name

If you start the publication application and HTTP post the provided DocBook sample file,⁴ you'll notice the following entries in the Mule console:

```
INFO [connector.http.0.receiver.2]
org.mule.transport.http.HttpMessageReceiver: Closing HTTP connection.
    Making portrait pages on USletter paper (8.5inx11in)

INFO [publicationJmsConnector.dispatcher.1]
org.mule.transport.jms.JmsMessageDispatcher:
    Connected: JmsMessageDispatcher{this=1611ed3,
    endpoint=jms://publicationQueue, disposed=false}
```

These entries follow the expected chronology:

- The message is received on the HTTP port and the connection is closed.
- The DocBook generation occurs (it'll generate a USletter document in this case).
- The result of the transformation is sent to the target JMS queue (named publicationQueue).

Right after these entries, you'll notice that the server stub consumes the message in the queue and logs its activity:

```
INFO [connector.file.0.dispatcher.1]
org.mule.transport.file.FileMessageDispatcher:
    Connected: FileMessageDispatcher{this=19576c3,
    endpoint=file://out, disposed=false}

INFO [connector.file.0.dispatcher.1]
org.mule.transport.file.FileConnector:
    Writing file to: /workdir/out/9bfb8a1b-55f3-11dd-b83c-61d8eb47eb0c.fo
```

As expected, the XSL-FO content has been read from the JMS queue and stored as a file (with a long uniquely generated name).

This integration scenario can look trivial from a quick look: it might look like we could've done the same with a few lines of code. Bear in mind that, with this short configuration, we've benefited from a lot of supporting features from Mule, such as XSL transformer pooling, JMS message generation, and transport connection (and reconnection in case of trouble). This allowed us to deliver a turnkey solution to the publishing company in a short time: a running prototype was ready in a matter of hours and production soon followed.

The existing Mule transformers are numerous and support a wide range of transformations. We can only encourage you to explore Mule Core, its modules, and libraries to discover all of them. But Mule isn't limited to the existing transformers you can find in its libraries. Mule supports the notion of custom transformers, a powerful means to plug in your own transformation logic. We'll now look at this ability and create our own custom transformer.

⁴ For example, by running

```
wget --post-file='data/test.docbook' localhost:8080/publicationService
```

5.7 Writing custom transformers

When would you need to write a custom transformer? This can happen for a variety of reasons:

- You might have a transformation requirement that's so specific that it's impossible to realize with an existing transformer or a chain of them.
- You want to use an existing transformation framework or a tool that has transforming capacities and isn't available as an existing transformer.

Clood, Inc., needs to generate a lot of custom emails to send to its clients: activity reports, incident status reports, and last but not least, invoices. For this they decided to use Mule, as it offers all the features they need: tapping different sources of data, transforming them into nice human-readable emails, and sending these over the wire to clients. Though XSLT could've been an option, they decided to leverage the Apache Velocity Engine, an open source templating framework.

We implemented two versions of this transformer:

- The first version we created used Velocity to transform only the payload of the message.
- It turned out that this wasn't enough: we came up with a second version, able to transform the payload and to get values from the message properties, too.

BEST PRACTICE Create payload transformers unless you absolutely need to have access to the `MuleMessage` object.

Let's start with reviewing what we did for the first version of the Velocity transformer.

NOTE In this section, we'll only look at compiled Java transformer implementation. In section 14.1.2, we'll look at the possibility of implementing transformers with a scripting language.

5.7.1 Transforming payloads

The first version of our custom transformer uses the message payload as the Velocity context and applies a template to it in order to generate a string. The idea was to apply the transformation on a bean representing a client's account details, resulting in the textual content of the email for the said client.

TIP *Transformers as good citizens* Whenever you decide to create your own transformer, ask yourself the following questions:

- *Should it be idempotent?* In other words, how should it behave if the source type is of the same type as the expected return type? It's often beneficial for a transformer to return the source object unchanged if it's already of the desired type. Similarly, a transformer that performs format transformation might decide to return the source unchanged if it's already in the desired format.

- *Should it have a round-trip alter ego?* Being able to transform back and forth from one form to another gives greater flexibility in the way the transformer will be used. If a transformation can easily be reverted by an alter-ego transformer, this can allow it to transform back to the original form in case of exception. This also applies to a format transformer, which can revert transformed data into its original form.

Now, let's look at the nitty-gritty. Even if all transformers implement `org.mule.api.transformer.Transformer`, it's recommended that you extend the abstract base class that implements this interface. This will shield you from changes in the interface, as the abstract base class will be updated so your custom code will still work. Moreover, the abstract base class takes care of pesky details such as enforcing the source and return types. The abstract base class to extend for transforming only payloads is `org.mule.transformer.AbstractTransformer`. This is the one that the `VelocityPayloadTransformer` extends, as shown in listing 5.4.

Listing 5.4 The Velocity payload transformer (imports and private members not shown)

```
public final class VelocityPayloadTransformer extends AbstractTransformer {
    private VelocityEngine velocityEngine;

    private String templateName;

    private Template template;

    public VelocityPayloadTransformer() {
        registerSourceType(Object.class);
        setReturnClass(String.class);
    }

    public void setVelocityEngine(final VelocityEngine velocityEngine) {
        this.velocityEngine = velocityEngine;
    }

    public void setTemplateName(final String templateName) {
        this.templateName = templateName;
    }

    @Override
    public void initialise() throws InitialisationException {
        try {
            template = velocityEngine.getTemplate(templateName);
        } catch (final Exception e) {
            throw new InitialisationException(e, this);
        }
    }

    @Override
    protected Object doTransform(final Object payload,
        final String encoding)
        throws TransformerException {
        try {
            final StringWriter result = new StringWriter();
            ↗ Sets acceptable source
            ↗ and return types
            ↗ Receives Velocity
            ↗ engine by injection
            ↗ Receives template
            ↗ name by injection
            ↗ Initializes
            ↗ Velocity
            ↗ template
            ↗ Performs transformation
            ↗ on payload
        
```

```
template.merge(new VelocityContext(
    Collections.singletonMap("payload", payload)),
    result);

return result.toString();

} catch (final Exception e) {
    throw new TransformerException(MessageFactory
        .createStaticMessage("Can not transform message with template: "
            + template), e);
}
}
```

There are several notable aspects of this code:

- We leverage the Spring-based configuration mechanism to get both the Velocity engine and the template name injected into the transformer.
- Though we could've deferred to Spring to initialize the Velocity template and inject it rather than the engine and the name, we let this happens in the initialization method so we can throw a proper exception referring to the transformer itself (this makes life easier for debugging, instead of a generic Spring context error).
- The source type is `Object` (as defined in ①). We don't want to limit where this transformer can be used, as Velocity can work on many types of objects. Most of the time a collection or a Java bean will be processed by this transformer.
- `doTransform` is the only method that must be implemented when extending `org.mule.transformer.AbstractTransformer`.
- We don't do anything with the encoding parameter because we work with Unicode strings during the transformation.
- We pass the message payload as a context entry named `payload` (not `src` or `data`) simply for consistency reasons, as other transformers, such as the script one (discussed in chapter 14) use `payload` too. Consistency is good.
- As Velocity throws an `Exception`, we have to catch it too. Note how we create a static message when this happens. We could've also created a message in a resource bundle so it could be translated, though the value at this level would be limited.

NOTE If you're wondering about the `initialise()` method, bear with us for now and just assume Mule will call it at the right time before the transformer gets used. The Mule internal API, including lifecycle events, will be covered in chapter 13.

Now, how do we use this transformer? Since it relies on Spring for the creation and injection of the Velocity engine, we need to use some Spring elements (see section 2.2.1) to make the magic happen. And because it's a custom transformer, we need to configure it with the `custom-transformer` element (shocking!). Here's the declaration of a global Velocity payload transformer for our email generation use case:

```

<spring:bean name="velocityEngine"
  class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
  <spring:property name="resourceLoaderPath" value="classpath:templates" />
</spring:bean>

<custom-transformer name="emailTransformer"
  class="com.muleinaction.transformer.VelocityPayloadTransformer">
  <spring:property name="velocityEngine" ref="velocityEngine" />
  <spring:property name="templateName" value="email.vm" />
</custom-transformer>

```

We rely on an Spring factory for instantiating and configuring the Velocity engine itself. Then we inject the Velocity engine into the transformer alongside the template name. This approach allows us to minimize the code of the transformer (listing 5.4 is only 48 lines, including the accessors), while benefiting from all the configuration flexibility Spring can buy us.

It's beyond the scope of this book to discuss Velocity templates, but here's what could be the beginning of an email template:

```
Dear ${payload.client.title} ${payload.client.lastName},
```

This template will be used with our transformer on messages whose payload is an “email context” bean that contains a `Client` bean, which exposes `getTitle()` and `getLastName()` accessors.⁵

As you've seen, it takes only a few lines of code to create a full-fledged transformer out of an existing transformation framework. Thanks to the usage of Spring, it's been possible to keep this code small and clear, while tapping some preexisting transformation logic. This should give you enough guidelines to reuse your existing transformation code inside Mule. But it should also give you enough guidance to get started with a complete transformer from scratch, should you need one.

BEST PRACTICE Transformers should be stateless and not assume anything about the order in which they process messages.

Just like we said before, this transformer can only use data in the payload. But what if we need to also use values extracted from the properties? If this is the case, we'd need to extend a different abstract class, as shown in the next section.

5.7.2 *Transforming messages*

We happened to add a current file reference as a property to the message triggering the email generation. Since we wanted clients to see this reference number in the generated emails (so they can refer to it when contacting Clood to complain about the outrageous fees), we needed to access the message properties from the Velocity transformer. How is this possible?

Whenever you need to read or modify message properties in a transformer, you need to have access to the `org.mule.api.MuleMessage` object itself, and not only its

⁵ In the next chapter, you'll see how we build the email context bean that's passed to this transformer.

payload as we've seen in the previous section. We'll modify the Velocity transformer we built in the previous section so it can read these properties. This second version will, of course, still give access to the payload: it has two objects that we'll pass in the Velocity context to the template. Note that a template designed for this second transformer won't work properly with the first one, as message properties will be unresolvable.

This time, the abstract base class to extend for transforming messages is `org.mule.transformer.AbstractMessageAwareTransformer`. This is the class that the `VelocityMessageTransformer` extends, as shown in listing 5.5.

Listing 5.5 The Velocity message transformer (imports not shown)

```

public final class VelocityMessageTransformer extends
    AbstractMessageAwareTransformer {
    private VelocityEngine velocityEngine;
    private String templateName;
    private Template template;
    public VelocityMessageTransformer() {
        registerSourceType(Object.class);
        setReturnClass(String.class);
    }
    public void setVelocityEngine(
        final VelocityEngine velocityEngine) {
        this.velocityEngine = velocityEngine;
    }
    public void setTemplateName(final String templateName) {
        this.templateName = templateName;
    }
    @Override
    public void initialise() throws InitialisationException {
        try {
            template = velocityEngine.getTemplate(templateName);
        } catch (final Exception e) {
            throw new InitialisationException(e, this);
        }
    }
    @Override
    public Object transform(final MuleMessage message,
        final String outputEncoding)
        throws TransformerException {
        try {
            final StringWriter result = new StringWriter();
            final Map<String, Object> context = new HashMap<String, Object>();
            context.put("message", message);
            context.put("payload", message.getPayload());
            template.merge(new VelocityContext(context), result);
        }
    }
}

```

- Sets acceptable source and return types** → `registerSourceType(Object.class);`
- Receives Velocity engine by injection** → `VelocityEngine velocityEngine;`
- Receives template name by injection** → `String templateName;`
- Initializes Velocity template** → `template = velocityEngine.getTemplate(templateName);`
- Performs transformation on message** → `template.merge(new VelocityContext(context), result);`

```

        return result.toString();
    } catch (final Exception e) {
        throw new TransformerException(MessageFactory
            .createStaticMessage("Can not transform message with template: "
                + template), e);
    }
}
}
}
}
```

If you compare the code of this second version of the Velocity transformer and the first one, you'll notice that we now pass two objects to the Velocity context: message and payload. Why do that when message should be enough, since you can get the payload from it? Again this is a matter of consistency with the existing transformers. You'll find the notion of passing both the payload and the complete message in other transformers. Moreover, this makes the second transformer backward-compatible with the first one. You can use the second one as a drop-in replacement for the first one, as existing Velocity templates will find the object they expect bound under the same name.

Configuring the Velocity message transformer is done exactly as for the payload transformer:

```

<spring:bean name="velocityEngine"
    class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
    <spring:property name="resourceLoaderPath" value="classpath:templates" />
</spring:bean>

<custom-transformer name="emailTransformer"
    class="com.muleinaction.transformer.VelocityMessageTransformer">
    <spring:property name="velocityEngine" ref="velocityEngine" />
    <spring:property name="templateName" value="email.vm" />
</custom-transformer>
```

The only difference is that with this version, we can now access message properties. Here's a new version of the previous email template, which will add a file reference to the header:

```
Our ref. ${message.getProperty('currentFileReference')}
Dear ${payload.client.title} ${payload.client.lastName},
```

Note how we access a particular message property using the `getProperty` method and its name.

At this point, you might wonder what it would take to modify the properties of a message. Not much, really. Let's say that we want our Velocity message transformer to mark a message with a time stamp whenever it's done with the transformation. Listing 5.6 shows the single-line addition to make this happen.

Listing 5.6 The transform method of the marking Velocity message transformer

```

@Override
public Object transform(final MuleMessage message,
    final String outputEncoding)
    throws TransformerException {
```

```

try {
    final StringWriter result = new StringWriter();

    final Map<String, Object> context = new HashMap<String, Object>();
    context.put("message", message);
    context.put("payload", message.getPayload());
    template.merge(new VelocityContext(context), result);

    message.setLongProperty(timeStampPropertyName,
                           System.currentTimeMillis());
    ← Sets long
    ← property to
    ← current time

    return result.toString();
} catch (final Exception e) {
    throw new TransformerException(MessageFactory
        .createStaticMessage("Can not transform message with template: "
        + template), e);
}
}
}

```

Because the `MuleMessage` instance is available in the `transform` method, you're basically free to alter its properties using the related methods (set, clear, remove). Note how we didn't hard-code the name of the property name we want to set with the timestamp value. This'll be externally configured with Spring the same way the template name is. Configuration eases reuse: it'll make your custom transformers more valuable.

TIP *Making your message transformer discoverable* Unlike payload transformers, which extend `org.mule.transformer.AbstractTransformer`, message transformers aren't discoverable by default. This means that if Mule looks for a transformer that can transform a particular source type to a particular return type, it won't discover a transformer that extends `org.mule.transformer.AbstractMessageAwareTransformer`. If you want your transformer to be discovered, you must then implement `org.mule.api.transformer.DiscoverableTransformer`. The transformer discovery mechanism supports the notion of weighting, which allows you to give precedence to a transformer above ones with similar source and return types.

Mule will look for discoverable transformers when you use the auto transformer that we mentioned earlier, but also when you use payload accessor methods (discussed in section 13.3.1).

As we said, auto transformation is an interesting option if the transformer you've created works with very specific source and return types (such as domain objects). Making a string-to-string transformer discoverable won't bring you much.

In this section, you've learned to implement custom transformers that can operate on message payloads and properties. You've also discovered some best practices in term of design and implementation.

Should Mule's extensive panoply of existing transformers miss a particular one, you're now ready to build it. There's no transformation need that you won't be able to tackle with Mule in a clean, efficient, yet simple manner.

5.8

Summary

Message transformation is a crucial feature of ESBs because it allows you to bridge the gap between different data types and formats. In this chapter, we've learned how transformation occurs in Mule, what it's good for, and how to leverage it in your integration projects.

You've discovered some of the existing Mule transformers. Some of them came from the core library, while others from specific modules or transports. Though they have different purposes, they're pretty similar in kind. This similarity makes them easy to learn and use. It also allows you to compose them in transformation chains to perform even more advanced or specific transformation operations.

Message transformation is yet another domain where Mule shines by its simplicity and extensibility. The several lines of code and configuration required to roll out your own custom transformers should've convinced you of this.

By now, you've also noticed how service components can be involved in message transformation. We'll look at this last part of Mule's core, the components, in the next chapter.

Working with components

In this chapter

- Understanding the role of components in Mule
- Common components from Mule Core
- Invoking SOAP and REST services
- Executing your business logic
- Advanced component configuration

If you've been around for a while in the happy field of software development, you've surely been exposed to different flavors of component models. CORBA, EJB, JavaBean, and now SCA have all helped familiarize us with the notion of the component. We understand that components represent entities that can perform specific operations, with well-defined interfaces and an accent on encapsulation.

Unsurprisingly, Mule supports its own component model. More surprisingly, it's often difficult to decide when to use or create a component in Mule. This difficulty stems from the extensive capacities of the routing, filtering, and transforming infrastructure that surrounds the components. The previous chapters explored these

capacities: you've discovered that you can achieve many complex integration scenarios without the need for any particular component.

So why bother about components?

In this chapter, we'll start by answering that question. Then we'll look at existing and custom components that perform message-level operations, remote logic invocation, and custom business logic execution. We'll also look closely at Cloud's usage of existing and custom components in their email generation, statistics, and file transfer services.

Components are at the core of Mule services: each service hosts a component. This component is the destination for messages after they've been received by the inbound endpoint, unless a filtering or forwarding router is used (see section 4.3.2). As illustrated in figure 6.1, a component can be the final destination of a message. This can happen if no outbound router is configured or if the component instructs Mule to stop further processing of the message.

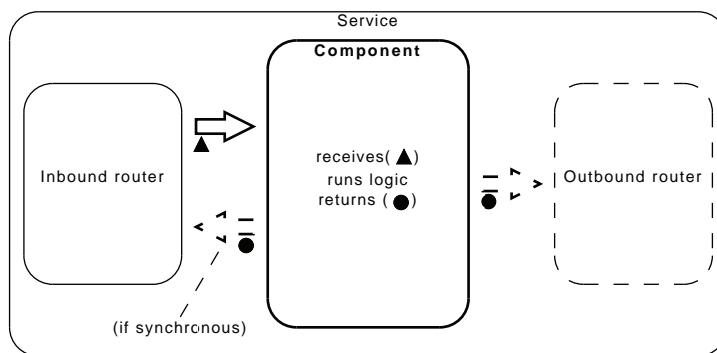


Figure 6.1 At the core of Mule services, components receive messages, process them, and return results.

NOTE Component invocations can also be wrapped by interceptors that allow you to share common behavior in a transversal manner. This advanced subject will be discussed in chapter 13.

Figure 6.1 also illustrates an important principle in the architecture of Mule. The response of the method that's been called on the component is both used as the return value of the inbound router endpoint (if it's synchronous) and sent to the outbound router for dispatching (if such an outbound router exists). This "one way in, two ways out" behavior is the cause of a lot of confusion and is also underestimated in its capacities. All the message exchange patterns supported by Mule, which are described in the online user guide,¹ are enabled by this behavior. The following sections will demonstrate how to leverage the component response in different scenarios.

WARNING *Components with attitude* A component can decide to return null or even nothing at all (if the method that's called on it is void): in this

¹ See <http://www.mulesource.org/display/MULE2USER/MEPs>.

case, the synchronous response will be null but no message will be sent to the outbound router.

A component can also take control of the message process flow. For example, it can cancel the routing that's configured on the service and perform a direct dispatch to another endpoint.

Don't go overboard in the routing logic you might be tempted to code in your component; instead you should prefer the more explicit and more readable configuration-based approach or consider leveraging the component binding feature (see section 6.3.4).

As you've learned in the previous chapters, transformers and routers already provide many message-processing capacities, such as splitting or enrichment. Where do components fit into this scheme? There are no hard rules, but here are a few use cases and guidelines to help you better understand:

- Some message-related operations aren't conceptually fitted anywhere else. For example, a transformer wouldn't be the best place for executing business logic or logging messages.
- Processing a message might require you to communicate with another service. Components can act as one-stop communication devices and perform operations such as synchronous RPC calls in a more straightforward manner than what you could do with, say, a chaining router and a bunch of transformers.
- Unlike other Mule moving parts, components don't mandate the implementation of a specific interface. This enables you to use any existing business logic POJO directly as a Mule component.
- Components can reify a preexisting business interface, defined, for example, with a WSDL in a contract-first approach.
- Components offer features that other Mule entities don't. For example, you can pool components only by means of configuration.
- Exceptions thrown at component level don't have the same semantics as exceptions thrown elsewhere. If your custom code executes business logic, throwing an exception from a transformer or a router wouldn't be interpreted and reported the same way by Mule as if you were throwing it from your component. The former would be handled by the connector's exception strategy, while the latter would be handled by the service exception strategy (see chapter 8 for more on exception handling).

"To component or not?" That's a question you should start to feel more confident answering.

If you're still hesitant, don't despair! The rest of the chapter will help you grok components. We'll start by looking at some existing components that ship with Mule: they're ready to be used and only need to be configured to start being useful. Then we'll detail the creation of custom components and all the possibilities Mule offers to configure them.

NOTE In the figures in the coming sections, you'll notice that transformers are represented on the inbound endpoints, as this is where they're configured. You might be puzzled by the fact that the messages represented as leaving the inbound router haven't been transformed yet. If you remember the discussion from the beginning of chapter 5, you should recall that the component will decide whether to take this transformer into account. Discussions in section 6.3.1 will make you realize that the reality is slightly more subtle than that. But, for now, considering that components decide to apply transformers or not is a fairly good approximation. This is why the transformation seems to happen inside the components in these figures.

6.1 Massaging messages

The core library of Mule contains components that can massage your messages in different manners. Though they don't execute any business logic-specific operations, they perform specific tasks that you'll find handy in a lot of different scenarios.

What can these core components do to your messages? Here's a list:

- *Bridge messages*—Pass messages from inbound to outbound routers
- *Echo and log messages*—Log messages and move them from inbound to outbound routers
- *Build messages*—Create messages from fixed or dynamic values

Let's review them in this order.

6.1.1 Building bridges

In the preceding chapters, you've seen many service examples that had no component configured. What does this mean? If components are at the core of Mule services, why do so many services not have any components? In fact, these examples rely on an implicit component that Mule uses when none is specified for a service.

This is the *bridge component*. As its name suggests, it bridges its inbound router to its outbound router, but doesn't perform any particular operation on the message. This said, it does honor any transformer defined on the inbound endpoint, as illustrated in figure 6.2. Often, you'll build services with this component because you won't have any specific logic to execute. In that case, the routers and transformers that kick in before or after the bridge will be enough to support the message-related operations you need.

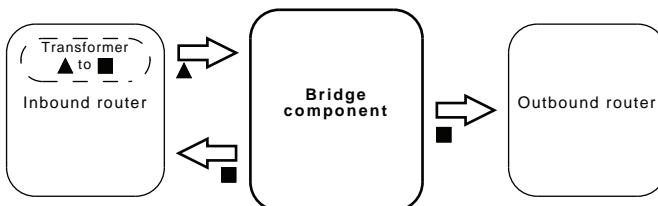


Figure 6.2 The bridge component passes inbound messages to the outbound router, while honoring the inbound transformer.

The bridge component returns the input message transformed by the inbound transformers, if any. In listing 6.1, messages sent to the VM inbound endpoint will be transformed by the Suffixer transformer and will be routed to the VM outbound endpoint. If the process that has sent a message to the VM inbound endpoint is waiting for a synchronous response, it'll receive the transformed message.

Listing 6.1 A service with an implicit bridge component

```
<service name="implicit bridge">
  <vm:inbound-endpoint path="ImplicitBridge.In">
    <transformer ref="Suffixer" />
  </vm:inbound-endpoint>
</inbound>
<outbound>
  <pass-through-router>
    <vm:outbound-endpoint ref="generic channel" />
  </pass-through-router>
</outbound>
</service>
```

Bridge component implicitly configured (no component defined)

The example of listing 6.1 isn't stunningly demonstrative, with its empty line where a component configuration element was expected. That said, relying on its implicit presence is the canonical way to define a bridge service in Mule. You might've seen configurations with a bridge component explicitly configured, as demonstrated in listing 6.2. Though the explicit configuration is currently technically equivalent to the implicit one, there's no guarantee that it'll stay like this in the future. It's possible that the implicit bridge might be refactored in a way that's even more efficient than the explicit one. Hence, you should prefer the implicit bridge approach.

Listing 6.2 A service with an explicit bridge component

```
<service name="ExplicitBridge">
  <inbound>
    <vm:inbound-endpoint path="ExplicitBridge.In">
      <transformer ref="Suffixer" />
    </vm:inbound-endpoint>
  </inbound>
  <brIDGE-COMPONENT /> <span style="border-left: 2px solid black; padding-left: 10px; margin-left: 10px; display: inline-block; vertical-align: middle; font-size: 0.8em; font-weight: bold;">Bridge component explicitly configured
  <outbound>
    <pass-through-router>
      <vm:outbound-endpoint ref="GenericChannel" />
    </pass-through-router>
  </outbound>
</service>
```

Sometimes it's useful to keep track of the messages that flow through a bridge service. The next component is useful for this.

6.1.2 Echoing and logging data

The *echo component*, which you discovered in the Echo World example in chapter 2, performs more than what its name indicates. Not only does it echo the incoming message, but it also logs it. Figure 6.3 illustrates its behavior.

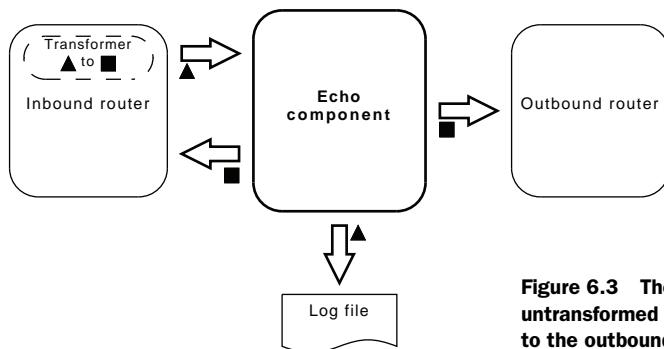


Figure 6.3 The echo component logs untransformed inbound messages and send them to the outbound router.

Like the bridge component, it returns the input message transformed by the inbound transformers. This makes the echo component usable with synchronous inbound endpoints and outbound routers. Listing 6.3 shows a configuration that is (at message level) functionally equivalent to the one shown for the bridge component. The only difference is that the inbound messages will be logged before being transformed and returned out of the echo component.

Listing 6.3 A service that logs and echoes messages back

```

<service name="Echo">
  <inbound>
    <vm:inbound-endpoint path="Echo.In">
      <transformer ref="Suffixer" />
    </vm:inbound-endpoint>
  </inbound>
  <echo-component />           ← Echo component
  <outbound>
    <pass-through-router>
      <vm:outbound-endpoint ref="GenericChannel" />
    </pass-through-router>
  </outbound>
</service>
    
```

The echo component logs a string rendition of the message at `INFO` level and for the `org.mule.component.simple` package. This log message is sent to the same logging framework as the one used by Mule itself. For a standalone server deployment, such as the one we use for this book's examples, the log files are located in `MULE_HOME/logs`. You can find more information about logging in chapter 8.

Even if functionally equivalent, the echo component is much less efficient than the bridge and should be used only if logging is necessary. A common approach is to start a configuration with echo components in services of particular interest and to replace these components with bridges when things work as expected.

NOTE *Log or echo?* You may have noticed the existence of another component in Mule's core schema named log-component and wondered how it differs from the echo component. At the time of this writing, there's no functional difference between the echo component and the log component. Prior to Mule 2.1.2, the log component used to return null, which limited its usage as a terminal component (messages weren't flowing out of it to the outbound router). This isn't the case anymore. The only difference that remains between these two components is the interfaces they implement: the log component implements org.mule.api.component.simple.LogService, whereas the echo component implements both org.mule.api.component.simple.LogService and org.mule.api.component.simple.EchoService.

Therefore, use the configuration element that best reveals your intention: use <log-component /> to indicate that the main intention of this service is to log messages and <echo-component /> if it's mainly about returning what it receives.

So far, the components we've looked at were only moving messages around. We'll now look at components that can build new messages.

6.1.3 Building messages

In chapter 4, you learned about leveraging routers to aggregate responses from different services. This aggregation mechanism is principally used to build a new message out of asynchronous responses. The *reflection message builder component*, which we'll now discover, can perform a similar aggregation, but out of synchronous service invocations. Figure 6.4 depicts this mechanism.

When a message is sent to the inbound endpoint of a service that uses a reflection message builder component, the following happens:

- The inbound transformers are applied, if any.
- The transformed message is synchronously sent to each endpoint of the outbound router.
- The payload of each response from each of the outbound endpoints is passed to the best matching method of the original untransformed inbound message.

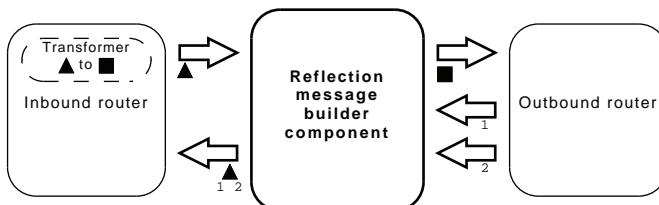


Figure 6.4 The reflection message builder component injects the results of remote calls in the inbound message.

If your head is spinning, this is normal! Let's look at actual usage of this component. For this, we'll return to our example of generating custom emails for Clood, Inc.'s clients. If you recall the discussion in section 5.7, we're using the Velocity transformer to transform an "email context" bean into email textual content. The UML diagram of this context class¹ is shown in figure 6.5.

When we create an instance of `com.clood.statistic.statistic.ActivityEmailContext`, we only set the client ID on it. Then we send it to a Mule service that uses the reflection message builder component to "enrich" it by calling other services to fetch the client bean and the activity report bean that match the client ID. This enricher service, named `EmailContextBuilder`, is illustrated in listing 6.4.

Listing 6.4 The reflection message builder can be used to enrich a message.

```
<service name="EmailContextBuilder">
  <inbound>
    <vm:inbound-endpoint path="EmailContextBuilder.In" />
  </inbound>
  <component>
    <singleton-object
      class="org.mule.component.builder.ReflectionMessageBuilder" />
  </component>
  <outbound>
    <multicasting-router>
      <vm:outbound-endpoint ref="ClientLookupChannel" />
      <vm:outbound-endpoint ref="ActivityReportLookupChannel" />
    </multicasting-router>
  </outbound>
</service>
```

1 Instantiates reflection message builder component

2 Endpoint list to get values for inbound bean

When the `ActivityEmailContext` object hits this service, it's broadcast to the outbound endpoints of the multicasting router ②. The response of the `ClientLookupChannel`, which unsurprisingly returns `Client` objects, will be passed to the relevant setter of `ActivityEmailContext` by the reflection message builder component ①. The same will be done with the response of the `ActivityReportLookupChannel`. Finally, the response of the service will be the original `ActivityEmailContext` instance whose properties have been set by values coming from the outbound endpoints. At this point, this object will be ready to be transformed into a meaningful email and sent to the relevant client.

TIP *Under construction* A good practice in software development consists of setting your IDE new method templates to contain a statement that throws an unsupported operation. This creates an unmissable reminder

¹ We spared you the usual accessors in this diagram.

| ActivityEmailContext |
|------------------------|
| id: long |
| client: Client |
| report: ActivityReport |

Figure 6.5 The context class used to generate client emails with the Velocity transformer

that something isn't quite done yet. Mule offers a component that supports a similar semantic: the *null component*. Although its name might suggest that it's a neutral component, this isn't the case. The null component will throw an exception if it receives a message. Use it in your development phase as a placeholder for "something needs to be done here," as shown in listing 6.5.

Listing 6.5 The null component is equivalent to an unsupported operation.

```
<service name="Null">
  <inbound>
    <vm:inbound-endpoint path="Null.In">
      <transformer ref="Suffixer" />
    </vm:inbound-endpoint>
  </inbound>
  <null-component />           ← Throws an
                                exception when called
  <outbound>                  ← Not-yet-used
    <pass-through-router>       outbound router
      <vm:outbound-endpoint ref="GenericChannel" />
    </pass-through-router>
  </outbound>
</service>
```

So far, the components we've looked at were mainly performing generic operations with messages. You've learned how to efficiently bridge the inbound and outbound routers of a service. You've also learned how to echo, log, and build messages using components available in Mule. We'll now start to look at components that can bring some logic into your Mule!

6.2 Invoking remote logic

You learned in the previous chapters that Mule can support many protocols and can connect to remote systems. We're pretty sure that the idea of calling some existing remote services has started to sprout in your mind. At this point, you might be unsure about what sort of endpoints, routers, and transformers you'd need to use to perform such calls.

If that's the case, we have good news for you: several Mule transports provide components that you can use to invoke remote logic in a synchronous RPC way. These components act as proxies for remote services, making them the one-stop solution you were looking for.

We'll consider two of these components that encompass two different approaches to remote service invocations, which the industry, in its infinite and collective wisdom, has christened SOAP and REST.¹ We won't detail these approaches, as it would be beyond the scope of this book, but we're sure you're familiar enough with them to understand what follows. Let's look at the components Mule offers to access services exposed with these two mainstream approaches.

¹ Taken in another context, "SOAP and REST" sounds like the mantra of a personal coach.

6.2.1 Feeling good with SOAP

After a little less than a decade in existence, web services have become the ubiquitous way to expose logic for remote access in corporate IT. It's more than probable that the environment in which you're developing offers SOAP-based web services. Mule offers a component that allows a seamless access to such remote services: it's called the *SOAP wrapper component*. This component exists in the different SOAP-supporting transports available for Mule.¹ Figure 6.6 illustrates its behavior. The notable feature of this component is that it performs its remote call within the normal message-processing flow of the message in the local Mule service, without mobilizing its routing infrastructure. Consequently, the routing infrastructure can be leveraged as with any nonremoting component, allowing the response of the SOAP service to be dispatched as needed.

Listing 6.6 demonstrates the usage of the Apache AXIS wrapper component. In this example, the remote service that's called is the free service from geocoder.us. Sending a string that contains an address to the GeoCoderSearch service inbound endpoint will make the wrapper component send a SOAP request. The wrapper component will use the configured address, method, and SOAP style and encoding for the request. It will then return the response synchronously. The response will be an array of GeocoderAddressResult, which is a value object that contains geolocation and address data.

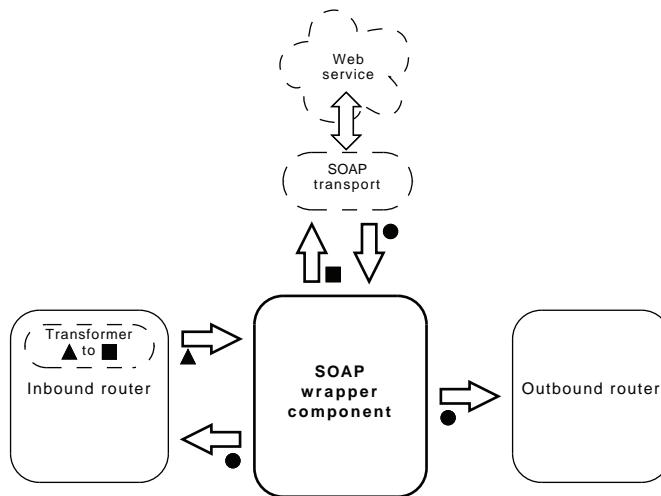


Figure 6.6 The SOAP wrapper component can invoke a remote web service within a Mule service.

Listing 6.6 Calling geocoder.us SOAP with the AXIS service wrapper component

```

<service name="GeoCoderSearch">
  <inbound>
    <vm:inbound-endpoint path="GeoCoderSearch.In" />
  </inbound>
  <axis:wrapper-component
    
```

SOAP wrapper component
from AXIS transport

¹ At the time of this writing, these transports are Apache AXIS and Apache CXF.

```

address="http://rpc.geocoder.us/service/soap/?method=geocode"
style="RPC" use="ENCODED" />
</service>

```

The simplicity of this configuration hides the heavy lifting performed behind the scenes by the SOAP transport to build the correct invocation message and map the response back to objects. This implies the need to create local proxy classes to map the result of the remote invocation, a practice that creates a sometimes undesirable tight coupling to the remote service.

Among other reasons, REST has been designed to alleviate this kind of rigidity.¹ We'll now look at the component that allows REST service access.

6.2.2 Taking some REST

Almost as old as SOAP, REST (short for *representational state transfer*) has started to gain traction in the past couple of years and has quickly become an important actor in Web 2.0 and WOA (Web-Oriented Architecture). The Mule HTTP transport contains a component that's specifically designed for accessing REST resources. This component, appropriately named *REST service component*, performs remote HTTP invocations within the normal message process flow that occurs in a Mule service, as illustrated by figure 6.7.²

REST services are based on standard HTTP features, such as methods and headers. Moreover, they're free to return data in the format they want (XML and JSON being two popular options). Therefore, you'll have to deal with these concerns when using the REST service component. Listing 6.7 illustrates this fact. This Mule service acts as a

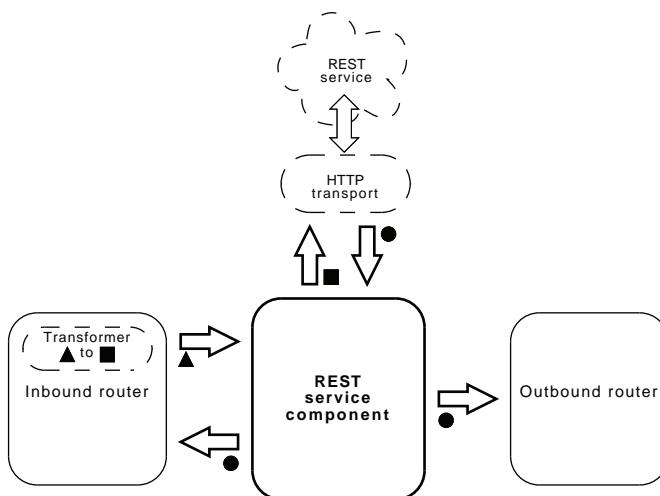


Figure 6.7 The REST service component can access a remote resource within a Mule service.

¹ We acknowledge that it's hard to transition from a section about SOAP to one about REST without hurting any feelings or cutting some corners.

² Mule also support exposing and consuming RESTful services, thanks to several transports (Abdera, Jersey, Restlet) available for download from the RESTpack home page: <http://www.mulesource.org/display/MULE/Mule+RESTpack>.

proxy for the Google REST search API. Sending a search term as a string to the inbound endpoint of this service will trigger a call to the Google search API. Thanks to a chain of transformers, the JSON response of Google search will be transformed into a `java.util.List` of web page titles from the hit list.

Listing 6.7 Calling the Google search API with the REST service component

```

<service name="GoogleSearch">
  <inbound>
    <vm:inbound-endpoint path="GoogleSearch.In">
      <transformers>
        <message-properties-transformer>
          <add-message-property
            key="http.custom.headers"
            value-ref="RefererHeader" />
        </message-properties-transformer>
      </transformers>
      <response-transformers>
        <byte-array-to-string-transformer />
        <custom-transformer
          class="com.muleinaction.transformer.JsonToXmlTransformer" />
        <xm:jxpath-extractor-transformer
          expression="titleNoFormatting/text()"
          singleResult="false" />
      </response-transformers>
    </vm:inbound-endpoint>
  </inbound>
  <http:rest-service-component
    httpMethod="GET"
    serviceUrl=
      "http://ajax.googleapis.com/ajax/services/search/web?v=1.0">
    <http:payloadParameterName value="q"/>
  </http:rest-service-component>
</service>

```

The code listing is annotated with several callouts:

- ① Adds referrer header as requested by Google AJAX API**: Points to the `<add-message-property>` element under the `<message-properties-transformer>`.
- Transforms REST response into list of strings**: Points to the `<byte-array-to-string-transformer />` and the `<custom-transformer>` block.
- REST component configured for Google Search API**: Points to the `<http:rest-service-component>` element.
- Passes message payload as q query parameter**: Points to the `<http:payloadParameterName value="q"/>` element.

NOTE Note how in ① we leverage a special property named `http.custom.headers` that's handled by the HTTP transport as a map of key/value pairs it'll use as extra HTTP headers. Note also that we refer to `Referer-Header`, which is a Spring-configured map not shown in the listing.

If you compare this configuration to the one from the SOAP example, you might find it overly complex. Bear in mind that the SOAP transport performs a lot behind the scenes and requires supporting stub classes. In contrast, when using the REST service component, all the nitty-gritty is visible and under your control. Moreover, your configuration mileage will vary depending on the requirements of the remote REST service you call, its response data format, and the data format your service will need to return.

You can now easily invoke remote business logic from your Mule services. But what if you want this logic to run within Mule itself? This is what we're now going to delve into.

6.3 Executing business logic

Suppose you have existing Java classes that can execute some business logic you want to use internally in your Mule instance. You might also want to expose this logic using one of Mule's many transports. What would it take to use these classes within Mule?

The answer is: not much. Mule doesn't mandate any change to your existing code; using it is just a matter of configuration. This is great news because the industry doesn't need yet another framework. Once-bitten-twice-shy developers have become leery of platforms that force them to depend on proprietary APIs. Aware of that fact, Mule goes to great lengths to allow you to use any existing class as a custom service component. Mule also allows you to use (or reuse) beans that are defined in Spring application contexts as custom components.

BEST PRACTICE Strive for creating Mule-unaware logic components.

This said, there are times when you'll be interested in getting coupled with Mule's API (detailed in chapter 13). One is when your component needs to be aware of the Mule context. By default, a component processes the payload of a Mule message. This payload is pure data and is independent of Mule. But sometimes, you'll need to access the current event context. At this point, you'll make your component implement `org.mule.api.lifecycle.Callable`. This interface defines only one method: `onCall(eventContext)`. From the `eventContext` object, it's possible to do all sorts of things, such as retrieve the message payload, apply inbound transformers, stop further processing of the message (no dispatching to the existing outbound routers), or look up other moving parts via the Mule registry. Another reason to get coupled with the Mule API is to receive by dependency injection references to configured moving parts such as connectors, transformers, or endpoint builders. We'll look at such a case in the configuration example in section 6.3.3.

NOTE *Instantiation policy* By now, you're certainly wondering how Mule takes care of instantiating your objects before calling the right method on them. Unless you decide to pool these objects, which is discussed in section 6.3.3, there are three main possibilities:

- Let Mule create a unique instance of the object and use it to service all the requests. This is demonstrated later, in listing 6.8 at ①.
- Let Mule create one new object instance per request it's servicing. This is done by using the `prototype-object` element or the short syntax where a class name is defined on the `component` element itself.
- Let Spring take care of object instantiation. In this case, Spring's concept of bean scope will apply.

Granting that creating custom components mostly amounts to creating standard business logic classes, there are technical aspects to consider when it comes to configuring these custom components. Since Mule doesn't force you to fit your code into a

proprietary mold, all the burden resides in the configuration itself. This raises the following questions that we'll answer in upcoming sections:

- How does Mule locate the method to call—aka the *entry point*—on your custom components?
- What are the possible ways to configure these components?
- Why and how should you pool custom components?
- How do you compose service components?

What are the benefits of an internal canonical data model?

WARNING Component versus component If you've looked at Mule's API, you might've stumbled upon the `org.mule.api.component.Component` interface. Is this an interface you need to implement if you create a custom component? Not really. This interface is mostly for Mule's usage only. It's implemented internally by Mule classes that wrap your custom components. We suspect that this interface was made public simply because Java doesn't have a "friend" visibility specifier.

Let's start with the important notion of entry point resolution.

6.3.1 Resolving the entry point

As we've just discovered, Mule components can be objects of any type. This flexibility raises the following question: how does Mule manage to determine what method to call on the component when a message reaches it? In Mule, the method that's called when a message is received is poetically called an *entry point*. To find the entry point to use, Mule uses a strategy that's defined by a set of *entry point resolvers*.

Figure 6.8 depicts this strategy. Entry point resolver sets can be defined at model level or at component level, with the latter overriding the former. If none is defined,

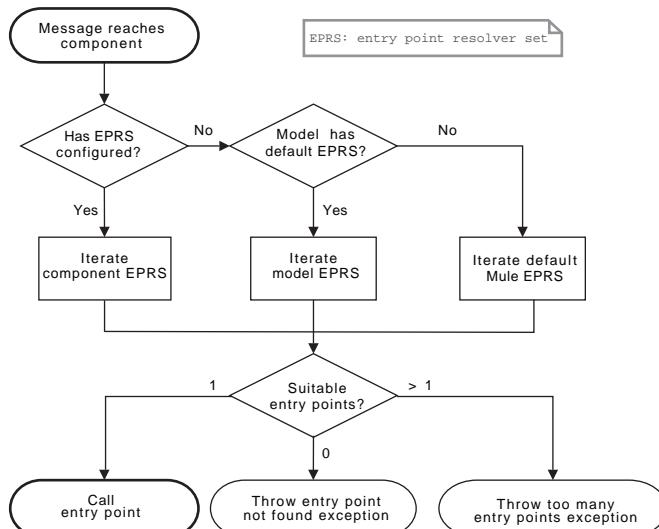


Figure 6.8 The component entry point resolution strategy used by Mule

Mule uses a default set. As you can see, if the strategy doesn't find exactly one suitable entry point, an exception will be thrown.

Mule offers a variety of entry point resolvers that you can leverage to ensure that messages reach the intended method on your components. Table 6.1 gives you a list of the available resolvers.

Each resolver supports different attributes and child elements, depending on its own configuration needs. The following attributes are shared across several resolvers:

- `acceptVoidMethods`—A component method is usually expected to return something. This attribute tells the resolvers to accept void methods.
- `enableDiscovery`—Asks a resolver to look for the best possible match (based on the message payload type), if a method name hasn't been explicitly provided.
- `transformFirst`—Automatically applies the transformers defined on the inbound endpoint to the message before trying to resolve the entry point.

Table 6.1 The available entry point resolvers can target component methods in a wide range of scenarios.

| Entry point resolver name | Behavior |
|-----------------------------------|--|
| array-entry-point-resolver | Selects methods that accept an array as a single argument. The inbound message payload must be an array. |
| callable-entry-point-resolver | Selects the <code>onCall (eventContext)</code> method if the component implements <code>org.mule.api.lifecycle.Callable</code> . This resolver is always in the default set that Mule defines. |
| method-entry-point-resolver | Selects the method that's configured in the resolver itself. |
| no-arguments-entry-point-resolver | Selects a method that takes no argument. Consequently, the message isn't passed to the component. |
| property-entry-point-resolver | Selects the method whose name is found in a property of the inbound message. The message property name to look for is defined on the resolver itself. |
| reflection-entry-point-resolver | Selects the method by using reflection to find the best match for the inbound message. If the message payload is an array, its members will be used as method arguments (if you really want to receive an array as an argument, you must use the <code>array-entry-point-resolver</code> discussed previously). If the message has a null payload, the resolver will look for a method without argument. |
| custom-entry-point-resolver | In rare cases, the previous resolvers can't satisfy your needs, and you can roll out your own implementation. For this, try subclassing <code>org.mule.model.resolvers.AbstractEntryPointResolver</code> rather than implementing <code>org.mule.api.model.EntryPointResolver</code> from scratch. |

WARNING *On call but no transformation duty* If you implement `onCall(event-Context)` and use the `callable-entry-point-resolver`, bear in mind that *the inbound transformers won't be automatically called for you*. In this case, you must call one of the `transformMessage` methods of the `eventContent` argument if you want to work with the transformed message in your callable custom component.

Similarly, if you disable transformation first at entry point resolver level, you'll have to acquire the event context (discussed in section 13.3) to perform the transformation programmatically.

If you remember the discussion from the beginning of chapter 5, you should remember that we said that *it's up to the component to decide whether to apply it*. Now you understand that the reality is slightly more complex: it's up to the entry point resolver or the component to transform the inbound message with the configured transformers, if need be.

Here are the child elements that are commonly available to resolvers:

- `exclude-object-methods`—This unique empty element instructs the resolver to ignore the standard Java object methods, such as `toString()` or `notify()`. It's the default behavior of all resolvers, but bear in mind that this default behavior is turned off if you use the following child element.
- `exclude-entry-point`—This repeatable element defines a single method name that the resolver must ignore. Caution: as soon as you use one `exclude-entry-point` element, you disable the intrinsic exclusion of the Java object methods. If there's a risk that messages may be dispatched to these undesired methods, use the `exclude-object-methods` element.
- `include-entry-point`—This repeatable element is used to strictly specify method names that the resolver must consider. If you specify several names, the resolver will use the first method with a matching name. This allows you to define a model-wide resolver that's applicable to several components.

At this point, you might feel overwhelmed by the versatility of the entry point resolution mechanism. The best hint we can give you at this point is not to overengineer your configuration with armies of finely tuned entry point resolvers. Rely first on the default set that Mule uses and add specific resolvers only if it's not able to satisfy your needs.

NOTE At this writing, the default set of entry point resolvers contains the following:

- A `property-entry-point-resolver`
- A `callable-entry-point-resolver`
- A `reflection-entry-point-resolver` configured to accept setter methods (they're normally excluded)
- A `reflection-entry-point-resolver` configured to accept setter methods and with transformation first disabled

It looks like a byzantine set of resolvers, but there's a reason for this. The default entry point resolver set is equivalent to the `legacy-entry-point-resolver-set`, which does all these contortions in order to be compatible with the behavior of Mule 1.

Let's look at a simple example. Listing 6.8 shows a bare-bones random integer generator service (we'll detail the component configuration side of things in the next section). The response to any message sent to its inbound router will be the value returned by a call on the `nextInt()` method of the random object. We use a no-arguments-entry-point resolver configured to precisely pick up the `nextInt()` method.

Listing 6.8 A random integer generator service

```
<service name="RandomIntegerGenerator">
  <inbound>
    <vm:inbound-endpoint path="RIG.In" />
  </inbound>

  <component>
    <no-arguments-entry-point-resolver>
      <include-entry-point method="nextInt"/>
    </no-arguments-entry-point-resolver>
    <singleton-object class="java.util.Random" />
  </component>
</service>
```

The code snippet shows a Mule service configuration. It includes an inbound endpoint and a component. The component contains a no-arguments-entry-point-resolver that includes the 'nextInt' method. An annotation points to this resolver with the text 'Targets nextInt() method of random object'. Another annotation points to the 'singleton-object' element with the text 'Instantiates standard Java random number'.

Using a transformer in conjunction with an existing entry point resolver might spare you some custom code and save your day. By pretransforming an incoming payload into a type that fits the argument of the desired method, you can finely guide the entry point resolver to hit the right target.

Let's go back once again to the client email generation that we used for Cloud, Inc. In section 6.1.3, we detailed the `EmailContextBuilder` service that calls other services in order to enrich a particular payload. You might recall that this service sends the incoming payload to the different outbound services it's configured with. This creates a mismatch between what's sent to these services (an instance of `com.cloud.statistic.statistic.ActivityEmailContext`) and what these services actually expect (a long representing the client ID). Listing 6.9 shows how we use a transformer to make the payload acceptable to the Client DAO object.

Listing 6.9 A transformer can guide the resolution of the desired component method.

```
<service name="ClientLookupService">
  <inbound>
    <vm:inbound-endpoint ref="ClientLookupChannel">
      <expression-transformer>
        <return-argument evaluator="bean" expression="id" />
      </expression-transformer>
    </vm:inbound-endpoint>
  </inbound>
```

The code snippet shows a Mule service configuration. It includes an inbound endpoint that references a channel named 'ClientLookupChannel'. This endpoint uses an expression transformer to evaluate a bean expression ('id') and return the result. An annotation points to this transformer with the text 'Extracts value from current payload'.

```

<component>
    <reflection-entry-point-resolver />
    <singleton-object class="com.clood.dao.ClientDao" />
</component>
</service>

```

Targets method that can accept extracted value

We use a bean expression-transformer (discussed in section 5.3.4) to extract the only value we’re interested in—`com.clood.statistic.statistic.ActivityEmailContext`, the client ID—and use this value as the new payload before relying on the reflection-entry-point resolver. Thanks to this pretransformation, the resolver can successfully locate and invoke `findById(long)` on the `com.clood.dao.ClientDao` component object.

BEST PRACTICE Adapt Mule to your components, not the other way around.

You’ve learned how to “direct” messages to the desired entry point on a service component object. We’ll now look at the different available options for configuring component objects.

6.3.2 Configuring the component

There are two main ways to configure a custom component:

- *Using Mule objects*—This approach is the simplest, though it offers the capacity to inject all sorts of dependencies, including properties and Mule moving parts. Its main drawback is that the component declaration is local to the service, hence not reusable.
- *Using Spring beans*—This is convenient if you have existing Spring application context files that define business logic beans. You can then use these beans as service components. Another advantage is the lifecycle methods Spring can call on your beans when the host application starts up and before it terminates.

To illustrate these two different approaches, we’ll configure a simple random integer generator using the stock JDK random class. We’ll set its seed to a predetermined configured value (don’t try this in your own online casino game).

WARNING *Property resolution challenges* There are cases when you’ll have to use Spring instead of Mule for configuring a component. Mule uses property resolvers, which sometimes get confused by the mismatch between the provided and the expected values. For example, if you try to set a byte array to a `java.lang.Object` property of a component, you’ll end up with a string representation of this array (the infamous “[B..” string) instead of having the value correctly set. This doesn’t happen if you use Spring to configure this component.

Listing 6.10 shows this service with its component configured using a Mule object. Note how the seed value is configured using a property placeholder. Note also that, like Spring, Mule supports the notion of singleton and prototype objects. Because

`java.util.Random` is thread-safe, we only need a unique instance of this object to serve all the requests coming to the service. This is why we use the `singleton-object` configuration element.

Listing 6.10 A fixed-seed random integer generator service

```
<service name="SeededRandomIntegerGenerator">
  <inbound>
    <vm:inbound-endpoint path="SRIG.In" />
  </inbound>
  <component>
    <no-arguments-entry-point-resolver>
      <include-entry-point method="nextInt" />
    </no-arguments-entry-point-resolver>
    <singleton-object class="java.util.Random">
      <property key="seed" value="${seed}" />
    </singleton-object>
  </component>
</service>
```

The code listing shows a Mule configuration for a service named "SeededRandomIntegerGenerator". It includes an inbound endpoint and a component. The component uses a "no-arguments-entry-point-resolver" to handle the "nextInt" method. It also contains a "singleton-object" configuration for the "java.util.Random" class, setting its "seed" property to "\${seed}". Two annotations are present: one pointing to the "nextInt" entry-point resolver with the text "Targets nextInt() method of random object", and another pointing to the "seed" property of the singleton-object with the text "Instantiates and calls setSeed() to configure random object".

There are cases where using a single component instance to serve all the service requests isn't desirable—for example, if the component depends on thread-unsafe libraries. In that case, if the cost of creating a new component instance isn't too high, using a prototype object can be an option. For this, the only change necessary consists in using the `prototype-object` configuration element instead of `singleton-object`. We'll come back to this subject in the next section.

TIP *Blissful statelessness* Stateful objects can't be safely shared across threads, unless synchronization or concurrent primitives are used. Unless you're confident in your Java concurrency skills and have a real need for that, strive to keep your components stateless.

Despite reducing local concurrency-related complexity, another advantage of stateless components is that they can be easily distributed across different Mule instances. Stateful components often imply clustering in highly available deployment topologies (this is further discussed in chapter 7).

Now let's look at the Spring version of this random integer generator service. As listing 6.11 shows, the main difference is that the service component refers to an existing Spring bean instead of configuring it locally. The bean definition can be in the same configuration file, in an imported one, or in a parent application context, which allows reusing existing Spring beans from a Mule configuration.

Listing 6.11 A Spring-configured fixed-seed random integer generator service

```
<spring:bean id="Random" class="java.util.Random">
  <spring:constructor-arg value="${seed}" />
</spring:bean>
<service name="SpringSeededRandomIntegerGenerator">
```

The code listing shows a Mule configuration for a service named "SpringSeededRandomIntegerGenerator". It includes a Spring bean definition for a "Random" object with a constructor argument "seed" and a service component. An annotation points to the "Random" bean with the text "Instantiates and configures random number generator".

```

<inbound>
    <vm:inbound-endpoint path="SSRIG.In" />
</inbound>
<component>
    <no-arguments-entry-point-resolver>
        <include-entry-point method="nextInt"/>
    </no-arguments-entry-point-resolver>

    <spring-object bean="Random" />
</component>
</service>

```

① References Spring-configured random object

Depending on your current usage of Spring and the necessity to share the component objects across services and configurations, you'll use either the Mule way or the Spring way to configure your custom service component objects.

Let's now look at advanced configuration options that can allow you to control the workload of your components.

6.3.3 Handling workload with a pool

Mule allows you to optionally pool your service component objects by configuring what's called a *pooling profile*. Pooling ensures that each component instance will handle only one request at a time. It's important to understand that this pooling profile, if used, influences the number of requests that can be served simultaneously by the service. The actual number of concurrent requests in a service is constrained by the smallest of these two pools: the component pool and the thread pool (refer to chapter 16 for more on this subject).

When would using a pooled component make sense? Here are a few possible use cases:

- *The component is expensive to create*—It's therefore important that the total number of component object instances remain under control.
- *The component is thread-unsafe*—It's then essential to ensure that only one thread at a time will ever enter a particular instance of the component.

NOTE *Pool exhaustion* By default, if no component object instance is available to serve an incoming request, Mule will disregard the maximum value you've set on the polling profile and create an extra component instance to take care of the request. You can change this and configure the polling profile to wait (for a fixed period of time or, unwisely, forever) until a component instance becomes available again. You can also define that new requests must be rejected if the pool is exhausted. For example, the following pooling profile sets a hard limit of 10 component objects and rejects incoming requests when the pool is exhausted:

```
<pooling-profile maxActive="10" exhaustedAction="WHEN_EXHAUSTED_FAIL" />
```

To illustrate pooled components, we'll look at a service Cloud uses to compute the MD5 hashcode of files they receive from their clients. The service performs this computation on demand: it receives a message whose payload is the file name for which the

hash must be calculated, performs the calculation, and returns the computed value. This is an important feature in validating that we've received the expected file and that we can proceed with it (like pushing it to the client's server farm in the cloud).

Because this computation is expensive for large files, we'll use pooled components. Listing 6.12 demonstrates the configuration for this service. The pooling profile element is explicit: one instance of the `com.clood.component.Md5FileHasher` component object will be created initially in the pool and a maximum of five active objects will exist at any point of time. We don't expect this service to receive heavy traffic, but should this happen, we allow a maximum of 15 seconds of waiting time in the event that the component pool becomes exhausted.

Listing 6.12 Controlling the resource usage of a service with a pooled component.

```
<service name="Md5FileHasher">
  <inbound>
    <vm:inbound-endpoint path="Md5FileHasher.In" />
  </inbound>
  <pooled-component>           +--- Declares component
    <prototype-object>          |   as pooled
      class="com.clood.component.Md5FileHasher">
      <property key="fileConnector"
        value-ref="NonDeletingFileConnector" />
      <property key="sourceFolder"
        value="${java.io.tmpdir}" />
    </prototype-object>          +--- Defines pooling profile
    <pooling-profile>           |
      initialisationPolicy="INITIALISE_ONE"
      maxActive="5"
      exhaustedAction="WHEN_EXHAUSTED_WAIT"
      maxWait="15000" /
    </pooling-profile>
  </pooled-component>
</service>
```

NOTE Note how in listing 6.12 we inject a reference to a Mule file connector instance. We use the file connector to read from a particular directory: you may wonder why we do that when it's a trivial task to read a file using plain Java code. In fact, using the Mule connector provides us with all the statistics, exception handling, and transformation features of Mule. In case you're wondering, here's the declaration of `NonDeletingFileConnector`:

```
<file:connector name="NonDeletingFileConnector"
  autoDelete="false" />
```

Similarly, you can inject global transformers and even global endpoints into your component. Of course, doing so couples your component with the Mule API (see chapter 13 for more on this subject).

You can leverage the pooling component configuration with Spring beans, too. For this, simply use a `spring-object` reference element ①, as you saw in 6.11, and you'll

be done. Bear in mind that you'll have to define the scope of the bean to prototype, so Mule will be able to populate the component pool with different object instances.

Before we close this section on pooling, here's a piece of advice: don't go overboard with pooling. Use it judiciously. Most of the time, nonpooled component objects will do the trick for you. Using pooling indifferently often amounts to premature optimization.

The last feature we're about to discover will allow your components to reach out and touch other services.

6.3.4 Reaching out with composition

Service composition is a powerful way to create new services out of existing ones. Mule offers a mechanism for performing such composition at component level called *component binding*. As figure 6.9 suggests, Mule can inject a dynamic proxy for a determined interface into your custom components. Calls to this proxy are then routed to a specified endpoint using the standard Mule message infrastructure. This means that the remote service is unaware that it participates in a composition: for this service, this is business as usual. It doesn't need to know anything about the interface to which it's been bound in the caller component. The remote service only needs to be able to accept an incoming message and return a synchronous response that satisfies the arguments of the method called on the interface and its return type, respectively.

You might wonder why this approach would be better than getting ahold of the Mule service inbound endpoint and dispatching a message to it from the client component. This would work, but would make your client component aware of Mule's message dispatching API, hence coupling your component code to the framework itself. A better approach consists in creating an interface that represents the contract you'll have with the remote service and let Mule bind this interface to the remote service inbound endpoint. This binding is done by the reflective injection of a dynamic proxy that implements the interface inside the client component.

Clood, Inc., leverages this binding mechanism to invoke the MD5 file hasher service that we detailed in the previous section. In fact, this is a feature we intensively use

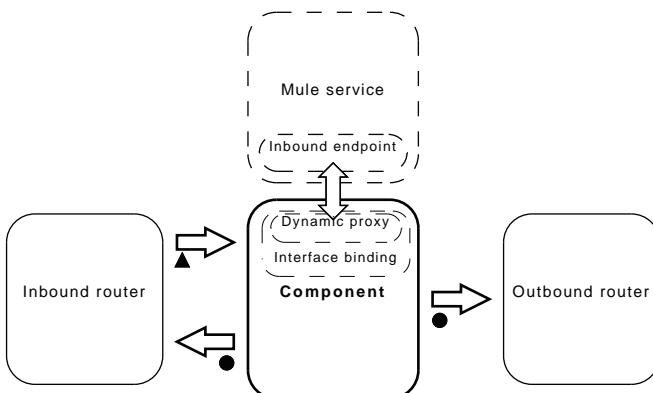


Figure 6.9 A component can compose other services through dynamic proxies of bound interfaces.

for all of our service composition needs. Let's look at how this binding happens. Here's the definition of the interface for the MD5 file hasher:

```
public interface Md5FileHasherService {
    String hash(String fileName);
}
```

And here's the setter in the client component that allows Mule to inject a dynamic proxy via reflection:

```
public void setMd5hasher(final Md5FileHasherService md5hasher) {
    this.md5hasher = md5hasher;
}
```

In this case, after instantiation, our component will hold in `md5hasher` a reference to a dynamic proxy that implements `Md5FileHasherService`. Whenever our component calls the `hash` method of `md5hasher`, Mule will transparently call an outbound endpoint and return the result of this remote invocation as the result of the call to the `hash` method.

Listing 6.13 shows how all this is configured. The binding element takes care of associating a particular method of the interface with an outbound endpoint. There's no need to define which setter must be called on `com.clood.component.Md5FileHasherClient`: Mule selects the most appropriate one based on its method signature.

Listing 6.13 A custom component can bind another Mule service to an interface.

```
<service name="Md5FileHasherClient">
    <inbound>
        <vm:inbound-endpoint path="MSC.In" />
    </inbound>

    <component>
        <singleton-object
            class="com.clood.component.Md5FileHasherClient" />
        ① Declares custom component
        <binding
            interface="com.clood.component.Md5FileHasherService"
            method="hash">
            ② Defines endpoint associated with this method
            <vm:outbound-endpoint
                path="Md5FileHasher.In"
                synchronous="true" />
        </binding>
    </component>
</service>
```

Note that in our case, because the interface contains only one method, the `method` argument could've been omitted on the binding element ①. Note also that the called outbound endpoint ② must be synchronous to ensure that the response of the other Mule service will be received. It's also possible to inject several different binding interfaces into the same component, provided Mule can identify in it a unique matching setter method for each of these interfaces.

WARNING Null indigestion When designing your remote service interface, try to use wrapper objects instead of their corresponding primitives (for example `java.lang.Integer` instead of `int`). This is because if the call to the proxied remote endpoint fails, a Mule message with a null payload will be returned. Primitives have digestion problems with nulls: a `NullPointerException` will be thrown in the dynamic proxy when the result of the invocation will be downcasted. Using a wrapper object allows the client service to receive a null reference and to deal with it.

This simple mechanism delivers the power of service composition to your custom component. It also raises the level of abstraction involved in interservice communications. When building complex integration configurations, you'll certainly come to appreciate this feature.

BEST PRACTICE Favor using Mule's infrastructure (via service composition or raw transport usage—see section 13.1.3) to connect to remote endpoints instead of coding it yourself.

We'll close this section by discussing how a canonical data model can also help with intercomponent communications needs.

6.3.5 Internal canonical data model

The following is a powerful approach to using business components within Mule:

- Using Mule-independent business classes as service components
- Configuring them with Mule or Spring mechanisms
- Exposing the services' methods over various endpoints
- Transforming data from transport specific forms to a canonical one that's common to all service components

This approach is illustrated in figure 6.10.

The transformations to and from the canonical form are usually done with custom transformers (such as the community-supported Smooks transformer) or with a combination of XSL-T and XML marshalling techniques. Building such an internal canonical data model simplifies many aspects of your configuration, including the unification of the entry point resolution mechanism or the handling of exceptional situations (caught exceptions will all have a payload of the same type).

It's up to you to decide whether building such a canonical data model makes sense for your project. If you only deploy a few services, the answer is most certainly no.

This section gave you an overview of the possibilities for running business logic in custom components. You've seen that the capacities of Mule in this domain go far beyond a simple object factory, as the platform offers component pooling and service composition.

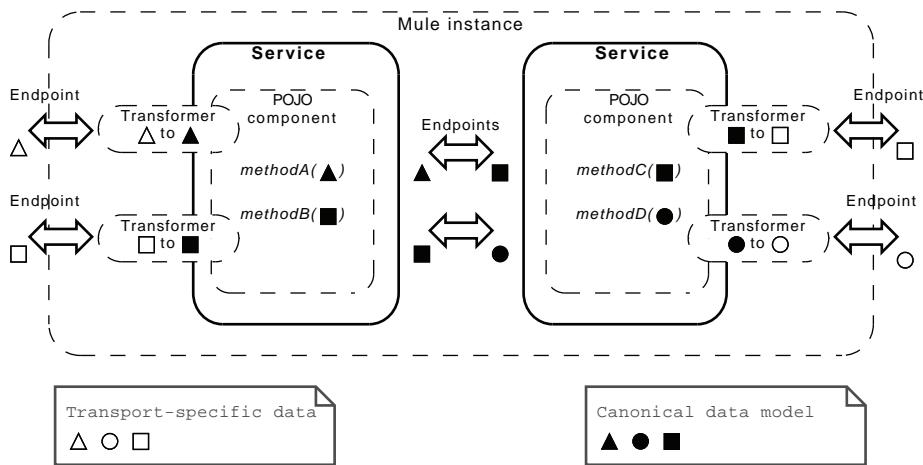


Figure 6.10 Canonical data transformations at play with POJO business components

6.4 Summary

Components are first-class citizens of a Mule configuration. They can be discreet (but efficient) like the ubiquitous bridge component. They can also encompass custom business logic under the form of standard Java objects.

You've discovered that Mule offers a wealth of standard components that allow you to perform simple message operations but also to invoke remote logic using standard web service protocols. You've also learned about the numerous options Mule provides when it comes to running your own business logic in custom components.

At this point, we've covered the configuration mechanisms of Mule and its main moving parts. By now, you should be able to create nontrivial integration projects, as you've learned to tap the extensive capacities of Mule's transports, routers, transformers, and components. In the coming chapters, we'll look at more advanced topics such as security and transaction management. We'll also look at deployment and monitoring strategies to ensure that Mule plays a prime role in your IT landscape in the best possible conditions.

Part 2

Running Mule

In part 1, you learned about the fundamentals of Mule. You discovered its philosophy, configuration principles, and major moving parts. You've also run examples that exercised the main building blocks of Mule services: transports, routers, transformers, and components. Part 2 will take you further and will guide you through the next steps toward running Mule in a production environment.

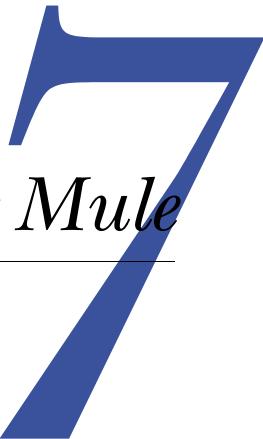
As an ESB and an integration platform, Mule supports several very different deployment strategies. In chapter 7, we'll review these strategies. We'll also explore the vast subject of deployment topologies and review different approaches for managing your deployments.

Problems happen. As tough as it is, Mule can stumble. This usually translates into exceptions being thrown. Chapter 8 will guide you in putting into place a sound error management strategy. You'll also learn how to recover from connection flakiness and what Mule offers in terms of logging.

All enterprise applications are subject to security concerns, and Mule doesn't elude these constraints. Chapter 9 will show you how to restrict access to certain resources with authentication and authorization mechanisms. You'll also learn how to encrypt and decrypt data.

Mule is often deployed in environments where data integrity is critical. This is when using the abilities of transaction-aware transports, such as JMS or JDBC, usually comes into play. You'll discover Mule's transaction management mechanism in chapter 10.

To wrap up the discussion about running Mule in production, we'll focus on monitoring in chapter 11. There, you'll see different approaches for keeping an eye on your Mule instances and ensuring they're healthy and behaving correctly.



Deploying Mule

In this chapter

- Discovering the deployment strategies supported by Mule
- The why and what of a few deployment topologies
- Managing the deployment of ESB instances
- Mule Galaxy

If you reached this chapter following the natural order of the book—you've learned how to configure Mule and have seen a few samples running—then at this point, you might be wondering how to move from a "works on my machine" situation to running Mule in production.

Or you may also have directly jumped to this chapter because your main concern is figuring out if and how Mule will fit into your IT landscape. You may have concerns about what you'll end up handing off to your production team: they may have operational or skills constraints that you must absolutely comply to. You might also be wondering about the different deployment topologies Mule can support.

Whatever path led you to this chapter, you'll find answers about the crucial topic of deploying Mule software and configuration files. We'll explore this matter by looking at three different aspects of deployment:

- *Deployment strategies*—where we'll consider the different runtime environments supported by Mule
- *Deployment topologies*—which will introduce the notions of instance-level and network-level topologies and how they relate to each other in Mule
- *Deployment management*—that will be focused on the challenges associated with managing your deployments and your different options for doing so

In all the upcoming sections, you'll come to realize that Mule is incredibly flexible and can adapt to your needs. In this, Mule differs from many of its competitors, which often mandate a unique way of doing each of these deployment-related activities. This diversity of choice can be overwhelming at first, but you'll soon realize that your needs and constraints will guide you in picking what's best for your project. At this point you'll be glad that Mule is such a versatile platform.

So, without further ado, let's get started exploring Mule's deployment strategies.

7.1 Deployment strategies

If you've already dealt with other ESBs, you probably had to use an installer of some sort in order to deploy the ESB application. After that, the ESB was ready to be configured, either via a GUI or directly by creating configuration files.

Mule differs from this shrink-wrapped approach because of its dual nature. Mule is described as “a lightweight messaging framework and highly distributable object broker,” which means that it supports more deployment strategies and flexibility than a traditional ESB does. As an object broker, Mule can be installed as a standalone server, pretty much like any application server. As a messaging framework, Mule is also available as a set of libraries¹ that you can use in any kind of Java application.

This flexibility leaves you with a choice to make as far as the runtime environment of your Mule project is concerned. Like most choices, you'll have to base your decision on your needs and your constraints. For example, you might need to connect to local-only EJBs in a particular server. Or the standard production environment in use in your company might constrain you to a particular web container.

TIP *No strings attached* Deciding on one deployment strategy doesn't lock you into it. With a few variations in some transports, such as the HTTP transport that can give way to the servlet transport in a web container, there's no absolute hindrance that would prevent you from migrating from one deployment strategy to another one.

In the following sections, we'll detail the five different deployment strategies that are possible with Mule and present their pros and cons. You'll discover that Mule is a contortionist capable of extremes such as running as a standalone server or being embedded in a Swing application. This knowledge will allow you to make an informed decision when you decide on the deployment strategy you'll follow.

¹ More specifically: a set of Maven artifacts.

NOTE At the time of this writing, none of the strategies that we're about to discuss support hot deployment. A restart of Mule is in order any time you change its configuration.

7.1.1 Standalone server

The simplest (yet still powerful) way to run Mule is to install it as a standalone server. This is achieved by downloading the complete distribution and following the detailed installation instructions that are available on the MuleSource web site: <http://mulesource.org/display/MULE2INTRO/Installing+Mule>. When deployed that way, Mule relies on the Java Service Wrapper from Tanuki Software to control its execution. Figure 7.1 shows a conceptual representation of this deployment model.

The wrapper is a comprehensive configuration and control framework for Java applications. By leveraging it, Mule can be deployed as a daemon on Unix-family operating systems or a service on Microsoft Windows. The wrapper is a production-grade running environment: it can tell the difference between a clean shutdown or a crash of the Java application it controls. In case the application has died in an unexpected manner, the wrapper will restart it automatically.

Use the following syntax to control the wrapper:

```
mulev [console|start|stop|restart|status|dump] [-configv <my-config.xml>]
```

If no particular action is specified, console is assumed: the instance will be bound to the terminal where it was launched. If no configuration filename is specified, Mule will look for a configuration file named mule-config.xml in the classpath and in its working directory. The startup script supports several other optional parameters, including -debug (to activate the JVM's remote debugging; see section 12.2.2) and -profile (to enable YourKit profiling; see section 16.2.1).

TIP *Going through the wrapper* Often you'll need to pass parameters to your Mule configuration by using Java system properties (for example, to provide values for property placeholders in your configuration file). Because of the nature of the wrapper, any system properties you try to set with the classic `-Dkey=value` construct will be applied on the wrapper itself. Hence they won't be set on the Mule instance that's wrapped. The solution is to use the `-M` command-line argument, which passes everything after itself directly to the Mule application. For example, the following code will set the `key=value` system property on Mule's instance:

```
mule -config my-config.xml -M-Dkey=value
```

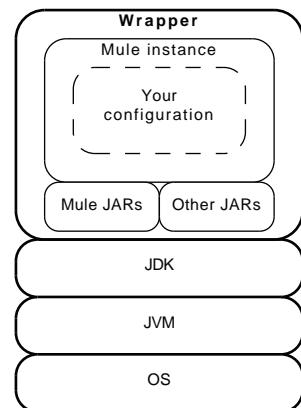


Figure 7.1 Conceptual deployment model of Mule standalone server

Figure 7.2 presents the structure created after installing Mule's complete distribution. The top-level structure presents no particular surprise: for example, as expected, bin is where to go to find the control scripts for different operating systems. We've expanded the lib subdirectory so you can see the structure where the different types of libraries that Mule depends on are stored. There's one notable directory in there: user. This is where you can drop any extra library you want Mule to be able to use. It's also the destination for patches, as its content is loaded before what's in the mule directory.²

In a production environment, there's a slight drawback to dropping your own libraries in the user directory. In doing so, you make it harder for your operations team to switch between Mule versions, as they'll have to tweak the server to deploy your libraries. If you run several differently configured Mule instances in the same server, the problem might even become more acute: different instances might have different, if not conflicting, libraries.

A clean solution for project deployment management is demonstrated by the examples that ship with the Mule distribution: you can create your own directory structure anywhere you want. You can then start Mule with a short script that calls the main script in Mule's standalone server, which is located via the MULE_HOME environment property. This is what we've used for our publication application of chapter 5, whose directory structure is shown in figure 7.3. Note how we've stored our required libraries and configuration files in this directory structure. Thanks to this approach, you can now change from one Mule version to another by simply changing the value of MULE_HOME.

BEST PRACTICE With Mule standalone, deploy complete project structures and reserve the /lib/user directory for patch installation and shared libraries (database drivers).

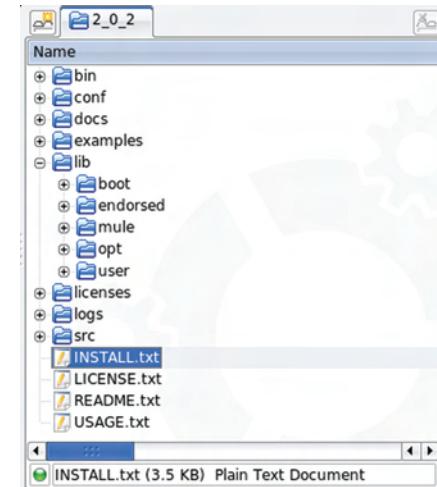


Figure 7.2 Directory structure of Mule standalone server

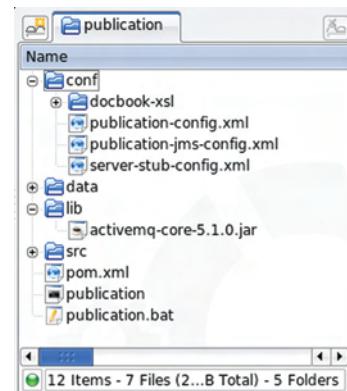


Figure 7.3 Directory structure of an application that uses the Mule standalone server

² Mule needs to be restarted to pick up any JAR that you drop in this directory.

Table 7.1 Pros and cons of the standalone server deployment option

| Pros | Cons |
|--|--|
| Install a standard Mule distribution and you're ready to go. | Can be an unfamiliar new piece of software for operations. |
| Proven and solid standalone thanks to the wrapper. | Dependency management can be tedious for large projects. |
| Very well-suited for ESB-style deployment. | Not suited for "Mule as a messaging framework" approach. |
| Direct support for patch installation. | |

Table 7.1 recaps the pros and cons of the standalone server deployment option.

Should any of the cons be a show stopper for you, don't despair! Mule has other deployment options in its bag of tricks. Let's now look at a variant of the standalone server that brings governance to your Mule.

7.1.2 NetBoot server

If handling Mule versions and deploying configuration files is too much hassle for your operations team, then NetBoot might be the right deployment strategy for your company.

The Mule NetBoot distribution is a slimmed-down version of the complete Mule distribution. It's designed to pull all the files it needs at startup from a specific repository, named Galaxy (discussed in section 7.3.2). As illustrated in figure 7.4, this includes Mule's libraries plus your own libraries, and of course, your configuration files. Like the standalone server, NetBoot leverages the wrapper too: all we've said in the previous section applies here as well.

Depending on how you organize your data in Galaxy, starting a NetBoot instance can be as simple as running this command from the NetBoot bin directory:

```
mule -M-Dgalaxy.app.workspaces=MyApplication
```

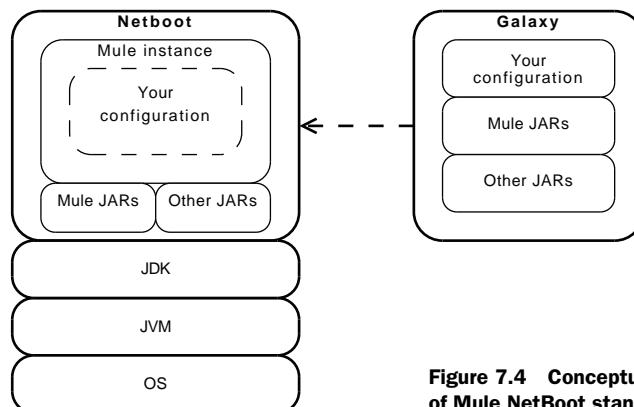


Figure 7.4 Conceptual deployment model of Mule NetBoot standalone server

As you can see, NetBoot parameters are passed as Galaxy-prefixed system properties through the wrapper (see the Tip in the previous section). There are a wealth of parameters that you can use, including `galaxy.host` and `galaxy.port` to control the connection to a remote Galaxy repository, and `galaxy.username` and `galaxy.password` for authentication purposes.

NOTE *Cashing in on the cache* NetBoot creates a local cache where it stores all the files pulled from Galaxy. At startup, NetBoot checks if more recent versions of these files exist in Galaxy and refresh them if need be. If Galaxy isn't reachable, NetBoot will rely on its local cache only. The cache directory structure represents the different workspaces NetBoot has already downloaded. For example, the cache for the previous sample will be created in the `netboot/lib/cache/MyApplication` directory.

As is the case for JARs in Mule's lib directory structure, all JARs anywhere in the cache directory hierarchy will be added to the classpath.

Don't hesitate to manually clean the content of the cache if you want to restart from scratch with NetBoot.

We'll come back to NetBoot and Galaxy when we discuss deployment management in section 7.3.2. At this point, let's just say that this deployment strategy allows you to manage multiple versions of Mule, along with your own JARs and configuration files, in a centralized manner.

Pros and cons of the NetBoot deployment option are listed in table 7.2.

Table 7.2 Pros and cons of the NetBoot deployment option

| Pros | Cons |
|--|--|
| Leverages the Galaxy SOA governance platform. | Can be an unfamiliar new piece of software for operations. |
| Proven and solid standalone thanks to the wrapper. | Need to deploy Galaxy in a production-grade manner. |
| Very well suited for ESB-style deployment. | Command lines can become hairy in nontrivial scenarios. |

Governance is an important factor to consider if you intend to deploy numerous Mule instances: with NetBoot you now have a tool that can help you to grow gracefully in SOA. Now let's look at how you could use Mule in a standard Java application.

7.1.3 *Embedded in a Java application*

If you've built applications that communicate with the outside world, you probably ended up building layers of abstractions from the low-level protocol to your domain model objects. This task was more or less easy depending on the availability of libraries and tools for the particular protocol. If at one point you needed to compose or orchestrate calls to different remote services, things started to get more complex. At

this point, using Mule as the “communicating framework” of your application could save you a lot of hassle.

Embedding Mule in an application is a convenient way to benefit from all the transports, routers, and transformers we discussed in part 1 of the book. Besides, your application will benefit from the level of abstraction Mule provides on top of all the different protocols it supports.

As shown in figure 7.5, Mule can be embedded in a standard Java application. This makes sense if the application isn’t destined to be run as a background service³ but, for example, as an interactive front end.

In the embedded mode, it’s up to you to add to the classpath of your application all the libraries that will be needed by Mule and the underlying transports you’ll need. Since Mule is built with Maven, you can benefit from its clean and controlled dependency management system by using Maven for your own project.⁴ See section 12.1 for a complete discussion of this subject.

Bootstrapping Mule from your own code is easy, as illustrated in the following code snippet, which loads a Spring XML configuration named my-config.xml:

```
DefaultMuleContextFactory muleContextFactory = new
    DefaultMuleContextFactory();
SpringXmlConfigurationBuilder configBuilder = new
    SpringXmlConfigurationBuilder("my-config.xml");
MuleContext muleContext =
    muleContextFactory.createMuleContext(configBuilder);
muleContext.start();
```

It’s important to keep the reference to the `MuleContext` object for the lifetime of your application, because you’ll need it in order to perform a clean shutdown of Mule, as illustrated here:

```
muleContext.dispose();
```

The `MuleContext` also allows you to instantiate a client to interact with the Mule instance from your application. The following code shows how to create the client out of a particular context:

```
MuleClient muleClient = new MuleClient(muleContext);
```

Note that the Mule client supports other construction parameters that we’ll discuss in chapter 13.

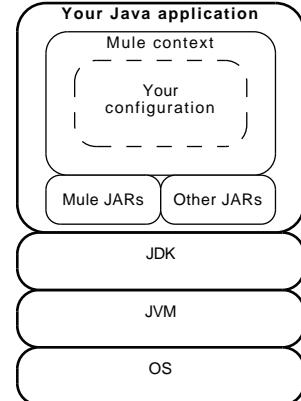


Figure 7.5 Conceptual deployment model of Mule embedded in a standard Java application

³ The standalone server deployment we talked about in the previous section is the way to go for running Mule as a service.

⁴ Ant can also pull dependencies from Maven repositories thanks to a specific task. We still strongly encourage you to use Maven, if you don’t already.

TIP *Mule über Spring?* The example in this section shows how Mule leads the dance relative to Spring: Mule takes the responsibility to load and bootstrap the bean factory where your beans will be managed. But what if you have an existing Spring application with its own context (or hierarchy) and you want Mule to be running within this environment and have access to its beans? In this case, you'll have to pass your existing Spring context to the Mule configuration builder so it can use Spring as its parent. This will allow Mule to use beans managed in a parent context as its service components:

```
ConfigurationBuilder builder = new
    SpringXmlConfigurationBuilder("my-config.xml");
builder.setParentContext(parentContext);
builder.configure(muleContext);
```

In the Tip, note how we used a configuration builder specific to Spring XML configurations. In chapter 2, we mentioned the possibility of using other builders, such as the scripting ones. Here's how you'd create a builder for a Groovy script-based configuration file:

```
ScriptConfigurationBuilder configBuilder = new
    ScriptConfigurationBuilder("groovy", "my-config.groovy");
```

Table 7.3 summarizes the pros and cons of embedding Mule in a standard Java application.

Table 7.3 Pros and cons of embedding Mule in a standard Java application

| Pros | Cons |
|---|--|
| Flexibility to deploy just what's needed | Have to deploy what's needed |
| Well suited for “Mule as an integration framework” approach | Have to manage Mule's lifecycle (start/stop) on your own |
| Perfect for a J2SE application, such as a Swing or Spring rich client GUI | |

You now know how to make your standalone Java applications leverage Mule to communicate and integrate with other applications. Let's now look at what you can do for your web applications.

7.1.4 **Embedded in a web application**

For the same reasons we evoked at the beginning of the previous section, you might be interested in embedding Mule in your web application. Mule provides all you need to hook it to your favorite servlet container. Why would this be desirable when it's possible to use the capable Jetty transport from a standalone Mule server? The main reason is familiarity. It's more than likely that your support team is knowledgeable about a particular Java web container. Deploying Mule in such a well-known application

environment context gives you the immediate support of operations for installing, managing, and monitoring your instance.

When Mule is embedded in a web application, as shown in figure 7.6, you need to make sure the necessary libraries are packaged in your WAR file. As we said in the previous section, if you use Maven, this packaging will be automatically done, including the transitive dependencies of the different Mule transports or modules you could use.

The Mule instance embedded in your web application is bootstrapped by using a specific `ServletContextListener`. This conveniently ties Mule's lifecycle with your web application's lifecycle (which itself is bound to the web container's lifecycle). The following demonstrates the entries you'd need to add to your application's `web.xml` to bootstrap Mule:

```
<context-param>
    <param-name>org.mule.config</param-name>
    <param-value>my-config.xml</param-value>
</context-param>
<listener>
    <listener-class>org.mule.config.builders.MuleXmlBuilderContextListener
        </listener-class>
</listener>
```

Needless to say, this context listener also takes care of shutting Mule down properly when the web application gets stopped. Note how the configuration of the Mule context listener is done by using context-wide initialization parameters.

Starting Mule is a great first step, but it's not sufficient: you need to be able to interact with it. From within your own web application, this can be achieved by using the Mule client. Since the listener took care of starting Mule, you have no reference to the context, unlike when you bootstrap Mule yourself. This is why the Mule client is instantiated with no context parameter, relying on the lookup it'll perform to locate a static instance of the context created by the listener:

```
MuleClient muleClient = new MuleClient();
```

There's another great benefit of this deployment model, which is the capacity to tap the servlet container directly for HTTP inbound endpoints. In a standalone server or standard Java application deployment scenario, your inbound HTTP endpoints actually rely on either the stock HTTP transport or the Jetty transport. But this need not be the case when you deploy in a web container. This container already has its socket management, thread pools, tuning, and monitoring facilities: Mule can leverage these if you use the servlet transport for your inbound endpoint.

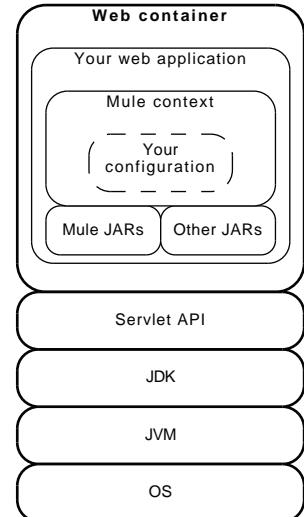


Figure 7.6 Conceptual deployment model of Mule embedded in a web application

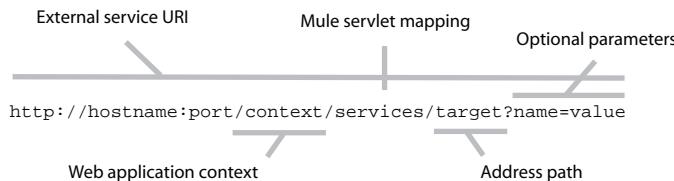
The servlet transport only works in coordination with an actual servlet configured in your web.xml: this servlet takes care of receiving the messages and sending them to the inbound endpoint that can accept it. Here's an example of how to configure this servlet:

```
<servlet>
  <servlet-name>muleServlet</servlet-name>
  <servlet-class>org.mule.transport.servlet.MuleReceiverServlet
    </servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>muleServlet</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

With this configuration in place, the following inbound endpoint can be used:

```
<servlet:inbound-endpoint address="servlet:arg1">
```

A Mule service using such a configuration will be accessible to the outside world with a URI following this pattern:



The actual payload of the message that'll be received in the service will depend on the HTTP method used by the caller.

NOTE Mule ships with another servlet that's more oriented toward REST-style services. Named `org.mule.transport.servlet.MuleRESTReceiverServlet`, this servlet can be used in lieu of or conjointly with the standard receiver servlet. Refer to the Mule servlet transport and the Mule REST-pack for more information on this.

Table 7.4 shows a list of pros and cons of the web application embedded deployment option of Mule.

Table 7.4 Pros and cons of embedding Mule in a web application

| Pros | Cons |
|--|--|
| Can tap your well-known servlet container | Must manage Mule's libraries yourself |
| Benefits from the web application lifecycle events | No direct support for scripted configurations |
| Familiar deployment platform for operations | Possible pesky class loading issues on some web containers |

Embedding Mule in a web application is a popular and powerful way to deploy Mule. Should you need tighter integration with the Java EE stack, especially EJBs, the last deployment strategy that we're going to look at is for you.

7.1.5 Embedded as a JCA resource

The Java EE Connector Architecture⁵ standardizes the contracts between a particular enterprise information system on the one hand, and a Java EE application server and the applications it hosts on the other hand. These contracts cover many aspects, such as connection, transaction, and security.

When you deploy Mule as a JCA resource, as depicted in figure 7.7, your other Java EE components will possibly have to gain access to it. They'll become able to use Mule in two different styles of scenarios: as synchronous clients or as asynchronous listeners.

Connecting to Mule as a synchronous client via its JCA resource adapter is somewhat similar to using the Mule client, as we've seen in the previous deployment options. To do so, you need to either look up Mule's connection factory in the JNDI tree or have it injected as a resource in your EJBs. Once you get hold of the connection factory, you can then communicate with the services defined in your Mule configuration, but also leverage the different transports to communicate directly with the outside world. Here's how you'd send a message to a service listening to a particular VM endpoint and get its response synchronously:

```
MuleConnection connection = muleConnectionFactory.createConnection();
MuleMessage response = connection.send("vm://MyService.In", payload,
    properties);
connection.close();
```

Note that we've spared you the necessary `try/catch/finally` construct. With such a connection reference, you can also synchronously read from a particular endpoint URI, as shown here:

```
MuleMessage response = connection.request("vm://MyService.In", timeOut);
```

Connecting to Mule as an asynchronous listener allows you to consume messages flowing out of Mule with Message-Driven Beans (MDBs) the same way you do with JMS destinations. This is done by using a particular messaging type and activation in the

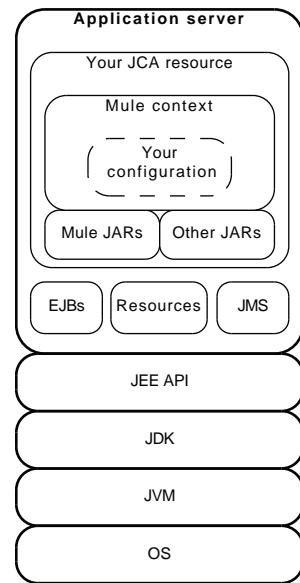


Figure 7.7 Conceptual deployment model of Mule deployed as a JCA resource

⁵ Improperly but almost universally abbreviated JCA, which was supposed to be the acronym of the Java Cryptography Architecture.

configuration of your MDB. To give you an idea, here's the standard example from Mule, where an MDB is configured to consume messages from a TCP socket:

```
<message-driven>
    <description>An MDB listening on a Tcp socket</description>
    <display-name>TcpReceiverMDB</display-name>
    <ejb-name>TcpReceiverMDB</ejb-name>
    <ejb-class>org.mule.samples.ejb.SimpleReceiverMessageBean</ejb-class>
    <messaging-type>org.mule.api.lifecycle.Callable</messaging-type>
    <transaction-type>Container</transaction-type>
    <activation-config>
        <activation-config-property>
            <activation-config-property-name>endpoint
                </activation-config-property-name>
            <activation-config-property-value>tcp://localhost:12345
                </activation-config-property-value>
        </activation-config-property>
    </activation-config>
</message-driven>
```

We won't dig further into the gory details of this deployment mechanism, as they're Java EE container specific: refer to the official Mule documentation to learn more about this approach. The pros and cons of deploying Mule as a JCA resource are listed in table 7.5.

Table 7.5 Pros and cons of deploying Mule as a JCA resource

| Pros | Cons |
|--|---|
| Java EE standard inbound and outbound access to Mule | Need in-depth understanding of the JCA implementation of your Java EE container |
| RAR template complete with all needed dependencies | Possibly hairy to debug in case of trouble |
| In-memory access to local EJBs and JMS destinations | Introduces a level of complexity that's seldom worth it |

We're done with our tour of the different deployment strategies for your Mule-powered applications. From the simple and sturdy standalone server to the advanced JCA resource adapter, you've discovered a palette of options that'll surely fit your needs.

Now that you understand how to deploy a Mule instance, your next question should be where to deploy it. Again, Mule is extremely flexible and supports many deployment topologies. Let's explore your options in this domain.

7.2 Deployment topologies

Because of the dual nature of Mule (framework and broker), its deployment topology is twofold: the instance-level topology and the network-level topology. The former is controlled by the configuration file you learned to write in part 1 of this book. The latter is defined by the number of Mule instances deployed, their locations, and the

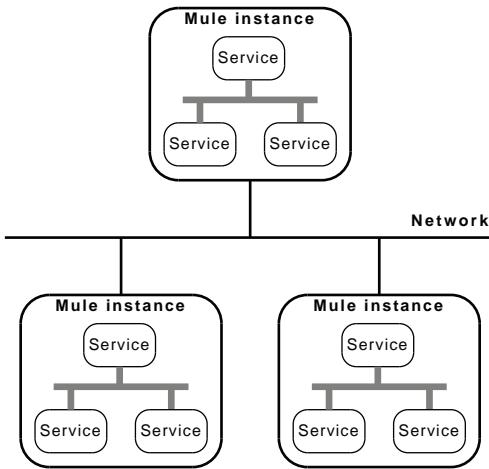


Figure 7.8 Instance-level and network-level topologies of a Mule deployment

transports they use. Moreover, they mutually influence each other to some extent. Figure 7.8 illustrates these two levels of deployment topologies.

There's a rich terminology for both instance-level and network-level topologies. The former, mostly coming from the Enterprise Integration Patterns corpus,⁶ contains terms such as *splitter*, *aggregator*, or *resequencer*. The latter talks about *ESB*, *ESN*, *pipeline*, *peer network*, *client/server*, or *hub-and-spoke*. There's enough here to make you feel dizzy, and certainly too much for us to try to present an exhaustive catalog. This would be beyond the scope of this book, and is covered in numerous authoritative books already.

To approach this subject in a pragmatic way, we'll consider some factors that can influence both instance-level and network-level topologies and detail a few related use cases. Consider the following a grab bag of concepts that you can reuse as you see fit. Each of the coming sections will be dedicated to a particular factor and what impact it has on your deployment topology. We'll start by looking at the impact of functional needs, then network concerns, high-availability expectations, and finally, fault tolerance requirements.

NOTE Security requirements can orient your deployment to particular topologies. Refer to chapter 9 for more on this subject.

Let's start with functional needs.

7.2.1 Satisfying functional needs

It's natural that the primary factors that influence your deployment topology are your functional needs. Suppose that you simply want to use Mule to expose some business logic as remotely accessible services. You'll end up with a *client/server* topology that

⁶ See section 1.3.

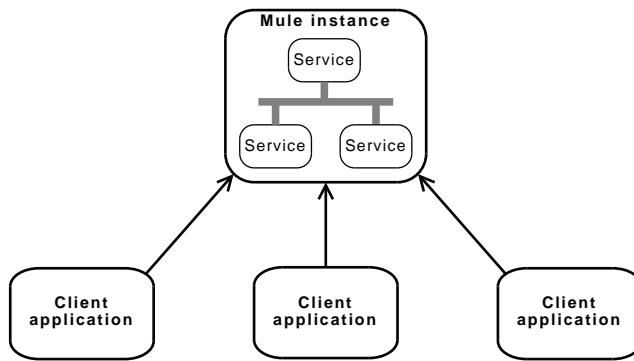


Figure 7.9 Mule can be leveraged to remotely expose business logic in a client/server topology.

pretty much looks like what's shown in figure 7.9. As the figure suggests, the services can themselves be engaged in ininstance communications: for example, they can be composed together using the bound interface mechanism we discussed in chapter 6.

Let's consider another example. Suppose you have a couple of applications that need to poll data regularly from different resources, say file systems and databases. You don't want to place such a burden on any of these applications, and have wisely commissioned Mule to take care of this work. Figure 7.10 shows that the topology you'll lean toward will be conceptually similar to a *hub-and-spoke* topology. The different services in such a Mule instance would not only perform polling but also all sorts of transformation, aggregation, or enrichment needed before sending the data to the target applications.

The two simple examples we've just looked at illustrate that functional constraints are the most obvious ones to take into account when defining your deployment topology.

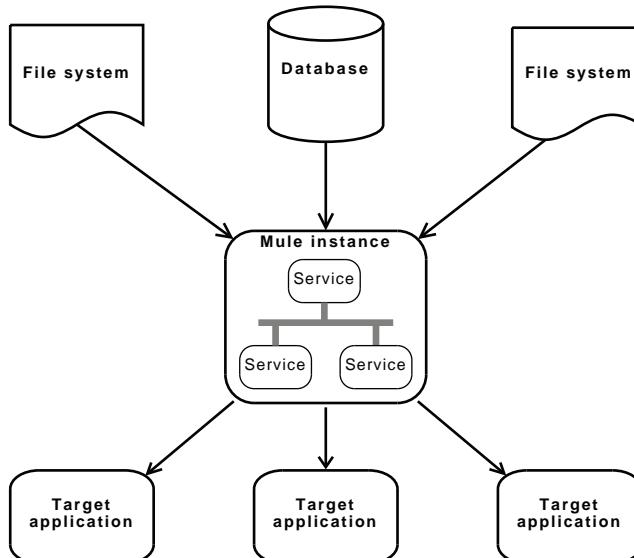


Figure 7.10 A Mule instance dedicated to polling data engages in a hub-and-spoke topology.

Everybody knows that things tend to work well “on their machine,” but start to show some attitude whenever they get deployed in the wild. Often, the network gets in the way, as we’ll see in the next section.

7.2.2 Dealing with the network

With cloud computing becoming mainstream, Sun’s singular vision “The Network Is the Computer” has turned out to be prophetic. This omnipresence of the network is what makes Cloud, Inc.’s business model possible. But life isn’t as rosy as it is in grand visions. Though omnipresent, fast, and reliable, the network is still fiendish sometimes. It’s all too easy to forget about the distance between services and the almost infinite time⁷ it takes to reach them. When this happens, the network will cruelly remind you that it exists and it can’t be factored out of your topology equation.

Let’s consider the case of an enterprise-wide network that spans several remote locations. Certain services exist in all the locations: it’s in the best interest of each location to always call the nearest service when several are available. Building an *ESN* (enterprise service network) topology, as illustrated in figure 7.11, can help achieve this goal. In this topology, each Mule instance has enough smarts to figure out what’s the most appropriate service to call in each site.

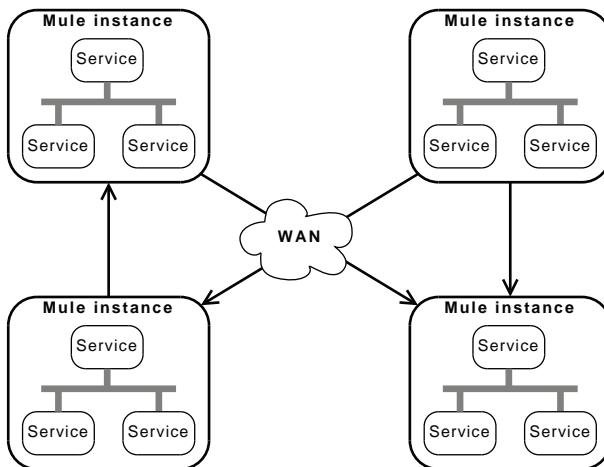


Figure 7.11 In an *ESN* deployment topology, Mule instances communicate directly as needed.

MULE-BASED PROXIES

If only for sending and receiving emails, companies have to communicate out of the comfort zone of their WANs and reach the outside world. Integrating remote systems is both an exhilarating and a daunting task. Even ignoring the security concerns, allowing too much knowledge of the external systems to spread out inside corporate applications can turn into a management nightmare. What if the remote address of the service changes? What if all communications with this service must now be

⁷ From the CPU’s standpoint.

audited? Using Mule as a proxy for remote services, as depicted in figure 7.12, is an elegant solution to this problem.

As you can see, this topology is just a variation of the client/server topology from figure 7.12: in this variation, Mule services act as a middleman for the client applications, taking care of invoking the remote service for them. All the knowledge of the remote service is concentrated in a single place, the Mule instance, which acts as a proxy. This knowledge consists not only of connection details, but can also cover security configuration or specific data transformation (for example, to transform data representations specific to the remote service into canonical ones in use in the company).

WARNING *Proxy frenzy* Using Mule as a proxy for all your service invocations, even internal ones, can be a tempting topology whose elegance shines in nifty deployment diagrams. After all, what could be wrong with centralizing all the service invocation knowledge in a central place? As Dr. Jim Webbers puts it,⁸ this can in fact merely consist of sweeping the mess under the carpet. Our advice is simple: proxy services if and only if there's value added in the indirection. You can achieve a lot in terms of indirection with simple tools such as DNS and load balancers. "Mule in the middle" will make sense only if there's a genuine gain in centralizing features such as protocol adaptation, transformation, auditing, or security.

As soon as you start using protocols that aren't built on top of HTTP or that use dynamic port allocations, you can get ready to hear about our best friend in the enterprise network: the firewall. Having a firewall stuck in the middle of your deployment

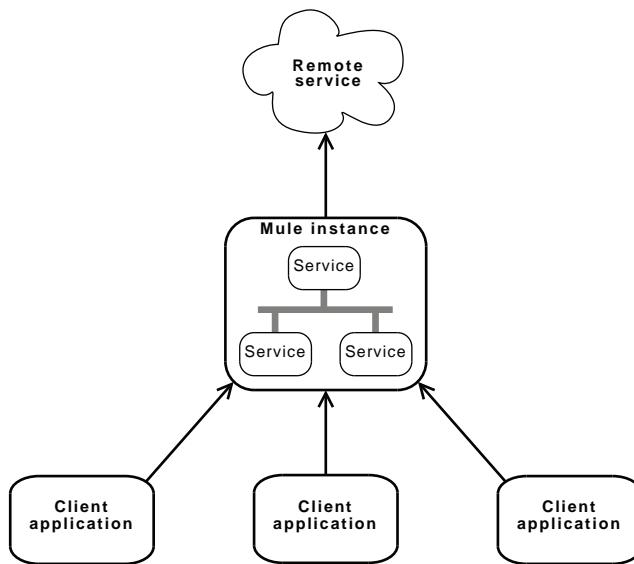


Figure 7.12 Mule can act as a proxy for remote services.

⁸ Discussions on "Guerrilla SOA": <http://bit.ly/g-soa>.

topology can force you to adopt a different protocol for sending and receiving messages. By using a localized *hub-and-spoke* topology at the edges of your networks, you can achieve protocol conversion without having to make other services or Mule instances aware of it. This approach is illustrated in figure 7.13.

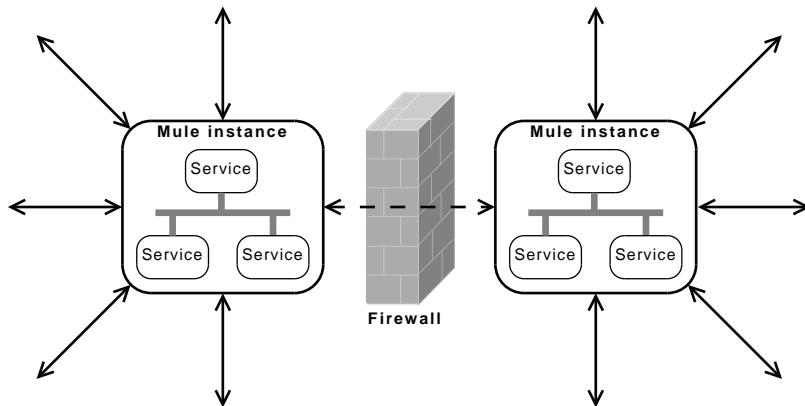


Figure 7.13 A hub-and-spoke topology used to bypass a network complication

CANONICAL ESB

If you're lucky enough to have access to a corporate-wide JMS provider, then you might want to consider the canonical *ESB* deployment topology. As illustrated in figure 7.14, this topology abstracts out the network by using JMS destinations as communication channels between Mule instances and applications. This approach frees each participant in the topology from pesky network details such as knowing host names or dealing with unreliable protocols. Provided they can connect to the common JMS provider, applications and Mule instances in an ESB topology will benefit from JMS characteristics such as guaranteed delivery, and local and distributed transaction support.

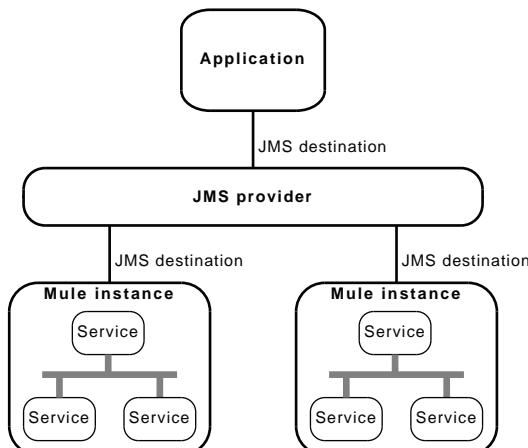


Figure 7.14 The canonical ESB deployment topology of Mule instances relies on a central JMS provider.

TIPS *For a good QoS* Using JMS as a messaging backbone in your deployment topology not only abstracts out the network but also enriches its semantics. JMS brings you quality of service (QoS) features that you can base your routing logic on. For example, you can decide to opt for a completely different routing logic if a message has its `Priority` property set above a certain level. As you can see, deciding for a canonical ESB topology at network level opens new possibilities in your instance-level topologies.

As you've seen through these few examples, the network will impose its own constraints on your deployment topology, whether you choose to abstract it out or decide to live with it.

The next topology influencing factor we'll consider is the need to run highly available Mule instances.

7.2.3 **Designing for high availability**

Being able to ensure business continuity is one of the main goals of any IT department. Your Mule-driven projects won't escape this rule. Depending on the criticality of the messages that'll flow through your Mule instances, you'll probably have to design your topology so it offers a high availability of service. High availability is generally attained with *redundancy* and *indirection*. Redundancy implies several Mule instances running at the same time. Indirection implies no direct calls between client applications and these Mule instances.

An interesting side effect of redundancy and indirection is that they allow you to take Mule instances down at any time with no negative impact on the overall availability of your ESB infrastructure. This allows you to perform maintenance operations, such as deploying a new configuration file, without any down time. In this scenario, each Mule instance behind the indirection layer is taken down and put back up successively.

BEST PRACTICE Consider redundancy and indirection whenever you need "hot deployments."

Using a *network load balancer* in front of a pool of similar Mule instances is probably the easiest way to achieve high availability. Obviously, this is only an option if the protocol used to reach the Mule instances can be load balanced (for example, HTTP). If you remember the client/server example we discussed in section 7.2.1, you'll recognize the example in figure 7.15. This is the same topology but with a network load balancer added. With this in place, one Mule instance can be taken down, for example for an upgrade, and the client applications will still be able to send messages to an active instance. As the name suggests, using a load balancer would also allow you to handle increases in load gracefully: it'll always be possible to add a new Mule instance to the pool and have it handle part of the load.

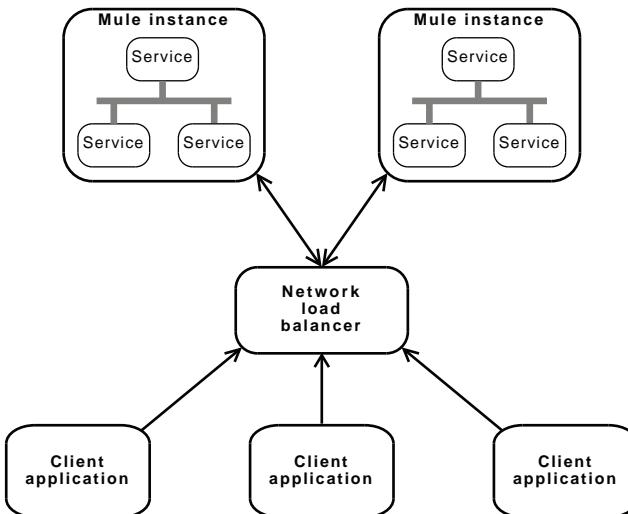


Figure 7.15 A network load balancer provides high availability to Mule instances.

Another type of indirection layer you can use is a JMS queue concurrently consumed by different Mule instances. No client application will ever talk directly to any Mule instance: all the communications will happen through said queue. Only one Mule instance will pick up a message that's been published in the queue. If one instance goes down, the other will take care of picking up messages. Moreover, if messages aren't processed fast enough, you can easily throw in an extra Mule instance to pick up part of the load. Of course, this implies that you're running a highly available JMS provider so it'll always be up and available for client applications. The canonical ESB topology, represented in figure 7.14, can therefore be easily evolved into a highly available one.

If your Mule instances don't contain any kind of session state, then it doesn't matter where the load balancer dispatches a particular request, as all your Mule instances are equal as far as incoming requests are concerned. But, on the other hand, if your Mule instance carries any sort of state that's necessary to process messages correctly, load balancing won't be enough in your topology and you'll need a way to share session state between your Mule instances.⁹ This is usually achieved either with a shared database or with clustering software, depending on what needs to be shared and on performance constraints.

TIP *State of the Mule* Session state can exist in different parts of Mule. If this is the case in your configuration, you'll have to cluster them if you intend to use a load balancer. Here are a few:

- Idempotent receivers, which must store the identifier of the messages they've already received. Behind the scenes, the idempotent receiver uses

⁹ One could argue that with source IP stickiness, a load balancer will make a client "stick" to a particular Mule instance. This is true but wouldn't guarantee a graceful failover in case of crash.

an internal API (`org.mule.api.store.ObjectStore`) to handle its persistence. You can use the provided implementations (file system or in-memory) and cluster them accordingly (shared file system or in-memory cluster). Or you can roll out your own implementation, for example a database-backed one.

- Message aggregators, which use in-memory collections to keep track of the message they're currently storing until the accumulation is done and they can be released. Currently, the most direct way to cluster aggregators is via JVM-level clustering.
- Your own components, though we discouraged you from doing so in chapter 6. At this point it should be clearer why we encouraged you to strive for statelessness when designing your components. If you need to make them stateful, it's up to you to cluster them in order to make them highly available.

Note that at the time of this writing, there's no officially supported clustering mechanism for Mule.

Let's now revisit our previous example and consider that our Mule instances need to share some state. As shown in figure 7.16, we've now added a piece of clustering software to take care of sharing the state across the instances in order to make them able to process any message at any time.

At this point, you should have a good understanding of what's involved when designing a topology for highly available Mule instances. This will allow you to ensure continuity of service in case of unexpected events or planned maintenance.

But it's possible that, for your business, this is still not enough. If you deal with sensitive data, you have to design your topology for fault tolerance, too.

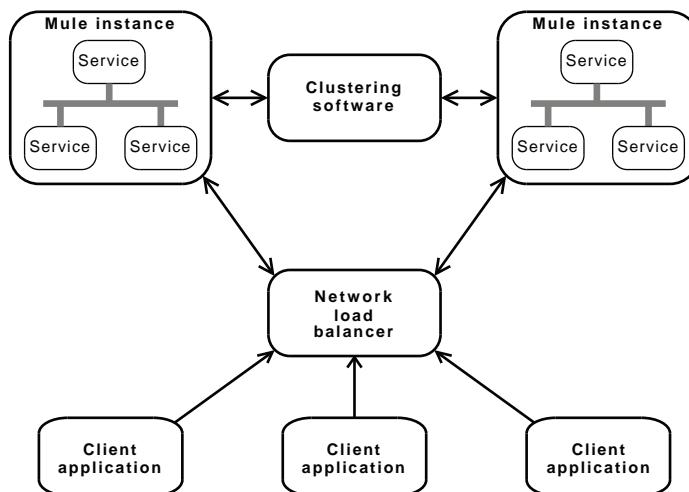


Figure 7.16 Clustering software is required to enable load balancing of stateful Mule instances.

7.2.4 Shooting for fault tolerance

If you have to ensure that, whatever happens during the processing of a request, no message gets lost at any time, you have to factor fault tolerance into your topology design. In traditional systems, fault tolerance is generally attained by using database transactions, either local or distributed (XA). In the happy world of Mule, since the vast majority of transports don't support the notion of transaction, you'll have to carefully craft both your instance-level and network-level topologies to become truly fault-tolerant.

Mule offers a simple way to gain a good level of fault tolerance via its persisted VM queues. When persistence is activated for the VM transport, messages get stored on the file system when they move between the different services of a Mule instance, as shown in figure 7.17. In case of crash, these stored messages will be processed upon restart of the instance. Note that, because it supports XA, the VM transport can be used in combination with other XA-compatible transports in order to guarantee a transactional consumption of messages.

The main drawback of VM persisted queues¹⁰ is that you need to restart a dead instance in order to resume processing. This can conflict with high availability requirements you might have. If this is the case, the best option is to rely on an external highly available JMS provider and use dedicated queues for all intrainstance communications. This is illustrated in figure 7.18.

So far, we've achieved fault tolerance within a Mule instance and at its boundaries if the protocol used supports XA. If all transports supported distributed transactions, if networks were highly reliable and latency didn't exist, life would be peachy. In this perfect world, we could enroll many systems in huge chains of distributed transactions, which would guarantee us the utmost level of fault tolerance possible. Unfortunately, we don't live in this world. Distributed transaction is a costly and limited mechanism that'll likely be minimally used in most of your integration projects.

So what's the solution? The widely adopted approach is to eventually reach correctness in regard to fault tolerance by building topologies that favor conversation

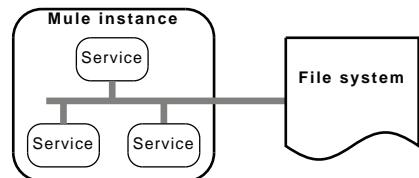


Figure 7.17 Simple file system-based persisted VM queues are standard with Mule.

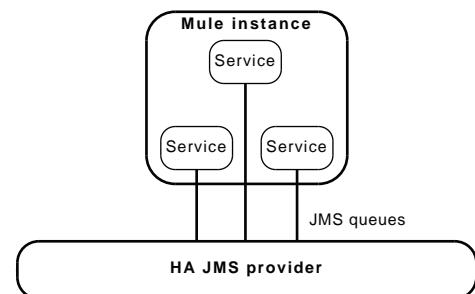


Figure 7.18 An HA JMS provider can host queues for all communications within a Mule instance.

¹⁰ Despite forcing you to move around serializable payloads only so they can be saved on disk.

and compensation over transactionality. This means that your network-level topology and the protocol it uses will influence your instance-level topology, should you need to be fully fault tolerant.

How does this translate in Mule? Let's suppose that you have a Mule instance that uses fully persisted ininstance communication channels. Suppose that, at one point in the processing of a message, this instance must call a remote service over HTTP and then process its response internally. As soon as you call the HTTP service, you'll be outside of a transactional behavior. What would happen if you never get its response? You can potentially lose the original message.

A possible solution here is to store a little conversational state in an aggregator and wait for a response from the HTTP service. If the response never comes, the aggregation will time-out and you'll be able to store the original message, which was waiting in the aggregator, in a safe place. This is illustrated in figure 7.19. The kinds of listeners you can use are exception handlers and routing notification listeners, which will be respectively discussed in chapters 8 and 13.

You've seen that fault tolerance can be achieved in different ways with Mule, depending on the criticality of the data you handle and the availability of transactions for the transports you use.

We're done with our exploration of Mule deployment topologies. If there's one concept you need to take away from this rather disparate section, it's that there's no prescriptive way to deploy Mule in your IT landscape. Mule is versatile enough to be deployed in a topology that best fits your needs. We've given you a few patterns and practices for dealing with this matter but, once again, Mule will go where you need it to.

As you've seen, some topologies rely on multiple deployments of Mule instances, leading to the necessity to housekeep the deployments themselves. This is a subject that we'll discuss in the next section.

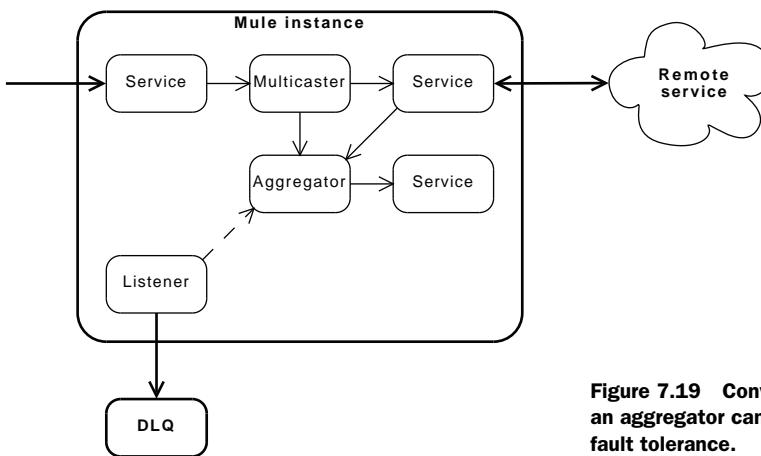


Figure 7.19 Conversational state in an aggregator can increase Mule's fault tolerance.

7.3 Deployment management

Whatever deployment strategy (or strategies) you opt for, your Mule applications will undergo the same deployment management difficulties as any of your other middleware applications. In this section, we'll review your options for how to tackle these difficulties by using standard development tools and by leveraging the Galaxy governance platform. But before doing so, let's start by detailing some common deployment challenges:

- *Multiple environments*—A standard corporate landscape is characterized by a multitude of environments, among which are often found development, integration, load test, QA, and production. The same rules that apply for transiting standard applications from one environment to another will apply to your Mule projects.
- *Multiple locations*—A production environment can span multiple physical locations. Two Mule instances providing the same services in each location can have completely different internal topologies, as some resources may not be uniformly accessible across production sites. For example, in one site, Mule may act as a service provider, while it will act as a proxy in another site.
- *Multiple roles*—Some Mule instances will play a supporting role (exposing public services), while others will play an enabling role (routing messages to services or enterprise resources). In all but the simplest deployment topologies, you'll need to coordinate the deployment of your Mule instances. Supporting instances will generally have to go live or be updated before other applications can start to use them. In contrast, enabling instances will be touched only after the systems they target have been readied.
- *Multiple versions*—Your Mule projects will evolve as time passes, the same way standard development projects evolve.
- *Multiple lifecycles*—A particular project often contains components that evolve at a different pace and have different lifecycles. In a Mule project, internal services evolve faster and more discretionally than publicly exposed services.

There are some more Mule-specific additional concerns that'll compound these difficulties:

- *Configurations have unobvious dependencies on libraries*—If you remember the discussion in chapter 2 about schema location and missing JARs, you should realize that something as benign as adding a new transport to your XML file might call for a few extra JARs to be added to the classpath.
- *Different deployment strategies implies different deployment management challenges*—For example, a standalone server deployment has all the Mule JARs installed by default, but of course, none of the business JARs. Conversely, an embedded deployment will contain your custom code but might miss necessary dependencies.

- *Different deployment topologies implies different configuration management constraints*—In a topology where Mule exposes services, such as a client/server or proxy service, maintaining backward compatibility between different configuration versions is essential. This isn't an issue if your Mule instances are deployed to only poll different external sources.

As we just said, public services imply commitment. Like any public API, a service exposed to the public becomes a commitment in terms of availability and graceful evolution. Strict production environments will impose SLAs¹¹ to your published Mule services. At your project level, these SLAs will translate to uptime requirements and backward compatibility constraints. Moreover, you'll have to devise a thorough testing strategy (discussed in chapter 12) and a clear policy for service evolution (transition times, end of life, and so on).

So how do you ensure that the right things get deployed at the right place and the right time? If you've made the choice to use NetBoot as your deployment strategy (see section 7.1.2), then the wisest option is to fully leverage Galaxy as your company-wide governance platform. If you've opted for another deployment strategy, Galaxy is still an option because it can still take good care of your configuration files.

But if you don't envision using Galaxy, you'll then have to leverage your standard development tools. Let's look first at this approach, before boldly going into Galaxy.

7.3.1 **Using development tools**

If you decide to use your standard development tools to manage the different Mule artifacts you need to deploy, the nature of these artifacts will mainly depend on the deployment strategy you've selected. With the exception of NetBoot, which implies using Galaxy, there are mainly two cases:

- *Standalone deployment*—You'll need to deploy your configuration files and the custom libraries they rely on. If you decide to deploy your project as an independent directory structure, as we discussed in section 7.1.1, a good strategy is to leverage the Maven Assembly Plugin to build a complete deployable project.¹² If instead you opt for dropping JARs in the lib/user directory, your best bet will consist of dropping your configuration files there too, as they'll end up in the classpath and will be easily picked up by Mule (while remaining editable, for example to have the operations team easily configure passwords).
- *Embedded deployment*—In this case, the deployment unit will be the host application. For example, for a web application, you'll typically create a WAR file that'll contain all your custom code and the Mule libraries and configuration files you need. You'll have to take care that all the necessary dependencies are packaged

¹¹ Service-level agreements, which usually “specify the levels of availability, serviceability, performance, operation” [Wikipedia].

¹² This is demonstrated in the /chapter07/publication-assembly example of the accompanying source code.

within the deployment unit. Again, using Maven for building your deployable would be of great help.

NOTE A mitigating factor for the need to add extra libraries to the classpath is to use scripting for custom components or transformers. Naturally, if you have your business logic already available as libraries, don't go rewriting it as scripts! We'll discuss scripting goodness in chapter 14.

Before going any further, let's make something clear: there's nothing special about a Mule project. Why do we say that? Because some may be tempted to consider that a Mule project escapes the rules and standards that apply to standard software engineering projects. After all, it's just a matter of tweaking a few XML files... Hacking the examples shipped with Mule is acceptable while learning the tool, but should stop as soon as you kick-start a real project on the platform.

All the sound practices of software engineering will apply to your Mule projects too:

- *Source control management*—All your Mule configuration files and custom code (components, transformers, and so forth) should be properly stored in an SCM.
- *Repeatable builds*—You should be able to produce a deployable of any version of your project at any time. We've mentioned Maven in both the deployment options listed previously for a reason: this tool is best fitted for systematically producing your Mule projects' artifacts (see section 12.1 for more).
- *Versioned artifacts*—Whether you target a standalone or an embedded deployment, your goal will be to produce a uniquely versioned deployable that can be handed off to your QA and production teams.
- *Configurable artifacts*—Externalize any parameter that others will have to adapt in the environment they administer (see section 2.2.2).
- *Testing*—All your custom code and configurations should be tested at all levels, which includes unit testing, integration testing, regression testing, and load testing (see section 12.3).
- *Issue/task tracking*—Issues happen, and so do feature requests: there's no reason for not tracking them with your Mule projects the same way you do with other projects. Doing so is essential when you have to coordinate your Mule projects with other ones.
- *Documentation*—At this point, some of you may cringe, but Mule projects deserve good documentation too. The bare minimum consists in operational procedures (deployment, support). An interesting addition to consider, depending on the SOA practices of your company, is a humane registry (see <http://www.martinfowler.com/bliki/HumaneRegistry.html>). Maven can be of great help here, as it can generate a documentation site at the same time it builds your artifacts.

As you can see, it's possible to leverage your existing development tools and practices to put in place a sensible management of your Mule deployables.

BEST PRACTICE Make the versions of your deployed Mule artifacts easily discoverable at runtime (via JMX, with a specific HTTP service).

But, because Mule instances will often host production-critical services in several distributed locations, you might feel the need for a more controlled approach to deployments. Enter Galaxy, the governance platform for Mule.

7.3.2 Hitchhiking Galaxy

In section 7.1.2, we introduced the NetBoot deployment strategy and mentioned how it relies on another product named Mule Galaxy. Is Galaxy just a repository of libraries? Far from it! Galaxy is a complete governance platform that allows you to fully manage all aspects of your Mule deployments. Let's detail what this means concretely:

- *Galaxy is an artifact repository*—You knew this already. In Galaxy, artifacts are stored in workspaces that are themselves organized as a hierarchy. Artifacts can be versioned and support lifecycle phases that represent their maturity state (such as development, test, production). Galaxy exposes its content over the Atom publishing protocol, which makes it reachable from any client application you want to use. It's also possible to select artifacts by using a specific query language: the Galaxy Query Language. As an example, figure 7.20 shows the content of the MuleInAction workspace: note the version number and lifecycle phase of the artifact named mia-nb-mule-config.xml.
- *Galaxy is a registry*—If you look again at figure 7.20 you'll notice that Galaxy has recognized that the artifact named mia-nb-mule-config.xml is a Mule XML configuration file. It has analyzed and “understood” its content, as the number of services, global endpoints, and models show. This capacity makes Galaxy a full-fledged registry because it can deduct dependencies between configuration files (including WSDL and schemas used by Mule configurations), hence helping you manage them efficiently.
- *Galaxy is a governance platform*—Because it understands what it stores, Galaxy can enforce custom policies to ensure that diverse constraints are respected. For example, you can enforce backward compatibility of all WSDL files, efficiently solving the configuration compatibility problem that we discussed in the introduction of this section. Galaxy also contains an integrated audit trail system that allows you to track all repository operations.

At startup time, the first thing Mule will look for is its configuration. So far, we've only used configuration files that were located somewhere in the classpath. Galaxy offers a new possibility: you can retrieve a configuration file directly from the registry by using a specific configuration builder named `org.mule.galaxy.mule2.config.GalaxyConfigurationBuilder`. This builder accepts a URI that points to the desired

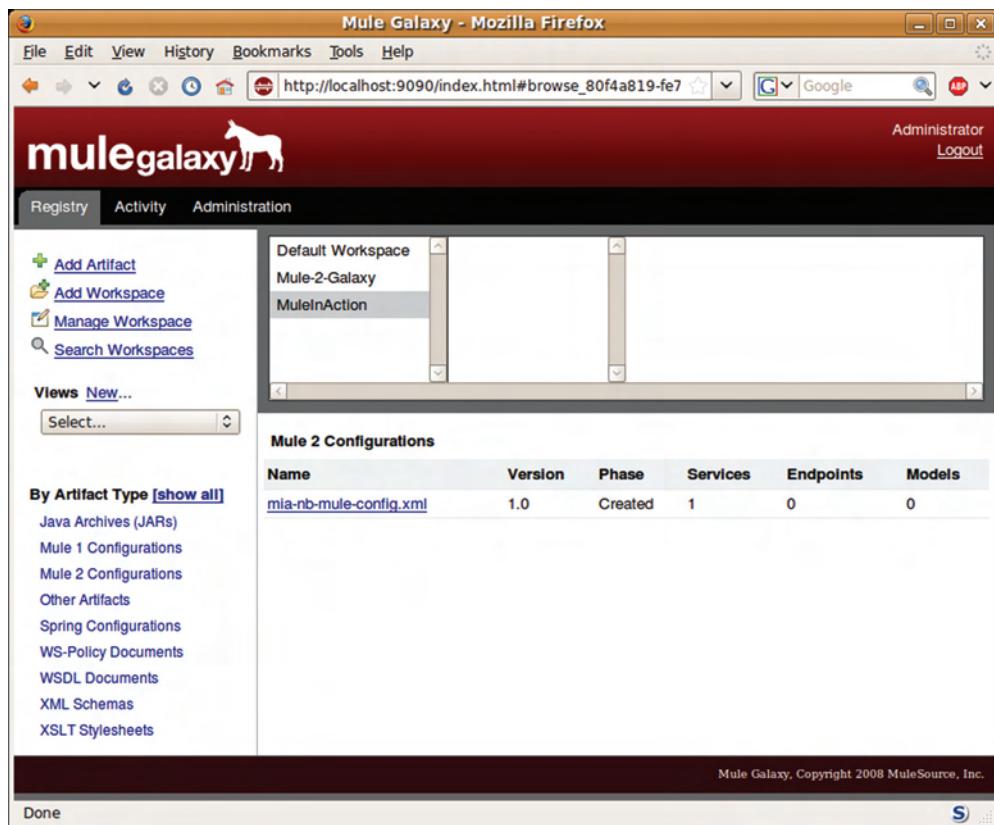


Figure 7.20 Browsing the MuleInAction workspace in a Mule Galaxy registry

configuration file and takes care of retrieving it from Galaxy's Atom API. This URI can contain an artifact query expressed in Galaxy Query Language. The Galaxy-ConfigurationBuilder can be used with any deployment strategy: for example, your Mule instance embedded in a web application can leverage it to fetch its configuration from a central Galaxy repository.

Let's look at how these two configuration fetching approaches (classpath and query) can be used with NetBoot and Galaxy:

- With the classpath approach, you need to package the configuration files in a JAR and let NetBoot download the workspace that contains this JAR. The following startup command assumes that the MuleInAction workspace contains a JAR where the configuration file mia-nb-mule-config.xml has been packaged:

```
./mule -config mia-nb-mule-config.xml \
-M-Dgalaxy.port=9090 \
-M-Dgalaxy.netboot.workspace=Mule-2-Galaxy \
-M-Dgalaxy.app.workspaces=MuleInAction
```

If you remember the discussion about NetBoot’s cache in section 7.1.2, we mentioned that all JARs downloaded locally are added to Mule’s classpath. This is how packaged configuration files become accessible to Mule’s standard configuration builders.

NOTE The Maven Publishing Plugin is of great help if you follow the classpath approach. It can take care of packaging your different artifacts in a JAR and uploading it directly into Galaxy’s repository. It also uploads the direct and transient dependencies of your project into Galaxy. It can even create a workspace for you if it doesn’t already exist.

- With the direct Galaxy query approach, you need to use the GalaxyConfigurationBuilder to query the registry. The following is a NetBoot startup command that directly fetches the mia-nb-mule-config.xml configuration file from Galaxy:

```
./mule -builder org.mule.galaxy.mule2.config.GalaxyConfigurationBuilder \
    -config http://admin:admin@localhost:9090/api/registry?q=select%20\
        artifact%20where%20name%20=%20'mia-nb-mule-config.xml' \
    -M-Dgalaxy.port=9090 \
    -M-Dgalaxy.netboot.workspace=Mule-2-Galaxy \
    -M-Dgalaxy.app.workspaces=MuleInAction
```

This command also pulls two complete workspaces: Mule-2-Galaxy, which contains the complete distribution of the Mule version we want to use, and MuleInAction, which could contain custom libraries if needed. A smarter way to write this query would be to add extra criteria to select a particular version or a specific lifecycle phase of the configuration file.

WARNING *Lost in space* Galaxy Query Language is picky about spaces: each keyword and operator must be well separated with spaces. For example, the following query, which selects the configuration file that is shown in figure 7.20, will work:

```
select artifact where name = 'mia-nb-mule-config.xml'
```

But the following query will fail because there’s no space around the equal sign (=):

```
select artifact where name='mia-nb-mule-config.xml'
```

In the same vein, you may have noticed we had to escape spaces into %20 in the previous command-line example, which clutters the command line and reduces its readability. In that case, the best approach is to store the command-line arguments in a properties file, where no escaping is needed, and have the GalaxyConfigurationBuilder read it.

It’s important to note that Galaxy can itself become a critical part of your deployment topology. If you rely on Galaxy for direct configuration querying, you’ll need to deploy it in a way that makes it highly available so your Mule instances will never have

trouble starting up. To that end, be aware that Galaxy EE, the enterprise edition of Galaxy, natively supports clustering.

NOTE Mule isn't the only application that can leverage Galaxy for configuration management: Spring-driven applications can do it too. Thanks to a specific `GalaxyApplicationContext`, it's possible to configure a Spring bean factory from a URI that contains a Galaxy query expression.

Galaxy's capacity to centrally manage the deployments of your Mule instances is a clear win whenever your integration infrastructure starts to grow. Moreover, thanks to its versatility, Galaxy can potentially become an important actor in your IT landscape as a central repository of configuration files.

Establishing sound management of your deployments is key to the success of your Mule projects. This section has given you some hints about ways to achieve this, whether you decide to use Galaxy or not.

7.4 **Summary**

In this section, we explored the act of deploying Mule from different standpoints. We considered the different possible strategies that you can use for deploying a particular instance. We also touched on the matter of deployment topologies by giving you some angles you can use to approach this broad subject. And, finally, we discussed the necessity to manage your Mule deployments and the options available in this domain.

It should have come as no surprise that, in all these aspects, Mule presents once again a great flexibility and offers a variety of options. Because of its capacity to act as an embedded messaging framework or as a distributed broker, Mule can truly embrace your integration needs at all levels.

In section 7.2.4, we mentioned the importance of exception handlers in building a fault-tolerant Mule instance. The next chapter, which is about exception handling in Mule, will give you the knowledge you need to strengthen your Mule.



Exception handling and logging

In this chapter

- Managing exceptions with exception strategies
- Using retry policies
- Logging with Mule

Dealing with the unexpected is an unfortunate reality when writing software. Through the use of exceptions, the Java platform provides a framework for dealing with events of this sort. Exceptions occur when unanticipated events arise in a system. These are things such as network failures, I/O issues, and authentication errors. When you control a system, you can anticipate these events and provide a means to recover from them. This luxury is often absent in a distributed integration environment. Remote applications you have no control over will fail for no apparent reason or supply malformed data. A messaging broker somewhere in your environment might begin to refuse connections. Your mail server's disk may fill up, prohibiting you from downloading emails. Your own code might even have a bug that causes your data to be routed improperly. In any case, it's undesirable for your entire application to fail because of a single unanticipated error.

Logging is closely related to exception recovery. You naturally want to know when error conditions occur. This enables you to identify where the issue is and recover from it. If you have a bunch of data-type exceptions on one of your endpoints, for instance, you'll want to know where they're coming from—even if you're correctly ignoring them. Logging also aids in debugging—giving you insight into what your system's doing.

Mule's exception handling and logging functionality recognize these facts. They let you plan for, react to, and log errors that would otherwise bring your integration process to a screeching halt. You'll find yourself leveraging Mule's exception handling ability to identify and troubleshoot failures in your endpoints, components, and routers.

In this chapter we'll be examining how Mule implements exception handling and logging. We'll first consider exception strategies, where we'll see how Mule lets you react to errors on your connectors and components. We'll see how you can use Mule's routing capabilities to control where exceptions are sent after they're generated. We'll then take a look at how Mule uses the SLF4J logging facade and log4j to simplify logging configuration. Finally, we'll see how you can use Apache Chainsaw as a graphical front end to view Mule's logging data.

8.1 Exception strategies

Mule uses *exception strategies* to handle failures in connectors and components. Runtime exceptions thrown in connectors and components have the potential to “trickle up” and wreak havoc. This could cause core parts of Mule and even Mule itself to fail. Exception strategies prohibit this—they catch an exception and perform an action as a result. The appropriate response might be as simple as logging the exception and moving on, or as complex as rolling back a transaction.

While you're free to implement your own exception strategies, Mule supplies default exception strategies that are flexible enough to handle basic exception handling requirements. We'll start off this section by examining these exception strategies. We'll then see how you can use Mule's routing capabilities in conjunction with exception strategies to intelligently route and handle errors.

8.1.1 Positioning exception strategies

Mule provides exception strategies for connectors and services. The default exception strategy for connectors is responsible for handling transport-related exceptions—such as SSL errors on an HTTPS endpoint, or a connection failure on a JMS endpoint. The default exception strategy for services handles exceptions that occur in components. As components generally host your custom code, exceptions thrown here will usually be related to your business logic. By default, both of these strategies handle exceptions in the same way—they'll log the exception and Mule will continue execution.

Being able to define separate exception strategies for connectors and components lets you handle each sort of error independently. This is often desirable. You may want connector-level exceptions logged at a higher level than component-level exceptions,

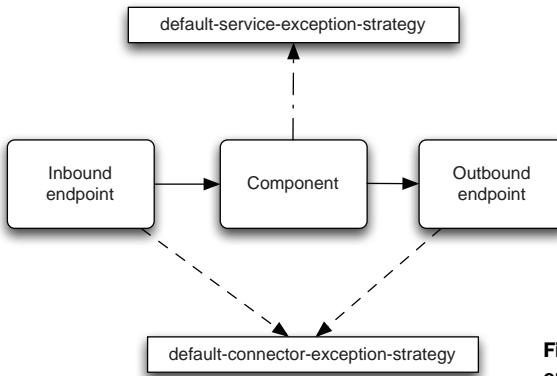


Figure 8.1 Handling exceptions on endpoints and components

for instance. As we'll see in the next chapter, this also gives you the flexibility to handle certain transaction-related responses, such as rollbacks, differently, based on where an exception occurs. The default exception strategies are illustrated in figure 8.1

You have the option of explicitly defining exception strategies in multiple places in your Mule configurations. Exception strategies can be configured on a *per-model* basis, before any service definitions. In this case, the exception strategy, either service or connector, will be applied to all subsequent services and connectors defined in the configuration. You can additionally define exception strategies on a *per-service* basis. This is done by defining the default-connector exception strategy or the default-service exception strategy at the end of each service definition.

While Mule will implicitly configure the default exception strategies for you, in order to override the defaults it's useful to see how to manually configure them. We'll demonstrate the placement of exception strategies in this section by showing where we can place the default exception strategies. You'll need this information in the next section, where the placement of an exception strategy will dictate how errors are routed out of a model or service. This will also be useful when you implement and place custom exception strategies.

The default exception strategy for connectors is configured by defining a default-connector-exception-strategy element on either a model or on a service. Defining the default-connector exception strategy on a model will cause all connectors used in that model to be handled by the defined exception strategy. Let's revisit listing 3.7 from chapter 3 and explicitly define the default-connector exception strategy for the model. The result is shown in listing 8.1.

Listing 8.1 Configuring the default-connector exception strategy on a model

```

<file:connector name="FileConnector"
    streaming="false"
    autoDelete="true"
    >
<file:expression-filename-parser/>

```

```

</file:connector>
<model name="smtpModel">
    <default-connector-exception-strategy/>

    <service name="smtpService">
        <inbound>
            <file:inbound-endpoint path=".data/invoice">
                <file:file-to-string-transformer/>
            </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <smtp:outbound-endpoint host="mail.clood.com"
                    from="mule@clood.com"
                    subject="Accounting Invoice"
                    to="accounting@clood.com">
                    <email:string-to-email-transformer/>
                </smtp:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>

    <service name="fileService">
        <inbound>
            <file:inbound-endpoint path=".data/snapshot">
                <file:filename-wildcard-filter pattern="SNAPSHOT*.xml"/>
            </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <file:outbound-endpoint
                    path=".data/archive"
                    outputPattern=
                        "#[header:originalFilename]-#[function:dateStamp].xml"/>
            </pass-through-router>
        </outbound>
    </service>
</model>

```

The default-connector exception strategy is configured after the model definition and before any service definitions, as we see in ①. This default-connector exception strategy will now handle all transport-related exceptions for the endpoints defined on ②, ③, ④, and ⑤. We can also define the default-connector exception strategy on a per-service basis as well. This is done by defining the default-connector exception as the last element of a service, as we see in listing 8.2.

Listing 8.2 Configuring the default-connector exception strategy on a service

```

<file:connector name="FileConnector"
    streaming="false"
    autoDelete="true"
    >
    <file:expression-filename-parser/>
</file:connector>

<model name="smtpModel">

    <service name="smtpService">
        <inbound>

```

```

<file:inbound-endpoint path="./data/invoice">
    <file:file-to-string-transformer/>
</file:inbound-endpoint>
</inbound>
<outbound>
    <pass-through-router>
        <smtp:outbound-endpoint host="mail.clood.com"
                                from="mule@clood.com"
                                subject="Accounting Invoice"
                                to="accounting@clood.com">
            <email:string-to-email-transformer/>
        </smtp:outbound-endpoint>
    </pass-through-router>
</outbound>
<default-connector-exception-strategy/> ❶
</service>

<service name="fileService">
    <inbound>
        <file:inbound-endpoint path="./data/snapshot">
            <file:filename-wildcard-filter pattern="SNAPSHOT*.xml"/>
        </file:inbound-endpoint>
    </inbound>
    <outbound>
        <pass-through-router>
            <file:outbound-endpoint
                path="./data/archive"
                outputPattern=
                    "#[header:originalFilename]-#[function:dateStamp].xml"/>
        </pass-through-router>
    </outbound>
    <default-connector-exception-strategy/> ❷
</service>
</model>

```

The `default-connector-exception-strategy` elements are defined on ❶ and ❷. Defining them here will cause connector-level exceptions thrown by the `smtpService` and the `fileService` to be handled independently of each other.

Now let's turn our attention to the default-service exception strategy, which defines how exceptions on our components are handled. Let's look at listing 6.8 from chapter 6 and see how to explicitly configure the default-service exception strategy to handle errors thrown by the `RandomIntegerGenerator`. The result is shown in listing 8.3.

Listing 8.3 Configuring the default-service exception strategy on a particular service

```

<service name="RandomIntegerGenerator">

    <inbound>
        <vm:inbound-endpoint path="RIG.In" />
    </inbound>

    <component>
        <no-arguments-entry-point-resolver>

```

```

<include-entry-point method="nextInt" />
</no-arguments-entry-point-resolver>

<singleton-object class="java.util.Random" />
</component>
<default-service-exception-strategy/>
</service>

```

1 Define default-service exception strategy

The default-service exception strategy configured on 1 will ensure all exceptions thrown by the RandomIntegerGenerator will be handled by the default-service exception strategy. You can also define a default-service exception strategy that all the services in a model use. The configuration is analogous to the global default-connector exception strategy we saw previously. Listing 8.4 illustrates how to accomplish this. We introduce the SeededRandomIntegerGenerator from chapter 6 and explicitly configure a default-service exception strategy that both will share.

Listing 8.4 Configuring the default-service exception strategy on all services

```

<model name="randomGeneratorModel">
    <default-service-exception-strategy/> 1 Define default-service exception strategy

    <service name="RandomIntegerGenerator">
        <inbound>
            <vm:inbound-endpoint path="RIG.In" />
        </inbound>

        <component>
            <no-arguments-entry-point-resolver>
                <include-entry-point method="nextInt" />
            </no-arguments-entry-point-resolver>

            <singleton-object class="java.util.Random" />
        </component>
    </service> 2 Define RandomIntegerGenerator service

    <service name="SeededRandomIntegerGenerator">
        <inbound>
            <vm:inbound-endpoint path="SRIG.In" />
        </inbound>
        <component>
            <no-arguments-entry-point-resolver>
                <include-entry-point method="nextInt" />
            </no-arguments-entry-point-resolver>
            <singleton-object class="java.util.Random">
                <property key="seed" value="${seed}" />
            </singleton-object>
        </component>
    </service> 3 Define SeededRandomIntegerGenerator service
</model>

```

The default-service exception strategy defined on 1 will now handle all exceptions thrown by the RandomIntegerGenerator defined on 2 and the SeededRandomIntegerGenerator defined on 3.

Let's now see how we can leverage the explicit exception strategy configuration with Mule's outbound routing capabilities. This will enable us to route exceptions to different endpoints for further processing and response.

8.1.2 Exceptions and routing

As we mentioned before, the default exception strategies will simply log exceptions and move on. This is often the right action to take, but sometimes you'll want to take more elaborate measures when an exception occurs. This is especially true in a large or distributed environment. Equally true in such an environment is the inevitability that a remote service will be unavailable. When such a service is the target of an outbound endpoint, you might want Mule to attempt to deliver the message to a different endpoint. We'll consider both scenarios in this section. Let's start by looking at how we can use routers in conjunction with exception strategies, which will allow us to do more than simply log exceptions as they occur. We'll then look at using exception-based routing. This will enable us to send data to alternate endpoints if one is unavailable.¹

While it's easy to keep an eye on log files on a handful of Mule instances, it becomes increasingly challenging when the number of Mule instances explodes or becomes distributed across a variety of locations. We'll discuss ways to mitigate this later on in this chapter when we discuss the Chainsaw tool, but you might need to distribute errors to someone who can't access or read the log data. Additionally, some errors are so critical that you want to be notified immediately when they occur. This can be difficult to accomplish by parsing log data alone. For situations like these, Mule allows you to route exceptions in much the same way we routed messages in chapter 4. This is accomplished by adding outbound endpoints to an exception strategy. This causes the exception strategy to send the exceptions through the endpoint as a message—in much the same way that a component does. Figure 8.2 illustrates how this works.

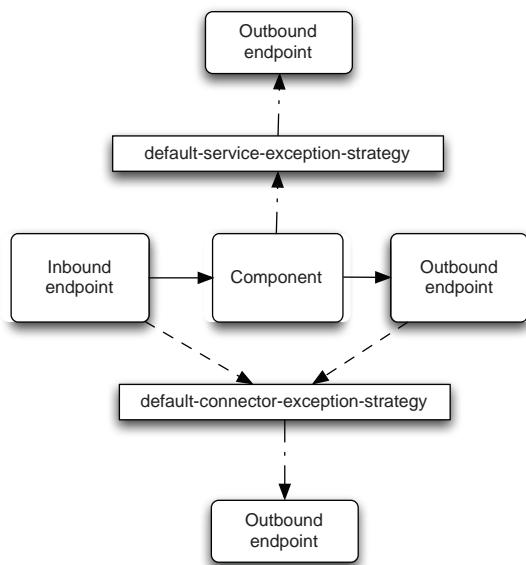


Figure 8.2 Routing exceptions with outbound endpoints

¹ We'll see later on in this chapter how to control logging in Mule.

Now let's see how we can configure Mule to do this. Listing 8.5 modifies listing 8.4 to send service exceptions and connector exceptions to separate JMS queues.

Listing 8.5 Sending all service-level exceptions to a JMS queue

```
<model name="randomGeneratorModel">
    <default-connector-exception-strategy>
        <jms:outbound-endpoint queue="transport-errors" />
    </default-connector-exception-strategy>

    <default-service-exception-strategy>
        <jms:outbound-endpoint queue="business-errors" />
    </default-service-exception-strategy>

    <service name="RandomIntegerGenerator">
        <inbound>
            <vm:inbound-endpoint path="RIG.In" />
        </inbound>

        <component>
            <no-arguments-entry-point-resolver>
                <include-entry-point method="nextInt" />
            </no-arguments-entry-point-resolver>

            <singleton-object class="java.util.Random" />
        </component>
    </service>

    <service name="SeededRandomIntegerGenerator">
        <inbound>
            <vm:inbound-endpoint path="SRIG.In" />
        </inbound>
        <component>
            <no-arguments-entry-point-resolver>
                <include-entry-point method="nextInt" />
            </no-arguments-entry-point-resolver>
            <singleton-object class="java.util.Random">
                <property key="seed" value="${seed}" />
            </singleton-object>
        </component>
    </service>
</model>
```



The JMS outbound endpoint defined on ② will route all exceptions thrown by RandomIntegerGenerator and SeededRandomIntegerGenerator to the JMS queue named business-errors. The outbound endpoint defined on ① will send all connector exceptions to the queue called transport-errors. Receivers on these queues can take some sort of action when a particular exception occurs. Let's look at how we can use Jabber to send messages to different parties depending on which queue an exception arrives on.

Listing 8.6 implements two services to handle errors on each of these queues.

Listing 8.6 Handling transport and business exceptions differently

```

<service name="transportErrorHandler">
    <inbound>
        <jms:inbound-endpoint topic="transport-errors"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <xmpp:outbound-endpoint recipient="ops-on-duty" />
        </pass-through-router>
    </outbound>
</service>

<service name="businessErrorHandler">
    <inbound>
        <jms:inbound-endpoint topic="business-errors"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <xmpp:outbound-endpoint recipient="engr-on-duty" />
        </pass-through-router>
    </outbound>
</service>

```

The diagram illustrates the message flow for two services:

- transportErrorHandler:** An incoming JMS message on the "transport-errors" topic is sent to an XMPP outbound endpoint with the recipient "ops-on-duty". This is labeled "Accept messages off transport-errors queue" and "Send exception contents to ops-on-duty".
- businessErrorHandler:** An incoming JMS message on the "business-errors" topic is sent to an XMPP outbound endpoint with the recipient "engr-on-duty". This is labeled "Accept messages off business-errors queue" and "Send exception contents to engr-on-duty".

The configuration is fairly straightforward. Exceptions that arrive on the `transport-errors` topic will be sent to an XMPP outbound endpoint where they arrive as Jabber-messages to the `ops-on-duty` user. Exceptions that arrive on the `business-errors` queue are delivered to the `engr-on-duty` user. Using topics to dispatch the errors ensures multiple parties can receive the error messages. For instance, there might be another service that subscribes to these errors and sends them as email alerts or forwards them to a logging database. As you saw in chapter 3, subscriptions to topics can also be made durable, ensuring that the message reaches the subscriber.

You might have a situation where you're only interested in routing certain types of exceptions through an outbound endpoint. Let's reconsider listing 4.9 from chapter 4 in the context of our friends at Clood, Inc. If you recall, listing 4.9 illustrated using a forwarding router to bypass component processing for messages containing a certain payload. Messages that didn't contain this payload were sent to the `messageEnricher` component for processing. Let's assume that Clood, Inc., is using this service to correct messages that aren't in an `OK` or `SUCCESS` state. If the messages aren't successfully put into the `OK` or `SUCCESS` state by the `messageEnricher` service, then an instance of `com.clood.BusinessException` is thrown. Since this is presumably a rare event, we want to send these exceptions to a JMS topic for further processing. We're satisfied with simply logging other types of exceptions. The exception-type filter is useful in cases like this—it'll cause the default exception strategy in question to only route exceptions that match the type in question. Listing 8.7 modifies listing 4.9 to accomplish this.

Listing 8.7 Using the exception-type router

```
<service name="forwardingConsumerService">
    <default-service-exception-strategy>
        <jms:outbound-endpoint topic="business-errors">
            <exception-type-filter
                expectedType="com.clood.BusinessException"/>
        </jms:outbound-endpoint>
    </default-service-exception-strategy> Only route instances of
    <inbound> com.clood.BusinessException ①
        <jms:inbound-endpoint queue="messages"/>
        <vm:inbound-endpoint address="vm://messages"/>
        <forwarding-router>
            <regex-filter pattern="^STATUS: (OK|SUCCESS)$"/>
        </forwarding-router>
        <selective-consumer-router>
            <regex-filter pattern="^STATUS: (CRITICAL)$"/>
        </selective-consumer-router>
    </inbound>
    <component>
        <spring-object bean="messageEnricher"/>
    </component>
    <outbound>
        <pass-through-router>
            <stdio:outbound-endpoint system="OUT"/>
        </pass-through-router>
    </outbound>
</service>
```

By adding the exception-type filter on ①, we're ensuring that only instances of `com.clood.BusinessException` are being routed to the `business-errors` topic. All other exceptions will be logged by the default-service exception strategy.

Routing exceptions using the default exception strategies is often a good idea, particularly in distributed environments where Mule usually runs. Routing all exceptions to a single JMS queue, for instance, aggregates errors in a central place for later triage, reporting, and management. More complex error handling can be accomplished by extending `org.mule.service.DefaultServiceExceptionStrategy` and overriding the default behaviors. You could, for instance, override the `routeException()` method to change how the exception message is sent. This would allow you to send the original message to the outbound endpoints instead of just the exception payload, serving the basis for a dead-letter queue strategy allowing you to retry failed message delivery at a later date.

BEST PRACTICE Override the default exception strategy routing to facilitate centralized error management and dead-letter queue functionality.

The default exception strategies provide a powerful means for error notification. We'll see in chapter 10 how we can further leverage exception strategies in the context of transactions—this will enable us to perform actions such as rolling back a transaction when an exception occurs. In addition to exception strategies, Mule provides another facility for reacting to errors—exception-based routing. Let's look at that now.

It's often useful in the context of outbound routing to provide multiple endpoints for a service to try in the event one of the endpoints is unavailable. The exception-based router exists for this purpose—it allows you to specify a list of endpoints that'll be tried in sequence until a message is accepted. Figure 8.3 illustrates this.

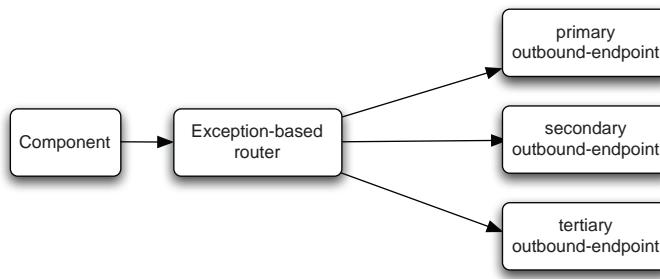


Figure 8.3 Using an exception-based router to send a message in sequence to multiple endpoints, halting when the message is successfully delivered

In the figure, the exception-based router will attempt to send the message to each endpoint in sequence until one succeeds. Let's reconsider listing 3.15 from chapter 3 to see how this works.

In listing 3.15 we were examining how one of Clood, Inc.'s partners might post reporting data to an HTTP inbound endpoint. In that example, Clood supplied the provider with a URL to post data to. Let's assume that Clood, Inc., has a new requirement to accept backup data from globally distributed partners. As such, Clood has made an effort to set up redundant endpoints for this data in its data centers in North America as well as Europe. A backup provider can choose to try the endpoint closest to it first, then fall back on a secondary endpoint in the event the first is unavailable. Listing 8.8 demonstrates how a provider in North America can use exception-based routing to accomplish this.

Listing 8.8 Falling back on multiple endpoints using an exception-based router

```

<model name="httpOutboundModel">
    <service name="httpInboundService">
        <inbound>
            <file:inbound-endpoint path=".//data/provider" />
        </inbound>
        <outbound>
            <exception-based-router>
                <http:outbound-endpoint
                    address=
                    "http://services.nyc.clood.com/backup-reporting"/>
                <http:outbound-endpoint
                    address=
                    "http://services.dub.clood.com/backup-reporting"/>
            </exception-based-router>
        </outbound>
    </service>
</model>
  
```

In this example, Mule would attempt to post the data from the file inbound endpoint to Cloud's data center in New York first. In the event this failed, the exception-based router would attempt to post the data to Cloud's data center in Dublin. If this failed as well, an exception would be thrown and handled by the exception strategy configured for the connector.

While the exception-based router is useful in failover scenarios, it might be preferable to retry the same resource repeatedly in the event it's unavailable. Let's look at how we can use retry policies to intelligently try to recover from connector failures.

8.2 Using retry policies

It's an unfortunate reality that services, servers, and remote applications are occasionally unavailable. Thankfully, though, these outages tend to be short-lived. Network routing issues, a sysadmin restarting an application, or a server rebooting all represent common scenarios that typically don't take long to recover from. Nonetheless, such failures can have a drastic impact on applications dependent on them. In order to mitigate such failures, Mule provides a retry policy mechanism to dictate how connectors deal with failed connections. We'll start off this section by examining Mule's retry policy support. We'll implement a simple retry policy that'll indefinitely attempt to connect to a failed resource. We'll then see how to use this policy with the JMS transport. Finally we'll show how we can use retry policies to allow Mule instances to start independently of remote services they may depend on.

Mule Enterprise Edition ships with a set of retry policies along with an associated XML schema. This saves you the effort of rolling your own retry policies and using the Spring injection of the retry policy template that we'll see in this section. If you're a Mule Enterprise user, you're encouraged to check the relevant documentation for your Mule Enterprise version and use the supplied retry policies and templates. You can use the information in this section to implement your own retry policies and template when the ones supplied by Mule Enterprise aren't sufficient.

8.2.1 Implementing a retry policy

A *retry policy* dictates how a connector should attempt to reconnect to a failed resource. You may want a connector to attempt to reconnect to the resource indefinitely every 5 seconds. In other scenarios you may want to connect to a resource every 2 minutes for 10 times and then stop. In order to define such behavior, you'll need to roll up your sleeves and work with the Mule API. Retry policies are defined by implementing the `RetryPolicy` interface. Listing 8.9 demonstrates a simple retry policy that'll instruct a connector to reconnect to the failed resource every 5 seconds.

Listing 8.9 An indefinitely reconnecting retry policy

```
public class SimpleRetryPolicy implements RetryPolicy {  
    public PolicyStatus applyPolicy(Throwable throwable) {  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {}  
    }  
}
```

1

```

        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return PolicyStatus.policyOk();
    }
}

```

This retry policy will sleep for 5 seconds on ①; if not interrupted it'll return a `PolicyStatus` of OK. This informs the connector to attempt to retry again. If the thread is interrupted, a `RuntimeException` will be thrown on ②. In addition to returning a `policyOK` state, `PolicyStatus` can also return an exhausted state. This is useful if you want to limit the number of retry attempts to a particular resource and is demonstrated in listing 8.10.

Listing 8.10 An exhaustible retry policy

```

public class ExhaustingRetryPolicy implements RetryPolicy {
    private static int RETRY_LIMIT = 5;
    private int retryCounter = 0;

    public PolicyStatus applyPolicy(Throwable throwable) {
        if (retryCounter >= RETRY_LIMIT) { ①
            return PolicyStatus.policyExhausted(throwable); ②
        } else {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            retryCounter++; ③
            return PolicyStatus.policyOk();
        }
    }
}

```

The `ExhaustingRetryPolicy` will attempt to connect to the remote resource five times. This is handled by incrementing the `retryCounter` on ③. If the `retryCounter` exceeds the `RETRY_LIMIT` on ①, then a `PolicyStatus` of exhausted is returned on ②, along with the cause of the retry. This causes the connector to stop retrying to connect to the failed resource.

Before we can use either of these retry policies, we must implement a *policy template*. This is accomplished by extending the `AbstractPolicyTemplate` class. Listing 8.11 illustrates a policy template for the `ExhaustingRetryPolicy`.

Listing 8.11 An exhaustible retry policy

```

public class ExhaustingRetryPolicyTemplate
    extends AbstractPolicyTemplate {

    public RetryPolicy createRetryInstance() {
        return new ExhaustingRetryPolicy();
    }
}

```

We simply need to implement the `createRetryInstance` method of `AbstractPolicyTemplate` to return an instance of the retry policy we wish to use. In this case, we'll return an `ExhaustingRetryPolicy`, which will cause the connector to attempt to connect to the failed resource a specified amount of times before giving up.

Now that we've defined our retry policies, let's see how to configure them for use in a connector.

8.2.2 Using the `SimpleRetryPolicy` with JMS

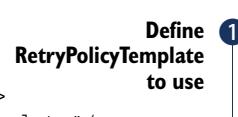
An endpoint will become unavailable if the underlying connector fails. Using the JMS transport with an external JMS broker is a good example. If the ActiveMQ instance the connector is configured on goes down, the connector will fail and JMS messages will stop being delivered—even if the ActiveMQ instance recovers. A retry policy enables the JMS connector to reconnect to the ActiveMQ instance. If it can reconnect, endpoints on the connector can begin sending and receiving JMS messages again. This allows your Mule instances to automatically recover from remote service failures without administrative assistance.

Let's return our attention to Clood, Inc., and see how they might use a retry policy. In listing 8.7 we reconsidered how Clood's backup partners might publish reports to globally distributed endpoints using the exception-based router. We discussed the receiving end of this publishing in listing 3.17, where Clood, Inc., publishes the reports to a JMS topic. To continue our example, let's assume Clood, Inc., is sharing an ActiveMQ infrastructure between its New York and Dublin POPs—this ensures reports will be received at both locations when published to a JMS topic. In order to increase resiliency for its JMS connections, Clood uses the `SimpleRetryPolicy` from listing 8.9 to automatically recover from ActiveMQ failures. Listing 8.12 illustrates how they do this.

Listing 8.12 Using the `SimpleRetryConnectionStrategy`

```
<jms:activemq-connector
    name="jmsConnector"
    specification="1.1"
    brokerURL="${jms.url}">
    <spring:property name="retryPolicyTemplate">
        <spring:bean class="SimpleRetryPolicyTemplate" />
    </spring:property>
</jms:activemq-connector>

<model name="jmsOutboundModel">
    <service name="jmsOutboundService">
        <inbound>
            <http:inbound-endpoint
                address="http://${http.host}:${http.port}/backup-reports"
                synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint>
        </inbound>
        <outbound>
```



The diagram shows a callout line originating from the `retryPolicyTemplate` attribute within the `SimpleRetryPolicyTemplate` bean declaration. The callout points to the text "Define 1 RetryPolicyTemplate to use".

```

<pass-through-router>
    <jms:outbound-endpoint topic="backup-reports" />
</pass-through-router>
</outbound>
</service>
</model>

```

We inject the `RetryPolicyTemplate` we wish to use on the ActiveMQ connector on ①. The JMS connector will now invoke the `SimpleRetryPolicy` when its connection to the ActiveMQ broker is interrupted.

In the event of an ActiveMQ failure, the JMS connector will attempt to reconnect to the ActiveMQ instance every 5 seconds for an indefinite amount of time. JMS messaging will fail during this time, but the Mule instance will still be up. Once ActiveMQ recovers, the JMS connector will reconnect to the endpoint and JMS messaging will continue as usual. A side effect of this behavior is that it allows Mule to start up when external dependencies are unavailable. Let's see how to do this.

8.2.3 Starting Mule with failed connectors using the Common Retry Policies

Mule will fail to start if an external dependency, such as a JMS broker or SMTP server, is unavailable. The `SimpleRetryPolicy` we implemented earlier doesn't handle this situation well—it'll block indefinitely until the remote resource becomes available, prohibiting Mule from starting up. Having the retry attempts occur in their own thread is one way to circumvent this issue. Such functionality is available to Mule EE users but is currently not available in the community Mule 2 release. Fortunately a MuleForge project exists to fill this gap—the Common Retry Policies

The Common Retry Policies Mule module is available on MuleForge at this address: <http://www.mulesource.org/display/COMMONRETRYPOLICIES/Home>. In addition to containing an implementation of the `SimpleRetryPolicy` we saw previously, the Common Retry Policies package also provides a multithreaded retry policy template. This template, called the *adaptive retry policy*, will cause retry attempts to occur in a thread separate from the main Mule thread when Mule is starting up. This allows Mule to start even if some connectors are unavailable. You'll need to download the JAR file from the Common Retry Policies home and make it available to your Mule installation prior to being able to use the policies.²

Let's see how we can have listing 8.11 start up when the JMS broker is unavailable. Listing 8.13 illustrates using the adaptive retry policy to accomplish this.

Listing 8.13 Allowing Mule to start when a connector is failed

```

<spring:beans>
    <spring:bean id="foreverRetryPolicyTemplate"
        class=
        "org.mule.modules.common.retry.policies.ForeverRetryPolicyTemplate" />

```

Define retry
policy template ①

² The Common Retry Policies is a MuleForge project maintained by the community. It's not part of the mainstream Mule distribution or maintained by MuleSource.

```

<spring:bean id="threadingPolicyTemplate"
    class=
"org.mule.modules.common.retry.policies.AdaptiveRetryPolicyTemplateWrapper">
    <spring:property name="delegate" ref="foreverRetryPolicyTemplate"/>
</spring:bean>
</spring:beans>

<jms:activemq-connector
    name="jmsConnector"
    specification="1.1"
    brokerURL="${jms.url}">
    <spring:property name="retryPolicyTemplate"
        ref="threadingPolicyTemplate"/>
</jms:activemq-connector>

<model name="jmsThreadingRetryModel">
    <service name="jmsThreadingRetryService">
        <inbound>
            <http:inbound-endpoint
                address="http://${http.host}:${http.port}/backup-reports"
                synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint topic="backup-reports"/>
            </pass-through-router>
        </outbound>
    </service>
</model>

```


We start off by defining the retry policy we want to use on ①. In this case, we're going to use the `ForeverRetryPolicyTemplate` supplied by the Common Retry Policies. The `ForeverRetryPolicyTemplate` behaves much like the `SimpleRetryPolicy` we looked at before. It'll attempt to reconnect to a failed connector every 5 seconds. The `AdaptiveRetryPolicyTemplateWrapper` defined on ② is what enables the `ForeverRetryPolicy` behavior to occur in its own thread when Mule starts. We finally inject the `threadingPolicyTemplate` into the ActiveMQ connector on ③. If the JMS broker is down when Mule is started, the `AdaptiveRetryPolicyTemplateWrapper` will use the `ForeverRetryPolicyTemplate` to attempt to reconnect to the broker independently of Mule bootstrapping. This will allow other services that aren't dependent on that JMS broker to start and behave normally.

In this section we saw how connection strategies enable us to tolerate failures in remote services without restarting Mule. We saw how we can automatically reconnect to failed services as well as start up Mule when remote dependencies are unavailable. Let's turn our attention now to Mule's logging facilities.

8.3 Logging with Mule

We've spent a lot of time discussing different ways to deal with errors that crop up in Mule deployments. One of the most common ways you'll deal with errors and other diagnostic events is by logging them. Mule uses the Apache Commons Logging package along with the SLF4J logging facade (<http://www.slf4j.org/>) to allow you to plug and play logging facilities. By default, Mule will use the popular log4j logging library without any intervention on your part. We'll see later on how we can change this behavior to allow Mule to use other logging implementations, such as `java.util.logging`. Let's start off with a look at how logging works in a freshly installed, standalone Mule instance.

8.3.1 Using log4j with Mule

Mule uses log4j as its default logging implementation. Log4j is a robust logging facility that's commonly used in many Java applications. Full documentation for using log4j is available on the project's web site at <http://logging.apache.org/log4j/>. Mule provides a default log4j configuration in the `$MULE_HOME/conf/log4j.properties` file. Let's look at this file now; listing 8.14 shows the default `log4j.properties` file.

Listing 8.14 The default `log4j.properties` file

```
#Default log level
log4j.rootCategory=INFO, console


- 1 Specify default log level


log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-5p %d [%t] %c: %m%n


- 2 Specify logging format


#####
# You can set custom log levels per-package here
#####


- 3 Specify logging level of org.apache package


# Apache Commons tend to make a lot of noise which can clutter the log.
log4j.logger.org.apache=WARN


- 4 Specify logging level of org.mule


# Mule classes
log4j.logger.org.mule=INFO


- 5 Specify logging level for your packages


# Your custom classes
log4j.logger.com.mycompany=DEBUG
```

If you've worked with log4j at all before, this should seem familiar to you. Log4j supports the concept of log levels for packages. The log levels available are DEBUG, INFO, WARN, ERROR, and FAIL, in ascending order of severity. The default log level is specified on ①, which is logging messages of level INFO to the console. The format of the log output is specified on ②. You can tweak this to customize how logging is output (see the log4j documentation for more information on how to do this). The log definitions on a per-package basis start on ③, where the libraries for the `org.apache` project (which are used extensively by Mule and Spring) are set at a logging level of WARN. The logging level for the Mule packages is specified next on ④. The default level is INFO, but you'll soon find it's convenient to set this to DEBUG for troubleshooting and

general insight into how Mule is behaving. Finally, you can change com.mycompany to your company's package prefix in order to set the debugging level for your custom components, transformers, routers, and so forth. For instance, in order to set DEBUG logging for Cloud, Inc.'s custom classes we'd change ⑤ to this:

```
log4j.logger.com.clood=DEBUG
```

By default, Mule will write to a log file called mule.log. This file is located in \$MULE_HOME/logs. You can change this location by editing the wrapper.logfile variable in \$MULE_HOME/conf/wrapper.conf. Mule will write 1 megabyte of data to the mule.log file before automatically rotating it. It'll archive up to 10 rotations with the stock configuration. This behavior is configured in wrapper.conf as well, by tuning the wrapper.logfile.maxsize and wrapper.logfile.maxfiles variables.

NOTE *Getting diagnostics from log4j* You occasionally need to gather debugging information from log4j itself. This can be accomplished by setting the log4j.debug property. If you're launching Mule in the standalone fashion discussed in chapter 7, this can be accomplished by appending the string -M-Dlog4j.debug at the end of the command to launch Mule. The following code shows how to do this:

```
mule -config my-config.xml -M-Dlog4j.debug
```

It's also possible to specify an alternative log4j.properties file by specifying the URL to the file. The following illustrates how to do this.

```
mule -config my-config.xml  
-M-Dlog4j.configuration=file:///path/conf/log4j.properties
```

Now that you've seen how to configure log4j and SLF4J, let's look at how we can use Chainsaw to make it easier to view and analyze log entries.

NOTE You might want to use a logger other than log4j with Mule. This is an easy task provided you're using a logging implementation supported by SLF4J. SLF4J supports JDK 1.4 logging, Logback, and the Apache Commons Logging project. Simply download the SLF4J implementation for your version of Mule, remove the existing SLF4J bridge and place the appropriate SLF4J bridge in \$MULE_HOME/lib/boot. On your next restart, Mule should log using the new implementation.

8.3.2 Using Apache Chainsaw with log4j

Dealing with text logs can be difficult at times. Even in a Unix-like environment, where a plethora of excellent text manipulation tools are available, it's occasionally useful to have a way to graphically look at and compare log data. This is especially true when log data must be analyzed from multiple sources, a common scenario when dealing with distributed environments and applications. Apache Chainsaw is an attempt to provide such a tool for log4j. In this section we'll investigate how to use Apache Chainsaw to view log data from Mule.³

³ You must be using log4j as your SLF4J logging implementation to use Apache Chainsaw.

Let's start by installing Apache Chainsaw. You can download Chainsaw from the project's web site here: <http://logging.apache.org/chainsaw>. There are three options for installation: running as a Java Webstart application, installing as on OS/X application, or running from the command line. The GUI is functionally identical between the three, so pick the means most convenient for you and start Chainsaw up. You should see a screen that looks something like figure 8.4.

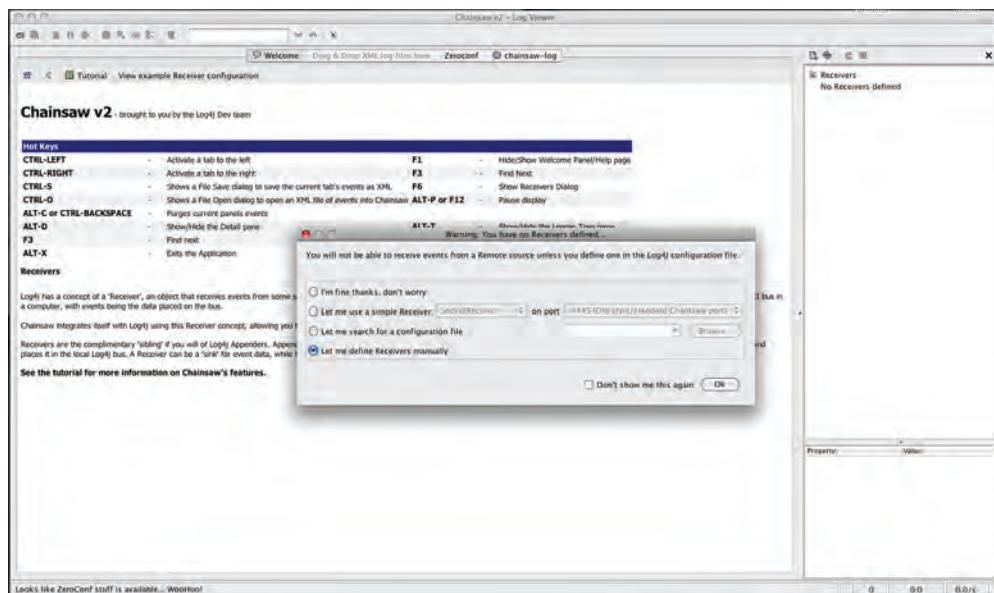


Figure 8.4 Running Chainsaw for the first time

For now, select *Let me define Receivers manually* and then click OK to continue. You should now be on the Welcome tab. Feel free to click around and explore the UI, and then we'll investigate how we can modify Mule's default logging configuration in order to see logs in Chainsaw.

Chainsaw uses log4j's receiver framework to accept remote logging events. One such receiver it supports is the `SocketHubAppender`. The `SocketHubAppender` is enabled in the log4j configuration to listen on a socket and publish logging events to connected clients. Listing 8.15 demonstrates how to modify the default `log4j.properties` file in `$MULE_HOME/conf` to get the `SocketHubAppender` going.

Listing 8.15 Configure a `SocketHubAppender` in the default `log4j.properties`

```
# Default log level
log4j.rootCategory=INFO, console, sockethub
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-5p %d [%t] %c: %m%n
```

Send logs to
SocketHubAppender

```

log4j.appenders.sockethub=org.apache.log4j.net.SocketHubAppender
log4j.appenders.sockethub.port=9999
#####
# You can set custom log levels per-package here
#####
# Apache Commons tend to make a lot of noise which can clutter the log.
log4j.logger.org.apache=WARN

# Mule classes
log4j.logger.org.mule=INFO

# Your custom classes
log4j.logger.com.mycompany=DEBUG

```

This configuration will set up a `SocketHubAppender` to listen for connections on port 9999. Clients connected to this socket will receive logging data from log4j. When you start Mule you should be able to connect to this port, using a tool such as netcat or telnet, and you'll see logging data sent to the socket. Let's now see how to get Chainsaw to connect to this port and receive logging events from Mule.

To receive logging events in Chainsaw we'll need to configure it to connect to the `SocketHubAppender` we just configured. The receiver is configured in the right panel of the Chainsaw GUI. You can configure a new receiver by clicking on the New Receiver button in the upper left side of the panel. The screenshot in figure 8.5 illustrates this.

After you click on this, you'll be presented with options to configure the receiver. We're going to set the host to localhost, the name to `Mule-1`, and the port to 9999. Once this is done, click the Refresh button and you should see the `Mule-1` receiver appear in the right panel along with a localhost tab on the top of the middle panel. This should look something like figure 8.6.

Assuming your Mule instance is generating logging events, you should start seeing these appear in the localhost tab. As you can see, you can now filter through the logging events by ID, timestamp, level, logger, message, and thread. The panel in the bottom center of the screen displays the full content of the logged message. On the right panel is a tree you can use to filter messages based on package hierarchy. This allows you to ignore or focus on messages logged by classes in a certain package. You can even change the color of log entries based on package or severity.

NOTE *The log component* You saw in chapter 6 how you can use the log component to send message payloads to the Mule log. We'll see further use of this component in chapter 11 when we begin talking about logging in the context of monitoring Mule instances.

The real value of a tool such as Chainsaw comes into play when you're dealing with several deployed Mule instances. Manually watching the log files on 10 Mule instances, for instance, quickly becomes unmanageable. With Chainsaw, you can set up a `SocketHubAppender` on each instance and monitor all 10 from a single window.

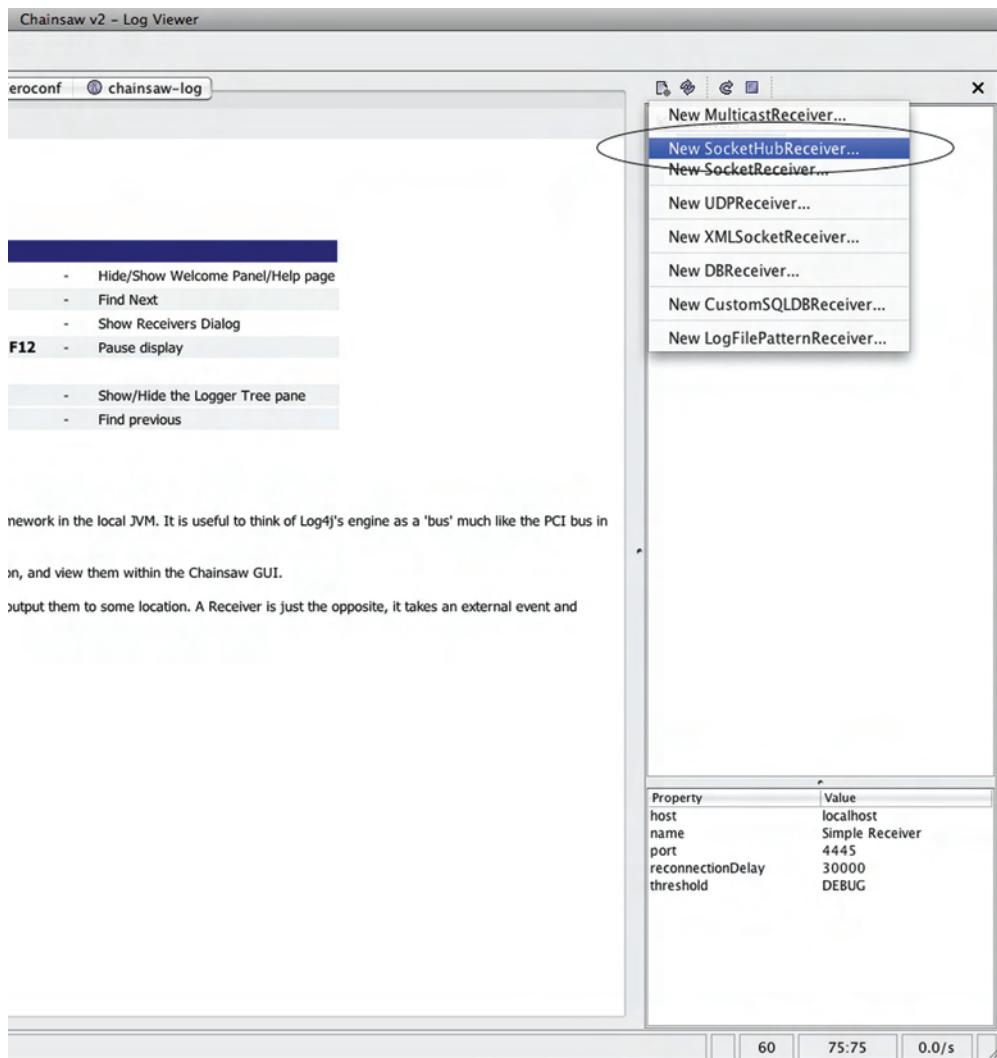


Figure 8.5 Creating a new SocketHubReceiver

on your desktop. Even with a small number of Mule instances, it's easy to see how this can be useful. You'll see more uses of Chainsaw in chapter 11, where we discuss how it can be useful in the context of monitoring Mule instances.

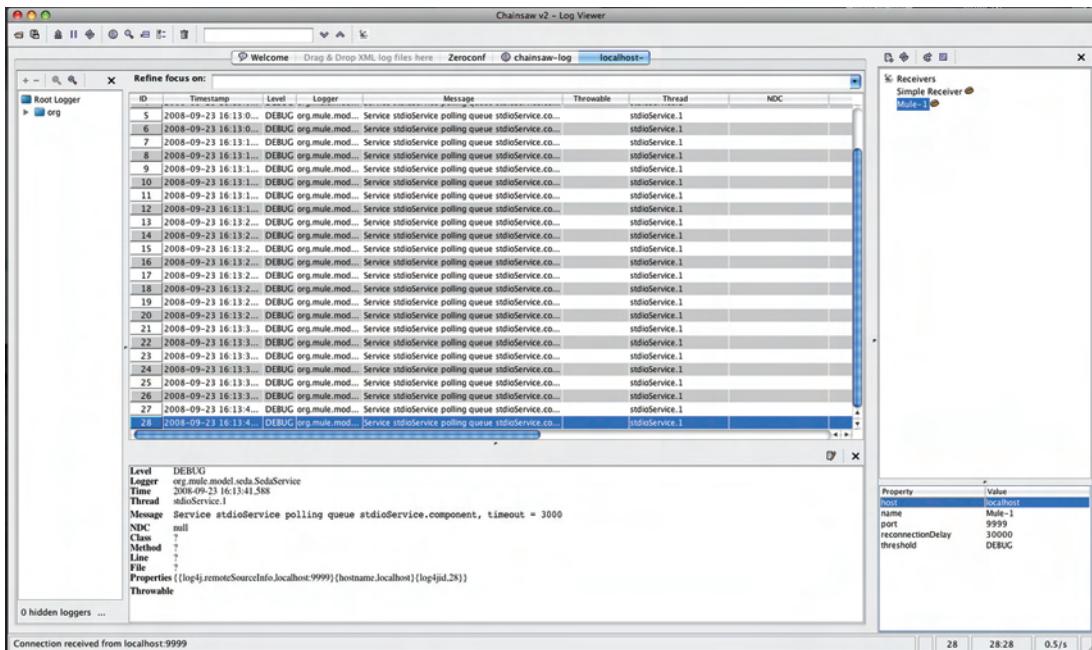


Figure 8.6 Connect Chainsaw to SocketHubAppender to receive Mule logging events.

8.4 Summary

In this chapter we investigated Mule's error handling capabilities. We saw how to use exception strategies to define how errors are handled on the connector and service levels. We then saw how to leverage exception strategies with outbound routing to manage what happens after an exception occurs. In the event of an endpoint failure, you saw how exception-based routing enables you to send the same message to multiple endpoints, trying each in succession. We turned our attention then to logging, investigating how Mule uses the SLF4J logging facade to support multiple logging implementations. We looked at how to configure Mule's default logger, log4j, as well as replace it with JDK 1.4 logging. Finally, we saw how to use Chainsaw as a graphical front end to Mule's logging data.

In the next chapters we'll be discussing other aspects of running Mule. We'll continue an aspect crucial to Mule deployments: security.

Securing Mule

In this chapter

- Securing Mule with Spring Security
- Using JAAS with Mule
- Using security filters to secure endpoints

Security is a challenge in application development and deployment—a challenge that’s exacerbated by application integration. Single sign-on technologies such as Kerberos, CAS, and LDAP minimize these burdens, but it’s unlikely that every application in your environment supports the SSO technology at hand. Even if this is the case, all bets are off when you’re integrating with applications outside of your company’s data centers. Thankfully, Mule employs the same architectural principles we saw in part 1 in its handling of security. This gives you the opportunity to decouple your security concerns from your routing, transformation, and components.

You’ll see in this chapter how Mule’s security architecture will enable you to quickly simplify what would otherwise be complex security tasks. These simplifications will cross-cut your authentication, authorization, and encryption concerns. We’ll see how Clood, Inc., uses Mule’s security features to perform authentication on endpoints, authorize users, and encrypt payloads. We’ll also demonstrate how Mule enables you to pull this all together to intelligently and quickly secure your integration infrastructure.

9.1 Demonstrating Mule security

Before we dig into Mule's security support, let's look at an example that demonstrates the flexibility of Mule's security system. Consider the model defined in listing 9.1.

Listing 9.1 Echoing data without encryption

```
<model name="auditModel">
    <service name="auditService">
        <inbound>
            <http:inbound-endpoint
                address="http://localhost:8080/audit-trail-data"
                synchronous="true"
            >
        </http:inbound-endpoint>
    </inbound>
    <outbound>
        <pass-through-router>
            <jms:outbound-endpoint address="topic://audit-data"/>
        </pass-through-router>
    </outbound>
    </service>
</model>
```

The diagram shows the flow of data in the auditModel. An arrow points from the `<http:inbound-endpoint` to the annotation **Accept data from HTTP inbound endpoint**. Another arrow points from the `<jms:outbound-endpoint` to the annotation **Forward data to JMS topic**.

Clood, Inc., uses this model to move data from an HTTP inbound endpoint to a JMS topic, where it's consumed by a remote office. There's recently been some concern about disquieting data entering the HTTP inbound endpoint. It seems like a malicious user is trying to get malformed data onto the JMS topic. To mitigate these concerns, you decide to force clients to encrypt the data being sent to the inbound endpoint using a key shared with the remote office. Additionally, you decide to enforce HTTP basic auth on the inbound endpoint. This will tie into Clood's LDAP infrastructure to require clients to supply a HTTP basic auth header for every request.

These requirements can be implemented by Mule's security support. Listing 9.2 illustrates how to use Mule's support for Spring Security to accomplish both tasks.

Listing 9.2 Using password-based encryption to encrypt the payload of JMS messages

```
<mule-ss:security-manager>
    <mule-ss:delegate-security-provider
        name="ldap-security-provider"
        delegate-ref="authenticationManager"/>
</mule-ss:security-manager>

<security-manager>
    <password-encryption-strategy
        name="passwordEncryption"
        password="password"/>
</security-manager>

<model name="securityManagerHttpPbeModel">
    <service name="securityManagerHttpPbeService">
        <inbound>
            <http:inbound-endpoint
```

The diagram highlights two configuration sections. A callout labeled **1 Define Mule Security Manager for LDAP** points to the `<mule-ss:delegate-security-provider` element. A callout labeled **2 Define Mule Security Manager for password-based encryption** points to the `<password-encryption-strategy` element.

```

        address="http://localhost:8080/audit-trail-data"
        synchronous="true">
      <mule-ss:http-security-filter realm="audit-trail"/> ←
    </http:inbound-endpoint>                                ③
  </inbound>
  <outbound>
    <pass-through-router>
      <jms:outbound-endpoint address="topic://audit-data">
        <transformers>
          <encrypt-transformer
            strategy-ref="passwordEncryption"/>
        </transformers> ←
      </jms:outbound-endpoint>                                ④
    </pass-through-router>
  </outbound>
</service>
</model>

```

The code listing shows XML configuration for a Mule application. Annotations are placed on specific lines:

- Enforce authentication on inbound endpoint** (3) points to the line containing the `<mule-ss:http-security-filter realm="audit-trail"/>` element.
- Encrypt payloads of messages leaving outbound endpoint** (4) points to the line containing the `<encrypt-transformer strategy-ref="passwordEncryption"/>` element.

We'll dig into the details of listing 9.2 as this chapter progresses. For now, though, you can see we implemented the security requirements by adding a small amount of XML. The security managers configured on ① and ② centralized the security mechanisms used by the endpoints on ③ and ④. This allows you to define security mechanisms, such as LDAP providers or encryption schemes, once and reuse them throughout your Mule configurations. It also means that you can make changes to the security managers in one place and have those changes propagate to all of your endpoints.

Let's take a deeper look into Mule's security features. We'll start off by investigating how security managers and providers allow you to integrate with external authentication and authorization systems.

9.2 Using security managers and understanding security providers

Mule's security functionality is supplied to components, endpoints, and transformers by *security managers*. Security managers implement an interface, `org.mule.api.security.SecurityManager`, which abstracts the details of an underlying security mechanism. The password encryption strategy configured in listing 9.2 demonstrates Mule's default security manager. The default security manager provides support for basic security functionality, such as password and secret-key-based encryption. Mule provides additional security manager implementations that support more sophisticated security implementations such as Spring Security, which you also saw in listing 9.2, or JAAS and PGP, which we'll see later in this chapter.

As you've seen, user authentication and authorization is handled by a security manager as well. This is done with the help of a Mule *security provider*. A security provider is an implementation of an interface, `org.mule.api.security.SecurityProvider`, that's responsible for authenticating and authorizing a message. Mule provides security managers and security providers for common back-end authentication schemes, such as Spring Security and JAAS.

In this section we'll look at how to use security managers and security providers to secure your Mule services. We'll start off by looking at using Spring Security, where we'll examine authenticating users against memory and LDAP user databases. We'll then turn our attention to JAAS, the Java Authentication and Authorization Service, and see how we can similarly leverage that to modularize our security concerns.

9.2.1 Using Spring Security

Spring Security, formerly known as Acegi, is an officially supported security product of the Spring Portfolio. Using Spring Security with Mule follows the pattern we described earlier. After defining your Spring Security configuration, which we'll see how to do shortly, you define Mule Spring Security providers and managers that can be used on your endpoints.

NOTE Before you can use Spring Security with Mule, you'll need to include the Spring Security and Mule Security namespaces. The schema definitions are defined as follows:

```
<mule
    xmlns="http://www.mulesource.org/schema/mule/core/2.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:mule-ss=
        "http://www.mulesource.org/schema/mule/spring-security/2.2"
    xmlns:ss="http://www.springframework.org/schema/security"
    xsi:schemaLocation=" http://www.springframework.org/schema/beans
http://www.springframework.org/schema/spring-beans-2.5.xsd
http://www.mulesource.org/schema/mule/core/2.2
http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
http://www.mulesource.org/schema/mule/spring-security/2.2
http://www.mulesou.../spring-security/2.2/mule-spring-security.xsd
http://www.springframework.org/schema/security
http://www.springfr.../schema/security/spring-security-2.0.xsd"
>
```

Mule uses the security manager to broker authentication to the security provider and the back-end security resource, such as a database of user accounts or an LDAP directory. A bit confusingly, Mule's Spring Security provider delegates to a Spring Security manager. The Spring Security manager then delegates to back-end authentication schemes, such as LDAP or an in-memory database of usernames. Figure 9.1 illustrates this relationship.

The additional level of indirection serves a purpose—it allows Spring Security to attempt multiple authentication mechanisms when authenticating users. This is useful when authentication data is spread across multiple data sources, such as LDAP and JDBC. You may want to try LDAP authentication first and, if that fails, try JDBC authentication. Let's take a look at two ways we can configure Spring Security with Mule. First we'll look at using an “in-memory” DAO to authenticate and authorize users. We'll then turn our attention to using Spring Security with an LDAP directory.

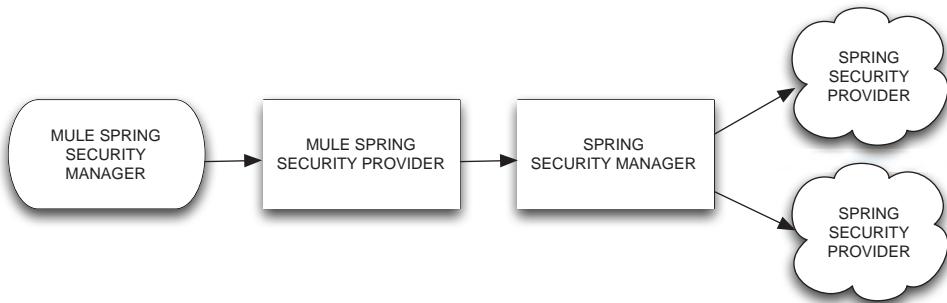


Figure 9.1 Delegation between Mule Spring Security and Spring Security

USER SECURITY WITH A MEMORY USER SERVICE

For simple applications, testing, and experimentation, it can be useful to use a static database for user information. Spring Security provides such a mechanism through its *user service*. The user service maintains a static map of users, passwords, and roles in memory. Listing 9.3 demonstrates configuring the user service along with a Spring Security manager and provider.

Listing 9.3 Defining a memory user service for endpoint authentication

```

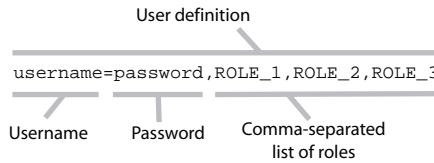
<spring:beans>           ◀ 1 Define Spring Security authentication manager
    <ss:authentication-manager
        alias="authenticationManager" />
    <ss:authentication-provider>          ◀ 2 Define Spring Security authentication provider
        <ss:user-service id="userService">
            <ss:user name="john"
                password="password"
                authorities="ROLE_ADMIN"/>
            <ss:user name="david"
                password="password"
                authorities="ROLE_ADMIN"/>
        </ss:user-service>
    </ss:authentication-provider>
</spring:beans>
<mule-ss:security-manager>      ◀ 4 Define Mule Spring Security manager
    <mule-ss:delegate-security-provider
        name="memory-dao"
        delegate-ref="authenticationManager" />
</mule-ss:security-manager>
</model>
...
</model>
  
```

◀ 3 Define user

◀ 5 Define Mule Spring Security provider

The Mule Spring Security manager is defined on ④. The security provider is then configured on ⑤ to delegate to the Spring Security manager defined on ①. The manager will implicitly use the Spring Security authentication provider defined on ②.

which is configured with the memory user service. The definitions for the user service start on ③. The format used to define a user is described here:



While the in-memory user service is a powerful tool for testing and simple authentication, you'll likely be facing more complex authentication schemes in your environment. One such authentication scheme is provided by an LDAP directory, such as OpenLDAP, Apache Directory Server, or Microsoft's Active Directory. Let's look at how to use Spring Security's LDAP support to authenticate against an LDAP directory.

USER SECURITY WITH LDAP

LDAP has emerged as a common directory implementation in many organizations, as evident from the popularity of products such as OpenLDAP and Active Directory. Because it's common to store authentication data alongside organizational data in a directory, it's important for a security framework to support LDAP as an authentication and authorization mechanism. Spring Security is no exception and as such provides rich support for LDAP directories. Let's look at how to configure Mule to use Spring Security to authenticate against an OpenLDAP directory. Listing 9.4 defines a security manager and provider that authenticate off Cloud's OpenLDAP server.

Listing 9.4 Defining an LDAP directory for endpoint authentication

```

<spring:beans>

    <ss:ldap-server
        root="dc=clood,dc=com"
        url="ldap://ldap.clood.com:389" />           ① Define the LDAP server

    <ss:ldap-authentication-provider
        user-dn-pattern="uid={0},ou=people"
        group-search-base="ou=groups"/>                 ② Define the LDAP
                                                        authentication provider

    <ss:authentication-manager alias="authenticationManager"/>
    <mule-ss:security-manager>
        <mule-ss:delegate-security-provider
            name="memory-dao"
            delegate-ref="authenticationManager"/>
    </mule-ss:security-manager>                      ③ Define Mule
                                                    Security manager ③

</spring:beans>                                     ④ Define Spring
                                                    Security provider
  
```

Cloud's LDAP server is defined on ①. We define the URL of the server along with the root *distinguished name* (DN) we want to query from. A distinguished name is used to uniquely identify an object in the LDAP directory. The Spring Security LDAP authentication provider is then defined on ②. We define two *DN patterns* here, one for user

lookups and another for group lookups. The user-dn-pattern tells Spring Security how to look up usernames from the directory. The {0} here will be filled in with the MULE_USER property as defined when a user attempts authentication on an endpoint. The group-search-base tells Spring Security where to begin searches for authority data. These are usually defined by groups the user is a member of in LDAP. Groups the user is a member of are capitalized and prepended with the string ROLE_. For instance, users in the LDAP group admin would have the ROLE_ADMIN authorities assigned to them when they're authenticated. The Spring Security authentication manager is defined in ③ and finally referenced by the Mule Spring Security manager on ④. Endpoints referencing this security manager will now use Cloud's LDAP server for authentication.

We've only touched on the flexibility Spring Security offers in this section. You're encouraged to check out the official documentation at <http://static.springframework.org/spring-security/site/index.html> for more info. Now that you're comfortable with configuring Spring Security for both in-memory and LDAP authentication scenarios, let's look at how to use JAAS with Mule.

9.2.2 Using JAAS

JAAS, the Java Authentication and Authorization Service, is a security framework that ships with the JRE. It allows you to plug in authentication mechanisms using an external configuration file. This enables you to keep authentication details independent of your Mule configuration. If you're familiar with Unix operating systems, JAAS is similar to PAM—it abstracts out authentication and authorization configuration from an application. In this section we'll configure Mule to authenticate using JAAS. For the purposes of this example we'll be using the DefaultLoginModule supplied by Mule. It allows us to define a static database of usernames, much like the in-memory user service we used with Spring Security. Listing 9.5 illustrates how to configure the DefaultLoginModule in jaas.conf.

Listing 9.5 Using jaas.conf to configure the DefaultLoginModule

```
jaas-simple {
    org.mule.module.jaas.loginmodule.DefaultLoginModule required
    credentials="john:password"
    debug=true;
};
```

In the jaas.conf file, we declare a LoginContext called jass-simple that requires the DefaultLoginModule. We then specify the map of login credentials and enable debugging. Listing 9.6 illustrates how to set up a JAAS security manager to reference this file.

Listing 9.6 Referencing jaas.conf with a JAAS security manager

```
<jaas:security-manager>      ←
    <jaas:security-provider     ① Define
        jaas:security-manager
```

```

name="jaas-provider"
loginContextName="jaas-simple"
loginConfig="jaas.conf"/>
</jaas:security-manager>

```

The Mule configuration is equally straightforward. We define a JAAS security manager on ① and then define the JAAS security provider on ②. The security provider is configured with the location of the `jaas.conf` file along with the `loginContextName` (`jaas-simple` in this case). You now have pluggable authentication module support for Mule. As you probably expect, a production `jaas.conf` will likely be more complex than what's in listing 9.5. We'll be seeing more of JAAS in the next section when we discuss authenticating JMS messages. For more information about configuring JAAS, please see <http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/JAASRef-Guide.html>.

By now you should have a good grasp of configuring Mule's security functionality. You've seen how security managers and security providers work together to authenticate and authorize users. We introduced Spring Security and saw how to configure in-memory and LDAP-based authentication and authorization. We then saw how to use JAAS to achieve externalized, pluggable authentication and authorization. You might be wondering, though, what good all of this is if we haven't actually authenticated anything yet. Let's see how we can use the techniques to achieve endpoint security with security filters.

9.3 Securing endpoints with security filters

Security filters allow you to control access, authorization, and encryption on your endpoints. We'll start off this section by looking at how to filter HTTP inbound endpoints using Spring Security and the HTTP security filter. We'll then see how we can use the JAAS security filter to authenticate JMS messages. We'll finally look at encrypting message payloads, using both password-based and public key encryption methods.

9.3.1 Securing an HTTP endpoint with Spring Security

Let's start our discussion on security filtering by looking at securing an HTTP inbound endpoint with the HTTP security filter. The HTTP security filter will attempt to authenticate every HTTP request it receives using HTTP basic authentication. Requests that don't contain a basic authentication header or whose header doesn't pass validation by the relevant security manager aren't passed by the filter. Listing 9.7 shows how to configure an HTTP security filter using Spring Security and a memory user service.

Listing 9.7 Defining a Basic HTTP security filter on an HTTP inbound endpoint

```

<spring:beans>           ← ① Define user service beans
    <ss:authentication-provider>
        <ss:user-service id="userService">
            <ss:user name="john" password="password" authorities="ROLE_ADMIN"/>
        </ss:user-service>

```

```

</ss:authentication-provider>
<ss:authentication-manager alias="authenticationManager">
</spring:beans>
<mule-ss:security-manager>           ② Define Mule
                                         security manager
    <mule-ss:delegate-security-provider
        name="memory-dao"
        delegate-ref="authenticationManager" />
</mule-ss:security-manager>

<model name="httpModel">
    <service name="httpService">
        <inbound>
            <http:inbound-endpoint
                address="http://localhost:8081/secure"
                synchronous="true">
                <mule-ss:http-security-filter realm="mule" />      ③ Define security filter
            </http:inbound-endpoint>                            on HTTP endpoint
        </inbound>
        <echo-component/>
    </service>
</model>

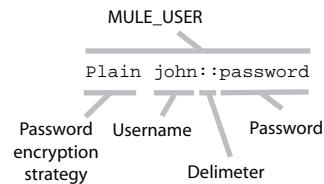
```

The memory user service is defined by the beans on ①, which are delegated to by the Mule security manager defined on ②. Now we'll see where these come into play. The HTTP security filter on ③, defined inside the HTTP inbound endpoint, will require a client to supply HTTP basic authentication credentials before accepting the message. These credentials are then passed to the Mule security manager for authentication. Remember that you can swap out the authentication mechanism, say from in-memory DAO to LDAP directory, by simply changing the delegate reference inside the security manager. No changes need to be made to the security filter. Now that we've seen how to do simple authentication with the HTTP security filter, let's look at how to secure JMS messages.

BEST PRACTICE Change the delegate reference in a security manager to swap out authentication mechanisms.

9.3.2 Performing JMS header authentication with JAAS

We can use JMS properties to authenticate JMS messages in much the same way we just used HTTP headers to authenticate HTTP requests. Mule abstracts the authentication data it receives from transports onto a special message property called `MULE_USER`. Mule, by default, will map JMS message properties onto the Mule message. We can take advantage of this fact and set a `MULE_USER` String property on each of our JMS messages. If we format the value of that property correctly, a security filter on a JMS endpoint can attempt to authenticate the message. The diagram at the right shows how to format the `MULE_USER` value.



The MULE_USER value consists of a password encryption strategy, a space, a username, a delimiter (::) and a password. The encryption strategy reference is always unencrypted. The username, delimiter, and password are either all encrypted or all plaintext. In the figure, we're specifying the encryption strategy as Plain (no encryption) and sending a credential string consisting of the username john and the password password. The security filter will split this string on the :: delimiter and submit the username and password to the security provider for authentication.

Let's see how this works. Listing 9.8 defines a JMS inbound endpoint with a JAAS security filter. It'll only pass messages that contain a valid MULE_USER property to the echo component.

Listing 9.8 Using JAAS to authenticate JMS messages

```
<jaas:security-manager>
    <jaas:security-provider
        name="jaas-simple"
        loginContextName="jaas-simple"
        loginConfig="jaas.conf" /> ①
</jaas:security-manager>

<model name="jaasModel">
    <service name="jaasService">
        <inbound>
            <jms:inbound-endpoint queue="messages">
                <jaas:jaas-security-filter/> ②
            </jms:inbound-endpoint>
        </inbound>
        <echo-component/>
    </service>
</model>
```

Assuming the JAAS security provider defined on ① is using the same jaas.conf we defined in listing 9.4, the JAAS security filter on ② will pass any JMS message containing a MULE_USER property that matches diagram on the previous page.

We're naturally wary about passing plaintext passwords around in JMS messages. We can remedy this by encrypting the credentials contained in the MULE_USER property. We can use a *password encryption strategy* to encrypt the password in the header. Adding a password encryption strategy to the security manager in listing 9.9 accomplishes this.¹

Listing 9.9 Using password-based encryption with JAAS

```
<jaas:security-manager>
    <jaas:security-provider
        name="jaas-simple"
        loginContextName="jaas-simple"
        loginConfig="jaas.conf" />
    <jaas:password-encryption-strategy>
```

¹ We'll see how to use the password encryption strategy to encrypt entire message payloads in the next section.

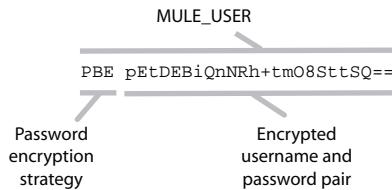
```

        name="PBE"
        password="password" /> ①
</jaas:security-manager>

<model name="jaasModel">
    <service name="jaasService">
        <inbound>
            <jms:inbound-endpoint queue="messages">
                <jaas:jaas-security-filter/>
            </jms:inbound-endpoint>
        </inbound>
        <echo-component/>
    </service>
</model>

```

We declare the password encryption strategy on ① along with the password we want to use. Under the covers, the password encryption strategy uses the Java Cryptographic Extensions' PBEWithMD5AndDES algorithm to decode the encrypted part of the MULE_USER value. The following diagram shows what the new value of MULE_USER looks like after encryption.



We first change Plain to PBE, which matches the name of the password encryption strategy defined on ①. This is followed by the encryption of Plain john::password using PBE with the specified password. When decrypted, the result matches the username, delimiter, and password pair from figure 9.3. The security manager can now decode and authenticate the encrypted credentials.

Sometimes we need to encrypt the entire payload of a message and not just the authentication credentials. This is useful when a secure transport such as SSL or a VPN is unavailable. Let's look at how we can use encryption to secure the contents of our payloads.

9.3.3 Using password-based payload encryption

We saw an example of password-based encryption at the beginning of this chapter. If you recall listing 9.2, we encrypted the payloads of JMS messages using a password. Recipients of these JMS messages would need to use the same password to decrypt the payload and process the contents. In that example, we only showed the encryption side of the payload transformation. Let's now take a look at how to perform end-to-end payload encryption of a message, with both encryption and decryption. Listing 9.10 illustrates a service that accepts a JMS message off a queue, encrypts it, and sends it to another JMS queue that decrypts the message and forwards it off.

Listing 9.10 Encrypting the payload of messages using password-based encryption

```

<security-manager>
    <password-encryption-strategy name="PBE" password="password" /> ←
</security-manager> Define encryption
<model name="pbeModel"> strategy ①

    <service name="pbeInService">
        <inbound>
            <jms:inbound-endpoint queue="messages.in" />
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="services.decryption">
                    <transformers>
                        <encrypt-transformer strategy-ref="PBE" /> ←
                        <jms:object-to-jmsmessage-transformer/>
                    </transformers>
                </jms:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>

    <service name="pbeOutService">
        <inbound>
            <jms:inbound-endpoint queue="services.decryption">
                <transformers>
                    <jms:jmsmessage-to-object-transformer/>
                    <decrypt-transformer strategy-ref="PBE" /> ←
                </transformers>
            </jms:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="messages.out" />
            </pass-through-router>
        </outbound>
    </service>
</model>

```

We configure the password encryption strategy on ① as we have previously. The *encrypt transformer* is then used on ② to encrypt the outbound message payload before it's placed on the JMS queue. When the message is received on ③ it's decrypted by the *decrypt transformer*. Both the encrypt transformer and decrypt transformer refer to the encryption strategy by its name, PBE.

NOTE In addition to message-level encryption, Mule offers SSL and TLS versions of some of the transports we discussed in chapter 3. Here are some of the transports that support SSL or TLS:

- HTTPS
- POP3S
- SMTSPS
- IMAPS
- XMPPS

These transports generally require you to supply information about your certificate keystores. The online Mule documentation will instruct you how to do this for the transport in question.

9.3.4 Decrypting message payloads with the PGP SecurityFilter

We just saw how to use password-based encryption to transparently encrypt message payloads. The main flaw with this sort of encryption is that the password must be shared between both parties wishing to exchange messages. As you share the password with more parties, the risk of it becoming compromised grows. This is a major motivation behind the popularity of *public key encryption*. Public key encryption uses key pairs rather than shared keys. The strength of public key encryption relies on the fact that each user has a closely guarded private key and a widely distributed public key. When a user wants to send a message, he can encrypt the message using the recipient's public key. The recipient is then able to decrypt the message using her private key. The sender is also able to "sign" a message using his private key. The recipient can subsequently verify this message using the sender's public key to guarantee the authenticity of the message.

Robust public key implementations are readily available. Mule uses the Cryptix library's PGP support to perform decryption and signature verification of messages. In addition to this, GNU Privacy Guard (GPG) and OpenPGP are popular client-side alternatives. But using public key encryption securely takes more than software. As public key encryption relies on the authenticity of public keys, having trustworthy processes and policies to disseminate and verify public keys is crucial. This infrastructure is referred to as *PKI*, or *Public Key Infrastructure*. For a small organization, this might consist of hand-delivering public keys to individuals on CD-ROM to ensure authenticity. For a larger authentication, the challenge of key distribution becomes more complex. Implementing a robust PKI is beyond the scope of this book, but numerous dedicated resources, both online and in print, can guide you in the right direction.

BEST PRACTICE Implement a robust PKI to facilitate the sharing of public keys.

PGP is a popular protocol for performing public key encryption. Mule's PGP module supplies an inbound security filter that can be used to verify signatures and decrypt messages. Let's look at using a PGP security filter to accept only PGP messages that we can decrypt and perform successful signature verification on. Listing 9.11 illustrates how to do this.

Listing 9.11 Decrypting PGP-encrypted JMS payloads using a PGP security filter

```
<spring:bean id="pgpKeyManager"
    class="org.mule.module.pgp.PGPKeyRingImpl"
    init-method="initialise">
    <!--
        ① Define pgpKeyManager
    -->
    <spring:property
        name="publicKeyRingFileName"
        value="conf/public.key.gpg"/>
    <!--
        Public key ring
    -->
<spring:property>
```

```

name="secretKeyRingFileName"
value="conf/secret.key.gpg"/>
    ↪ Private key ring

<spring:property
    name="secretAliasId"
    value="0x7927DE78"/>
    ↪ Alias of private key used for decryption

<spring:property
    name="secretPassphrase"
    value="mule"/>
    ↪ Private key pass-phrase

</spring:bean>

<spring:bean id="credentialAccessor"
    class="org.mule.security.MuleHeaderCredentialsAccessor"/>
    ↪ Define credentialAccesor 2

<pgp:security-manager>
    <pgp:security-provider
        name="pgpSecurityProvider"
        keyManager-ref="pgpKeyManager"/>
        ↪ Configure PGP security manager

    <pgp:keybased-encryption-strategy
        name="keyBasedEncryptionStrategy"
        keyManager-ref="pgpKeyManager"/>
        ↪ Configure PGP security manager

</pgp:security-manager>

<model name="pgpModel">
    <service name="pgpService">
        <inbound>
            <jms:inbound-endpoint
                queue="messages.encrypted" />
        </inbound>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint
                    path="services.decryption" />
            </pass-through-router>
        </outbound>
    </service>

    <service name="out">
        <inbound>
            <vm:inbound-endpoint path="services.decryption">
                <pgp:security-filter
                    strategyName="keyBasedEncryptionStrategy"
                    signRequired="true"
                    keyManager-ref="pgpKeyManager"
                    credentialsAccessor-ref="credentialAccessor" />
            </vm:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="messages.decrypted" />
            </pass-through-router>
        </outbound>
    </service>
</model>
```

Declare inbound endpoints

Pass valid messages through to outbound endpoint 3

We start off by defining a key manager to handle the details required for PGP on ①. As you see here, we specify the location of our key rings, the key alias, and secret password. The credential accessor is defined on ②. This instructs Mule how to infer the user who encrypted or signed the payload from the message. In this case we'll use the `MuleHeaderCredentialsAccessor`, which will have Mule use the `MULE_USER` message property to identify who encrypted or signed the message. The decryption and signature verification occur on ③. Messages that can't be decrypted or verified are handled by the exception strategy for the service. Messages whose payloads can be decrypted and verified are passed.²

NOTE If you're using a Sun JVM, you'll most likely need to install the Unlimited Strength Jurisdiction Policy files to use Mule's PGP support. These are available on the Java SE downloads page for the Java version you're using. The Java 5 Unlimited Strength Jurisdiction Policy files, for instance, are located at http://java.sun.com/javase/downloads/index_jdk5.jsp. Installation instructions are included in the archive.

In this section we saw how security filters equip us with the means to verify the authenticity of messages arriving and leaving our endpoints. We first saw how to use Spring Security in conjunction with an HTTP security filter to achieve basic authentication on HTTP endpoints. We then saw how to use JAAS and the JAAS security filter to perform header authentication on JMS messages. We finally saw how to leverage public key encryption to securely decrypt and verify the payloads of Mule messages.

9.4 Summary

We saw in this chapter how Mule simplifies security for integration projects. This simplification is hopefully evident from our discussion of security managers and security providers. You saw that you can use different security managers, such as Spring Security and JAAS, to independently provide security services for endpoint filters. We saw how to use endpoint filters to provide basic authentication for HTTP endpoints, header-based authentication for JMS messages, and public key security for message payloads with PGP.

Now that you're comfortable with security in Mule, we'll spend the next two chapters talking about two other cross-cutting concerns in many integration scenarios—transactions and monitoring.

² You'll need a PGP implementation to generate a key pair and encrypt messages. Good alternatives are Cryptix (<http://www.cryptix.org/>) for programmatic manipulation and GNU Privacy Guard (<http://www.gnupg.org/>) for manipulation via the command line.

10

Using transactions with Mule

In this chapter

- Introducing transactional concepts
- Using single resource transactions
- Using XA transactions with multiple resources

A *transaction*, whether occurring in software or in real life, is a series of events that need to occur in a specified sequence, leaving all participants in some predetermined state. Leaving the software world aside for a moment, let's consider a real-world example: paying for groceries in a supermarket. In such a transaction, the following steps need to occur in the specified order:

- 1 You place your groceries on the counter.
- 2 The cashier scans each item, adding the price of the item to the register.
- 3 The cashier sums the prices of the items and communicates the total for you.
- 4 You present the cashier with a credit card to pay for the groceries.
- 5 The cashier charges your credit card with the amount for the groceries.
- 6 The cashier returns the credit card to you.
- 7 The cashier bags your groceries.
- 8 You take the bag of groceries and leave the supermarket.

This seemingly mundane undertaking is actually a fairly complex orchestration of events. The events need to happen in the order specified—the cashier can't scan your items until you've put them on the counter; he can't total their prices unless he scans them; he won't let you leave the store without paying for them; and so on. The success or failure of each event is also important. If the cashier won't tell you the price of the items, you probably won't give him your credit card. Likewise, if your credit card is declined, the cashier is unlikely to bag your groceries (or let you leave the store with them). In this sense, the preceding transaction is an all-or-nothing proposition. When you walk out of the grocery store, only two states should be possible: you leave with the groceries and their monetary amount charged to your credit card, or you leave without any groceries and your credit card is uncharged.

Similar scenarios are present in software applications. Updating related data in a database, for instance, usually requires that all or none of the tables are updated. If some failure occurs halfway through the database update, then the database is left in an inconsistent state. To prevent this state, you need some mechanism to *roll back* the data that has been updated to the point of failure. This makes the database update *atomic*—even though a sequence of disconnected events are taking place (the updating of different tables), they're treated as a single operation that's either completely successful or completely rolled back.

From this atomicity we can also start to make assumptions about consistency. Because the database operations are treated in a singular fashion, the database is guaranteed to be in defined states whether the transaction has completed or failed, making the operation consistent.

While a transaction is taking place, it's important other transactions aren't affected. This is closely related to consistency. If another process were querying a table while the database update we discussed was occurring, and then the update was subsequently rolled back, the other process might've read data that's now invalid. This can be avoided, for instance, by not only rolling back the failed database updates but also by locking the tables being updated. We refer to such behavior as being *isolated*.

Transactions must also be permanent in nature. If the cashier's card reader tells him your card has been charged, but in reality it hasn't, then the grocery store is out the cost of your groceries. Conversely, if the cashier hides some of your groceries under the counter to take home with him after his shift, you've been charged for goods you haven't received. In both cases we want to make sure the transaction has been completely applied to each resource being affected. When this is guaranteed, the transaction is referred to as *durable*.

These four properties—atomicity, consistency, isolation, and durability—when taken together form the acronym *ACID*. This term is commonly used to discuss the transactions we'll be examining in this chapter. Such transactions can play an important role in integration scenarios—the nature of distributed data and systems often necessitates their use. Unfortunately, dealing with transactions programmatically can be esoteric and error-prone. Mule, thankfully, makes it easy to declare transactional

properties on your endpoints. We'll see in this chapter how to add transactional properties to our Mule services. We'll be covering the two major types of transactions supported by the Java and JEE ecosystems—single and multiple resource transactions. We'll start off by examining how single resource transactions let us operate on a single resource, such as a database or JMS broker, transactionally. We'll then see how we can use transactions across multiple resources. Finally, we'll look at how we can use exception strategies in conjunction with transactions to provide custom rollback and commit behavior. As usual, we'll examine each of these features through the lens of Clood, Inc. You'll see how Mule's transactional support augments the reliability of the integration projects.

10.1 Using transactions with a single resource

Let's start off by examining how to operate on a single resource transactionally with Mule. A single resource transaction implies a set of operations executed on a single provider, such as a particular database or JMS broker. In the context of Mule, transactions of this sort will occur on or across endpoints using the same connector. For instance, you might use a single-resource JMS transaction on a JMS inbound endpoint or a single-resource JDBC transaction on a JDBC outbound endpoint. Single-resource transactions can also be used across inbound and outbound endpoints, provided the underlying connector is the same. You could, for instance, accept a JMS message on an inbound endpoint and send the message to multiple JMS queues using a multicasting router on an outbound endpoint. Assuming the queues involved were all hosted on the same JMS broker, a failure in sending the message to one of the remote JMS queues could trigger a rollback of the entire operation, up to and including the message being received on the inbound endpoint.

In this section we'll start off by looking at how to operate on JDBC endpoints transactionally. We'll then see how we can use these same techniques to consume and send JMS messages in transactions.

10.1.1 Using JDBC endpoints transactionally

Being able to operate transactionally against a database is critical for many applications. The nature of relational databases usually means that data for a single business entity is stored across numerous tables—joined to each other with foreign key references. When this data is updated, care must be taken that every required table is updated or it isn't. Anything else could leave the data in an inconsistent state. Implicit transactional behavior can also be required of inserts to a single table. Perhaps you want a group of insert statements to occur atomically to ensure that selects against the table are consistent.

Clood, Inc., extensively uses relational databases for its day-to-day operations. We saw in chapter 3 how Clood was using monitoring data from emails to populate a table of alert data. To augment this data, Clood's operations team has deployed a performance monitoring application to run against Clood's clients' web applications. This

application periodically runs a series of tests against a client's web site and writes the results of the tests, represented as XML, to a file. The contents of this file are then sent at a certain interval to a JMS topic for further processing.

Mule is being used to accept this data and persist it to Cloud's monitoring database. The payload of these messages must be persisted to a database in an all-or-nothing manner. If any of the row inserts fail then the entire transaction should be rolled back. This ensures the monitoring data for a given client is consistent when it's queued by a web-facing analytics engine. Let's see how to use a JDBC outbound endpoint to enforce this behavior. Listing 10.1 illustrates how Cloud accomplishes this.

NOTE For transactions with MySQL to work properly, they must be supported by an underlying engine that supports transactions, such as InnoDB.

Listing 10.1 Using a JDBC outbound endpoint transactionally

```

<spring:beans>
    <spring:import resource="spring-config.xml" />
</spring:beans>

<jms:activemq-connector
    name="jmsConnector"
    specification="1.1"/>
<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="statsInsert"
        value="" style="text-decoration: underline; color: blue; font-weight: bold;">1 Declare INSERT statement
    INSERT INTO PERF_METRICS VALUES
        (0,#[map-payload:CLIENT_ID], 'AVG_RESPONSE_TIME',
         #[map-payload:AVG_RESPONSE_TIME],
         #[map-payload:TIMESTAMP]),
        (0,#[map-payload:CLIENT_ID], 'MED_RESPONSE_TIME',
         #[map-payload:MED_RESPONSE_TIME],
         #[map-payload:TIMESTAMP]),
        (0,#[map-payload:CLIENT_ID], 'MAX_RESPONSE_TIME',
         #[map-payload:MAX_RESPONSE_TIME],
         #[map-payload:TIMESTAMP])
    " />
</jdbc:connector>

<model name="URLAlertingModel">
    <service name="URLAlertingService">
        <inbound>
            <jms:inbound-endpoint topic="monitoring.performance" />
        </inbound>
        <component class="com.cloud.monitoring.URLMetricsComponent" />
        <outbound>
            <pass-through-router>
                <jdbc:outbound-endpoint queryKey="statsInsert" style="text-decoration: underline; color: blue; font-weight: bold;">2 Execute INSERT statement


```

Our insert statement is declared on ①. It inserts data into the `PERF_METRICS` table using data from the map populated by the `URLMetricsComponent`. `URLMetricsComponent` builds this map from the JMS message received on the `monitoring.performance` topic. The insert statement is then executed on ②.

The action parameter on ③ tells Mule how to initiate the transactional behavior. The `ALWAYS_BEGIN` value here indicates that the inserts should begin in a new transaction, independent of any other transactions that might be present. Table 10.1 lists the valid action values for a Mule transaction.

Table 10.1 Available options for configuring a transaction action

| | Description |
|------------------|--|
| NONE | Never participate in a transaction. |
| ALWAYS_BEGIN | Always start a new transaction, committing any previously existing transaction. |
| BEGIN_OR_JOIN | If there's an existing transaction, join that transaction. If not, start a new transaction. |
| ALWAYS_JOIN | Always expect and join an existing transaction. Throws an exception if no previous transaction exists. |
| JOIN_IF_POSSIBLE | Join an existing transaction if one exists. If no transaction exists, run nontransactionally. |

Since the JDBC outbound endpoint in listing 10.1 isn't participating in any other transaction, we're naturally using `ALWAYS_BEGIN` to start a new transaction for the insert. Now that we've seen how transactions work with the JDBC transport, let's look at how we can send and receive JMS messages transactionally.

10.1.2 Using JMS endpoints transactionally

JMS messaging can also be performed transactionally. Transactions on JMS inbound endpoints ensure that JMS messages are received successfully. Messages that aren't received successfully are rolled back. When a JMS transaction is rolled back, the JMS message is put back on the queue with the `JMSRedelivered` header set, allowing a different receiver to attempt to consume the message. A count is also incremented for the message, which when used in conjunction with the `maxRedelivery` property of the JMS connector can limit the number of times delivery for the message is attempted. A committed transaction on a JMS outbound endpoint indicates that the message was sent successfully and the message reception is acknowledged. As we'll see shortly, JMS transactions can be used in conjunction with the multicasting router, allowing multiple messages to be dispatched in the same transaction. First, let's see how to accept JMS messages transactionally.

In listing 4.9 in chapter 4, we saw how to use the forwarding-consumer router to perform selective enrichment of messages. Messages with a payload containing a

status of OK or SUCCESS were forwarded along to an outbound endpoint, whereas non-conforming messages were passed through the component for correction. Let's assume that Clood, Inc., wants to start using this service in a production manner for important data, such as messages containing order or provisioning information. Clood initially wants to be sure that messages are acknowledged by the JMS inbound endpoint and processed successfully by the `messageProcessor`. To accomplish this, Clood will receive JMS messages off the messages queue transactionally. Listing 10.2 demonstrates this, ensuring either that the message is successfully received by the endpoint and processed by the component or the transaction is rolled back (the message is placed back on the queue).

Listing 10.2 Sending a JMS message transactionally

```
<model name="forwardingConsumerModel">
    <service name="forwardingConsumerService">
        <inbound>
            <jms:inbound-endpoint queue="messages">
                <jms:transaction action="ALWAYS_BEGIN"/>
            </jms:inbound-endpoint>
            <forwarding-router>
                <regex-filter pattern="^STATUS: (OK|SUCCESS)$"/>
            </forwarding-router>
        </inbound>
        <component>
            <spring-object bean="messageProcessor"/>
        </component>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="processed.messages"/>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

The diagram illustrates the flow of a JMS message transaction. It starts with an arrow labeled "1 Accept JMS message in transaction" pointing from the "messages" queue to the "inbound" section of the configuration. From there, an arrow labeled "2 Enrich message" points to the "component" section, where the `messageProcessor` Spring object is defined. Finally, an arrow points from the "component" section to the "pass-through-router" section, indicating the message's path to the "processed.messages" queue.

Messages are received off the `messages` queue on ①. If a message has a payload that matches the regular expression defined by the regex filter, it's passed over the component on ② and sent directly to the pass-through router for outbound routing. If a message doesn't have a payload matching the regular expression, it's processed by the `messageProcessor` Spring object defined on ②. If a failure occurs on the JMS inbound-endpoint, the message isn't acknowledged and is placed back on the queue for redelivery.

Failures in the component will also influence the transaction committing or rolling back. Unless overridden by the methods discussed in chapter 8, Mule will use the default exception strategy to process the exception thrown by the component. Such a failure will, by default, trigger a rollback on the transaction. We'll see later in this chapter how we can override this behavior and trigger a commit based on the type of exception thrown by a component.

Once the message is passed to the pass-through router, either from the forwarding router or from passing through the `messageProcessor`, it's routed to the processed.

messages queue via the JMS outbound endpoint. But what happens if there's a failure on the JMS dispatch to the `processed.messages` queue? We'd hardly be honoring the business case if we went through the trouble of correcting the inbound message only to lose it once it's routed. To solve this problem, let's see how we can have the outbound endpoint join the transaction created on ①. Once the outbound endpoint is joined in the transaction started on ①, the transaction will span message reception on the inbound endpoint, processing by the component, and dispatch by the outbound endpoint. A failure in dispatching the message will trigger a rollback, ensuring that the message received on ① will be rolled back and the JMS message placed back on the queue for subsequent redelivery. Listing 10.3 illustrates how to do this.

Listing 10.3 Using `ALWAYS_JOIN` to join in an existing transaction

```
<model name="forwardingConsumerModel">
    <service name="forwardingConsumerService">
        <inbound>
            <jms:inbound-endpoint queue="messages">
                <jms:transaction action="ALWAYS_BEGIN"/>
            </jms:inbound-endpoint>
            <forwarding-router>
                <regex-filter pattern="^STATUS: (OK|SUCCESS)$"/>
            </forwarding-router>
        </inbound>
        <component>
            <spring-object bean="messageProcessor"/>
        </component>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="processed.messages">
                    <jms:transaction action="ALWAYS_JOIN"/>
                </jms:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

By specifying the transaction action as `ALWAYS_JOIN` on ②, we're specifying that the sending of the JMS message will join the transaction started on ① as we just described.

It's also possible to send multiple messages transactionally. Using a multicasting router with JMS outbound endpoints along with a JMS transaction will cause the messages sent by the JMS outbound endpoints to either all be committed or all be rolled back atomically. Let's see how this functionality is useful for Clood, Inc. Clood currently collects various real-time analytical data about its customers' web applications. Web application response times, network metrics, and billing data are all collected from Clood's data centers and published to JMS queues for processing by Clood's Mule instances. This data must ultimately be processed and fed into Clood's operational database as well as its data warehouse. In order to decouple the operational database and the data warehousing, Clood has decided to republish the data in an appropriate format for each destination. The resulting data is then placed on a

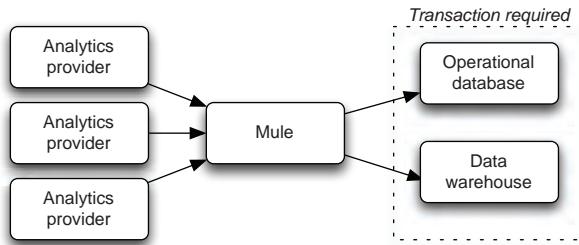


Figure 10.1 Cloud, Inc.'s approach for decoupled data management

dedicated JMS queue where it's consumed and saved. Since the operational and data warehouse data must be in sync with each other, it makes sense to group the publishing of this data atomically in a transaction. Figure 10.1 illustrates this.

Listing 10.4 shows the corresponding Mule configuration.

Listing 10.4 Using the multicasting router transactionally

```

<service name="transactedMulticastingRouterService">
    <inbound>
        <jms:inbound-endpoint topic="application-response-times">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:inbound-endpoint>
        <jms:inbound-endpoint topic="network-metrics">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:inbound-endpoint>
        <jms:inbound-endpoint topic="billing-statistics">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:inbound-endpoint>
    </inbound>
    <component>
        <spring-object bean="analyticsService" />
    </component>
    <outbound>
        <multicasting-router>
            <jms:outbound-endpoint
                queue="operational-database">
                <jms:transaction
                    action="ALWAYS_BEGIN"
                    timeout="300000"/>
            </jms:outbound-endpoint>
            <jms:outbound-endpoint
                queue="data-warehouse">
                <jms:transaction
                    action="ALWAYS_BEGIN"
                    timeout="300000"/>
            </jms:outbound-endpoint>
        </multicasting-router>
    </outbound>
</service>
  
```

1 Accept JMS messages from analytics providers

2 Process analytical data

3 Define the multicasting router

4 Send analytical data transactionally

Send analytical data transactionally

The JMS endpoints defined on ① will accept messages from each analytics provider transactionally. This data will be processed by the `analyticsService` Spring object configured on ②. If there's a failure on the JMS endpoints, the transaction will be rolled back. Exceptions thrown by the `analyticsService` will also trigger a rollback of the transaction. The processed message will then be passed to the multicasting router defined on ③. The JMS transaction defined on ④ will ensure that the message is sent to both JMS queues successfully. If there's a failure on either queue, then the transaction on ④ won't begin and the message will be lost. The `timeout` value defined on ④ specifies how many milliseconds to wait before rolling back the transaction. The default is usually acceptable for the transport in question. In this case, we're overriding the default and explicitly setting a value of 5 minutes for each endpoint.

In order to ensure messages aren't lost by such a failure, we can make this entire message flow transactional, from JMS inbound endpoints to JMS outbound endpoints. Listing 10.5 shows how to do this.

Listing 10.5 Making an entire message flow transactional

```
<service name="metricsService">
    <inbound>
        <jms:inbound-endpoint topic="application-response-times">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:inbound-endpoint>
        <jms:inbound-endpoint topic="network-metrics">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:inbound-endpoint>
        <jms:inbound-endpoint topic="billing-statistics">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:inbound-endpoint>
    </inbound>
    <component>
        <spring-object bean="analyticsService" />
    </component>
    <outbound>
        <multicasting-router>
            <jms:outbound-endpoint
                queue="operational-database">
                <jms:transaction
                    action="ALWAYS_JOIN" timeout="600000"/>
            </jms:outbound-endpoint>
            <jms:outbound-endpoint
                queue="data-warehouse">
                <jms:transaction
                    action="ALWAYS_JOIN" timeout="600000"/>
            </jms:outbound-endpoint>
        </multicasting-router>
    </outbound>
</service>
```

① Join previously existing transaction

Join previously existing transaction

We've changed the transaction action from `ALWAYS_BEGIN` to "`ALWAYS_JOIN`" on ①. Now a failure on either queue in the multicasting router will cause the transaction to

roll back up to the message reception on the inbound endpoint. Such a configuration will make the transaction resilient against a failure on one of the queues. For instance, if Mule doesn't have the appropriate rights to access the data-warehouse queue, then the entire operation will roll back to message reception on the JMS inbound endpoint.

In this section, we saw how to use transactions with a single resource, such as a single database or JMS provider. It's also possible to run transactions across multiple resources, such as two databases or a database and JMS provider. Let's investigate Mule's support for that now.

NOTE The VM transport can also be used transactionally. For instance, to begin a new VM transaction instead of a JMS transaction in listing 10.5, you'd use the `vm:transaction` element as follows:

```
<outbound>
    <multicasting-router>
        <vm:outbound-endpoint
            path="operational.database">
            <vm:transaction
                action="ALWAYS_JOIN"/>
        </vm:outbound-endpoint>
        <vm:outbound-endpoint
            path="data.warehouse">
            <vm:transaction
                action="ALWAYS_JOIN"/>
        </vm:outbound-endpoint>
    </multicasting-router>
</outbound>
```

10.2 Using multiple resource transactions

Performing transactions on a single resource is appropriate when the operations to be conducted transactionally all use the same connector. We saw examples of this in the previous section, where we showed how you can use the JDBC and JMS transports transactionally. What if the operations you wish to group atomically span more than one resource? Perhaps you need to accept a JMS message on an inbound endpoint, process it with a component, and then save the message payload to a database with a JDBC outbound endpoint. You want to make this operation transactional so that failures in either the JMS endpoint or the JDBC endpoint trigger a rollback of the entire operation. Figure 10.2 illustrates this scenario.

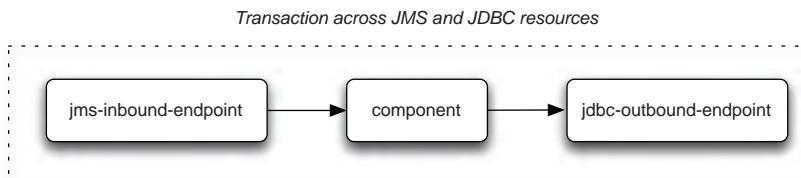


Figure 10.2 Performing a transaction across multiple resources

The XA standard is a distributed transaction protocol designed to meet this need. For resources that support XA transactions, such as many JDBC drivers and JMS providers, this is possible through use of the Java Transaction API. An XA transaction uses the *two-phase commit* (2PC) protocol to ensure all participants in the transaction commit or roll back. During the first phase of the 2PC, the transaction manager issues a PREPARE command to each transaction participant. The participants “vote” during this call to indicate whether or not the transaction can be committed. In the second phase on the 2PC, if any of the participants indicates that its portion of the transaction can’t be committed, the transaction manager instructs each participant to roll back. If each participant can commit the transaction, the transaction manager instructs them each to do so and the transaction is completed.

To take advantage of XA transactions, use of a specific driver is often required. JMS and JDBC providers usually provide connection factories or data sources prefixed with *XA* to differentiate them. You’ll need to consult the documentation for your provider and see what these differences are.

XA transactions can be complex beasts. As we just mentioned, you usually need to use different JDBC or JMS drivers that specifically have XA support. Resources in XA transactions can also make decisions about rolling back a transaction outside the scope of the transaction manager. These scenarios, often caused by locking or network issues, cause `HeuristicExceptions` to be thrown. You should be aware of these exceptions and configure your exception strategies accordingly. Finally, XA transactions can introduce scalability issues when locking occurs in XA participants. Be aware of this when deciding to use XA transactions in your projects.

BEST PRACTICE Exercise caution when using XA transactions, as they can have adverse scalability, complexity, and performance impacts on your projects.

We’ll see in this section how Mule uses the Java Transaction API (JTA for short) to allow you to declaratively configure such transactions via XML. We’ll start off by looking at how to perform standalone XA transactions using JBoss Transactions. We’ll then see how to access a transaction manager when running Mule embedded in an application running in a container, such as an application server or servlet container.

NOTE MuleEE offers support for Multi-TX. This allows you to span transactions across multiple resources without the overhead of JTA. Consult the MuleEE documentation for more information.

10.2.1 Spanning multiple resources with JBossTS

It used to be that running JTA transactions required the use of a JEE application server, such as JBossAS or WebLogic, or a standalone, commercial JTA implementation. Thankfully there are now open source JTA implementations that don’t require an application server or expensive proprietary solution. One such implementation, which is supported out of the box by Mule, is JBossTS. Let’s see how we can use Mule’s JBossTS support to improve Clood, Inc.’s data warehousing service.

We saw in listing 10.5 how Clood was transactionally receiving and republishing analytical data using JMS inbound and outbound endpoints. Two separate transactions were occurring in this scenario. The first transaction was spanning the message reception and subsequent component processing. The second transaction was spanning the publishing of each JMS message to a queue by the multicasting router.

This approach is robust enough if the providers only care that their JMS messages are received by Mule. It's less appropriate if the provider needs to be sure whether the entire action was successful. This might be the case for the data on the billing endpoint. In this case, a provider wants to be certain that the billing data is saved to the operational database and the data warehouse successfully. To support this, Clood has added another service that's dedicated to receiving billing data. Figure 10.3 illustrates the new service, which forgoes the JMS outbound endpoints and will write to the database and data warehouse directly.

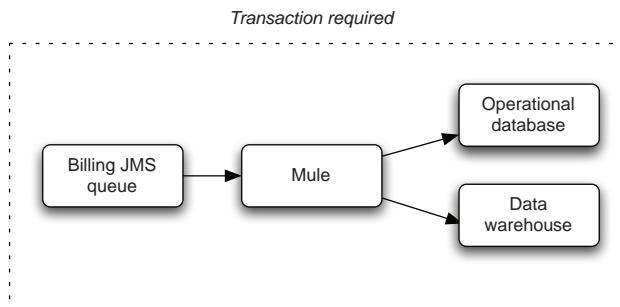


Figure 10.3 Wrapping billing data reception into a single transaction

Assuming Clood's JMS provider and its JDBC drivers for the database and data warehouse support XA transactions, we can use Mule's JBossTS support to wrap this entire operation in a single transaction. Listing 10.6 illustrates how Clood has accomplished this.

Listing 10.6 Sending outbound messages to list of endpoints using an XA transaction

```

<jms:activemq-xa-connector name="jmsConnector"
                           specification="1.1" />
  
```

1 Define ActiveMQ XA connector for JMS data source

```

<jdbc:connector name="operationalDb"
                dataSource-ref="operationalDataSource">
    <jdbc:query key="operationalBillingInsert"
                value="
                    INSERT INTO billing_stats VALUES (0,#[map-payload:STAT]);"/>
  
```

2 Define JDBC connector for operational data source

```

</jdbc:connector>
  
```

```

<jdbc:connector name="warehouseDb"
                dataSource-ref="warehouseDataSource">
    <jdbc:query key="warehouseBillingInsert"
                value="
                    INSERT INTO billing_stats VALUES (0,#[map-payload:STAT]);"/>
  
```

3 Define JDBC connector for warehouse data source

```

</jdbc:connector>
  
```

```

<jbossts:transaction-manager/>                                ← Use JBossTS to
<model name="BillingModel">                                         manage XA transactions
    <service name="BillingService">
        <inbound>
            <jms:inbound-endpoint queue="billing-data">
                <xa-transaction action="ALWAYS_BEGIN"
                    timeout="60000"/>                         ← Begin new XA
            </jms:inbound-endpoint>                            transaction
        </inbound>
        <component class="com.cloud.billing.BillingService"/>
        <outbound>
            <messaging-router>
                <jdbc:outbound-endpoint
                    connector-ref="operationalDb"
                    queryKey="operationalBillingInsert">
                    <xa-transaction action="ALWAYS_JOIN"/>           ← Expect and
                </jdbc:outbound-endpoint>                      join existing
                <jdbc:outbound-endpoint
                    connector-ref="warehouseDb"
                    queryKey="warehouseBillingInsert">
                    <xa-transaction action="ALWAYS_JOIN"/>           ← Expect and
                </jdbc:outbound-endpoint>                      join existing
            </messaging-router>
        </outbound>
    </service>
</model>

```

We start off by defining an ActiveMQ connector that supports XA transactions on ①. Since our database and data warehouse require separate data sources, we need to configure two JDBC connectors to reference each data source. These are configured on ② and ③. The dataSources for both of these connectors, configured in Spring, should support XA transactions. We specify that we're using JBossTS to manage the XA transactions on ④. Specifying ALWAYS_BEGIN will cause the JMS inbound endpoint to start an XA transaction and attempt to receive a message off the billing-data queue. If a message is received within a wait period, the message is processed by the component and routed to the outbound endpoints in the same transaction. A failure in either will trigger the transaction to roll back. If no message is received in the wait period, then the transaction is also rolled back and a new transaction is started. This polling is to ensure that messages received by the endpoint are within the XA transaction. The message is processed by the component and sent to the multicasting router.

NOTE Using ALWAYS_BEGIN in an XA context will suspend an existing transaction. Once the new transaction has completed, the previous transaction will be resumed.

The XA transaction configuration on ④ has an action of ALWAYS_JOIN. This means it expects a previous XA transaction to be open and will join to it (as opposed to committing the previous transaction and starting a new one). The message will then be written to both JDBC outbound endpoints. A failure in either the JMS message

reception or either of the JDBC outbound endpoints will trigger the XA transaction to roll back. This will trickle back up to the JMS inbound endpoint and cause the JMS provider to engage in redelivery of the message, using the semantics we discussed for JMS single resource transactions.

Let's now look at how Cloud could modify this configuration to run in one of their application servers.

10.2.2 Using XA transactions in a container

If you're running Mule embedded in an application that's deployed in a container, such as an application server or Servlet container, you have the option to use the container's JTA implementation (if one exists). As we mentioned previously, many of the popular JEE application servers ship with JTA implementations. Mule facilitates using these implementations by providing a `*-transaction-manager` configuration element. This lets you specify a `LookupFactory` to locate the appropriate JTA transaction manager for your environment. Table 10.2 lists the supported application servers along with their associated configuration elements.

Table 10.2 Transaction manager lookup factories

| Application server | Configuration element |
|--------------------|---|
| JBoss AS | <code><jboss-transaction-manager/></code> |
| JRun | <code><jrun-transaction-manager/></code> |
| Resin | <code><resin-transaction-manager/></code> |
| Weblogic | <code><weblogic-transaction-manager/></code> |
| WebSphere | <code><websphere-transaction-manager/></code> |

Listing 10.6 assumed we were running Mule standalone and as such were leveraging JBossTS outside of any JBossAS context. Let's now assume that the Mule configuration in listing 10.6 is running inside as a deployed WAR application in Resin. To use Resin's JTA implementation, we'd simply replace the `jbossts:transaction-manager` element with Resin's as illustrated in listing 10.7.

Listing 10.7 Using an application server's transaction manager

```

<jms:activemq-xa-connector
    name="jmsConnector"
    specification="1.1"/>

<jdbc:connector name="operationalDb"
    dataSource-ref="operationalDataSource">
    <jdbc:query key="operationalBillingInsert"
        value="
            INSERT INTO billing_stats VALUES (0,#[map-payload:STAT]);"/>
</jdbc:connector>

```

```
<jdbc:connector name="warehouseDb"
    dataSource-ref="warehouseDataSource">
    <jdbc:query key="warehouseBillingInsert"
        value="">
        INSERT INTO billing_stats VALUES (0,#[map-payload:STAT]);" />
    </jdbc:connector>

<resin-transaction-manager/>           ←
                                         Use Resin's
<model name="BillingModel">           JTA support
    <service name="BillingService">
        <inbound>
            <jms:inbound-endpoint queue="billing-data">
                <xa-transaction action="ALWAYS_BEGIN"
                    timeout="60000"/>
            </jms:inbound-endpoint>
        </inbound>
        <component class="com.clood.billing.BillingService"/>
        <outbound>
            <messaging-router>
                <jdbc:outbound-endpoint
                    connector-ref="operationalDb"
                    queryKey="operationalBillingInsert">
                    <xa-transaction action="ALWAYS_JOIN"/>
                </jdbc:outbound-endpoint>
                <jdbc:outbound-endpoint
                    connector-ref="warehouseDb"
                    queryKey="warehouseBillingInsert">
                    <xa-transaction action="ALWAYS_JOIN"/>
                </jdbc:outbound-endpoint>
            </messaging-router>
        </outbound>
    </service>
</model>
```

If you need access to a JTA provider that isn't explicitly supported by Mule, you can use the jndi-transaction-manager. This allows you to specify the JNDI location of a JTA implementation for Mule to use. For instance, to access a JTA implementation with the JNDI name of `java:/TransactionManager` you'd use the following transaction manager configuration:

```
<jndi-transaction-manager jndiName="java:/TransactionManager" />
```

So far we've seen how to perform transactions against single and multiple resources. We still haven't seen what to do when exceptions are thrown by components. Let's look at how we can leverage the error handling techniques we learned in chapter 8 to deal with exceptions when they arise in components.

10.3 Managing transactions with exception strategies

In chapter 8 we saw how Mule uses exception strategies to take action when unexpected events occur. In this section we'll see how to use exception strategies in conjunction with Mule's transaction support. First we'll see how we can use the default-service exception strategy to roll back transactions when an exception is thrown by a

component. We'll then see how we can use the default-connector exception strategy to commit transactions when there's a failure in a connector.

10.3.1 Handling component exceptions

We saw previously that, by default, the default-service exception strategy will wrap exceptions thrown by a component and log them. Mule will then stop subsequent processing of the message (the outbound routers won't be invoked). This is the behavior we'd expect from our discussion of exception strategies in chapter 8. In the context of a transaction, a failure in the component will cause the transaction to be rolled back. This is usually the behavior we want—reverting the system to the state it was in before the transaction was started.

With a transport like JMS, though, this rollback state will cause redelivery attempts to occur if certain headers are present on the JMS message. If you know the exception causing the rollback is unrecoverable, you may want to commit the transaction to prohibit the message from a redelivery attempt. There also may be business logic that requires you to commit the transaction. For instance, if the received messages have invalid credentials, it's probably appropriate to stop further message processing and commit the transaction. You also may want to forward this message to an error queue for further analysis. Mule allows you to configure this behavior by specifying commit and rollback semantics on the default-service exception strategy in a model. Listing 10.8 illustrates how Cloud uses this to commit transactions when a component throws a com.cloud.BusinessException.

Listing 10.8 Using an exception strategy to commit a transaction

```
<model name="forwardingConsumerModel">

    <default-service-exception-strategy>
        <commit-transaction
            exception-pattern="com.cloud.BusinessException"/>
    </default-service-exception-strategy>

    <service name="forwardingConsumerService">
        <inbound>
            <jms:inbound-endpoint queue="messages">
                <jms:transaction action="ALWAYS_BEGIN"/>
            </jms:inbound-endpoint>
            <forwarding-router>
                <regex-filter pattern="^STATUS: (OK|SUCCESS)$"/>
            </forwarding-router>
        </inbound>
        <component>
            <spring-object bean="messageEnricher"/>
        </component>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="processed-messages">
                    <jms:transaction action="ALWAYS_JOIN"/>
                </jms:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>
</model>
```



```

        </pass-through-router>
    </outbound>
</service>
</model>
```

The default-service exception strategy has a commit-transaction element configured on ①. The exception-pattern defines what exceptions will trigger a transaction to commit. Other types of exceptions will be handled as usual and rolled back. You can also specify exception patterns at a package level using wildcards. You could, for instance, set the exception pattern to com.clood.exception.* and commit any open transaction when an exception in the com.clood.exception.* package is thrown.

The default-connector exception strategy also provides facilities for dealing with exceptions and transactions. Let's look at those now.

10.3.2 Committing transactions with an exception strategy

You can configure the default-connector exception strategy to influence transactional behavior when exceptions are thrown by a connector. This is useful if you want to commit a transaction that would otherwise be rolled back. Let's see how this might be useful in the context of Clood's JMS to JDBC XA transaction configuration we discussed in listing 10.6. Perhaps some inconsequential exception is being thrown by the data warehouse's JDBC driver. Rather than roll back the entire XA transaction when these exceptions crop up, you instead decide to catch the exception, log it to an error queue, and commit the transaction. Mule supports this behavior via the commit-transaction element. Listing 10.9 illustrates how to do this.

Listing 10.9 Using an exception strategy to commit a transaction

```

<model name="BillingModel">
    <service name="BillingService">
        <inbound>
            <jms:inbound-endpoint queue="billing-data">
                <xa-transaction action="ALWAYS_BEGIN" timeout="60000" />
            </jms:inbound-endpoint>
        </inbound>
        <component class="com.clood.billing.BillingService"/>
        <outbound>
            <static-recipient-list-router>
                <xa-transaction action="ALWAYS_JOIN" />
                <recipients>
                    <recipients>
                        <spring:value>jdbc://operationalDb</spring:value>
                        <spring:value>jdbc://warehouseDb</spring:value>
                    </recipients>
                </recipients>
            </static-recipient-list-router>
        </outbound>
        <default-connector-exception-strategy>
            <commit-transaction
                exception-pattern=
                    "com.db.jdbc.NotAReallyImportantException" />
        </default-connector-exception-strategy>
    </service>
</model>
```

Commit transaction
if specified exception
is thrown

```
<jms:outbound-endpoint queue="errors" />
  </default-connector-exception-strategy>
</service>
</model>
```

If a `NotAReallyImportantException` instance is thrown by the JDBC connector, the default-connector exception strategy will log the exception to the errors queue and commit any open transaction.

In this section you saw how the exception strategies we introduced in chapter 8 can be used to impact transactional behavior. We saw how the default-service exception strategy can be used to commit transactions in components. This allows you to avoid rollbacks when such a behavior would violate a business case or cause adverse effects on a service. We saw how similar behavior can be used with the default-component exception strategy, allowing us to override the default transaction behavior when exceptions occur on connectors and endpoints.

10.4 Summary

Transactions play a critical role when grouping otherwise distinct operations together atomically. They can be indispensable in an integration scenario where the nature of such operations is often distinct. We saw in this chapter how Mule makes this potentially difficult task straightforward. By making minor modifications to an endpoint's configuration, a range of transactional behavior can be enabled. This behavior can be used with single-resource transactions or, by using Mule's JTA support, transactions using multiple resources. Mule allows exception strategies to partake in the transactional flow by committing or rolling back a transaction based on the exception thrown by a component or connector. This, too, is easily configurable by modifying an exception strategy's XML configuration.

Now let's take a look at another critical element of running Mule in production: monitoring.

Monitoring with Mule

In this chapter

- Using standard tools to monitor Mule instances
- Strategies for auditing the ESB
- Building human-friendly dashboards

Whether you use Mule to bridge systems together or to directly expose services, business activities and processes throughout your company will soon rely on the availability of your Mule instances. As an intermediation tier, it's common for Mule to become a critical actor in the IT landscape. In chapter 7, we reviewed different approaches for running Mule in an highly available manner. In this chapter, we'll look at another important aspect of running Mule in production: monitoring. The ability to trigger alerts before end users or business processes start to suffer is an immediate benefit of monitoring. But it also helps on other related topics such as SLA enforcement, operational reporting, and audit trail (for legitimate or unauthorized activities).

In the coming sections, we'll review different aspects and techniques involved in monitoring Mule instances. We'll first consider the need to perform health checks on these instances. We'll then look at the different options available to track the activity that occurs in these instances. And we'll close with a mention of dashboards, the human-facing part of monitoring. While doing so, we'll look at the different

ways Cloud, Inc., monitors its applications and those of its clients, including the particular web dashboard they've created.

NOTE Our focus is on monitoring instances of Mule Community Edition. The Enterprise Edition (EE) of Mule comes complete with MuleHQ, a centralized monitoring platform based on HypericHQ. MuleHQ goes further than monitoring Mule instances, as it's able to monitor all the systems in an IT landscape. Mule EE and MuleHQ are beyond the scope of this book.

While reading this chapter, you'll realize that *management agents* are prominent actors of Mule monitoring. The agents are responsible for exposing internal state or events of a Mule instance to the outside world. They exist under a wide variety of implementations, where each gives access to Mule's internals in a different manner. By default, Mule doesn't activate any agent. This is why you may have noticed the following line in the startup sequence of our first example shown in 2.1:

```
Agents Running: None
```

Mule's management module contains dozens of agents. Who are these agents and what are they capable of? This information isn't classified, so we'll be able to discuss it publicly. Instead of listing the agents one by one, we'll discover them, and what they're good for, while we review the different aspects of monitoring that we'll cover in this chapter.

So let's start by reviewing some approaches to assess the health of your Mule.

11.1 Checking health

Monitoring systems with periodic health checks is a standard practice in the industry. Like checking the health of a living thing, checking the health of a software system implies taking measures of predetermined parameters at regular time intervals. Most of the time, these parameters are analog by nature (such as pulse, blood pressure, or temperature). For these, thresholds and trends provide insight on the sanity of the system. For example, we can check whether the CPU load is above a predefined alert limit or whether the memory usage is constantly increasing. Sometimes these parameters are truly boolean. For those, it's either all of nothing, as is the case for a network ping, where we're only concerned with the presence of the remote host (and not the response time).

Checking the health of a Mule instance can be performed at different levels:

- It runs on a network-attached server, so it must be monitored at network and system levels.
- It's a Java application, so ensuring the JVM that hosts it is in good health is essential.¹
- It's a Mule instance, so there are specific moving parts of the ESB that you want to keep an eye on.

¹ For an extensive discussion of this subject, refer to "Java run-time monitoring," from Nicholas Whitehead: <http://www.ibm.com/developerworks/library/j-jrtm1>.

To perform these checks, you can use different tools and technologies. Sometimes, these technologies overlap in their capacities: for example, SNMP enables both system-level and JVM-level checks. JMX can also be used to monitor both the JVM and Mule itself. You'll likely end up with a mix of these technologies, using the ones that you're familiar with or for which you have some preexisting tools.

We'll now review some approaches that you can follow to perform systematic health checks of your Mule instances for the different levels we've just mentioned.

11.1.1 Checking health at network level

One of the most basic health checks you can perform is to ensure that a system is alive. The easiest way to check that a Mule instance is up and running is to perform health checks at network level. This is usually achieved by exposing a basic service that returns a fixed value and “ping” it for availability. Of course, this doesn't reveal the actual state of the whole Mule instance, as the ping service can be the only one active, but for a basic check, this is all we need.

Network-level checks are often required by peripheral systems such as load balancers. These systems usually perform regular calls on the servers they manage, and if a predefined response isn't received on time, they can decide that a particular server isn't responsive. This is illustrated in figure 11.1: the load balancer calls a specially created service on each Mule instance to assess its readiness to process normal service calls.

This kind of ping service is easy to configure in Mule. Clood, Inc., had to add such a service to its publication application. The publishing company was looking for a highly available deployment of its application, so Clood decided to deploy several instances of Mule behind a network balancer (see section 7.2.3 for more discussion on load balancing). The full configuration of this service is shown in listing 11.1. Note how we leverage a string expression evaluator in a response transformer to return a fixed value (refer to appendix A for more on this subject). Any call to the ping service will appropriately receive a PONG response.

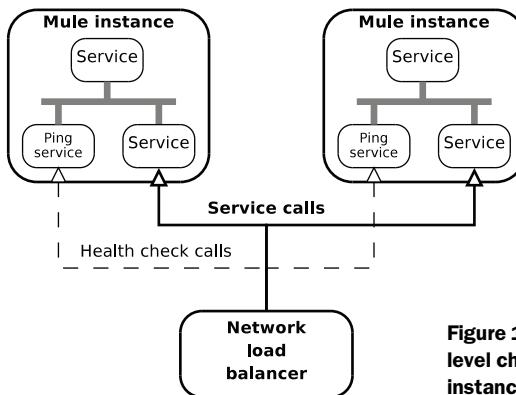


Figure 11.1 A load balancer can perform network-level checks to ensure the healthiness of the Mule instances it manages.

Listing 11.1 The ping service defined replies “PONG” to all calls.

```
<service name="pingService">
  <inbound>
    <http:inbound-endpoint
      address="http://${esb.bind.address}:${esb.web.port}/ping"
      synchronous="true">
      <response-transformers>
        <expression-transformer>
          <return-argument evaluator="string" expression="PONG" />
        </expression-transformer>
      </response-transformers>
    </http:inbound-endpoint>
  </inbound>
</service>
```

This simple ping service works for an HTTP load balancer, but it's easy to see that it could be bound to another type of transport (such as plain TCP). Of course, such a service can also be used outside of the context of a load balancer. A monitoring tool, for example, can leverage a ping service and decide to raise an alert if the expected response doesn't come on time.

A refinement of this service would consist of creating a specific component that would leverage the JVM and Mule's APIs (see chapter 13) to more accurately assert the internal state of the instance, returning something other than “PONG” in case of unhappiness.

As we said in the introduction, network-level checks are useful when a coarse assessment of an instance's health is acceptable. Let's now look at a more scrutinous means to monitor your instances, at system and JVM levels, with SNMP.

11.1.2 Checking health at system and JVM levels

The *Simple Network Management Protocol* (SNMP) is a standard defined by the Internet Engineering Task Force (IETF) for monitoring network-attached devices. SNMP is traditionally used to assess the health of servers at operating system-level by checking parameters such as CPU load, memory usage, or network traffic. Your organization is most likely already using SNMP to monitor its different servers. But SNMP can be used to check parameters of applications themselves, such as the JVM.

The JVM has a built-in SNMP agent that's configured to expose general configuration information and also memory pools and threads usage. It can also issue notifications on low-memory conditions. This allows you to use any SNMP-aware monitoring tool to ensure that the JVM in which Mule instances are running is in good shape. This is illustrated in figure 11.2.

There are many commercial and open source tools that can be used to perform SNMP monitoring. Some are focused on problem detection and alerts only, while others can actually graph the evolution of particular JVM properties. Figure 11.3 shows the evolution of the JVM heap utilization and thread count over a period of two days, as produced by Cacti, an open source web-based network graphing solution.

Using SNMP to monitor your Mule instances' JVMs can be a good option if you already have compatible monitors in-house. It should be limited to intranet usage only,

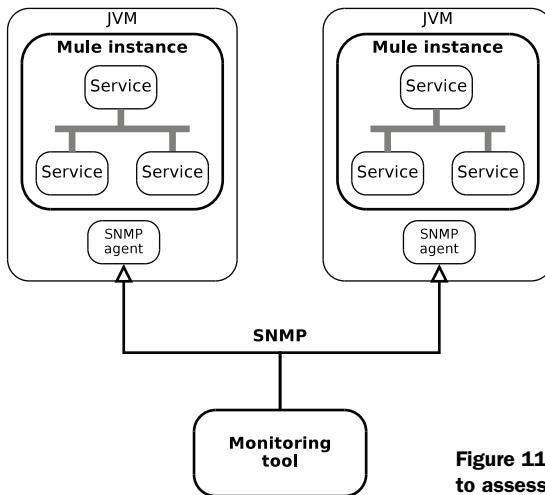


Figure 11.2 The JVM SNMP agent can be used to assess the health of a Mule instance.

though, as the particular version of the SNMP protocol supported by the JVM agent is mostly unsecured. Moreover, the agent is limited to JVM health assessment, and doesn't allow you to get any information about the actual health of the Java application running on the JVM.

Consequently, if you're interested in monitoring not only at the JVM level but also Mule itself, you'd better opt for the next option we'll detail: JMX monitoring.



Figure 11.3 Cacti is a tool that can leverage SNMP to monitor the evolution of some JVM properties, such as memory usage and thread count.

11.1.3 Checking health at JVM and Mule levels

Mule comes complete with agents that leverage the Java Management Extension (JMX) technology. These agents remotely expose (over RMI) a wealth of information about the JVM and the Mule instance itself in a standardized manner, which makes them ideal for in-depth monitoring scenarios. In addition to monitoring, these agents can also be used to actually act on the Mule instance itself. For example, it's possible to suspend services or even shut down an instance from a distance.

So how do we activate the Mule JMX agents? The quick and easy way is to use the `jmx-default-config` configuration element, which registers a bunch of predefined agents on fixed ports. This is done as shown in this configuration fragment:²

```
<management:jmx-default-config />
```

If you add this element to your configuration, you'll notice that the list of agents shown at Mule's startup now looks like this:

```
Running: jmx-agent:
service:jmx:rmi:///jndi/rmi://localhost:1099/server Rmi Registry:
rmi://localhost:1099 Default Jmx Agent Support jmx-log4j Jmx
Notification Agent (Listener MBean registered)
```

What have we gained here? Mainly, the bootstrapping of an RMI server and the registration of numerous MBeans (we'll come back to the log4j and the JMX notification agents in the next section). With this configuration in place, it's then possible to use a standard JMX tool, such as JConsole, and connect to the listed JMX RMI URI as shown in figure 11.4. Note how the innermost parts of Mule (connectors, models, services, and so on) are visible in the tree view of the MBean server, alongside the statistics.

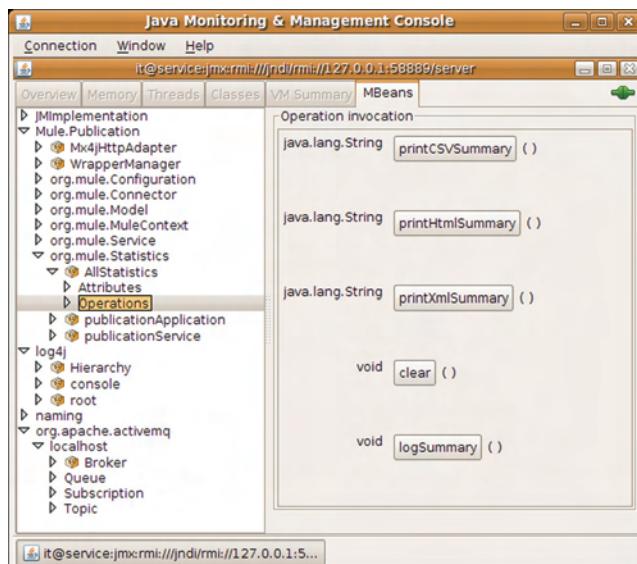


Figure 11.4 Mule registers numerous MBeans to expose its internals, such as comprehensive usage statistics.

² The correct declaration of the management schema is assumed.

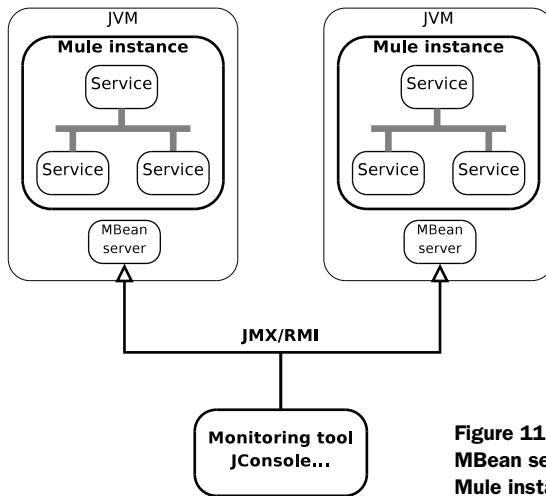


Figure 11.5 Remote connectivity to the JVM MBean server enables in-depth monitoring of a Mule instance.

It's also now possible to monitor your Mule instance by periodically checking the values of particular MBean attributes, as conceptually shown in figure 11.5.

Some notable MBean attribute values to monitor include

- *JVM memory*—A good strategy is to observe the size of the different memory pools after the Mule instances you intend to monitor have been up for a significant period of time and then set usage thresholds above these deemed normal values.
- *Lifecycle states*—Connectors and services expose boolean flags for their lifecycle states, such as disposed, paused, or stopped. This allows you to check that none of these critical moving parts is in a state that prevents it from performing its duty.
- *Service error counts*—Services track the number of errors that occurred while they were processing messages. Monitoring for increases in these error counts can warn you about serious issues happening in a Mule instance.

TIP *Server moniker* Unless instructed otherwise, Mule generates a unique server ID for each running instance. If you run Mule as a standalone server, you may have noticed this ID in the logs. The server ID is printed in the standard start-up boilerplate, as shown here:

```
*****
* Mule ESB and Integration Platform
* ...
* Server ID:54f6bf50-81df-11dd-a889-61bfa310f59c
* ...
```

This server ID is used to form the name of each MBean Mule exposes. For example, here's a typical MBean object name:

```
Mule.54f6bf50-81df-11dd-a889-61bfa310f59c:type=org.mule.Notification,
↳ name=MuleNotificationBroadcaster
```

Note the unique ID right after the `Mule.` prefix. As you can guess, this is less than practical if you intend to pull values from or invoke methods on Mule's MBeans. In that case, you'd better set the server ID to a known value so the MBean names won't change at each restart of the instance.

The good news is that this is easily achieved by setting the value of a Java system property named `mule.serverId`.³ If you look again at figure 11.4, you'll notice that we've used this technique to set the server ID to `Publication`. All MBeans names are prefixed with `Mule.Publication`, which make them easily reachable.

Our previous usage of `jmx-default-config` was bare-bones: we used the element as-is without setting any properties on it. In real life, you'll want to have control over the URL and ports the management agents will use, as well as over the users who'll be allowed to connect to these agents. You don't want to let just anybody connect to the MBean server of your production Mule instances. For this, you'll have to add more configuration in and around this configuration element.

TIP *Wrapper manager* Take a look at figure 11.4 and note the MBean named `WrapperManager`. This MBean allows you to control the Tanuki wrapper, which we talked about in section 7.1.1. From this MBean, you can request a thread dump to be written to the log, but can also stop or restart the whole Mule instance. This is another reason for not exposing Mule's MBean server to unauthenticated users.

Let's look at the management configuration shown in listing 11.2. It's the configuration we created for the publication application. Note how we've configured the desired bind address, ports, and security credentials. We'll come back to the second management element we added in shortly.

Listing 11.2 The full management configuration of the publication application

```
<management:jmx-default-config port="${esb.jmxrmi.port}" >
  <management:credentials>
    <spring:entry key="${esb.admin.username}"
                  value="${esb.admin.password}" />
  </management:credentials>
</management:jmx-default-config>

<management:jmx-mx4j-adaptor
  jmxAdaptorUrl="http://${esb.bind.address}:${esb.console.port}"
  login="${esb.admin.username}"
  password="${esb.admin.password}" />
```

With this configuration in place, the log boilerplate of the publication application looks like listing 11.3.

³ The command-line syntax is `-M-Dmule.serverId=Publication` (see tip in section 7.1.1).

Listing 11.3 The log boilerplate details the active management agents

```
*****
* Mule ESB and Integration Platform
* Version: 2.2.0 Build: 12377
* MuleSource, Inc.
* For more information go to http://mule.mulesource.org
*
* Server started: 12/01/09 12:34 PM
* Server ID: Publication
* JDK: 1.6.0_06 (mixed mode)
* OS: Linux (2.6.24-19-generic, i386)
* Host: mule-workhorse (127.0.1.1)
*
* Agents Running:
*   jmx-log4j
*   Wrapper Manager: Mule PID #17050, Wrapper PID #17048
*   Rmi Registry: rmi://127.0.0.1:58889/server
*   Jmx Notification Agent (Listener MBean registered)
*   jmx-agent: service:jmx:rmi:///jndi/rmi://localhost:58889/server
*   Default Jmx Support Agent
*   MX4J Http adaptor: http://127.0.0.1:8089
*****

```

Server ID set to known value

List of agents running

You may have noticed in listings 11.2 and 11.3 the appearance of a new agent: the *MX4J HTTP adaptor*. What's this new guy doing? It's a convenient way to expose the whole MBean server hierarchy as a web-based console (think of it as a browsable JConsole). As illustrated in figure 11.6, the MX4J HTTP adaptor locally connects to the JVM MBean server and makes it available to a simple browser as a console.

Figure 11.7 illustrates how convenient and consummate this HTTP console is. It's Cloud, Inc.'s tool of choice when a quick review of an instance's statistics is needed. It's also convenient when a firewall bars access to RMI ports or protocols. As flabbergasting as it may be, it can also be used for monitoring if it's absolutely impossible to hook any tool directly to your Mule instances. Monitoring tools are good at scraping HTML pages for content; hence they can easily pull all the MBean attribute values we've mentioned. As rough as it may sound, having this option handy as a last resort can save the day.

The Mule JMX agents enable in-depth monitoring of your instances. They provide you with the means to assess the health of both the JVM and the Mule moving parts that are critical for your business.

BEST PRACTICE Use JMX to monitor the JVM and Mule's critical services.

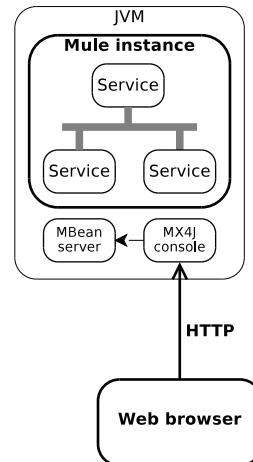


Figure 11.6 The MX4J HTTP adaptor exposes the whole MBean server as a browsable web console.

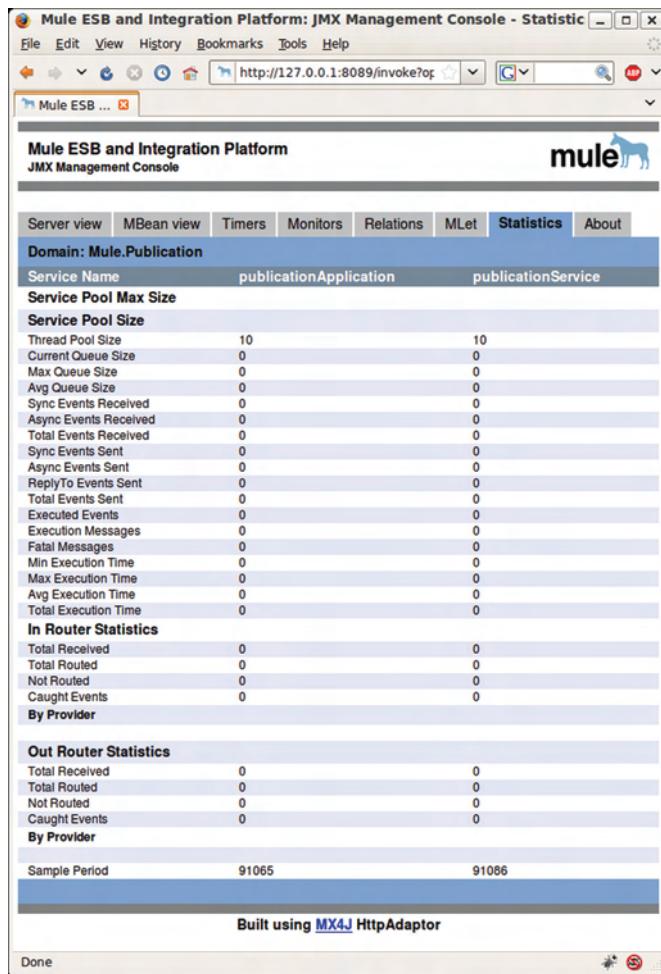


Figure 11.7 The MX4J HTTP adaptor agent exposes a JMX console which allows access to all the registered MBeans with a simple browser.

Whether it's for SLA reasons or because a particular network device needs it, both the JVM and Mule provide you with all the information you need to effectively ascertain their health. But monitoring isn't limited to checking the health of an application: it's often necessary to track the actual behavior of this application. We'll now review some techniques that you can use to track the activity of your Mule instances.

11.2 Tracking activity

If checking the health of your Mule instances turned you into a nurse, tracking its activity will turn you into a policeman. Why go for a different uniform? Because monitoring an application extends beyond the need for keeping an eye on its health: it's also important to ensure its behavior is actually what's expected. It may also be a business or a security requirement to track the activity on particular instances or services, for compliance or audit reasons.

The behavior of a Mule instance can be complex to describe. Inputs over a wide variety of transports, including schedulers, will trigger outputs toward a great many destinations. Based on the message context, conditional routing will orient message processing in different directions. Moreover, errors will occur at all levels and for diverse reasons: validation or transformation errors for bogus data, security errors for invalid credentials or rights, transport exceptions when remote destinations decide to show some attitude.

Facing such complexity, what could we potentially do to keep track of the behavior of our Mule instances? The good news is that activity leaves traces. By configuring Mule to leave useful traces behind itself, we can gain pretty good insight of what's happening inside of Mule, both in real time or after the fact. This is possible by using log files, whose verbosity can be configured, and also by leveraging the extensive notification framework inside Mule. We can also gain a good understanding of the activity of a Mule instance by monitoring the data it produces in external systems. Though not always possible, this is an option that's worth considering.

We'll start by looking at how the log files of a Mule instance can help you to keep track of its activity.

11.2.1 Using log files

Log files are still the most popular way of keeping track of an application's activity. Consequently, one of the most basic means to track the activity of a Mule instance is to keep an eye on what gets reported in its log files. This eye can be a monitoring system, looking for specific patterns, such as error messages. Monitoring tools are often well equipped to efficiently and continuously read log files. But often, it's a support person who'll have to skim through these log files, in search of some evidence needed to diagnose an issue. So what should you log in order to facilitate this forensic work?

You have three options for producing logs that enable you to efficiently track the activity of a Mule instance:

- *Using Mule's core logs*—Mule can emit an incredible amount of information at DEBUG level. While running at DEBUG level is appropriate in development, it's cumbersome in production, where the sheer amount of data it produces becomes a hindrance. This can be mitigated by tuning each logger (aka category) to the desired threshold, effectively controlling the overall log output. But this requires a good deal of knowledge of Mule's internals.
- *Logging Mule's notifications*—Mule can broadcast notifications for each event that occurs internally, and it's possible to log them. We'll discuss this fine-grained option in the next section.
- *Using custom logging*—As presented in chapter 8, you can leverage the logging infrastructure used by Mule to output your own log entries. This can be done in different places, for example in your own transformers, in custom components, or in Spring-driven interceptors (for Spring managed beans). We'll focus on this approach for the rest of this section.

TIP *Verbosity moderator* In the previous listings showing the list of running agents, you may have noticed one named jmx-log4j. This agent allows you to control the logging threshold of loggers and appenders at run time, via JMX. As we just mentioned, running constantly at DEBUG level in production isn't an option. But thanks to the jmx-log4j agent, it's possible to temporarily increase the verbosity of a particular logger or appender while a Mule instance is running. This is convenient for capturing a detailed activity snapshot while performing a particular action.

A convenient way to achieve custom logging is to tap a channel and send the snooped message to an audit channel (the wiretap router is discussed in chapter 4). This audit channel, which is usually a VM transport queue, is then consumed by a dedicated service that takes care of logging the messages. These logged messages can then be directed to a dedicated file or any other destination supported by log4j.

Bear in mind that messages often need a specific transformation before being in a form that can be logged. This transformation can't generally be done in the audit service, because a unique audit channel is used by wiretap routers in services processing messages of all kinds. The transformation should be done on these routers, where the data type is known. This isn't necessary if you transform all your messages to an internal canonical form and only log messages under this form.

TIP *No correlation, no love* You may remember that we introduced the log component in chapter 6. Can it be leveraged for activity tracking? Not really. This component only logs a string representation of the message payload. It misses an important bit of information for activity tracking: the message correlation ID.

The correlation ID is the Ariadne's thread you follow when you need to perform forensics in a messaging environment. Always ensure that you keep track of it. This is the only way to follow a message path in a highly distributed deployment. And the same applies for a single instance deployment: concurrent message processing weaves log entries beyond what a normal human brain can follow (or at least what my brain can follow).

Let's look at what we've done for the publication application. For auditing reasons, Clood's client wanted Clood to keep track of each book content upload done by the authors' application. Listing 11.4 shows the configuration of the auditor service we've created: it uses a specific component⁴ that logs the message properties and payload as shown in listing 11.5. Note how the correlation ID created by Mule is clearly visible, among the other transport and Mule-specific properties.

⁴ The code of the AuditComponent is available at the companion web site for this book.

Listing 11.4 An activity audit service logs messages using a custom component.

```
<service name="activityAuditorService">
  <inbound>
    <inbound-endpoint ref="AuditChannel" />
  </inbound>
  <component>
    <singleton-object class="com.clood.component.AuditComponent">
      <property key="logName" value="publication.mule.audit" />
    </singleton-object>
  </component>
</service>
```

Listing 11.5 An activity audit based on detailed issue tracking

```
15:18:44,807 INFO [publication.mule.audit] {
  Accept=*/
  Connection=true
  Content-Length=10487
  Content-Type=application/x-www-form-urlencoded
  Host=localhost:8080
  MULE_CORRELATION_ID=76653493-876a-11dd-8b08-cb6ad264076e
  MULE_ENDPOINT=vm://audit.channel
  MULE_ORIGINATING_ENDPOINT=AuditChannel
  MULE_REMOTE_CLIENT_ADDRESS=/127.0.0.1:38438
  User-Agent=Wget/1.10.2
  http.method=POST
  http.request=/publicationService
  http.version=HTTP/1.0
}
Uploaded title: Unit Test: refentry.007
-----
```

If you look at the log entry in listing 11.5, you'll realize that we don't log the whole content of the uploaded DocBook document, but only the value of the first title element. This is achieved by using an expression transformer in the wiretap router that sends to the audit channel, as shown in listing 11.6.

Listing 11.6 A transformer makes snooped messages writable in an audit log.

```
<wire-tap-router>
  <outbound-endpoint ref="AuditChannel">
    <expression-transformer>
      <return-argument
        evaluator="groovy"
        expression="'Uploaded title: '+
          org.apache.commons.lang.StringUtils.substringBetween(
            payload,'<title>','</title>')"/>
    </expression-transformer>
  </outbound-endpoint>
</wire-tap-router>
```

Whether you rely on Mule's core logs or decide to specifically log certain messages, logging plays an important role when tracking the activity of an instance. Mule gives you the tools you need for an effective logging strategy.

BEST PRACTICE Devise a logging strategy that fits your business, security, and production requirements.

Exploring logs is like exploring Mule’s intimate journal after the facts. Sometimes you’d prefer that Mule call you directly when a particular event occurs. This is when the notification framework, which we mentioned earlier in this section, comes into play. Let’s now detail how you can benefit from it.

11.2.2 Using notifications

The management module defines several agents specialized in handling Mule’s internal notifications that enable fine-grained tracking of an instance’s activity. These notifications are generated when specific events occur inside Mule, such as when an instance starts or stops, when an exception has been caught, or when a message is received or dispatched.⁵ In essence, the notification agents are ready-made listeners that process events they receive in predetermined and specific ways. As we introduced in the previous section, logging these notifications is one possibility, but there are many others. Let’s review them:

- The jmx-notifications agent rebroadcasts Mule notifications as standard MBean notifications. By registering a javax.management.NotificationListener to the MuleNotificationBroadcaster MBean, you can remotely receive these notifications in your own code. The jmx-notifications agent can also register an MuleNotificationListener MBean that basically accumulates messages in a list on your behalf. If you use this option, it’s up to you to query and flush this list regularly.
- The publish-notifications agent dispatches the notifications to an arbitrary endpoint. This opens the door to broadcasting event objects to virtually any destination you could want.
- The log4j-notifications agent sends a text representation of these events to the log4j logging framework. Thanks to the extensive list of appenders supported by log4j, it’s possible to send these log entries to a wide variety of destinations (including SNMP).

NOTE There’s also a chainsaw-notifications agent that seems to be ineffective in the version of Mule in use at this writing. This isn’t detrimental since the log4j-notifications agent can also broadcast to Chainsaw, the GUI-based log viewer for log4j that we presented in chapter 8.

When a publishing company came to Clood, Inc., and asked for a simple way to monitor in real time the activity of their authors in a system Clood had previously built for them, we decided to use the log4j agent and make it broadcast to a centralized Chainsaw console. The addition to the existing Mule configuration was minimal, as you can see in listing 11.7.

⁵ The notification framework is further detailed in section 11.2.2.

Listing 11.7 Configuration of notifications and a related log4j agent

```

<notifications>
    <notification event="ENDPOINT-MESSAGE" /> 1 Activates generation of
    </notifications> notifications for endpoint events

    <management:log4j-notifications logName="publication.mule.notifications" <span style="color: blue;">2 Declares log4j agent
        ignoreAdminNotifications="true" to process these
        ignoreComponentNotifications="true"
        ignoreConnectionNotifications="true"
        ignoreManagementNotifications="true"
        ignoreManagerNotifications="true"
        ignoreModelNotifications="true"
        ignoreSecurityNotifications="true" />

```

Note how we first have to activate the generation of notifications in ①: without this configuration entry, Mule wouldn't generate events when messages are received or dispatched. The notable aspects of configuring the agent itself ② are that we configure it to ignore the notifications we aren't interested in and that it's set to log under a particular name (this name is usually a class name, but can be arbitrary as is the case here).

WARNING *Death by a thousand notifications?* If you don't configure your agent to ignore the notifications you're not really interested in, you must be aware that it'll potentially generate significant traffic in the event-handling infrastructure you intend to use to process these notifications. Whether you use the log4j agent or another agent, it's recommended that you tailor the notifications they'll be interested in to your actual needs. Failure to do so will potentially induce a secondary load on your systems that'll follow the main load your Mule instances receive.

Be careful also not to establish a feedback loop if you use the publish-notifications agent: if you dispatch notifications to an endpoint in a Mule instance where message notifications are listened to, the notifications can raise events that will themselves be dispatched, and so on, leading to infinite generation of notifications.

Mule will then log endpoint message notifications to this category name at INFO level (as we haven't specified a particular level). Therefore, we have to use the same name to define the log4j category that appends directly to the Chainsaw console, as you can see in listing 11.8.

Listing 11.8 The log4j configuration that broadcasts Mule notifications to Chainsaw

```

<appender name="CHAINSAW" class="org.apache.log4j.net.SocketAppender">
    <param name="remoteHost" value="chainsaw.publisher.com" />
    <param name="port" value="4445" />
    <param name="locationInfo" value="true" />
</appender>

<category name="publication.mule.notifications" additivity="false">
    <priority value="INFO" />
    <appender-ref ref="CHAINSAW" />
</category>

```

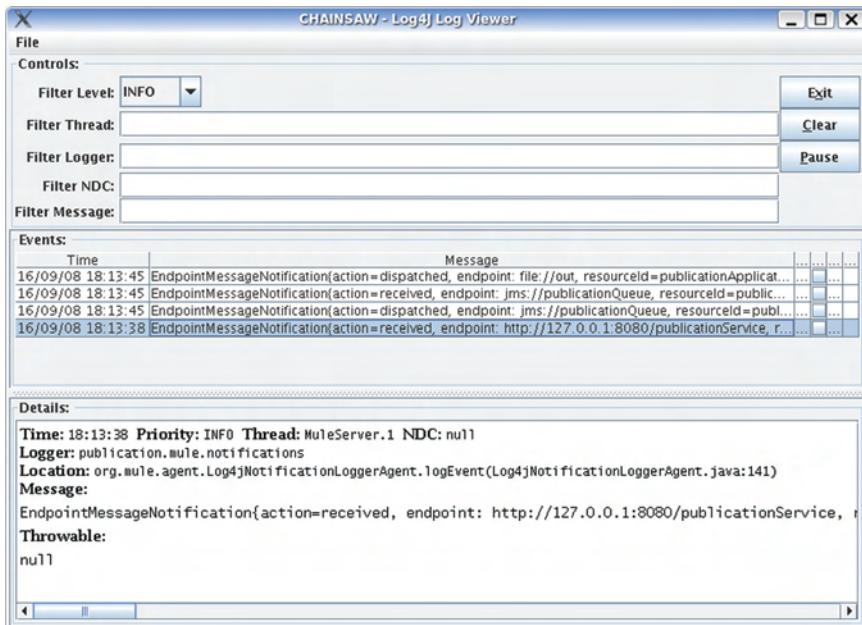


Figure 11.8 Chainsaw can be used to display Mule endpoint message notifications in real time.

Figure 11.8 shows the log entries received in Chainsaw right after an author has sent a document to the publication application. The first entry, which is highlighted, shows when and where the message has been received. The other entries let us see the message progress through JMS to the file system.

Should Cloud, Inc., rejoice that they've now created a simple console that displays the frantic activity of book authors? Technically, let's say yes, as all this has been achieved using standard Mule features and with the help of log4j.

The notification framework is a convenient way to monitor the activity of a Mule instance, as it's not only extensive but also highly configurable.

We'll now explore the last aspect of activity tracking, which consists of monitoring the impact Mule has on its surroundings.

11.2.3 Periodic data monitoring

Mule is seldom used in isolation: as an integration platform, it usually interacts with many different systems and enterprise resources. Once a message is done processing in a Mule instance, it's generally sent to an external destination, such as a messaging system, a web service, a file system, or a database. Mule's activity leaves traces. Monitoring these traces, whenever it's possible, is a viable way to ensure that Mule's performing its duty as expected.

Typically, this kind of monitoring is put in place when Mule performs regular activities that modify the state of an external system. For example, if Mule is used to poll a currency exchange rates feed once per day, you'll want to ensure that the database table where the values are stored doesn't have data older than 24 hours.

Thanks to its numerous transports, Mule has the capability to reach external systems of all kinds. It can be a challenge to speak all the protocols needed to connect to all the systems that a Mule instance can reach in order to ensure that data is modified as expected. The best approach is to leverage Mule itself for this data collection activity and to monitor... the monitor! This approach is illustrated in figure 11.9: a Mule service takes care of collecting data state in external systems and is regularly polled by a monitoring system to check whether a defect flag has been raised.

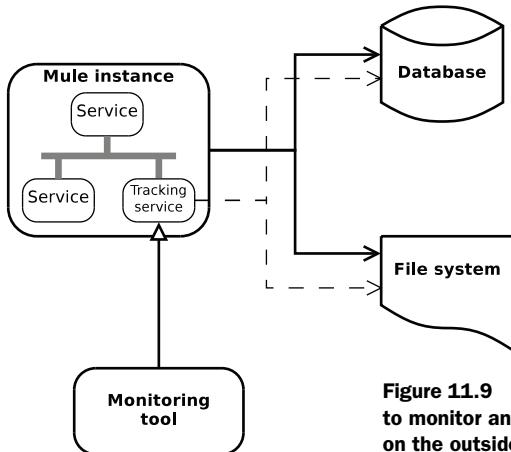


Figure 11.9 A service in Mule can be a convenient way to monitor an instance activity by looking at its impact on the outside world.

Clood, Inc., uses a slight variation of this approach: their tracking service is periodically triggered by an embedded Quartz scheduler and reports any problem to a dedicated error channel. This is illustrated in figure 11.10. If you wonder why we still use an external monitor, the answer is simple: we simply want to ensure that the activity tracking service is up and running. Again we monitor the monitor, but this time not to trigger its activity, but just to ensure its health. We'll come back to this particular use case of a scheduler in chapter 15.

Sometimes periodic data monitoring can't be transparently applied to all the external destinations that a Mule instance reaches. For example, unless it's been designed for it, a web service can't be randomly queried for the state of the data it encapsulates. In other situations, these external destinations are easy to monitor. Consider message queues, for example: if you're using Mule to consume or produce messages, you'll want to monitor the different destination queues. You can, for example, set a high watermark limit on a particular queue and raise an alarm if too many messages are pending there.

BEST PRACTICE Monitor enterprise resources that your Mule instances talk to.

We've reviewed different approaches to keep track of the activity of your Mule instances. In this domain, Mule is again feature-rich, providing you with the capacity to log and monitor the business-critical parts of your messaging infrastructure.

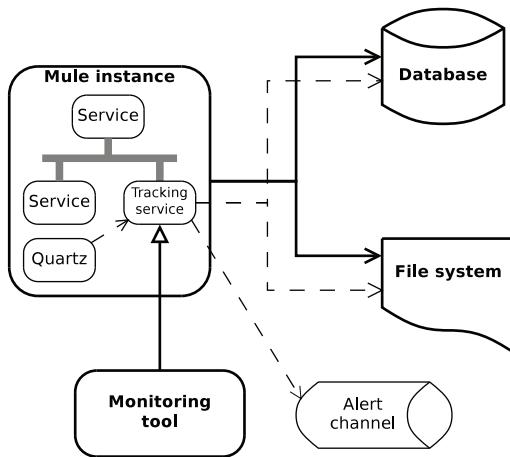


Figure 11.10 A Quartz scheduler can be used to trigger the activity-tracking service in a Mule instance.

The monitoring techniques we've talked about so far were mainly oriented toward machines such as load balancers and monitoring tools. But human beings also want to be able to sense the state of a Mule instance. For this, the best option, which we'll now discover, is to build a dashboard.

11.3 Building dashboards

Whereas monitoring is oriented toward machines and applications, dashboards are intended to be used by humans. They're built to aggregate and summarize technical information about a system into synthetic representations that allow you to get the feel of a situation simply by glancing at a display. Though it may at first look futile to build stuff for human beings to look at, it's in fact of paramount importance. Once Mule instances start to multiply in your network, you'll feel the urge to gain awareness of what's happening in these instances. Monitors will assure you that things are going well; activity tracking will give you the capability to diagnose issues, but you'll still need to take a look at what's going on.

This is when building a dashboard that makes sense to you becomes important. Discussing the qualities of a good dashboard is beyond the scope of this book: there's abundant literature about this subject available out there. Instead we'll review the general-purpose dashboard Clood, Inc., has created.

As shown in figure 11.11, our dashboard is generated by a component⁶ that's deployed in the Mule instance itself. It's rendered as HTML, so a standard web browser can display it. Sev-

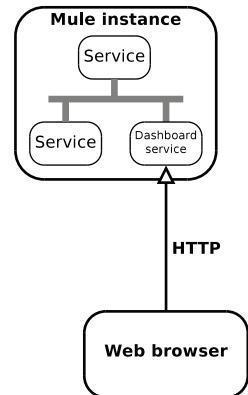


Figure 11.11
An external dashboard can connect to Mule's JMX notification MBeans to gather information to display.

⁶ The code of the `HtmlDashboard` is available at the companion site for this book.

eral of these dashboards, generated on different Mule instances, can be aggregated with something as crude as a frameset page hosted on a shared host.

Because the dashboard is generated by a component hosted in Mule, it has direct access to the complete Mule API. This API, discussed in chapter 13, is much richer and more direct to use than the MBeans exposed by Mule. Our dashboard piggybacks on the extensive statistics API that Mule offers: it basically displays traffic patterns with different colors so you can capture the activity that occurs in an instance with a glimpse.

Figure 11.12 presents a few examples of different states displayed by our dashboard while deployed in the publication application. From top to bottom: the ping service is correctly exercised while other services are idle; the publication service has received a message that makes the document processor choke; the document processor service has been paused. An interesting aspect of this dashboard is that it displays variations between refreshes: this means that if you don't display it for a week, you'll see a week of activity the next time you start it up. This flattens a lot of details but also gives you a great way to catch up with the latest news: no red light while I was in the Bahamas?

Behind the scenes, the dashboard is a standard custom component that you can expose using the connector that's relevant for your configuration (Servlet, Jetty, or HTTP). An interesting aspect of its configuration, shown in listing 11.9, is that it receives the set of services to monitor via a direct Spring injection. The referenced beans are actual Mule services. This is a foolproof way to configure a component, as it relies on actual object references and not symbolic names.

Listing 11.9 The log4j configuration that broadcasts Mule notifications to Chainsaw

```
<spring:bean id="HtmlDashboardComponent"
            class="com.clood.component.HtmlDashboard">
    <spring:property name="observedServices">
        <spring:set>
            <spring:ref bean="publicationService" />
            <spring:ref bean="documentProcessor" />
            <spring:ref bean="pingService" />
        </spring:set>
    </spring:property>
</spring:bean>
```

A viable alternative to the embedded approach we've followed consists of creating a dashboard as a standalone application (web or rich). This can be done by leveraging

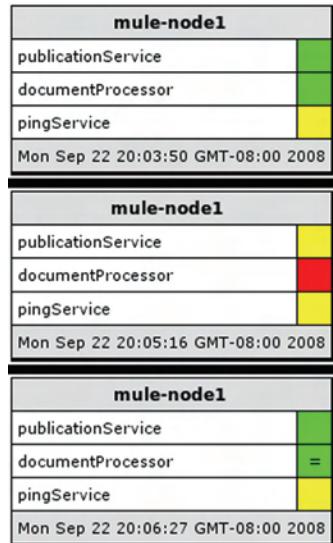


Figure 11.12 The HTML dashboard summarizes recent activities and service states with color codes and symbols.

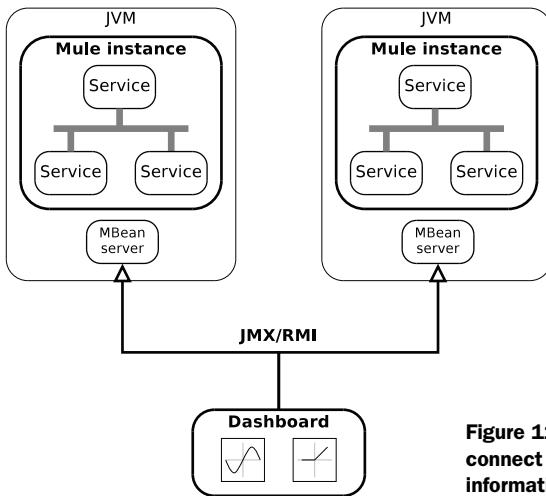


Figure 11.11 An external dashboard can connect to Mule's JMX MBean server to gather information to display.

the different JMX agents to regularly query interesting MBeans or receive notifications. This approach is illustrated in figure 11.13. It's a good option if you want to use advanced graphing components, such as delicious pie charts or good old gauges.

Building custom dashboards for Mule is both useful and fun. It's by no mean a one-size-fits-all activity: you'll have to identify the type of information and the representation that best fit your needs or those of your production team. But, whether you decide to use an existing graphing tool or to create your own renderer, the diversity of information exposed by Mule will certainly allow you to build an effective dashboard.

11.4 Summary

Putting in place a well-grounded monitoring strategy for your Mule instances is sometimes required by network devices such as load balancers, but is always a necessity in a production-grade environment. Moreover, if this monitoring strategy is a cake, then dashboards are the icing that provides a satisfactory feeling of control over what can be a complex situation.

As we've discovered, the JVM, Mule, and custom components can all contribute to achieving your monitoring goals. We've also reviewed the available options offered by Mule, whether you opt for leveraging standard protocols, simple log files, or Mule's extensive notification framework.

We're now done with our review of guidelines and practices for happily running Mule in your environment. You've learned how to deploy, secure, and monitor Mule instances. You've also discovered how to establish sound logging, exception handling, and transaction management practices in your Mule applications. In the remainder of the book, we'll introduce you to techniques and tactics that'll help you travel even further with Mule. We'll come closer to programming by looking at some developer tools and the Mule API. We'll also explore scripting, scheduling, orchestration, and tuning.

Part 3

Traveling further with Mule

While all the fundamentals of Mule have been covered in parts 1 and 2, we aren't done yet with our exploration of all the good things Mule can do for you! This last part of the book will bring additional bits of knowledge and best practices in order for you to travel further with Mule.

A wealth of tools is available that can help you be more productive while developing with Mule. In chapter 12, we'll review how Maven can help you to build and manage all the dependencies of your Mule-driven projects. We'll also look at developing with an IDE, such as Eclipse, and discuss your different options when it comes to thoroughly testing your integration application.

Chapter 13 will be the most code-intensive of this book: in it, we'll delve into the API of Mule by looking at some of its key classes. We'll detail the Mule client and its numerous use cases, the different contexts from which you can extract a lot of information, and the notification and interceptor frameworks.

Dynamic languages have become major players in a developer's toolbox. Chapter 14 will demonstrate how Mule allows you to leverage the power of scripting languages to improve the flexibility and versatility of your integration applications.

One of the main virtues of services is that they can be composed and orchestrated: here again Mule can help. In chapter 15, we'll explore some techniques that will allow you to use Mule to run business processes and scheduled tasks.

We'll close this book with an in-depth analysis of Mule's threading model in chapter 16. We'll also discuss how thread pools can be tuned and what techniques you can use to profile and performance-boost your Mule-driven applications.

12

Developing and testing with Mule

In this chapter

- Leveraging Maven with Mule
- Using an IDE to aid development with Mule
- Writing tests for Mule projects

Developers all have different styles and preferences for writing software and managing the development process. Some embrace the command line, using vi to edit their code along with configuration files and scripts to package and deploy their applications. Others prefer to centralize their development using a powerful IDE. Most of us probably use some combination of the two approaches, perhaps using an IDE, such as Eclipse or IDEA, to write code and then using the Unix shell tools for packaging and deployment. While these proclivities may vary, one thing developers can usually agree on is testing. Testing techniques such as unit, integration, and load testing allow us to ensure our code works in isolation and conjunction—letting us refactor in confidence.

Mule simplifies these tasks by providing facilities for build management, IDE integration, and testing support. We'll examine all of these in this chapter, starting

with Mule's support for Maven, a popular, open source build management framework. Once you're comfortable managing your Mule projects with Maven, we'll examine how an IDE can simplify Mule development. Finally, we'll see how to perform integration and load testing with Mule using the JUnit test framework, Mule's Test Compatibility Kit, and the JMeter load testing tool.

12.1 **Managing Mule projects with Maven**

Dependency management, testing, and packaging are integral parts of the development cycle. Nontrivial applications need to manage external libraries, run automated test suites, and compile and package for deployment. Code you write for your Mule projects is no exception. This'll become apparent once you start writing your own transformers and components. Let's look at some of the steps you need to take after you've written a custom component:

- Add any libraries you need for compilation, including the Mule libraries, to the classpath of javac.
- Use javac to compile the Java source into class files.
- Run any unit or functional tests.
- Package the Java class into a JAR file.
- Deploy the JAR file.

This collection of activities might be tolerable if it weren't repeated frequently. This unfortunately isn't the case—a developer might need to repeat this process several times in succession when developing and debugging. More problems are introduced as the process is independently repeated by other developers. One developer may forget to run the tests, while another may build his JAR files slightly differently, for instance.

Automation is a solution for these steps. Tools such as Ant, make, or a scripting language could be used for this purpose. While these approaches are perfectly reasonable, some problems arise when moving from one project to the next. The build processes for different projects invariably begin to diverge. This introduces developer inertia when moving from one project to the next. Learning how the build infrastructure works quickly becomes more difficult than comprehending the code of the project itself! This can be managed within a single organization, but resurfaces when developers of that organization need to work on an external project—as is often the case when working with open source software.

Maven was introduced as a solution to this problem. It provides a framework for managing the build infrastructure of Java projects. Its facilities should negate most of the need for custom build infrastructures, allowing developers to move from project to project without spending a lot of time learning how to build the new project. In this section we'll start off by seeing how to create a Maven project. We'll then see how to manage dependencies to Mule libraries in our Maven project. Finally, we'll explore how Maven simplifies packaging and deployment of our Mule code.

12.1.1 Setting up a Maven project

Let's start off by downloading and installing Maven. Maven can be downloaded from <http://maven.apache.org/>. Full installation instructions can be found in the release or at <http://maven.apache.org/download.html>. You essentially need to do the following:

- Add an `M2_HOME` environment variable to either your Windows system or Unix shell.
- Add the `bin` subdirectory of `M2_HOME` to your system's path so you can execute the Maven commands.

Once Maven is installed, you can create your first project. The following command (listing 12.1) creates a Maven project for a `message-enricher` component Cloud will use to add headers to messages.

Listing 12.1 Creating a Maven project structure for a custom component

```
mvn archetype:create -DgroupId=com.cloud -DartifactId=message-enricher
```

We need to specify the group and artifact IDs for our project. Maven exhibits tight control over a project's dependencies. It does this by organizing JAR files in a hierarchy of groups, artifacts, and versions. We'll see more of this in a bit when we explicitly identify our Mule requirements. For now, we're telling Maven that the group for our project is `com.cloud`. The artifact name is going to be `message-enricher`. When we build our project, Maven will use these conventions to store the JAR file in a local repository. We'll then be able to use the combination of group ID, artifact ID, and version to identify and use the JAR file in other projects.

When you run the command in listing 12.1, you should see Maven spin into a flurry of activity. When it finishes, there'll be a subdirectory called `message-enricher` in the directory from which you run the command. This directory contains three things: a `pom.xml`, file and source directories for your code, and its corresponding unit tests. The structure is illustrated in Figure 12.1.

Maven will create the two source trees for our project: one for the main project code and another for JUnit tests. The source directories will form a package structure

| ▼ | message-enricher | | |
|---|------------------|-----------------|------|
| | pom.xml | Today, 11:08 AM | -- |
| ▼ | src | Today, 11:08 AM | 4 KB |
| ▼ | main | Today, 11:16 AM | -- |
| ▼ | java | Today, 11:16 AM | -- |
| ▼ | com | Today, 11:16 AM | -- |
| ▼ | cloud | Today, 11:08 AM | -- |
| | App.java | Today, 11:08 AM | 4 KB |
| ▼ | test | Today, 11:08 AM | -- |
| ▼ | java | Today, 11:16 AM | -- |
| ▼ | com | Today, 11:16 AM | -- |
| ▼ | cloud | Today, 11:08 AM | -- |
| | AppTest.java | Today, 11:08 AM | 4 KB |

Figure 12.1 The directory structure created by Maven for a project

that matches the group ID you supplied to the `mvn` command. In addition to creating the directories, Maven has also created a dummy application that'll print "Hello World" to the screen. If you look at these source files you'll see that they've been automatically placed in a package that matches the group ID.

Let's build and execute the test app to get a feel for how builds with Maven work. Listing 12.2 will build `App.java`, run its associated JUnit test, and create a JAR file.

Listing 12.2 Compile, test, and build a JAR file of App.java

```
mvn clean compile test package
```

Running this command from the directory with `pom.xml` in it will clean any previously compiled classes, compile all the Java sources, run any JUnit tests, and package the resulting Java classes into a JAR. If any of the steps fail, the build will be stopped at that point. For instance, if any of the tests fail, then a JAR file won't be built. The build process will add an additional directory to your project, as illustrated in figure 12.2.

| | | | |
|-----------------------------------|--|-----------------|------|
| message-enricher | | Today, 11:33 AM | -- |
| pom.xml | | Today, 11:08 AM | 4 KB |
| src | | Today, 11:16 AM | -- |
| main | | Today, 11:16 AM | -- |
| java | | Today, 11:16 AM | -- |
| com | | Today, 11:16 AM | -- |
| cloud | | Today, 11:22 AM | -- |
| App.java | | Today, 11:08 AM | 4 KB |
| test | | Today, 11:16 AM | -- |
| java | | Today, 11:16 AM | -- |
| com | | Today, 11:16 AM | -- |
| cloud | | Today, 11:22 AM | -- |
| AppTest.java | | Today, 11:08 AM | 4 KB |
| target | | Today, 11:33 AM | -- |
| classes | | Today, 11:33 AM | -- |
| maven-archiver | | Today, 11:33 AM | -- |
| message-enricher-1.0-SNAPSHOT.jar | | Today, 11:33 AM | 4 KB |
| surefire-reports | | Today, 11:33 AM | -- |
| test-classes | | Today, 11:33 AM | -- |

Figure 12.2 The directory structure after a successful build

If you inspect the target directory, you'll see that the compiled class files are located in `classes` and the compiled test classes in `test-classes`. A JAR file has also been created. Let's execute the jar file (listing 12.3) and see what happens.

Listing 12.3 Run the Hello World app

```
java -cp target/message-enricher-1.0-SNAPSHOT.jar com/clood/App
```

This should result in "Hello World!" being printed to your screen. You can probably infer that the JAR file is named after the artifact ID we used to create the Maven project in listing 12.1, but where is the version coming from? The answer is from the `pom.xml` file in the root of your project directory. This is the file you'll use to tell Maven about your project. Listing 12.4 illustrates the one for our `message-enricher` artifact.

Listing 12.4 The Maven project definition for the message-enricher project

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://maven.apache.org/POM/4.0.0
     http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.clood</groupId>
  <artifactId>message-enricher</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>message-enricher</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

The `pom.xml` file contains all the information Mule needs to build our project. The `groupId` and `artifactId` elements on ① and ② are populated by the arguments to the `mvn` command we ran in listing 12.1. The packaging mechanism is specified on ③. In this case, we want our end result from the build to be a JAR file, so we have `jar` specified. Other targets are possible. If we were building a web application, for example, the target here would be `war`. The full range of available targets can be found on the Maven web site. The version of our project is specified on ④. In this case we're going to use the default, `1.0-SNAPSHOT`, indicating that we're working on a development version of the message enricher.

The dependencies for `message-enricher` begin on ⑤. Dependency management is one of the key features of Maven. By defining our dependencies, Maven will automatically resolve the dependency graph of your project—bundling the appropriate libraries in your builds. The only dependency for the Hello World application is JUnit, and this is declared on ⑥. We're specifying that we want to use JUnit 3.8.1 for our tests. Since we don't need JUnit to compile our main classes, it's given a scope of `test`. This would exclude the bundling of the JUnit library if we opted to build a JAR with all dependencies included. The default scope is `compile`.

Maven isn't limited to building just JAR files. By changing the packaging mechanism in the `pom.xml` file, you can build WAR or EAR files, for instance. This allows you to use Maven with the deployment options discussed in chapter 7. Maven generally makes smart decisions when packaging these projects. Mule projects embedded in web application WAR files, for instance, will have all the required JAR dependencies in their `WEB-INF/lib` directories when packaged.

Now that you're comfortable setting up Maven and building a project, let's get to work on our custom component.

12.1.2 Using the Mule Maven dependencies

As we mentioned previously, one of the core features of Maven is dependency management. Maven calls dependencies *artifacts*. An artifact is usually a JAR file located in a Maven repository that your project needs for some phase of its build lifecycle, generally for compilation, testing, or debugging. Before we dig into how to add Maven artifacts to our project, let's look at the source for Clood's message enricher. Clood is writing a message enricher that'll add a property to messages containing metadata about the organization they belong to. This data will ultimately come from Clood's LDAP directory, but for the proof-of-concept phase they're simply adding a static header to each message. This is illustrated in listing 12.5.

Listing 12.5 A simple component to add a property to a message

```
package com.clood;
import org.mule.api.MuleEventContext;
import org.mule.api.MuleMessage;
import org.mule.api.lifecycle.Callable;

public class MessageEnricher implements Callable {
    public Object onCall(MuleEventContext muleEventContext) {
        MuleMessage message = muleEventContext.getMessage();
        message.setProperty("ORGANIZATION", "CLOOD");
        return message;
    }
}
```

The source for this class will be placed in `src/main/java` along with a unit test in `src/test/java`. As we'll be covering unit testing in depth in section 12.3, we'll just add a placeholder `TestCase` for now and revisit it later on. Listing 12.6 illustrates the skeleton test case.¹

Listing 12.6 A skeleton test case

```
package com.clood;

import junit.framework.TestCase;

public class MessageEnricherTest extends TestCase {
    public void testOnCall() {
        // ToDo IMPLEMENT A REAL TEST!
        assertTrue(true);
    }
}
```

¹ A property transformer could also be used in place of this component, until the additional LDAP lookup functionality was required.

Let's build our project again and see what happens. The output is illustrated in listing 12.7.

Listing 12.7 Attempt to build the message-enricher project

```
mvn clean package
```

This time you'll notice that the build has failed. You should see output on your screen similar to listing 12.8.²

Listing 12.8 The component fails to compile because of missing dependencies.

```
[INFO] -----
[ERROR] BUILD FAILURE
[INFO] -----
[INFO] Compilation failure

... MessageEnricher.java:[3,19] package org.mule.api does not exist
... MessageEnricher.java:[4,19] package org.mule.api does not exist
... MessageEnricher.java:[5,29] package org.mule.api.lifecycle
does not exist

... MessageEnricher.java:[7,40] cannot find symbol
... etc
```

It seems like our build has failed because the required Mule JAR libraries aren't available to the project. This can be corrected by adding a dependency to Mule's core library in the project's `pom.xml` file. Listing 12.9 illustrates how to do this.

Listing 12.9 Add a dependency to Mule's core library for the project.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  "http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.cloud</groupId>
<artifactId>message-enricher</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>message-enricher</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>           ← Group ID of dependency
    <artifactId>junit</artifactId>     ← Artifact ID of dependency
    <version>3.8.1</version>           ← Version of dependency
    <scope>test</scope>
  </dependency>
</dependencies>
```

² Note that we've omitted the `compile` and `test` arguments. These arguments, called *goals* by Maven, are implied and will be executed by default.

```

<groupId>org.mule</groupId>
<artifactId>mule-core</artifactId>
<version>2.2.0</version>
</dependency>
</dependencies>
</project>

```

Adding a dependency to version 2.2.0 of the mule-core artifact will make Mule's core API available to our project. The project should now build without incident, producing a JAR file suitable for deployment to Mule in the target directory.

Maven does more than just add the Mule core library to your project. All dependencies of Mule's core library are added to the project as well. You can view the entire dependency tree of your project by running the mvn dependency:tree goal. Listing 12.10 shows what this looks like for Cloud's message-enricher project.

Listing 12.10 Printing the dependency tree of the message-enricher project

```

[INFO] [dependency:tree]
[INFO] com.cloud:message-enricher:jar:1.0-SNAPSHOT
[INFO] +- org.apache.activemq:activemq-core:jar:5.1.0:compile
[INFO] |  +- commons-logging:commons-logging-api:jar:1.1:compile
[INFO] |  +- org.apache.camel:camel-core:jar:1.3.0:compile
[INFO] |  |  +- javax.xml.bind:jaxb-api:jar:2.1:compile
[INFO] |  |  |  \- javax.xml.stream:stax-api:jar:1.0-2:compile
[INFO] |  |  \- com.sun.xml.bind:jaxb-impl:jar:2.1.3:compile
[INFO] |  +- org.apache.geronimo.specs:geronimo-jms_1.1_spec:jar...
[INFO] |  +- org.apache.activemq:activeio-core:jar:3.1.0:compile
[INFO] |  |  +- commons-logging:commons-logging:jar:1.1:compile
[INFO] |  |  |  +- logkit:logkit:jar:1.0.1:compile
[INFO] |  |  |  +- avalon-framework:avalon-framework:jar:4.1.3:compile
[INFO] |  |  |  |  \- javax.servlet:servlet-api:jar:2.4:compile (version ...
[INFO] |  |  |  \- backport-util-concurrent:backport-util-concurrent:jar:...
[INFO] |  |  +- org.apache.geronimo.specs:geronimo-j2ee-management_1.0_sp...
[INFO] |  |  \- org.apache.activemq:activeio-core:test-jar:tests:3.1.0:co...
[INFO] +- org.mule.transports:mule-transport-jms:jar:2.2.0:compile
[INFO] |  +- org.mule:mule-core:jar:2.2.0:compile
[INFO] |  |  +- commons-beanutils:commons-beanutils:jar:1.7.0-osgi:compile
[INFO] |  |  +- org.safehaus.jug:jug:jar:asl:2.0.0-osgi:compile
[INFO] |  |  +- commons-cli:commons-cli:jar:1.0-osgi:compile
[INFO] |  |  +- commons-collections:commons-collections:jar:3.2-osgi:compile
[INFO] |  |  +- commons-io:commons-io:jar:1.3.1-osgi:compile
[INFO] |  |  +- commons-lang:commons-lang:jar:2.4:compile (version man...
[INFO] |  |  +- commons-pool:commons-pool:jar:1.4:compile
[INFO] |  |  +- javax.activation:activation:jar:1.1:compile
[INFO] |  |  +- org.apache.geronimo.specs:geronimo-jta_1.0.1B_spec:jar...
[INFO] |  |  +- org.apache.geronimo.specs:geronimo-j2ee-connector_1.5...
[INFO] |  |  +- org.slf4j:jcl104-over-slf4j:jar:1.5.0:compile
[INFO] |  |  +- org.slf4j:slf4j-api:jar:1.5.0:compile
[INFO] |  |  +- org.slf4j:slf4j-log4j12:jar:1.5.0:compile
[INFO] |  |  |  \- log4j:log4j:jar:1.2.14:compile
[INFO] |  |  +- org.mule.modules:mule-module-spring-config:jar:2.2.0:compile
[INFO] |  |  |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  |  |  \- jaxen:jaxen:jar:1.1.1:compile

```

```
[INFO] |  | \- jdom:jdom:jar:1.0:compile
[INFO] |  \- org.springframework:spring-context:jar:2.5.6:compile
[INFO] |    +- org.springframework:spring-beans:jar:2.5.6:compile
[INFO] |    \- org.springframework:spring-core:jar:2.5.6:compile
[INFO] +- com.muleinaction:velocity-transformer:jar:1.0-SNAPSHOT:compile
[INFO] |  +- org.mule.transports:mule-transport-vm:jar:2.2.0:compile
[INFO] |  |  \- org.mule.modules:mule-module-xml:jar:2.2.0:compile
[INFO] |  |    +- org.apache.geronimo.specs:geronimo-stax-api_1.0...
[INFO] |  |    +- commons-jxpath:commons-jxpath:jar:1.3-osgi:compile
[INFO] |  |    +- com.thoughtworks.xstream:xstream:jar:1.2.2-osgi:compile
[INFO] |  |    +- xpp3:xpp3_min:jar:1.1.3.4.0-osgi:compile
[INFO] |  |    +- org.codehaus.woodstox:wstx-asl:jar:3.2.6-osgi:compile
[INFO] |  |    +- net.java.dev.stax-utils:stax-utils:jar:200807...
[INFO] |  |    +- net.sf.saxon:saxon:jar:8.9.0.4-osgi:compile
[INFO] |  |    \- net.sf.saxon:saxon-dom:jar:8.9.0.4-osgi:compile
[INFO] |  +- org.mule.modules:mule-module-client:jar:2.2.0:compile
[INFO] |  +- org.springframework:spring-context-support:jar:2.5.6:compile
[INFO] |  |  \- aopalliance:aopalliance:jar:1.0:compile
[INFO] |  |  \- org.apache.velocity:velocity:jar:1.5:compile
[INFO] |  \- oro:oro:jar:2.0.8:compile
[INFO] +- org.mule.modules:mule-module-builders:jar:2.2.0:compile
[INFO] |  \- org.springframework:spring-web:jar:2.5.6:compile
[INFO] +- org.mule.tests:mule-tests-functional:jar:2.2.0:test
[INFO] |  \- org.mule:mule-core:test-jar:tests:2.2.0:test
[INFO] \- junit:junit:jar:4.4:test
```

The entire graph of our project's dependencies are output to the screen. All of `mule-core`'s dependencies are added to the message enricher's `compile` classpath, saving you the trouble of having to do this manually.

You might've noticed Maven fetching JARs after you added the `mule-core` dependency to your POM and rebuilt the project. Maven automatically downloads necessary dependencies from a central site, called a *repository*. These artifacts are then cached to a local repository on your filesystem. The central Maven repository is located at <http://repo1.maven.org/maven2/>. If you poke around the repository for a while, you'll see the artifacts are organized in a directory structure that matches their group IDs. If you navigate to <http://repo1.maven.org/maven2/org/mule/> you'll see the start of Mule's Maven artifacts.

You can use this to select the appropriate module to include in your own projects. For instance, if you were working on a custom JMS transformer, your code would probably need access to Mule's JMS transport, the JMS API libraries, and so on. As such, you might have a `pom.xml` file that looks like listing 12.11.

Listing 12.11 Adding a dependency to the Mule JMS transport to a `pom.xml`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation=
         "http://maven.apache.org/POM/4.0.0
          http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.clod</groupId>
```

```

<artifactId>message-enricher</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>clood-jms-transformer</name>
<url>http://maven.apache.org</url>
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mule.transports</groupId>
        <artifactId>mule-transport-jms</artifactId>
        <version>2.2.0</version>
    </dependency>
</dependencies>
</project>

```

This would make the JMS transport and all its dependencies, including `mule-core`, available to your transformer.

Let's now see how to simplify Mule project creation by using the Mule Maven archetypes.

12.1.3 Simplifying Maven projects with the Mule Maven archetypes

Mule provides a set of Maven archetypes that simplify some of the tasks we previously enumerated. You can take advantage of these archetypes when starting a new Mule project or extending Mule by creating a new transport or Mule module.

The Mule project archetype can be used to create a new project. The archetype will ask you a series of questions, including things such as the Mule version and the transports you wish to use, and create a Maven project for you. This project will include all the required dependencies in its `pom.xml` file along with a template Mule configuration that includes all the required namespace definitions. Execute the Maven command in listing 12.12 to create the project.

Listing 12.12 Starting a Maven project with the Mule project archetype

```
mvn org.mule.tools:mule-project-archetype:create \
-DartifactId=message-enricher \
-DmuleVersion=2.2.0
```

You should now have a Maven project with the artifacts we just described. These files contain additional gems in the forms of TODO pointers—these will guide you through what you need to complete to get your Mule project running and tested. The Mule Maven archetypes can also be used as starting points to create transports and modules. Mule modules are pieces of functionality, like XML or PGP support, that don't fall into the category of transports. You can use the `mule-transport-archetype:create` goal when starting a new transport and the `mule-module-archetype:create` goal when beginning work on a new module.

By now you should be comfortable building your Mule projects using Maven. You saw how to set up, test, and build a Maven project. You also saw how to manage your project's dependencies by using the artifacts in a Maven repository. This allowed us to automatically make Mule's libraries available for Cloud's message enricher. Let's now turn our attention to another way to ease developing with Mule—an integrated development environment.

12.2 Using Mule with an IDE

It's completely feasible to perform all your Mule-related work—development, testing, debugging, and deployment—using command-line tools. Integrated development environments exist for those of us who need a bit more help. This is particularly the case with projects that make heavy use of XML. You might already be aware that working with XML using a bare-bones text editor can be slightly painful. By using a well-documented XML schema for configuration, Mule eases much of this burden. To take full advantage of these features, though, you need a schema-aware development environment.

Starting and stopping Mule can be another inconvenience. When you've finished modifying a Mule configuration and want to test it, you generally need to stop and start a Mule instance to see your changes take effect. This generally involves leaving your development environment and executing a shell script, then switching back over to your development environment to test and debug. Wouldn't it be nice if this functionality were integrated into your IDE?

In this section we'll look at Mule's support for integrated development environments. We'll start by seeing how the XML features of two popular Java IDEs, Eclipse and IDEA, can simplify the authoring of XML Mule configurations. We'll then see how to use Mule IDE, an Eclipse plug-in, to start and stop Mule instances from within the Eclipse development environment.

12.2.1 XML editing for Mule

For some people, XML is challenging. Even those of us comfortable working with XML sometimes need a bit of help. Whatever your level of comfort with XML, we highly recommend you use a schema-aware XML editor to author Mule's configuration files. Using such an editor will provide two main features that'll make your life easier and the overall XML edition process a little happier. First, you'll immediately know if your configuration file is well-formed and valid, or said differently, that it's really XML and it complies with the schema rules. Second, you'll benefit from the content-authoring assistance that'll guide you when creating the XML file, pretty much like with the code completion feature of a Java editor. Most recent IDEs offer such an editor, and there are also capable standalone editors. Figure 12.3 shows a content assist pop-up in Eclipse. Note how both insertion suggestions and documentation are contextual to the element that's edited.

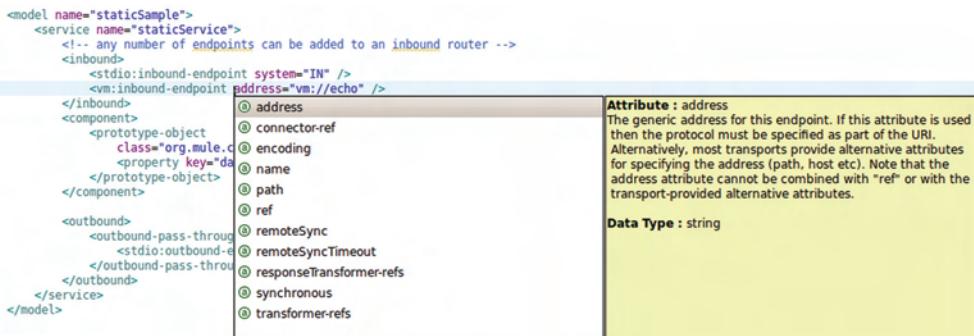


Figure 12.3 Editing a Mule configuration file with Eclipse: content assist and contextual help are provided based on the schema

NOTE *My XML editor tries to connect to the Internet!* Depending on the tool you use, an Internet connection might be required to fetch Mule schemas when a configuration file is opened. This happens with tools that don't understand the local redirection mechanism explained in chapter 2. Some XML editors require you to build a local catalog of schemas to be able to offer validation and code assistance. In that case, you can point the tool to the schemas embedded in the Mule libraries you're using in your project.

Your IDE's templating features can also simplify XML authoring. Templating allows you to create a skeleton Mule config that you can use as a baseline for new Mule projects. The template could include schema declarations typically used in your projects, shared connectors, or boilerplate comments. As you can tell, Clood makes extensive use of Mule. Most of their Mule configurations contain the same elements, such as a reference to a Spring config file containing bean definitions or a connector definition for their JMS broker. The Java developers at Clood who prefer IDEA as their IDE use a configuration template to simplify starting these configurations. Figure 12.4 illustrates a template that includes the elements we just mentioned.

This template includes the schema definitions commonly used in Clood projects, a Spring bean import element, a connector for Clood's ActiveMQ broker, and a stub model definition.

Now that we've seen some general ways that IDEs can simplify Mule development, let's look at Mule's explicit support for Eclipse through the use of the Mule IDE plug-in.

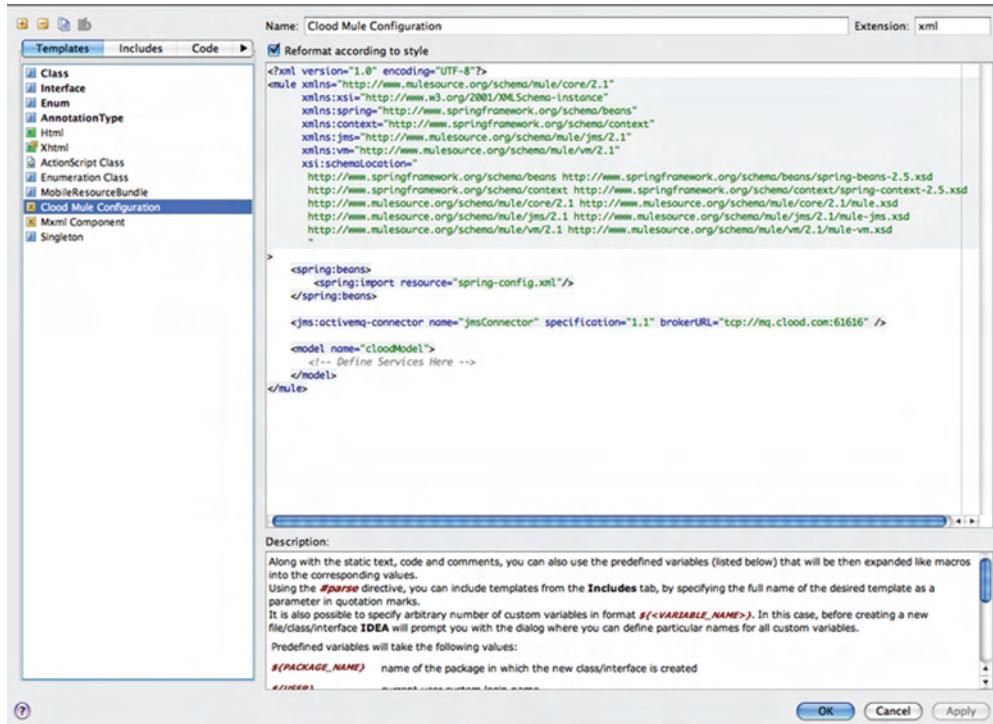


Figure 12.4 Creating a Mule configuration template in IDEA

12.2.2 Using Mule's IDE plug-in

If you're an Eclipse user, you can take advantage of the Mule IDE plug-in. This freely available plug-in provides convenience for working with Mule projects in Eclipse. In this section we'll look at how to install Mule IDE in Eclipse. We'll then use Eclipse to start a new Mule project using the Echo Example from chapter 1. Finally, we'll run the example from inside Eclipse using the Mule IDE.

If you don't have Eclipse installed, you can download it from <http://www.eclipse.org/>. Once it's running, you're ready to install Mule IDE. This is straightforward—select Software Updates from the Help menu. This'll bring you to the Software Updates and Add-ons panel. From here, click on Add Site and enter the following: <http://snapshots.dist.muleforge.org/mule-ide/updates-2.0-M1/>. This is illustrated in figure 12.5.

Clicking on OK will install the Mule IDE. You'll be asked to restart Eclipse when it's finished. Once this is done, you'll need to point Mule IDE at the Mule installation location on your machine. This'll enable you to start and stop Mule from Eclipse. To do this, open the Preferences menu, select Mule, and set it to the same value as your MULE_HOME value, as illustrated in figure 12.6.

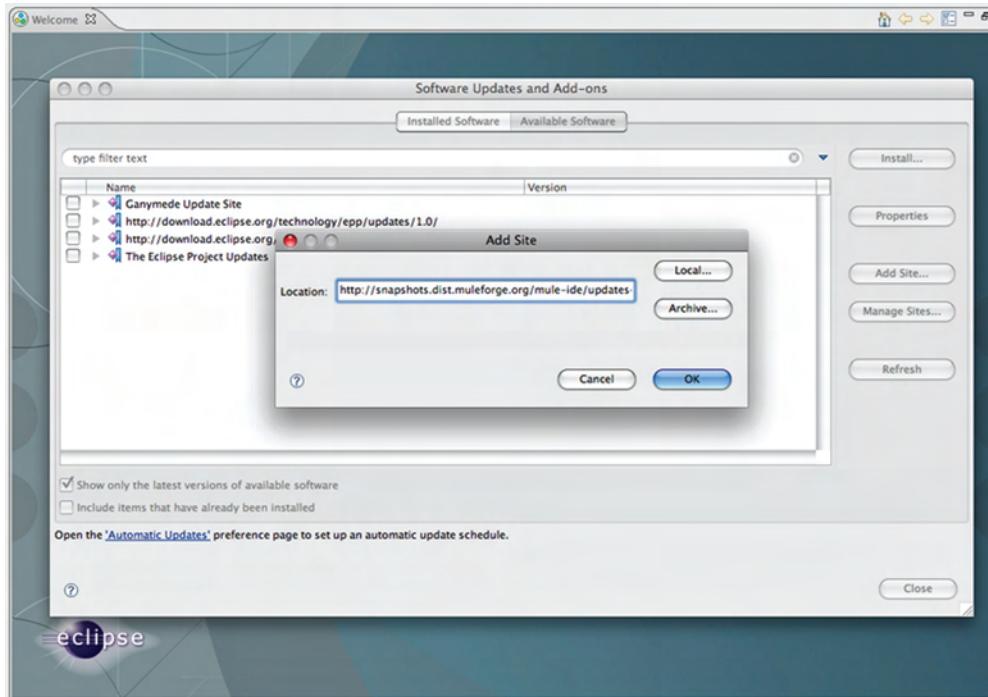


Figure 12.5 Installing the Mule IDE in Eclipse Ganymede

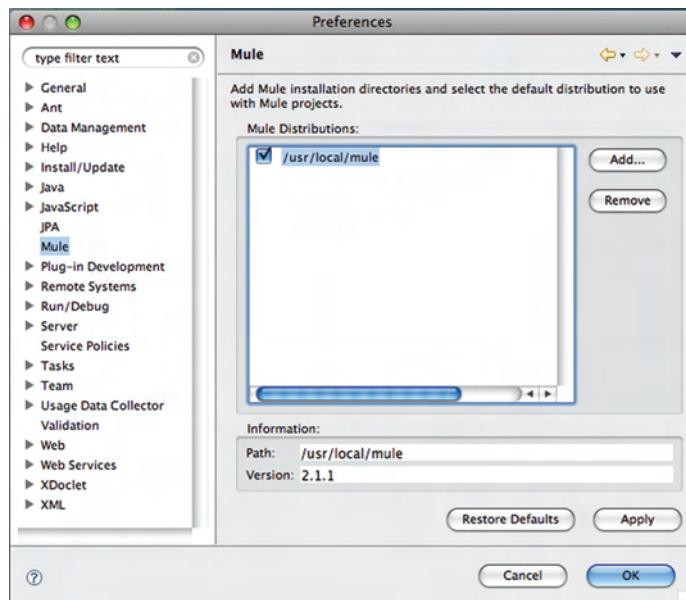


Figure 12.6 Setting the location of Mule for Mule IDE

Now let's start and run a Mule project. If you navigate to File > New > Project, you'll be presented with a New Project dialog. From here, select Mule Project. Give your project a name and at the bottom of the dialog, select Add Sample project content, and select the Echo Example. Once this is done, navigate to the WorkBench and you should wind up with a screen that looks something like figure 12.7.

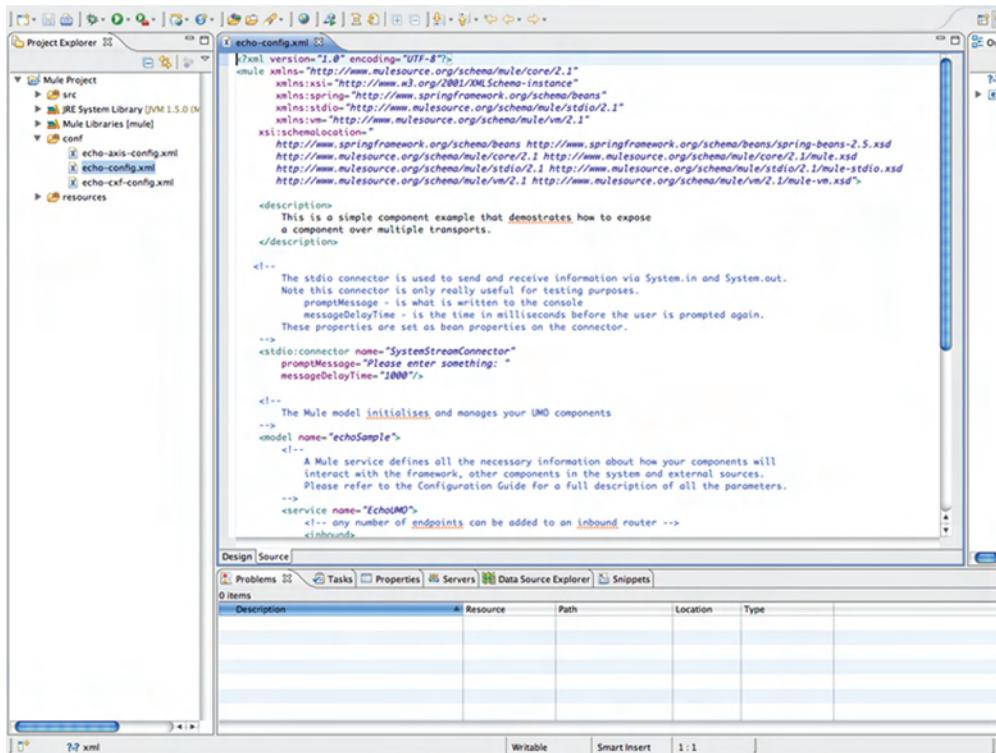


Figure 12.7 The Eclipse WorkBench after starting a new Mule project

We're now ready to run the Echo Example from inside Eclipse. Do the following to accomplish this:

- 1 Save your project from the File menu.
- 2 Select Run Configurations from the Run menu.
- 3 Select Local Mule Server on the left side of the panel.
- 4 Select the project to run.

Now you need to set the configuration file to use. As you can see from figure 12.8, this project has three config files: echo-axis-config.xml, echo-config.xml, and echo-cxf-config.xml. Click on the Add button next to the Configuration Files pane, and then select the echo-config.xml file. Your screen should resemble figure 12.8.

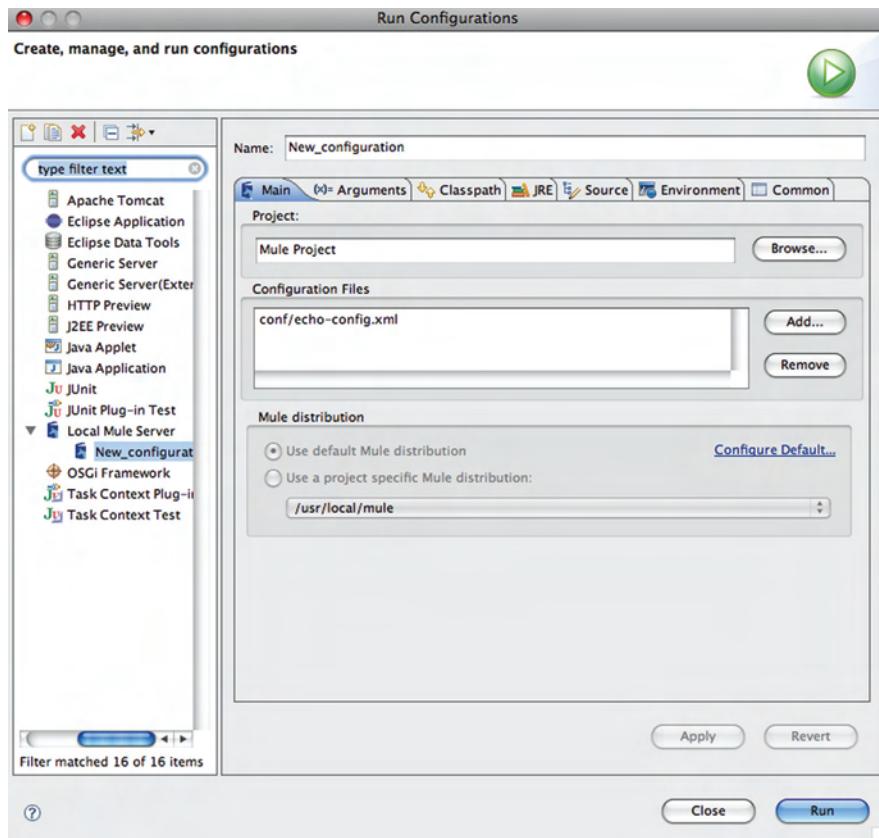


Figure 12.8 Setting up a run configuration for Mule in Eclipse

Clicking on Run should now start Mule running. You should see something like figure 12.9 on your console window in Eclipse.



Figure 12.9 A Mule instance running within Eclipse

NOTE *Provider org.apache.xerces.jaxp.SAXParserFactoryImpl not found* If you see Mule throw an exception that says something like “Provider org.apache.xerces.jaxp.SAXParserFactoryImpl not found” when you attempt to run the configuration, you need to add the Xerces JAR to Eclipse’s classpath. This can be done by going to the Run Configurations

menu, clicking on the Classpath tab, and adding the Xerces JAR to the classpath of your Mule project. A Xerces JAR is available in the lib/endorsed directory of the Mule distribution.

Congratulations! You can now run Mule directly from your development environment. Clicking on the red Stop button will stop the Mule process. If you want to start Mule again, you just need to click on the green Run icon in the toolbar.

NOTE Most modern IDEs feature extensive debugging capabilities. One such feature is remote debugging. This allows you to set up breakpoints and step debug a remote Java process while it's running. If your IDE supports JPDA (Java Platform Debugger Architecture), you can take advantage of this functionality with Mule once you get a little configuration out of the way. To enable JPDA in Mule, simply add the -debug flag when you run the Mule startup script. You can now connect your IDE's debugger to Mule on port 5005.

Mule's well-documented XML schema and IDE support can greatly simplify your experience working with Mule. You saw how an IDE, or any other competent XML editor, can take advantage of Mule's XML schemas to assist you when building your Mule configurations. You also saw how Mule IDE provides tight integration with the Eclipse IDE platform, enabling you to start and stop a Mule instance without leaving your development environment. We'll now see how to tackle another crucial part of the development process—testing.

12.3 Testing with Mule

Mule provides rich facilities to test your integration projects. We'll start off by seeing how to use Mule's functional testing capabilities to perform integration tests of your Mule configurations. We'll then take advantage of Mule's test namespace to use the test component to mock component behavior. We'll wrap up by taking a look at JMeter, an open source load testing tool, to load test Mule endpoints.

12.3.1 Functional testing

One way to perform integration testing on a Mule project is to manually start a Mule instance, send some messages to the endpoint in question, and manually verify that it worked. During development, this can be an effective technique, as we saw in section 12.2 with the Mule IDE. This process should ultimately be automated, allowing you to automatically verify the correctness of your projects. The `FunctionalTestCase` of Mule's Test Compatibility Kit (TCK) can be used for this purpose. The TCK can be used to test various aspects of Mule projects. The `FunctionalTestCase` allows you to bootstrap a Mule instance from a `TestCase` and then use the `MuleClient` to interact with it.

To demonstrate Mule's functional testing capabilities, let's start with a simple Mule project that uses the component and transformer we discussed in this chapter. The configuration in listing 12.13 illustrates a Mule service that accepts a JMS

message off a queue, processes it with the message enricher, transforms it with the VelocityTransformer, and then sends it to another JMS queue. Clood is using this process to accept data from a vendor, add an ORGANIZATION header to the message, format the message with a velocity template, and send the message to another queue for further processing.

Listing 12.13 Using the message enricher and VelocityTransformer

```

<spring:bean name="velocityEngine"
    class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
    <spring:property name="resourceLoaderPath" value="classpath:templates"/>
</spring:bean>                                Define VelocityEngine bean

<custom-transformer name="velocityPayloadTransformer"
    class="com.muleinaction.transformer.VelocityPayloadTransformer">
    <spring:property name="velocityEngine" ref="velocityEngine"/>
    <spring:property name="templateName" value="test-payload.vm"/>
</custom-transformer>

<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="vm://localhost"/>                ① Define ActiveMQ connector

<model name="FunctionalTestModel">
    <service name="FunctionalTestService">
        <inbound>
            <jms:inbound-endpoint queue="in"/>
        </inbound>
        <component class="com.clood.MessageEnricher"/>
        <outbound>
            <pass-through-router>
                <jms:outbound-endpoint queue="out">
                    <transformers>
                        <transformer ref="velocityPayloadTransformer"/>
                        <jms:object-to-jmssmessage-transformer/>
                    </transformers>
                </jms:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>
</model>

```

The code listing includes several annotations:

- A callout labeled "Define VelocityEngine bean" points to the first `<spring:bean>` element.
- A callout labeled "Define VelocityPayloadTransformer" points to the `<custom-transformer>` element.
- A callout labeled "Define ActiveMQ connector" points to the `<jms:activemq-connector>` element, which is annotated with a circled number "①".
- A callout labeled "Declare MessageEnricher component" points to the `<component class="com.clood.MessageEnricher"/>` element.
- A callout labeled "Declare outbound transformers" points to the `<transformers>` block under the outbound endpoint.

This configuration is fairly straightforward. One thing to note is the use of the VM broker for ActiveMQ on ①. This creates an in-memory ActiveMQ broker that lives for the duration of the functional test. Clood wants to avoid sending test messages through its production JMS infrastructure, so using an in-memory broker is a good compromise. Similar options exist for other transports. For example, you might want to use an in-memory database such as Apache Derby or HSQL for testing JDBC endpoints, or the GreenMail libraries for testing SMTP and IMAP endpoints.

BEST PRACTICE Use ActiveMQ's `vm` broker to test JMS endpoints without relying on an external JMS infrastructure.

Let's write a functional test for this configuration. Our goal is to start Mule, send a JMS message to the in queue, then consume a message off the out queue and confirm that the header and transformed payload are present. We'll use the `FunctionalTestCase` along with the `MuleClient` to get this done. Listing 12.14 illustrates this.

Listing 12.14 Performing an integration test of a Mule configuration

```

public class MessageEnricherFunctionalTest
    extends FunctionalTestCase {
    protected String getConfigResources() {
        return "conf/message-enricher-conf.xml";
    }

    public void testMessageTransformation() throws Exception {
        MuleClient muleClient = new MuleClient(muleContext);
        muleClient.sendAsync("jms://in", "TEST_PAYLOAD", null);
        MuleMessage response = muleClient.request("jms://out", 2000);
        assertEquals("****[MESSAGE=TEST_PAYLOAD]****",
            response.getPayload());
        assertNotNull(response.getProperty("ORGANIZATION", true));
        assertEquals("CLOUD",
            response.getProperty("ORGANIZATION", true));
    }
}

```

Specify location of Mule configuration

1 Construct MuleClient instance

2 Wait 2 seconds for message

3 Test that payload has been transformed

4 Test that appropriate header has been added

5 Test that header has appropriate

We start off by specifying the location of the Mule configuration on the classpath. If you're using Maven for your tests, the configuration file would be located in `src/test/resources/conf` (you'll need to create this directory if it doesn't exist.) We create a `MuleClient` on ① that uses the `muleContext` present in the `FunctionalTestCase` superclass. You'll learn much more about the `MuleClient` in the next chapter. For now, all you need to know is that the `MuleClient` is a facility for directly interacting with a Mule instance, which in this case is the Mule environment created by the `FunctionalTestCase`.

We're sending a JMS message where we don't expect a synchronous response, so we use the `sendAsync` method of `MuleClient` to send `TEST_PAYLOAD` to the in queue. The message will now be sent to the in-memory ActiveMQ broker we configured in listing 12.13. On ② we request a message off the JMS out queue with a 2-second timeout. Assuming everything went well, our transformed message with the added headers should be waiting for us patiently on the out queue. If we don't receive a message off the queue in that time frame, the test will fail. Assuming a message has been received, we'll begin our test assertions on ③, where we start by testing whether the payload has been transformed appropriately. If it has, we then test that the header has been set and has the appropriate value in ④ and ⑤.

NOTE The AbstractTransformerTestCase can simplify (and speed up) transformer testing at the expense of greater exposure to Mule internals. You should generally stick to extending the FunctionalTestCase and using a simple Mule config to test transformers, but you might consider the AbstractTransformerTestCase when the FunctionalTestCase proves to be too slow.

12.3.2 Mocking component behavior

The functional test we just saw tested whether a message was successfully processed. We also may want to test what happens when message processing fails or when invalid data is returned from a component. One way to approach this would be to mock the component implementation to return the dummy data. The downside of this approach is that you may need to maintain multiple mock implementations for each of your failure scenarios. For instance, you'd have a mock component implementation to return invalid data or another mock that would artificially delay processing to simulate a time-out of some sort.

The Mule TCK provides functionality to make this sort of testing easier. By using Mule's test component, you can return arbitrary data from a component or introduce delays in component processing. Let's see how this works by introducing a 3-second delay to the message processing demonstrated in listing 12.11. Listing 12.15 shows how to configure the test component to introduce the delay.

Listing 12.15 Using the test component to introduce a delay in component processing

```
<model name="FunctionalTestModel">
    <service name="FunctionalTestService">
        <inbound>
            <jms:inbound-endpoint queue="in"/>
        </inbound>
        <test:component waitTime="3000" /> ① Introduce delay in
        <outbound>                                component processing
            <pass-through-router>
                <jms:outbound-endpoint queue="out">
                    <transformers>
                        <transformer ref="velocityPayloadTransformer" />
                        <jms:object-to-jmsmessage-transformer/>
                    </transformers>
                </jms:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

Replacing the original component class definition with ① will cause Mule to pause for three seconds when the message is received by the component. You can use this to test that no message is emitted to the out queue during this interval, as illustrated in listing 12.16.

Listing 12.16 Confirming that no message is output before the delay interval expires

```
public class MessageEnricherFunctionalTestCase
    extends FunctionalTestCase {
    protected String getConfigResources() {
        return "conf/test-component-delay.xml";
    }
    public void testClientTimedOut() throws Exception {
        MuleClient muleClient = new MuleClient(muleContext);
        muleClient.sendAsync("jms://in", "TEST_PAYLOAD", null);
        MuleMessage response = muleClient.request("jms://out", 2000);
        assertNull(response);
    }
}
```

When testing components using this method, be sure to use the `MuleClient`'s synchronous send method. Doing otherwise will cause your test to exit before the messages reach the component in question. If this isn't an option, consider using the latching facilities in `java.util.concurrent` in conjunction with Mule's event notification features to block the test thread until the event is processed by the component.

BEST PRACTICE Use `MuleClient`'s synchronous send method when testing component behavior.

The test component can also be used to simulate exceptions in a component. This is useful when testing exception strategies work as expected. Listing 12.17 shows how to configure the test component to accomplish this.

Listing 12.17 Using the test component to simulate exceptions

```
<default-service-exception-strategy>
    <jms:outbound-endpoint queue="errors"/>
</default-service-exception-strategy>

<service name="FunctionalTestService">
    <inbound>
        <jms:inbound-endpoint queue="in"/>
    </inbound>
    <test:component
        exceptionToThrow="java.lang.RuntimeException"
        throwException="true"/>           ↪ 1 Throw declared
    <outbound>
        <pass-through-router>
            <jms:outbound-endpoint queue="out">
                <transformers>
                    <transformer ref="velocityPayloadTransformer"/>
                    <jms:object-to-jmsmessage-transformer/>
                </transformers>
            </jms:outbound-endpoint>
        </pass-through-router>
    </outbound>
</service>
```

When the component is invoked it'll now throw the exception defined by exception-ToThrow on ①. We can now test for this message on the errors queue.

In addition to introducing delays and throwing exceptions, the test component also allows you to return arbitrary content. This can be useful to test how a transformer or filter on an outbound endpoint responds to malformed data. Listing 12.18 illustrates how to configure the test component to return a defined String.

Listing 12.18 Using the test component to return an arbitrary String

```
<test:component>
    <test:return-data>
        TEST PAYLOAD
    </test:return-data>
</test:component>
```

You can also configure the test component to return the data in a specified file, as illustrated by listing 12.19.

Listing 12.19 Using the test component to return the contents of a file

```
<test:component>
    <test:return-data file="payload.txt"/>
</test:component>
```

By now you should be comfortable writing functional tests with Mule and mocking component behavior using the test component. We'll now see how we can perform load tests on a running Mule instance.

12.3.3 Load testing with JMeter

Extensive functional tests coupled with automated integration testing should provide a fair bit of reassurance that your Mule projects will run correctly when deployed. But the real world might have different plans. Perhaps Cloud's message transformation and enrichment service becomes wildly popular within the organization. Unaware of the demand for such a simple service, you deployed Mule on a spare blade server loaned to you by a colleague in operations. Mule performs admirably on this blade, easily processing a hundred or so messages every hour. Suddenly, you notice the service is crawling and the load on your lowly spare blade server is through the roof. You check with Cloud's JMS administrator and realize that thousands of messages are being sent to the in queue every hour. Since Cloud is a cloud computing company, after all, you decide to deploy Mule to Cloud's cloud—automatically provisioning Mule instances to consume messages when the message volume becomes to high. To do this, though, you need to determine how many messages a single Mule instance can handle.

Apache JMeter is an open source Java load generation tool that can be used for this purpose. JMeter allows you to generate different sorts of load for a variety of services, including JMS, HTTP, JDBC, and LDAP. You can use these facilities to load test Mule endpoints that use these transports. In this section, we'll examine how to use JMeter with JMS as we load test the service we performed integration testing on previously.

To start off, you'll need to download JMeter from http://jakarta.apache.org/site/downloads/downloads_jmeter.cgi. The example in this section will be testing JMS, so we need to provide our JMS provider's client libraries to JMeter before we start it up. This can be done by copying the appropriate JAR files to the lib/ subdirectory in the root of the JMeter distribution. Once this is done, you can start JMeter from the bin/ subdirectory by either double-clicking on the JMeter JAR file or by running the appropriate shell script for your platform. If all of this went correctly, you should see a screen resembling figure 12.10.

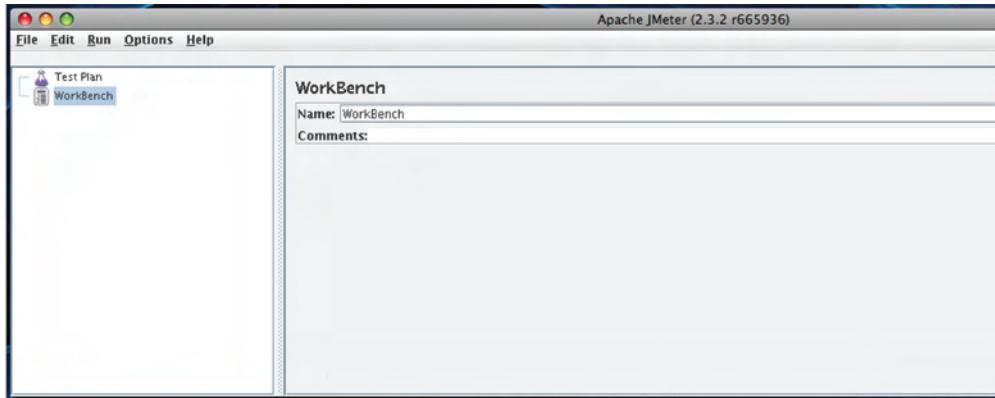


Figure 12.10 Starting JMeter for the first time

We'll start by specifying a test plan for a load test. Do this by right-clicking on the Test Plan icon, selecting Add, and then selecting Thread Group. The thread group is what we'll use to control the amount and concurrency of the JMS messages we'll send. After talking to Clood's JMS administrator a bit more, you manage to nail down that load problems start to occur with five concurrent users sending about 20 messages each. You figure this is as good a baseline as any to begin your load testing experiments, so you set their values in JMeter as illustrated in figure 12.11.

We set the name, and if you want, any comments. We want our tests to continue if there are any errors along the way. This could allow us to detect and debug failures in the JMS infrastructure itself, for example. Selecting Stop Thread here will cause the



Figure 12.11 Specifying a thread group to simulate five concurrent users sending 20 messages

thread that encountered the error to exit and not send any more messages. Stop Test would cause the entire test to stop when an error is encountered in any thread. The Number of Threads (users) is then set to 5. The Ramp-Up Period indicates the amount of time to delay starting each thread. In this case, we want all five threads to start at the same time, so we set the value to zero. The Loop Count indicates the number of messages we want each thread to send. In this case, we want each user to send 20 messages, and then stop. Clicking Forever here will cause each user to send messages indefinitely.

We now need to add a JMS *sampler* to the load test. A JMeter sampler is responsible for generating the traffic used for the test. Let's set up a JMS sampler by right-clicking on JMS Test, selecting Add, selecting Sampler, and then selecting Point-to-Point. The Point-to-Point sampler indicates that we want to generate data for a JMS queue. If you were publishing data to a topic instead, you'd use the JMS Publisher sampler. Conversely, if you wanted to test consuming off a topic, you'd use the JMS Subscriber sampler. This should lead to a screen that allows you to configure the details of your JMS broker. Figure 12.12 illustrates the configuration for Cloud's staging ActiveMQ broker.

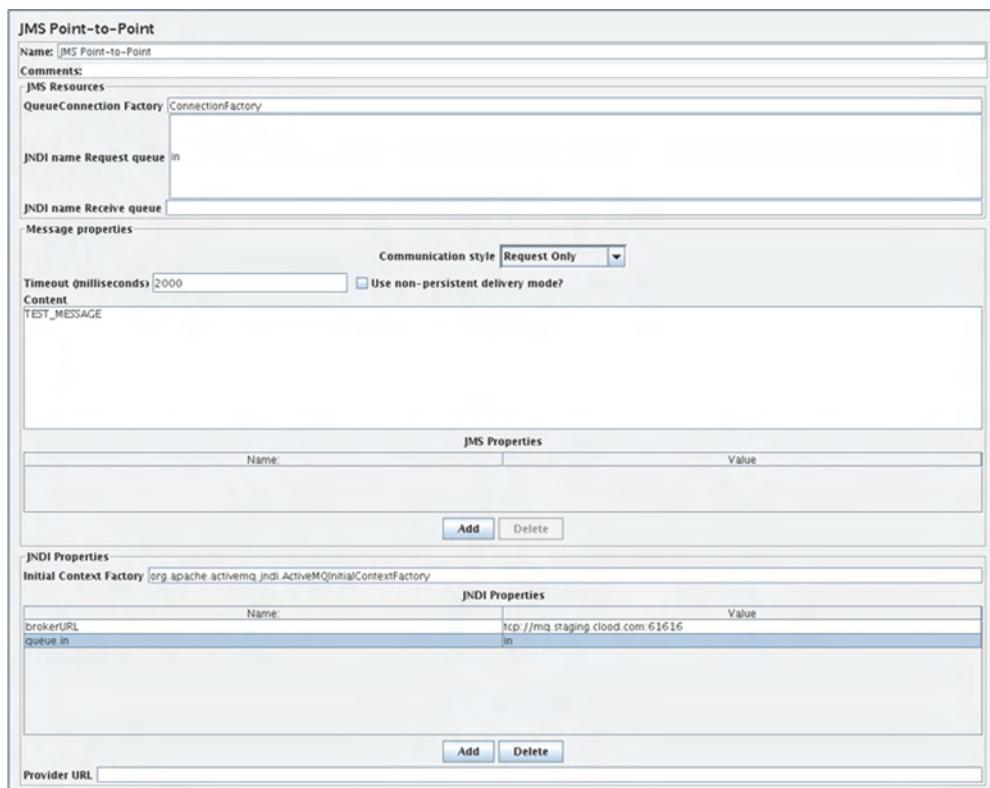


Figure 12.12 Configuring a JMS Point-to-Point sampler with JMeter

We're not interested in what happens after the messages are sent out by Mule, so the Communication style is set to Request Only. After filling out the remainder of the properties for our JMS broker, we're almost ready to start our load test. We could start the test now, but we want to collect some data as we run the tests. This is accomplished by adding *listeners* to the thread group. Right click on the thread group, select Add, then select Listener and select a listener you'd like to use to interpret the test results. For our case, we're going to add the Aggregate Graph, View Results in Table, and Graph Results. We can now run the test and view the results. Click on the Run menu and then select Start. You should see a green light on the upper-right side of the screen go on as the test runs.³

If you click on any of the listeners as the test is running, you should see the results being populated. This should resemble figure 12.13.

The listener results will be appended as subsequent tests are run. This'll allow you to trend things such as throughput as you increase the load of your tests. You can now use this test to load test the message enricher on the VM. By examining the message throughput as you increase the number of threads and messages they send, you should be able to identify the point at which performance 16 to try to squeeze more performance out of the Mule instance. Ultimately you should be able to get a good idea of how many messages the given Mule instance on the given platform can handle—allowing you to make intelligent scaling decisions.

We've only scratched the surface of JMeter's capabilities. As mentioned in the introduction, JMeter can test transports other than JMS. You can even implement your

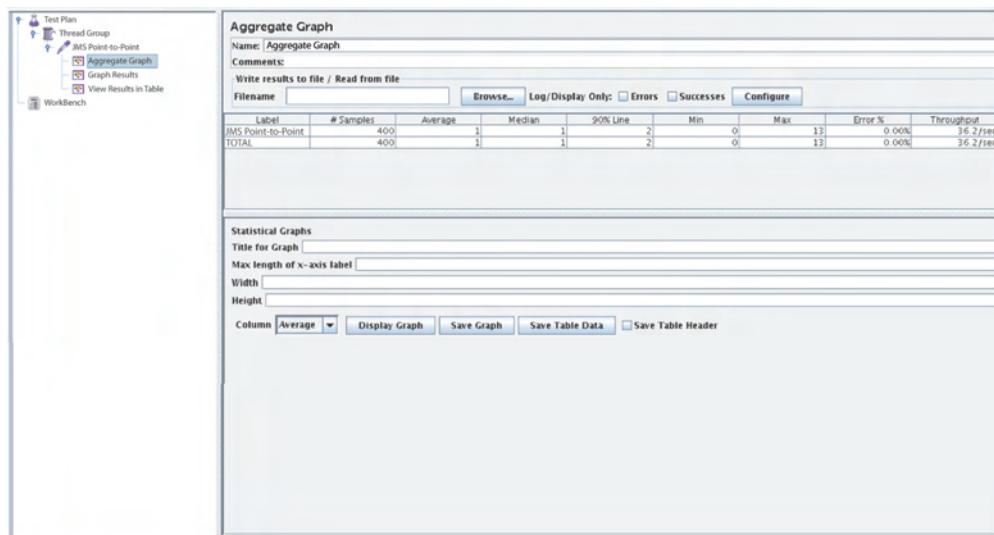


Figure 12.13 Examining results of a JMeter listener

³ If your test fails to run, or if any failures occur during the test, you can view them in the `jmeter.log` file located in the `bin` subdirectory you used to launch JMeter.

own samplers if there isn't a presupplied one for the endpoint you wish to test. Test automation is also available, allowing you to integrate load tests into your build processes or IDE. Full documentation is available from the JMeter site.

By now you should be comfortable testing your Mule projects. We saw how Mule's Test Compatibility Kit facilitates functional testing of your components, transformers, and Mule configurations. We then saw how JMeter, an open source load testing framework, can be leveraged to load test your Mule endpoints.

12.4 Summary

The value of good development tools can't be underestimated. Mule's ease of use with popular build management, IDE, and testing tools greatly eases the burden typically associated with complex integration projects. By now, you should be confident managing your Mule builds with Maven, using Mule with your IDE of choice, and extensively testing your Mule projects.

In this chapter you also saw more of Mule's API. You saw how the Mule TCK simplifies unit and integration testing. We also saw a glimpse of the `MuleClient`, a powerful facility for interacting directly with Mule instances. This exploration into Mule's internals will be continued now as we investigate the Mule API.

13

Using the Mule API

In this chapter

- Communicating with the MuleClient
- Using context objects to interact with Mule
- AOP with component interceptors
- Leveraging the notification framework

The previous 12 chapters of this book should've convinced you that a lot can be achieved with Mule with minimal coding, or at least without any Mule-specific code to write. For example, we've seen that Mule goes to great lengths to let you reuse your existing business logic as-is. This is great news because no one wants to write code that's coupled to a particular framework, as it creates a potential implementation lock-in and weakens the code by making it sensitive to API changes.

But there are times when it's worth considering the trade-off between framework coupling and the advanced features it offers. Using Mule's API allows you to implement advanced behaviors that are cumbersome or even impossible to roll out when staying away from the framework. In fact, since Mule is "a lightweight messaging framework and highly distributable object broker," staying away from its API would amount to denying half of its nature.

Because there are more than 2,100 classes in Mule 2.2, this chapter can by no means be an exhaustive tour of all of them. Essentially, this chapter will give you the necessary pointers you need to approach Mule's API in an efficient and productive manner. Instead of considering the API as a static resource, we look at it as a gateway to Mule's live moving parts. Along the way, we'll come back to several Cloud examples we've mentioned in the previous chapters. You'll see how Cloud, Inc., makes good use of the API, which'll give you a better and deeper insight on how things work inside of Mule.

Let's start by discovering the Mule client, a useful part of the API that you'll happily piggyback soon!

13.1 Piggybacking the Mule client

The Mule client isn't part of the core library, but is bundled in the mule-module-client module. As we'll see in the coming sections, its main purpose is to allow you to interact with a Mule instance, whether it's local (running in the same JVM) or remote (running in another JVM). We'll also look at how the client can bootstrap and shut down an embedded Mule instance for you, which can be used to directly tap into the transports infrastructure of Mule.

The Mule client provides you with a rich messaging abstraction and a complete isolation from the transports' particularities. The client supports the three main types of interactions that are common in Mule:

- *Send message*—A synchronous operation that waits for a response to the message it sent (relies on the message dispatcher infrastructure)
- *Dispatch message*—An asynchronous operation that expects no response for the message it dispatched (also relies on the message dispatcher infrastructure)
- *Request message*—An operation that doesn't send anything, but synchronously consumes a message from an endpoint (relies on the message requester infrastructure)

All these operations are offered under numerous variations that allow you, for example, to directly target a service instead of its inbound endpoint, receive a future¹ message, or disregard the response of a synchronous call.

NOTE It's possible, but not trivial, to involve several operations of the Mule client in a single transaction, provided of course that the transports used are transactional. This is seldom necessary and won't be detailed here. You can study `org.mule.test.integration.client.MuleClientTransactionTestCase` for more information.

The Mule client main class is `org.mule.module.client.MuleClient`, which is thread-safe. This class offers several different constructors: the one you need to use depends

¹ As in `java.util.concurrent.Future`—the pending result of a computation that can be waited on or polled regularly for a result.

on the context in which you'll utilize the client. The main idea is that by using a particular constructor, you instruct the client to use an existing Mule context (detailed in section 13.2), look for an existing context in memory, or create a new one by bootstrapping a Mule instance. We'll review some of these constructors in the upcoming examples.

Let's start by discovering how the client can be used to interact with a local Mule instance.

13.1.1 Reaching a local Mule

The most common usage of the Mule client is to interact with an already running local Mule instance—an instance that runs within the same JVM. This is illustrated in figure 13.1.

In-memory calls from the client to a local Mule are common in integration testing (see the discussion in chapter 12). The whole Mule test suite, as well as the majority of the examples that accompany this book, uses the Mule client, for that matter. Listing 13.1 shows an excerpt of a method that Clood, Inc., uses to run integration tests on their MD5 file hasher component (we talked about this component in chapter 6).

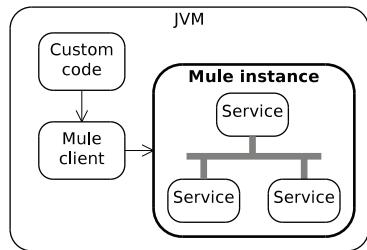


Figure 13.1 The client is a convenient way to reach a Mule instance within the same VM.

Listing 13.1 The client facilitates testing by granting access to a Mule instance.

```
MuleClient muleClient = new MuleClient(muleContext); ① Create client for existing Mule instance context
String actualHash =
    muleClient.send("vm://Md5FileHasher.In",
        tempFileName, null).getPayload(); ② Use client to perform synchronous send to endpoint
assertEquals(expectedHash, actualHash);
```

Because this test runs in a class where the Mule context is available, we use the specific client constructor that accepts it on ①.

When your code runs in the same JVM where a Mule instance is already running, the client can locate it without the need to pass it a reference to the context. In our case, we could've just done that and used the parameterless constructor, as shown here:

```
MuleClient muleClient = new MuleClient();
```

BEST PRACTICE If you have access to the Mule context, pass it to the client constructor as a way to eliminate ambiguity for anybody else reading your code.

You may have noticed that in listing 13.1 we send a message to a VM transport URI. You may wonder if we're constrained to only use the VM transport when the client is

connected to a local Mule instance. This isn't the case, and to illustrate that other transports can be used, we'll rewrite Clood, Inc.'s file hasher component using the Mule client.

Listing 13.2 shows how the invocation of the file transport is performed by constructing a URI that it understands. By this means, the component synchronously requests a file to be read from the filesystem. If you turn back to listing 6.12, you'll notice that the source folder is configured on the component itself. In fact, the only variable part in this invocation is the file name.

Listing 13.2 A client connected to an in-JVM Mule instance can use any transport.

```
MuleClient muleClient = new MuleClient(eventContext
    .getMuleContext());

MuleMessage requestedFileMessage =
    muleClient.request("file://" +
        + sourceFolder
        + "/" + fileName
        + "?connector="
        + fileConnectorName, 0);
```

Note how the Mule client is instantiated by using the constructor that accepts a Mule context, which itself comes from the event context (discussed in section 13.3). We could've relied on the autodetection of the current Mule context by the client, but since we had direct access to it, we opted for the constructor that accepts it explicitly. Instantiating a Mule client for each request processed by the component isn't efficient: we'll review this component again in a coming section.

TIP *Connector selector* You've surely noticed that the URI we build in 13.2 ends with a connector parameter added to the file URI. Since the client API can't accommodate all the possible parameters that can be involved in a particular request, you have to pass any extra argument in the URI itself.

In our case, we want to use a specific file connector configured to not delete the files it reads. Our component has been configured with a reference to this connector, and we refer to it by its name when we form a request so we're sure Mule will use it.

You've just discovered how the Mule client facilitates connectivity with a local instance running in the same JVM. What if the Mule instance you want to connect to runs in another JVM? What if it runs in a remote server? The Mule client can be used in this case also, as we'll now see.

13.1.2 Reaching a remote Mule

The Mule client can connect to a remote instance and use the same richness of interaction it uses with a local instance. For this to work, a specific remote dispatcher agent must be configured in the Mule instance that the client wants to connect to, as illustrated in figure 13.2. Because the dispatcher agent is simply a particular consumer for a standard endpoint, any protocol that supports synchronous communications can be

used. On top of this protocol, a configurable (and customizable) format is used to represent the data that's sent over the wire in both directions (the client request to Mule and its response).

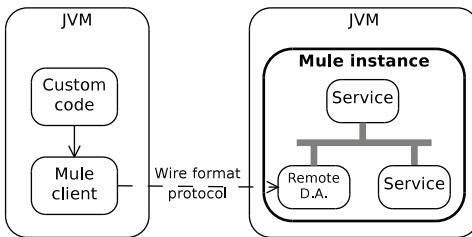


Figure 13.2 The client can be used to reach a remote Mule instance by sending messages to a dedicated dispatcher agent.

Listing 13.3 shows the configuration of a Mule instance that'll accept client connections over TCP, using the default wire format. Note that the remote dispatcher agent is provided by the mule-module-client module. Technically, the standard wire formats are pairs of round-trip transformers that take care of transforming `MuleMessages` (payload and properties) to and from byte arrays. The default wire format relies on standard Java serialization, hence can only carry serializable payloads.

Listing 13.3 A remote dispatcher agent using the default wire format over TCP

```

<client:remote-dispatcher-agent>
    <client:remote-endpoint
        address="tcp://localhost:5555"
        synchronous="true" />
</client:remote-dispatcher-agent>

```

Once the remote dispatcher agent has been configured, it's possible to connect to it and then, through this connection, access all the messaging infrastructure of the remote Mule. The client we've seen so far is unable to do this directly but must be used to spawn a specific client for the remote dispatcher agent. This is shown in listing 13.4.

Listing 13.4 The client acts as a factory for creating remote dispatchers.

```

MuleClient muleClient = new MuleClient(false);           ① Creating context-
                                                        less stopped client
RemoteDispatcher remoteDispatcher =
    muleClient
    .getRemoteDispatcher("tcp://localhost:5555");
FutureMessageResult asyncResponse =
    remoteDispatcher
    .sendAsyncRemote("TickerLookupChannel",
                    "GOOG",
                    null);

```

② Using client to build
remote dispatcher
for TCP URI

③ Sending message to endpoint
named TickerLookupChannel

The client for the remote dispatcher agent, which is an instance of `org.mule.module.client.RemoteDispatcher`, is created in ②. This object is thread-safe, too.

In this example, we perform an asynchronous send to an endpoint named TickerLookupChannel with a message payload of GOOG and no properties. Note that the sendAsyncRemote method does the same thing as the local Mule client's send-Async method. Other methods have slightly different names: for example, the methods that send directly to a component are suffixed with ToRemoteComponent instead of Direct.

Note in ① how we create a Mule client without providing it a context and asking it, with the false parameter, not to look for or create one. Because we're only interested in the remote Mule we connect to, there's no reason for the Mule client to try to interact with a local instance.

Once we're done using the remote dispatcher, we can get rid of it by simply disposing of the Mule client that created it:

```
muleClient.dispose();
```

Note that disposing a Mule client terminates all the remote dispatchers it could've created.

WARNING *All your clients are belong to us* The remote dispatcher is a convenient way to reach a remote Mule instance through a feature-rich back door. Should it be the only way? Of course not! It's still valid to use any relevant transport to reach services hosted on Mule. For example, you can still use an HTTP API to form a request to a service hosted on Mule and exposed over HTTP. Or you can still use a JMS client to send to a destination consumed by a Mule service.

Consequently, if you contemplate using the Mule client, take into account its footprint in terms of transitive dependencies and the tight coupling between your client code and Mule that its usage induces. You'll also want to assess your actual need for the messaging abstraction the Mule client proposes.

We've mentioned that the default wire format is based on standard Java serialization. This implies that you can only send serializable objects to a remote Mule instance. If this limitation is an issue for you or if sending binary content over the wire isn't an option, then you can consider using the XML wire format. This format relies on XStream, which can serialize any Java object to an XML representation. Listing 13.5 presents this wire format option combined with an HTTP endpoint. With this configuration, your client and the remote Mule instance will communicate with XML over HTTP, something that might be more palatable for a firewall than the raw binary over TCP of the previous example.

Listing 13.5 A remote dispatcher agent using the XML wire format over HTTP

```
<client:remote-dispatcher-agent>
  <client:remote-endpoint ref="RemoteDispatcherChannel" />
  <client:xml-wire-format />
</client:remote-dispatcher-agent>
```

```
<http:endpoint name="RemoteDispatcherChannel"
    host="localhost"
    port="8181"
    synchronous="true" />
```

You've surely observed in listing 13.5 how we created a global HTTP endpoint and referenced it from the remote dispatcher agent. This is by no means a requirement: we could've declared the HTTP endpoint on the `remote-endpoint` element, as in listing 13.3. The idea was to demonstrate that the remote dispatcher agent leverages the standard transport infrastructure of the Mule instance that hosts it.

Now look at listing 13.6, which demonstrates how we create a client for this HTTP remote dispatcher agent. Except for the URI that has changed, there isn't much difference from the previous code: where's the configuration of the XML wire format? It's nowhere to be found, because the client doesn't decide the wire format: it's imposed on it by the distant Mule instance during the initial handshake that occurs when the remote dispatcher is created.

Listing 13.6 The client can create a remote dispatcher that connects over HTTP.

```
MuleClient muleClient = new MuleClient(false);
RemoteDispatcher remoteDispatcher =
    muleClient.getRemoteDispatcher("http://localhost:8181");
```

If none of the existing wire formats satisfy your needs, you can roll your own by creating a custom implementation of `org.mule.api.transformer.wire.WireFormat`.

NOTE *Did you say back door?* Yes, we referred to the remote dispatcher agent as a back door to a running Mule instance because this is what it is. This should naturally raise some legitimate security concerns. At the time of this writing, the remote dispatcher agent doesn't support security during the handshake phase, so it's not possible to secure the endpoint it uses.

This said, it's possible to secure the services hosted by the remote Mule instance, as detailed in chapter 9. If you follow that path, you'll need to pass a user name and a password to the remote dispatcher when it's created:

```
RemoteDispatcher remoteDispatcher =
    muleClient.getRemoteDispatcher("http://localhost:8181",
        username,
        password);
```

These credentials will then be used for the subsequent remote calls sent through it.

You now have a new power tool in your box: the remote dispatcher. It's a convenient means for sending messages to any remote Mule instance that has a corresponding agent enabled, but its usage must be considered with circumspection, as it has security and coupling trade-offs.

If you find the messaging abstraction offered by the client to be seductive, then the next section will be music to your ears. In it, you'll learn how to use the client to benefit from Mule's messaging infrastructure without even configuring it.

13.1.3 Reaching out with transports

In the opening discussion of this section, we said that the client can bootstrap a Mule instance if no context is passed to it or is detectable by it. This capability can be leveraged to directly exploit Mule's transports by bootstrapping a serviceless instance and using the messaging abstraction directly offered by the client. This approach, which is illustrated in figure 13.3, truly promotes Mule as an integration framework on which an application can lean to connect to remote services without having to deal with the particularities of the protocols involved.

Listing 13.7 shows the code that makes this happen. The client is started without any parameter, which makes it look for an existing Mule instance in memory. Because it doesn't find one, it bootstraps an empty instance. This empty instance can be used from the client to reach any URI whose related transport library is available in the classpath.

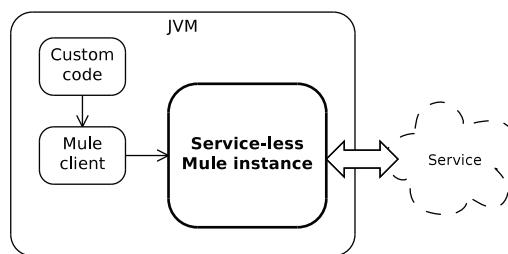


Figure 13.3 The client can bootstrap a minimal Mule instance for the sole purpose of tapping its transport layer in order to reach remote services.

Listing 13.7 Using the client to directly tap Mule's messaging infrastructure

```

MuleClient muleClient = new MuleClient(false);
MuleMessage response =
    muleClient.send(
        "http://www.google.com/finance/historical?",
        q="NASDAQ:GOOG&histperiod=weekly&output=csv",
        null, null);
String payload = response.getPayloadAsString();
muleClient.dispose();   ← Disposes client
  
```

Creates empty Mule instance and client connected to it

Performs synchronous send using HTTP transport

Retrieves result of HTTP call as string

In the previous example, the `getPayloadAsString()` may look benign, but behind the scenes, it relies on the powerful message adaptation mechanism we talked about in section 5.1. Independently of the transport used, Mule will be able to give you a byte or String representation of the message thanks to this mechanism. The message adaptation mechanism also took care of copying all the HTTP headers returned by Google in the `response` object properties.

Be aware that the default transformers defined on the transport itself have kicked in during the `send` operation. Should you need to configure other default transformers, as discussed in section 3.1.1, or should the transport you use need to be configured,

you'll have to create a minimal Mule configuration and load it from the client. This would be required if you use the JMS transport, as shown in listing 13.8. The raw-jms-muleclient-config.xml configuration file² still doesn't contain any service.

Listing 13.8 The client can bootstrap a Mule instance and tap transports directly.

```
MuleClient muleClient =
    new MuleClient("conf/raw-jms-muleclient-config.xml"); Creates client and configures Mule

muleClient.getMuleContext().start(); 1 Starts the Mule instance

MuleMessage response =
    muleClient.request("jms://" +
        queueName +
        "?connector=amqConnector", 1000); 2 Performs synchronous read using JMS transport

muleClient.getMuleContext().dispose(); 3 Disposes configured Mule instance, then client
muleClient.dispose();
```

Because there's no constructor of the client that loads a configuration in Mule and starts it, we had to do it explicitly in ①. As we showed before, we defined the connector name we wanted to use in the request URI in ②. This would allow us to configure several connectors in the same Mule configuration and select the one we want to use in the client code. The same way we started the instance explicitly, we have to dispose of it in ③ before terminating the client.

As we've seen, directly tapping Mule's transports is made easy thanks to the client. This approach brings the power of Mule's connectors, adaptors, and transformers to any of your applications. By delegating all transport-handling aspects to Mule, you can build communicating systems on top of a neat messaging abstraction layer.

By now, the Mule client must've become an important tool in your Mule toolbox, as it's a convenient gateway to an instance, whether this instance is running locally or in a remote JVM. We've mentioned that to connect to a local instance, the client needs a reference to its Mule context. This context is like a box of chocolates waiting to reward the audacious explorer. Let's now discover the treasures it holds.

13.2 Exploring the Mule context

In the previous section of this chapter, and in previous chapters as well, you've seen and heard about the Mule context. You surely understand that it's an in-memory handle to a running Mule instance. We've been using it to configure the Mule client, so it connects to the instance whose context was in our possession. But there's way more you can do with the Mule context. In fact, once you get ahold of it, it's party time: all Mule's internals become accessible!

Figure 13.4 is a partial dependency graph that represents the main ways for retrieving an instance of the Mule context and some moving parts you can reach from it. It's far from a complete representation of reality: if you look at the JavaDoc of the

² Available in the provided code samples.

`org.mule.api.MuleContext` class and follow some of the classes it links, such as the registry, you'll find way more objects than represented here. Nevertheless, for the purpose of this discussion, we'll focus on a few significant aspects and let you explore the rest of the context on your own.

The Mule context is accessible in several ways:

- *Create it*—You can create a Mule context by using a factory and loading a specific configuration in it. This has been demonstrated in section 7.1.3.
- *Get it from the client*—If you use the client to bootstrap a Mule instance, you can ask it for the context it's created behind the scene. We used this approach in listing 13.8.
- *Get it from the server*—The `org.mule.MuleServer` class exposes a static accessor to the context of the currently running Mule instance. This is the method to use in a custom object that runs within Mule but doesn't implement any of its interfaces or extend any of its superclasses, like a pure POJO component.
- *Get it from the event context*—Components implementing `org.mule.api.lifecycle.Callable` receive an event context which, among other things, gives direct access to the Mule context. We used it in listing 13.2.
- *Receive it by injection*—If you write a custom class that implements `org.mule.api.MuleContextAware` and is stored in the registry, you'll receive a reference to the current Mule context just before it gets initialized (see section 13.4.1).

Let's now review some of the actions you can perform once you have a reference to the context.

NOTE Using the expression manager is discussed in appendix A.

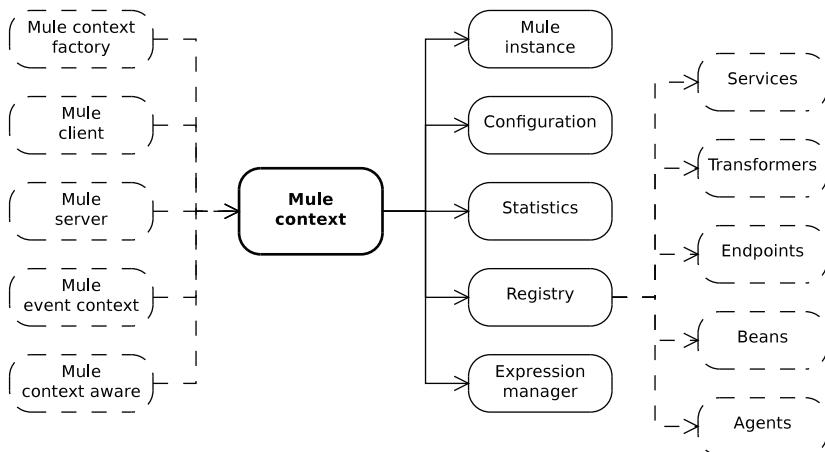


Figure 13.4 There are many ways to get hold of the Mule Context, which is the gateway to Mule's inner workings.

13.2.1 Controlling a Mule instance

The primary function of the context is to control the overall lifecycle of a Mule instance. Consequently, if you create a context using the factory, you need to call the following method after loading your configuration file(s):

```
muleContext.start();
```

Similarly, when you want to cleanly terminate a Mule instance, you need to execute the following:

```
muleContext.dispose();
```

NOTE The context also offers a stop() method that can be used to suspend a Mule instance, which can be resumed with a subsequent call to start(). Note that, as of this writing, Mule 2.2 doesn't unregister all its MBeans on a stop operation, which prevents a successful restart. Hence suspension can only be used on a Mule instance with no JMX management features activated.

13.2.2 Reading the configuration

The Mule context holds a reference to an object that details the system configuration of the instance. This object, which is immutable after the instance is started, is an instance of org.mule.api.config.MuleConfiguration. Though the vast majority of all configuration values you'll handle will come from XML or properties files and be directly injected into your objects, some values are computed by Mule itself and made available via the configuration object.

You may recall the discussion in section 11.1.3 concerning the necessity to name your Mule instance so you can form MBean object names and query them. If you don't name your instance, Mule generates a unique name for it on the fly. Because the configuration object allows you to get this ID, if you have access to the Mule context, you can build MBean names that are guaranteed to work whether you name your instance or not. This is illustrated in listing 13.9.

Listing 13.9 The Mule context gives access to the whole system configuration.

```
String serverId = muleContext.getConfiguration().getId();

ObjectName listenerObjectName = ObjectName
    .getInstance(JmxSupport.DEFAULT_JMX_DOMAIN_PREFIX
    + "."
    + serverId
    + ":"
    + JmxServerNotificationAgent.LISTENER_JMX_OBJECT_NAME);
```

13.2.3 Accessing statistics

Mule keeps detailed statistics for all its moving parts. In an instance with numerous services, these statistics are a convenient means to keep track of the activity of a Mule instance. In section 11.3, we reviewed the dashboard that Cloud, Inc., built that

leverages these statistics to build a snapshot of the situation of a particular instance. We use another component to regularly dump the full statistics of an instance in XML files. The logic inside this component, which is shown in listing 13.10, is simple, thanks to the supporting classes offered by Mule.

Listing 13.10 The Mule context gives access to the global statistics.

```
AllStatistics allStatistics = muleContext.getStatistics();
allStatistics.logSummary(new XMLPrinter(xmlStatisticsWriter));
```

13.2.4 Looking up the registry

One of the richest objects the context gives you access to is the registry, where, as its name suggests, all Mule's moving parts are registered and accessible via getters or lookup methods. When using a Spring XML configuration, the registry is created and populated for you. In that case, it's composed of a Mule-specific transient registry, which contains non-Spring objects, and that can delegate to a read-only Spring bean factory. This means that you can access all the Mule objects and your Spring-handled objects through the registry.

In term of software design, dependency injection should be preferred to registry lookups: this is why most of the time you'll directly inject the necessary dependencies into your custom objects. Lookups become relevant when the required resource is only known at runtime, hence can't be statically configured. This is the case for Cloud's XML statistics component. As we said earlier, this component can output the whole of the statistics of an instance, but can also do so for only one service, if its name has been sent to the component. Because of the dynamic nature of this behavior, we use the registry to look up the component by name and then render its statistics, as shown in listing 13.11.

Listing 13.11 In the context, the registry gives access to all Mule's moving parts.

```
Service service = muleContext.getRegistry()
    .lookupService(serviceName);

service.getStatistics().logSummary(
    new XMLPrinter(xmlStatisticsWriter));
```

There is another situation when you should prefer lookup over injection: when no configuration element can be referenced (thus injected). In chapter 11, we talked about the `jmx-default-config` configuration element that automatically configures a bunch of agents. This element registers each agent it creates under a known name, so if you needed to reach one of these agents, as shown in listing 13.12, you'd need to perform a registry lookup.

Listing 13.12 The Mule registry can be used to look up JMX agents.

```
JmxAgent jmxAgent =
    (JmxAgent) muleContext.getRegistry()
        .lookupAgent("jmx-agent");

MBeanServer mBeanServer = jmxAgent.getMBeanServer();
```

TIP *Registry reloaded* It's possible to store your own objects in the registry with the `registerObject` method and retrieve them with `lookupObject`. Though thread-safe, the registry shouldn't be mistaken for a general-purpose object cache. If this is what your application needs, you'd better use a dedicated one and have it injected in your components or transformers.

It's also possible to access in the registry the `ApplicationContext` that's created behind the scenes when you use the Spring XML configuration format:

```
ApplicationContext ac =
    (ApplicationContext) muleContext.getRegistry().lookupObject(
        SpringRegistry.SPRING_APPLICATION_CONTEXT);
```

This application context contains not only your Spring beans but also all Mule's moving parts configured in the XML file.

As suggested by figure 13.4, the Mule context gives access to the innards of the whole instance. The few examples we've just gone through should've whetted your appetite for more. We can't possibly fully explore the Mule context in this book, as we may end up putting the whole JavaDoc API in print, but we believe the pointers we've given you will allow you to find your way.

We'll now discover another useful context that exists only when a message event is processed in Mule.

13.3 Digging the Mule event context

The event context is conceptually a façade to Mule's messaging infrastructure that's scoped to the event currently under processing. It's an instance of `org.mule.api.MuleEventContext` and shouldn't be confused with `org.mule.api.MuleContext`, which we discovered in the previous section. As represented in figure 13.5, the event context methods delegate to several objects: the current event, the current transaction, and the current session. It also gives direct access to the message, transaction, session, and service objects, should you need to use some of their methods that aren't exposed at the event context level.

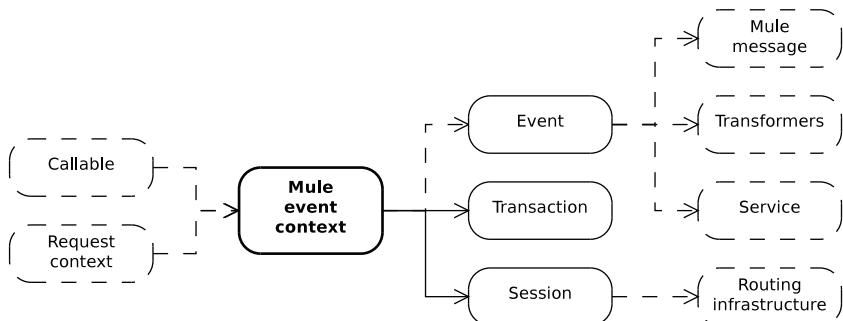


Figure 13.5 The Mule event context is a façade on objects that expose operations relevant to the event currently being processed.

In the coming sections, we'll look closely at the message features accessible via the event context and how you can leverage them in your components. We'll also discuss how the event context allows you to influence Mule's message-processing flow.

If you remember our discussion about messages and events in section 1.3.6, you should recall that technically Mule processes events, and not messages directly. This is why the event context delegates to the event to access message-related objects and actions. The transaction exposes methods that allow checking and modifying the state of the current transactional context. The session gives access to the message-routing infrastructure of Mule, with the familiar dispatch/send/request operations.

The event context is accessible in several ways:

- *Receive it when called*—Components implementing `org.mule.api.lifecycle.Callable` receive an event context instance as the unique argument of the `onCall` method.
- *Get it from the request context*—The `org.mule.RequestContext` class exposes a static accessor to the event context relevant for the message currently under processing.

Let's start by looking at the message-related features exposed by the event context.

13.3.1 Prospecting messages

The event context exposes a few methods that give access to the message payload (which we will detail in a moment). Thanks to the event reference it holds, the event context also exposes a reference to the `org.mule.api.MuleMessage` object, represented in figure 13.6, so all its content and operations become accessible. By now, you should be familiar with the properties, payload, and attachment parts of the message object. Most of the time, once you have access to the message, you can alter these values, but some messages have an immutable adapter that prevents payload modifications. This adapter is the moving part that performs the hard work of abstracting out the transport-specific particularities (as we discussed in section 5.1).

The message can also hold a reference to an instance of `org.mule.api.ExceptionPayload` if an error occurred while Mule was processing the current event. If you intend to build a custom audit trail log based on the content of Mule messages, don't forget to look for a non-null exception payload, as this is where you can find the details that allow you to perform effective forensics (see section 11.2).

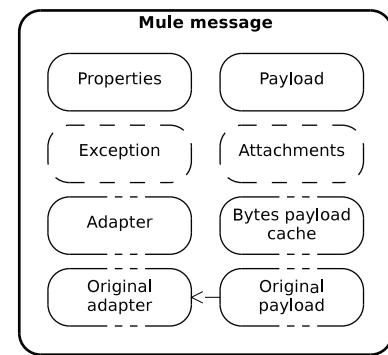


Figure 13.6 A Mule message object contains properties and a payload, and optionally an exception, some attachments, and cached values.

To understand how the byte payload cache works, we need to look further into the message payload accessor methods of the event context. These methods target string and byte restitution of the current message's payload:

```
byte[] message = eventContext.getMessageAsBytes();
String message = eventContext.getMessageAsString();
String message = eventContext.getMessageAsString(String encoding);
```

Behind the scenes, these methods delegate to the message itself: this is where the transformation to string or bytes happens. In both cases, the result of the transformation will be cached as bytes in the message itself. Subsequent calls to these methods will hit the cache and not trigger any further transformation. The transformer used is automatically selected by Mule based on the actual message payload and the desired rendering (string or bytes), using a mechanism similar to the one used by the auto transformer (see the tip about discoverable transformers in section 5.7.2). If no discoverable transformer has been found apt to perform the transformation, Mule will default to a standard object-to-string transformer or object-to-byte-array transformer.

We use one of these message payload accessor methods in line ① of Cloud, Inc.'s file hasher component shown in listing 13.13. This is a good use case for it because we don't expect any inbound transformer to be configured before this component, and we don't dispatch anything out of this component. By their nature, these methods are only relevant when the outcome of the transformation won't be used as the new payload of the message currently being processed. Should this be the case, then a more permanent transformation of the message payload is needed.

The main method that the event context exposes for applying the inbound transformers is the following:

```
Object result = transformMessage();
```

There are also variants of this method that perform a subsequent transformation to reconstitute the transformed message into a particular format:

```
Object result = transformMessage(Class expectedType);
byte[] result = transformMessageToBytes();
String result = transformMessageToString();
```

These variants use the same autotransformation mechanism leveraged by the message payload accessor methods that we just talked about.

Cloud uses the `transformMessage` method in their client validation service, shown in listing 13.14, in order to honor the defined inbound transformers in this component. More generally, if the entry point resolver (see section 6.3.1) used to select the method called on your component isn't configured to apply transformers first, you'll have to do it yourself programmatically by calling one of these methods. This is the case for components implementing `Callable` used in configurations that rely on the default entry point resolvers set.

The last gem in the Mule message you may want to dig is the original payload, which is available through the original adapter. This gives you a chance to retrieve the

message as it was before its last transformation. The original payload is kept only if the message adapter used by Mule before the transformation is immutable, which is the case for messages created by transports. But if you create messages using `org.mule.transport.DefaultMessageAdapter` (implicitly used by `org.mule.DefaultMuleMessage`), you won't be able to retrieve the original payload, as this adapter is mutable.

TIP *Size matters* If the byte payload and the original payload caches are a concern to you in terms of memory consumption, you can deactivate them by setting the `mule.message.cacheBytes` and `mule.message.cacheOriginal` Java system properties to `false`, respectively.

This should be envisioned only for Mule instances where memory size has actually been a concern or if huge message volumes or sizes are expected to be processed.

In this section, you discovered that there's way more in a Mule message than meets the eye. You also learned how to honor inbound transformers from a component if its entry point resolver doesn't do it for you. Let's now see how you can programmatically influence the way messages are processed in Mule.

13.3.2 Influencing message processing

Mule's standard message processing flow can be used in complex scenarios thanks to Mule's support of routing and transforming features. Often a particular need that seems to require custom code can be achieved with the addition of an extra service and a smart routing infrastructure around it (see chapter 4). Service composition though interface binding (described in section 6.3.4) also enables advanced component interactions without the need to couple your code to the Mule API.

BEST PRACTICE What you can't do with one service, do with two!

But there are times when you need to take control of the message processing flow by using the API directly. Let's consider the case of components. By default, the value returned from a component's method is used both as the response of the inbound request, if it's synchronously waiting for one, and as the payload to be dispatched to the configured outbound router. One way to prevent anything from being dispatched out of the component is to return `null`. But what if you want to return to the inbound router a non-null value that's different than the one sent to the outbound router?

In these scenarios, you'll need to get ahold of the event context and use the methods it offers for influencing the message processing flow and adapting it to your needs. These methods include

- `setStopFurtherProcessing`—To prevent the current event from being forwarded to the configured outbound router.
- `markTransactionForRollback`—To instruct Mule to roll back the current transaction, if one exists. Refer to chapter 10 for a complete discussion on transaction control.

- *send/dispatch to the configured outbound router*—To send the current message or a new one to the router that's been defined for the current service in the configuration.
- *send/dispatch to an arbitrary endpoint*—To send the current message or a new one to any endpoint referred to by its name or URI.

As is the case with the Mule client, the messaging methods offered by the event context can also return future messages and support synchronous requesting of messages. Clood, Inc., uses this last feature in their file hasher component, whose event-handling code is shown in listing 13.13.

Listing 13.13 The event context allows us to modify the message-processing flow.

```
public Object onCall(MuleEventContext eventContext)
    throws Exception {

    String fileName = eventContext.getMessageAsString();

    MuleMessage requestedFileMessage =
        eventContext.requestEvent("file://" + sourceFolder
            + "/" + fileName
            + "?connector=" + fileConnectorName, 0);

    eventContext.setStopFurtherProcessing(true);

    return requestedFileMessage != null ? DigestUtils
        .md5Hex(requestedFileMessage.getPayloadAsBytes()) : null;
}
```

1 Gets file name from incoming message payload

2 Synchronously requests content of file

3 Response needs not be sent to outbound router

Response consists of MD5 hash of file's content

Because this component isn't designed to be used with an outbound router, in ③ we use the event context method to instruct Mule not to send the return value to the outbound router (if one has been mistakenly configured). We also use the event context in ① to transform the inbound message to a string, as discussed before, and to synchronously request the content of a file from its complete URI in ②.

Coming back to Clood's client emailing services, we'll look at a component that further manipulates the message process flow. This component confirms that a message payload is a valid instance of com.clood.model.Client and dispatches it accordingly. The event-handling code of this component is shown in listing 13.14. Depending on the validity of the Client object, it conditionally dispatches the received message to the outbound router or to an error channel, and systematically returns a simple string acknowledgment message to the inbound router.

Listing 13.14 Disconnecting the response from the dispatched outbound message

```
public Object onCall(MuleEventContext eventContext)
    throws Exception {

    eventContext.setStopFurtherProcessing(true);
```

1 Prevents dispatching to outbound router

```

Object payload = eventContext.transformMessage(); 1 Applies any configured  
transformers
try {
    validatePayloadIsValidClient(payload);
}
} catch (IllegalArgumentException iae) {
    try {
        eventContext.dispatchEvent(
            eventContext.getMessage(),
            errorProcessorChannel);
    } catch (MuleException me) {
        processMuleException(me);
    }
    return "ERROR: " + iae.getMessage();
}
eventContext.dispatchEvent(payload);
return "OK"; 5 Returns acknowledgment  
message to inbound router
}
2 Dispatches invalid  
message to error  
channel
3 Returns error message  
to synchronous  
inbound router
4 Dispatches valid  
message to configured  
outbound router

```

We achieved this by first instructing (in ①) Mule to never dispatch the return value of the component, because the return values (in ③ and ⑤) are simple strings that make sense only to the client connected to the inbound router. Depending on the outcome of the validatePayloadIsValidClient method call, the message will either be dispatched in its transformed form to the outbound router in ④ or in its original form to the error channel in ②.

Even though this component has the drawback of being coupled to Mule's API, note how it doesn't hard-code anything about the destinations where messages should be dispatched. The error channel is configured by injection and the outbound router is configured using the standard configuration mechanism.

You've discovered that the event context is a rich façade to Mule's messaging infrastructure that can be leveraged to reach further than a message payload or to concisely implement complex process flows.

We'll now proceed to the last leg of our journey in the API, where we'll discover advanced means to inject custom behaviors in Mule.

13.4 Keeping abreast with Mule

During the lifetime of an instance, many things happen inside Mule. Its API allows you to keep abreast with all this activity and graft the custom behavior you need at different strategic locations. In this section we'll focus on the following places:

- *Lifecycle events*—The API allows you to leverage lifecycle events that occur from startup time, when moving parts get instantiated, configured, and put into service, to shutdown time, when they're transitioned back to oblivion.
- *Interceptors*—The API allows you to elegantly inject cross-cutting logic on top of the normal processing of messages by using component interceptors.

- *Notifications*—Any significant operation or event that happens in Mule ends up in a notification being fired, which the API allows you to listen to.

Let's start by looking at how your applications can benefit from becoming aware of Mule's lifecycle.

13.4.1 Leveraging lifecycle events

Between the time you bootstrap a Mule instance and the time it's up and ready, many things happen, as the numerous log file entries can testify. Moving parts are created and made ready for production. These moving parts are configured and transitioned through lifecycle phases, which technically amounts to calling specific methods in a predefined order. The same happens when you shut down a Mule instance: all moving parts are transitioned to their ultimate destruction through another series of lifecycle method calls.

Your own custom moving parts can benefit from these configuration and lifecycle method calls the same way Mule's do. Table 13.1 gives an overview of the methods that are called and the order in which this happens for the main types of custom objects you can create. On top of the four standard Mule lifecycle methods (`initialize`, `start`, `stop`, `dispose`), this table also shows the custom properties setters, the Mule-specific setters (context and service injection), and the Spring-specific lifecycle methods.

Table 13.1 Configuration and lifecycle methods are honored differently depending on the type of your custom object.

| Configuration and lifecycle methods | Prototype component (for each instance) | Singleton component | Spring Bean component | Any Spring Bean | Transformer | Filter |
|-------------------------------------|---|---------------------|-----------------------|-----------------|-------------|--------|
| Custom properties setters | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Mule context setter | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Spring init-method | n/a | n/a | ✓ | ✓ | n/a | n/a |
| Mule service setter | ✓ | ✓ | ✓ | n/a | | |
| initialise | ✓ | ✓ | | | ✓ | ✓ |
| start | ✓ | ✓ | ✓ | ✓ | | |
| stop | ✓ | ✓ | ✓ | ✓ | | |
| dispose | ✓ | ✓ | | | | |
| Spring destroy-method | n/a | n/a | ✓ | ✓ | n/a | n/a |

How does Mule decide whether it should call a particular lifecycle method or setter on a custom object? It simply looks for particular interfaces that the custom object must have implemented. Here's the list of the four standard lifecycle interfaces:

```
org.mule.api.lifecycle.Initialisable
org.mule.api.lifecycle.Startable
org.mule.api.lifecycle.Stoppable
org.mule.api.lifecycle.Disposable
```

TIP *One interface to rule them all* If your custom object needs to implement the four standard lifecycle interfaces, you can save yourself a lot of typing by simply implementing `org.mule.api.lifecycle.Lifecycle`, which extends the four standard interfaces.

And here are the interfaces for the Mule-specific setters:

```
org.mule.api.context.MuleContextAware
org.mule.api.service.ServiceAware
```

We mentioned implementing `MuleContextAware` in section 13.2 as a convenient way to get ahold of the Mule context. Service awareness allows a component to access the service it's hosted by and all its internals: state, statistics, inbound, and outbound routing infrastructure.

WARNING *Started but not ready* When your components are started, don't assume that the whole Mule instance is up and running. Some components start way before the complete boot sequence is done. If your component needs to actively use Mule's infrastructure, it should wait until it's ready. The best way to achieve this is to listen to notifications, as we'll see in section 13.4.3.

The code shown in listing 13.15 demonstrates how Cloud's client validation service (introduced in the previous section) is configured and initialized. Note that we've implemented the `initialise` method to be idempotent, should Mule call it several times, but we'll perform the actual initialization sequence only once.³ The `initialize` method will be called because the `ClientValidatorService` component implements the `org.mule.api.lifecycle.Initialisable` interface we just mentioned.

Listing 13.15 Using property injection and lifecycle methods to initialize a service

```
public void setErrorProcessorChannel(
    EndpointBuilder errorProcessorChannelBuilder) {
    this.errorProcessorChannelBuilder = errorProcessorChannelBuilder;
}

public void initialise() throws InitialisationException {
    if (initialised) {
        return;
    }
}
```

Receives error
channel endpoint
builder via
setter injection

Ensures that component
hasn't been initialized

³ You may notice that the `initialise` method isn't designed with thread-safety in mind. This is acceptable because, though it may be called several times, these calls will happen sequentially and not simultaneously.

```

    }

    try {
        errorProcessorChannel = errorProcessorChannelBuilder
            .buildOutboundEndpoint();
        initialized = true;
    } catch (EndpointException ee) {
        throw new InitialisationException(ee, this);
    }
}

```

**Builds actual
error channel
outbound endpoint**

The configuration of the component itself is trivial:

```
<singleton-object class="com.cloud.component.ClientValidatorService">
    <property key="errorProcessorChannel" value-ref="ErrorProcessorChannel" />
</singleton-object>
```

In this configuration, we inject a reference to a global endpoint named `ErrorProcessorChannel` in the designated setter method. As we've seen in listing 3.15, what we actually inject is a reference to an endpoint builder, from which we derive an outbound endpoint. This makes sense when you take into account that a global endpoint can be referred to both from inbound and outbound endpoints.

NOTE The same way entry point resolvers (see section 6.3.1) allow you to target component methods without implementing any Mule interface, it's also possible to create custom lifecycle adapters⁴ that'll take care of translating Mule lifecycle methods into any methods on your objects. Is it worth creating such an adapter for the sake of coding Mule-free components when it's also possible to rely on the noninvasive Spring lifecycle methods? Our own experience is that the latter is more than enough.

You can now make custom logic execute when lifecycle events occur in Mule. Let's now learn how to attach cross-cutting logic to components.

13.4.2 Intercepting messages

Components encapsulate standalone units of work that get invoked when events reach them. In any kind of component-oriented framework, the question of sharing transversal behavior is inevitable. Mule's components don't escape this question. Though it's possible to follow object-oriented approaches (composition and inheritance) with Mule components, the most satisfying solution to the problem of implementing cross-cutting concerns comes from aspect-oriented software development (AOSD).

To this end, Mule supports the notion of component interceptors, which represent only a small subset of aspect-oriented programming (AOP) but still offer the capacity to attach common behaviors to components. A component interceptor, which is an implementation of `org.mule.api.interceptor.Interceptor`, receives an invocation to the component it's attached to and has all the latitude to decide what to do with it. For example, it can decide not to forward it to the component, or to be

⁴ Namely, custom implementations of `org.mule.api.component.LifecycleAdapterFactory` and `org.mule.api.component.LifecycleAdapter`.

more accurate, to the next interceptor in the stack, as interceptors are always members of the stack (the last member of the stack is the component itself). The invocation is itself an instance of `org.mule.api.interceptor.Invocation` and is represented in figure 13.7.

Of course, besides stopping further processing, an interceptor can also perform actions around the invocation of the next member of the stack, while keeping its own state during the life of its own invocation. This is, for example, what the `timer-interceptor` does: it computes the time spent in the rest of the stack by storing the time before passing to the next interceptor and comparing it with the time when the execution flow comes back to it.

BEST PRACTICE Harness Mule's interceptors for implementing messaging-level cross-cutting concerns.

Mule also comes complete with an abstract interceptor implementation named `org.mule.api.interceptor.EnvelopeInterceptor`, whose semantics are limited to adding behavior before and after the invocation of the next member of the stack, but without the possibility to control this invocation itself. It's more suited for interceptors that don't need to maintain some state around the invocation of the next interceptor in the stack. The `logging-interceptor` is built on this principle: it simply logs a message before forwarding it to the next interceptor and another message after the execution flow comes back to it.

NOTE *Springing into AOP* If you're a savvy Spring user, you might be interested in leveraging your Spring AOP skills with Mule. Using Spring AOP can allow you to go further than where the Mule interceptors can take you, as it's a full-fledged aspect-oriented programming framework. This said, be aware that it requires a great deal of knowledge on both the Mule and Spring sides. For example, Spring auto-created AOP proxies can confuse the entry point resolver mechanism of Mule, as the proxy will expose different methods than what your original object was exposing, leading to confusing errors in Mule. This is especially true for plain POJO components that don't implement `org.mule.api.lifecycle.Callable`. Therefore, it's possible that advising components can lead you to rework part of your configuration.

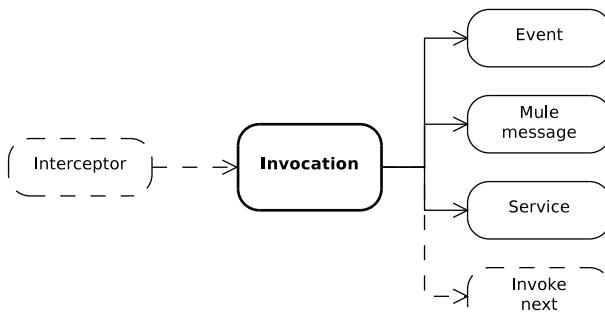


Figure 13.7 The `Invocation` object allows the interceptor to access the current event and to propagate it.

Listing 13.16 shows the interceptor stack that Clood uses in front of components that are costly to call and whose results can be cached. The stack first defines the standard timer-interceptor that's used to record how efficient the custom-interceptor configured after it is. The actual cache is provided by Ehcache (see <http://ehcache.sourceforge.net>), itself configured and injected into our interceptor by Spring.

Listing 13.16 Clood's caching interceptor stack defines two interceptors.

```
<interceptor-stack name="PayloadCacheInterceptors">
    <timer-interceptor />
    <custom-interceptor class="com.clood.interceptor.PayloadCacheInterceptor">
        <spring:property name="cache" ref="ehCache" />
    </custom-interceptor>
</interceptor-stack>
```

The PayloadCacheInterceptor implementation itself is fairly trivial,⁵ as shown in listing 13.17. Of course, the implementation is concise because all the heavy lifting is done by Ehcache.

Listing 13.17 The cache interceptor leverages Ehcache to store and replay payloads.

```
public class PayloadCacheInterceptor implements Interceptor {
    private Ehcache cache;

    public void setCache(Ehcache cache) {
        this.cache = cache;
    }

    public MuleMessage intercept(Invocation invocation)
        throws MuleException {
        MuleMessage currentMessage = invocation.getMessage();
        Object key = currentMessage.getPayload();
        Element cachedElement = cache.get(key);
        if (cachedElement != null) {
            return new DefaultMuleMessage(cachedElement.getObjectValue(),
                currentMessage);
        }
        MuleMessage result = invocation.invoke();
        cache.put(new Element(key, result.getPayload()));
        return result;
    }
}
```

Annotations for Listing 13.17:

- Defines class as Interceptor implementation**: Points to the line `private Ehcache cache;`
- Uses incoming message payload as cache key**: Points to the line `Object key = currentMessage.getPayload();`
- Cache hit: return response immediately**: Points to the line `return new DefaultMuleMessage(cachedElement.getObjectValue(), currentMessage);`
- Cache miss: proceed with invocation**: Points to the line `MuleMessage result = invocation.invoke();`
- Store invocation result in cache**: Points to the line `cache.put(new Element(key, result.getPayload()));`

Note in ① that in case of a cache hit, we build the response message by using the cached payload and the current message. This is because we want to preserve all the preexisting properties and other extra context from the current message (see section 13.3.1) alongside the new payload.

⁵ The multithread-minded readers will notice that no effort is made to ensure that only a single thread actually ever invokes the next interceptor: this is acceptable in Clood Inc.'s usage scenarios but may need to be enforced in other ones.

Clood's file hasher component is a perfect candidate for this interceptor stack. It receives a message whose payload is a file name and returns a message whose payload is the MD5 hash of the file's content. Because files don't change in content, we can then use the file name as the key and the MD5 hash as the value in a standard cache. Adding this interceptor stack to our existing file hashing service (shown in listing 6.12) is a no-brainer, as shown in this configuration excerpt:

```
...
<pooled-component>
  <interceptor-stack ref="PayloadCacheInterceptors" />
  <singleton-object class="com.clood.component.Md5FileHasher">
...

```

The following console transcript clearly shows the efficiency of the payload cache interceptor, as reported by the timer-interceptor:

```
27-Jun-2009 15:34:14 org.mule.interceptor.TimerInterceptor intercept
INFO: Md5FileHasher took 21ms to process event [25e11b6f-a488-11dd]
27-Jun-2009 15:34:24 org.mule.interceptor.TimerInterceptor intercept
INFO: Md5FileHasher took 1ms to process event [25e5884c-a488-11dd]
27-Jun-2009 15:34:34 org.mule.interceptor.TimerInterceptor intercept
INFO: Md5FileHasher took 0ms to process event [25e5fd85-a488-11dd]
```

TIP *Explicit bridge interception* You may wonder where to declare an interceptor stack on an implicit bridge (see section 6.1.1) as there's no XML component element in the configuration. The solution is to use an explicit bridge component and add the stack child element in it.

Thanks to the interceptor framework, you can now share transversal logic across components in an elegant and efficient manner. This allows you to enrich the message process flow with your own custom code. But what if you want to execute some logic only when particular events occur in Mule? This is when you'll need to tap the notification framework that we'll detail now.

13.4.3 Receiving notifications

As we said in chapter 11, notifications are generated whenever a specific event occurs inside Mule, such as when an instance starts or stops, when an exception has been caught, or when a message is received or dispatched. We extensively discussed how notifications can be leveraged to keep track of the activity of a Mule instance in section 11.2.2. There, we mainly talked about using preexisting listeners for Mule's internal notifications, such as the log4j-notifications agent.

There's much more you can do by directly tapping into the notification framework. For this you need to write custom classes that can receive notification events. We evoked this possibility at the end of section 7.2.4, where we mentioned creating a listener that can be notified of routing problems that occur when messages don't get correlated on time in an aggregator router. Routing notifications are just one type of notification that can be listened to in Mule. Figure 13.8 shows an overview of the

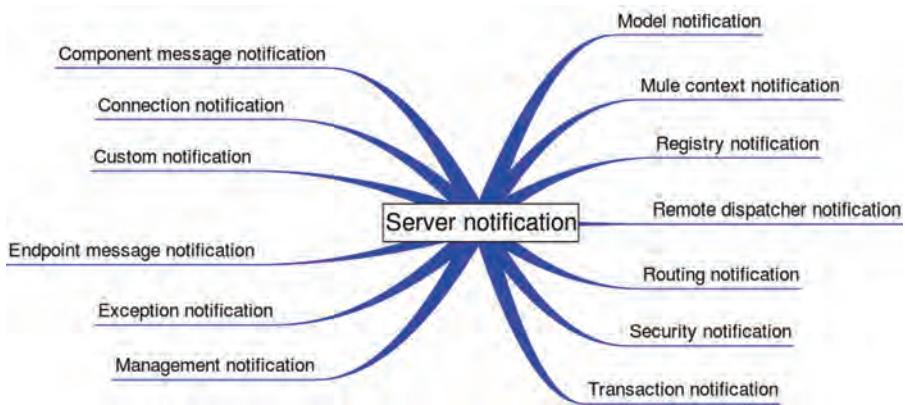


Figure 13.8 The family of Mule server notifications is rich and diverse.

notification types, which are all descendants of `ServerNotification`. Each notification type also defines actions that further specify the context in which they happen.

Listing 13.18 shows part of the implementation of the correlation time-out notification listener.

Listing 13.18 A custom class can receive specific notifications like routing issues.

```

public class CorrelationTimeOutListener
    implements RoutingNotificationListener {
    ...
    public void onNotification(ServerNotification notification) {
        if (notification.getAction() != RoutingNotification.CORRELATION_TIMEOUT) {
            return; ① Disregard any notification event
                    that's not CORRELATION_TIMEOUT
        }
        MuleMessageCollection messageCollection =
            (MuleMessageCollection) notification.getSource();
        ...
    }
}
  
```

Defines class as
RoutingNotificationListener
implementation

Extracts expected
content from
notification source

Note in ① how we programmatically narrow the scope of the notification received to the correlation time-out action only. We have to do this because routing notifications encompass more types of events than just correlation time-outs. Because of the way the notification listener API is designed, note also that this listener, like all listeners, receives instances of `org.mule.api.context.notification.ServerNotification`. A cast to the particular type of notification object your listener can receive is required if you need to access one of its specific methods. For example, we could cast the received notification to `org.mule.context.notification.RoutingNotification` in this example.

For such a listener to start receiving notifications, you need to register it with Mule and activate the notification family it's interested in, as shown in the configuration fragment of listing 13.19.

Listing 13.19 A custom object can be registered with Mule to receive notifications.

```

<spring:bean
    name="correlationTimeOutListener"
    class="com.muleinaction.service
        ↪ CorrelationTimeOutListener">

    <spring:property name="dlqAddress"
        value="${dlq.address}" />
</spring:bean>

<notifications>
    <notification event="ROUTING" />
    <notification-listener ref="correlationTimeOutListener" />
</notifications>

```

1 Instantiates and configures notification listener
2 Activates routing notification family
3 Registers listener with Mule's notifications infrastructure

The activation of the notification family is done in ② and the registration of the listener in ③. The instantiation and initialization of this listener shown in ① is by no means normative: an object doesn't need to be a Spring-handled bean to be able to listen to notifications, as the upcoming example will demonstrate.

The following demonstrates an advanced usage of notifications, by combining several of the concepts you've learned in this chapter (context injection, interceptors, and notification listeners). Some of Cloood's services are unable to process any event until the Mule instance they run into is fully initialized (because they dispatch messages to other services that may not themselves be ready). To prevent an incoming event from reaching the component of these sensitive services, Cloood created an interceptor that rejects any traffic until the whole Mule instance is fully initialized. Listing 13.20 shows the full code of this interceptor.

Listing 13.20 An interceptor that rejects events until Mule is fully initialized

```

public class BrokerNotReadyInterceptor implements MuleContextAware,
    MuleContextNotificationListener, Interceptor {

    private volatile boolean brokerReady = false;

    public void setMuleContext(MuleContext context) {
        try {
            context.registerListener(this);
        } catch (final NotificationException ne) {
            throw new RuntimeException(ne);
        }
    }

    public void onNotification(ServerNotification notification) {
        int action = notification.getAction();

        if (action ==
            MuleContextNotification.CONTEXT_STARTED) {
            brokerReady = true;
        }
        else if (action ==
            MuleContextNotification.CONTEXT_STOPPED) {
            brokerReady = false;
        }
    }
}

```

1 Registers listener with notifications infrastructure
2 Sets broker ready when context is started
3 Sets not ready when context is stopped

```

        }
    }

    public MuleMessage intercept(Invocation invocation)
        throws MuleException {
        if (!brokerReady) {
            throw new IllegalStateException("Invocation of service "
                + invocation.getService().getName()
                + " impossible at this time!");
        }
        return invocation.invoke();
    }
}

```

Rejects any event until the broker is ready

Because an interceptor configuration doesn't expose an ID that we could use to register it with Mule's notifications infrastructure, we have to perform this registration programmatically on ①.

This interceptor is configured using the custom-interceptor element, as we saw in the previous section. Notice how there's no ID attribute that could allow us to register it as a notification listener by configuration:

```
<custom-interceptor
    class="com.cloud.interceptor.BrokerNotReadyInterceptor" />
```

Therefore, the notification configuration related to this interceptor consists only in the activation of the context family of events:

```
<notifications>
    <notification event="CONTEXT" />
</notifications>
```

NOTE Mule's notification framework offers advanced features such as

- Associating a custom subinterface of `org.mule.api.context.notification.ServerNotificationListener` with an existing event family
- Disabling standard listeners to receive events from the family they naturally listen to (for example, preventing all implementations of `org.mule.api.context.notification.RoutingNotificationListener` from receiving routing events)
- Creating custom notifications (as represented in figure 13.8), listening to them, and broadcasting them programmatically

In this section, we reviewed how listening to notifications from your own custom code can allow you to implement advanced Mule-aware logic. Along with lifecycle events and component interceptors, the notification framework is another tool offered by Mule to allow you to keep abreast with its internal activity.

13.5 Summary

The Mule API is a rich and comprehensive gateway to all moving parts of the ESB. Though using the API creates coupling between your code and Mule itself, we've

explored a wealth of scenarios where the drawbacks of such a coupling are overcome by its benefits.

We saw in this chapter how it's possible to leverage this API to perform actions of all kinds, including communicating with a running Mule instance, exploring the inner parts its context gives access to, performing message transformation, and altering the normal course of message processing. We also learned how to leverage Mule's lifecycle events, interceptors, and notification framework to trigger the execution of business logic at specific moments.

If the prospect of having to compile your code first before deploying it to Mule is a daunting one for you, it's time to rejoice, as the next chapter will be dedicated to using dynamic languages with Mule.

14

Scripting with Mule

In this chapter

- Using Groovy and Rhino with Mule
- Implementing component and transformer logic with scripts
- Leveraging Spring's scripting functionality with Mule

The undeniable success of the popular web development framework Ruby on Rails epitomizes the benefits of working with scripting languages. The conciseness of Ruby coupled with the lack of a compilation phase makes working on this platform quick and productive. Changes to Ruby on Rails applications can be made on the fly while the application is running. This instant feedback is invaluable for debugging, testing, and prototyping. Contrast this with the compile, deploy, and debug cycle that's present in a compiled language such as Java or C. A typical web application in Java, for instance, needs to be compiled, packaged, and deployed to an application server. To apply modifications to the web application, the cycle must be repeated. Up until now, this has also been the case for your Mule projects. Any changes you've made to a component or transformer, for instance, would require you to restart Mule to take effect. This usually isn't too big of a deal, but wouldn't it be nice if you could test the component or transformer while Mule was running?

In chapters 5 and 6 you saw how to use the Mule API to implement custom transformation and component logic. The examples in those chapters used Java, but perhaps you feel like your transformation logic might be more concisely expressed using Groovy. Maybe you'd rather implement your component logic with Python instead of Java. You could also be in a position where you need to do a "one-off" transformation and don't want to go through the hassle of writing a Java class, packaging it, and deploying it. We'll see in this chapter how Mule's scripting support can help you accomplish all of these things. We'll start off by seeing how to use Rhino, the default scripting language of the JVM, to implement component logic as well as bind to service interfaces. Next we'll turn our attention to Groovy, another Java scripting language, and see how it enables us to implement transformation and header evaluation logic in Mule. Finally, we'll see how Spring's scripting support augments Mule's dynamic language features, allowing you to implement arbitrary Mule functionality via scripts as well as auto-refresh scripts without restarting Mule.

14.1 Using Rhino

Since the release of Java 6, the JDK has shipped with scripting support and a scripting language to take advantage of it: a JavaScript engine called *Rhino*. Quickly following suit were a host of other scripting platforms either ported or designed to work from scratch on the JVM. These included *JRuby*, a Java implementation of Ruby, as well as *Groovy*, a deliberately Java-esque scripting language. Mule takes advantage of the scripting functionality in Java 6 and can use any language that implements JSR-223—the specification that defines dynamic language support in the JVM.¹

In this section we'll begin to explore Mule's dynamic language support by implementing Mule component logic using Rhino. We'll then revisit a feature you saw in chapter 6—service interface binding—and see how to use that functionality in a scripting context.

14.1.1 Implementing component logic with Rhino

You might find yourself in a situation where you want to add simple component logic to a service but don't want to go through the hassle of developing a Java class, packaging it, and deploying it to Mule. Embedding a script in Mule's configuration can be an attractive option in cases like this. Let's assume you want to add a property to messages as they pass through a service. Perhaps you want to evaluate a message's payload and, based on that, set a property header stating the message's priority. This property could subsequently be used by selective consumers to determine whether to accept the message. Listing 14.1 illustrates how this might be accomplished by using a JavaScript component.

¹ Groovy support for Mule is bundled by default. The Mule Scripting Pack, available from <http://www.mulesource.org/display/MULE/Download+Mule+CE>, provides support for Rhino, JRuby and Jython.

Listing 14.1 Using a JavaScript component to enrich a message

```

<service name="rhinoMessageEnrichmentService">
    <inbound>
        <vm:inbound-endpoint path="in"/>
    </inbound>
    <scripting:component>
        <scripting:script engine="js">
            if (payload.search("STATUS: CRITICAL") != -1) {
                message.setProperty("PRIORITY", 'HIGH');
            }
            result = message
        </scripting:script>
    </scripting:component>
    <outbound>
        <pass-through-router>
            <vm:outbound-endpoint path="out"/>
        </pass-through-router>
    </outbound>
</service>

```

By using the scripting namespace, we can begin to inline scripts into our Mule configuration. We first configure a scripting component on ①, telling Mule that the component logic will use a JSR-223-compliant scripting engine. Mule will subsequently spin up an appropriate environment for the script to execute with common Mule references defined in the script's context—as we'll see in a moment. Since the logic for this component is fairly straightforward, we choose to inline the script directly in the XML configuration. ② indicates the start of the inlined script along with the scripting engine to use. Every JSR-223-compliant scripting engine will declare a name, which is set on ③. In this case, we have it set to js, indicating our script is written in JavaScript and will be executed using Rhino. The script itself begins on ④, where we're searching the payload variable for presence of the specified expression. You might be wondering where we've defined the payload variable, and that's because we haven't—Mule has. Mule will prepopulate the script's context with variables you're most likely to need. This saves you the hassle of obtaining a reference to the `MuleContext`. These variables are listed in table 14.1.²

We use the `message` variable on ④ to set the `PRIORITY` message property (header) to `HIGH`. The component then exists and the modified message is sent out through the outbound endpoint.

Embedding a script in your Mule configuration can be convenient when the script is small enough, but managing larger scripts in the context of a Mule XML configuration can quickly become unwieldy. As such, it's possible to reference a script stored in an external file. In listing 6.12 we implemented a component that took the MD5 hash of a specified file. Let's take a variation of this functionality and see how it can be implemented using an external script. We're going to implement a Rhino script that'll

² Please make sure you've installed The Mule Scripting Pack if you intend to use a language other than Groovy.

Table 14.1 Variables made available to a scripting context

| Variable name | Description |
|-----------------|--|
| message | The current message being processed |
| payload | The payload of the current message |
| originalPayload | The original payload of the current message, before any transformation |
| muleContext | A reference to the MuleContext |
| eventContext | A reference to the EventContext |
| id | The ID of the current event |
| service | A reference to the service object processing the current message |
| result | A variable to explicitly set the result of a script |

take the MD5 hash of the payload of a message and attach it as a property of that message. This property can then be used to ensure that the payload of a message hasn't changed when processed by future services. Listing 14.2 illustrates a Rhino script external to the Mule configuration to accomplish this.

Listing 14.2 An external Rhino script to add an MD5SUM property to a message

```
importPackage(org.apache.commons.codec.digest);
var PROPERTY_NAME = "MD5SUM";
if (!message.getProperty(PROPERTY_NAME)) {
    log.debug("Setting " + PROPERTY_NAME +
        " property for message: " + id);
    message.setProperty(PROPERTY_NAME,
        getMD5Sum(payload));
    result = message;
}
function getMD5Sum(input) {
    return DigestUtils.md5Hex(input);
}
```

This script is pretty similar to that of listing 14.1. We have a bit more ceremony with listing 14.3, though, so it seems more natural to store it external to the Mule configuration. This will also give us the flexibility to change the script as Mule is running, as we'll see in a bit. The Mule configuration to load the externally defined script is illustrated in listing 14.3.

Listing 14.3 Using an externally defined Rhino script

```
<service name="md5MessageEnrichmentService">
    <inbound>
        <vm:inbound-endpoint path="in"/>
    </inbound>
    <scripting:component>
        <scripting:script file="md5.component.js"/>
    </scripting:component>
    <outbound>
        <pass-through-router>
            <vm:outbound-endpoint path="out" />
        </pass-through-router>
    </outbound>
</service>
```

1 Declare location of externally defined script

By setting the `file` parameter on ①, we can set the location of the script relative to Mule's classpath. You may have noticed that we've dropped the `engine` parameter. When this value is missing, Mule will try to infer the appropriate engine using the script's extension. In this case, since the script ends with a `.js` extension, Mule will run the script using the Rhino engine. We'll see this service again later in this chapter when we discuss reloading scripts and using Spring's scripting support with Mule.

14.1.2 Using service interface binding in scripts

In section 6.3.4, you saw how component bindings allowed Mule to transparently implement interfaces used in components. The interface implementation constructed dynamically by Mule could then invoke other services to provide the results of method calls. This allowed component code to take advantage of Mule services without being coupled to Mule code, say by using the Mule client. By using the `java-interface-binding` element, Mule provides the same functionality for scripts.

In listing 6.13, we saw how the `Md5FileHasherClient` was injected with an implementation of `Md5FileHasherService`. By using an interface binding, Mule would proxy calls to the `process` method of this class to a VM outbound endpoint. The result of this synchronous endpoint would then be the return method of the process call. Let's see how we can use the same functionality in the script. Listing 14.4 shows an inline Rhino script that duplicates the functionality of listing 6.13.

Listing 14.4 Binding a service interface in a script using `java-interface-binding`

```
<model name="vmSimpleModel">
    <service name="greeting">
        <inbound>
            <vm:inbound-endpoint path="MSC.In" />
        </inbound>
        <scripting:component>
            <scripting:script engine="js">
                result = Md5FileHasherService.hash(payload)
            </scripting:script>
            <scripting:java-interface-binding
                interface="com.cloud.component.Md5FileHasherService"
```

1 Execute method on bound interface

```

method="hash">
<vm:outbound-endpoint
    path="Md5FileHasher.In"
    synchronous="true"/>
</scripting:java-interface-binding>
</scripting:component>
</service>
</model>
```

The `java-interface-binding` element in the scripting namespace allows us to specify the interface and method to bind to. On ② we're binding to the `Md5FileHasherService` interface's `hash` method. This exposes an `Md5FileHasherService` to the scripting context that will be proxied to the supplied outbound endpoint. When the `hash` method is called on ①, a synchronous request will be made on the VM outbound endpoint. The result will then be returned to the component.

JSR-223 wouldn't be of much use if it only supported a single scripting language. Let's look at another JSR-223-compliant language, Groovy, and see how we can use it to implement transformation and header evaluation logic in Mule.

14.2 Using Groovy

Groovy is a JSR-223-compliant dynamic scripting language for the JVM. It's deliberately Java-esque, and its syntax closely resembles Java's—with some interesting twists. Groovy ships with Mule, so there's no need to install any additional JARs to take advantage of it. As we'll see in this section and the next, Mule affords Groovy scripts some additional functionality as well. More information and full documentation about the Groovy language is available at <http://groovy.codehaus.org/>.

14.2.1 Implementing transformers with Groovy

In chapter 5 we saw how custom transformers can be used to perform data transformations that Mule doesn't supply in its distribution. In that chapter, we implemented a custom transformer in Java that transformed message payloads using Velocity. Using the scripting namespace, we'll now see how we can implement custom transformation logic using Groovy.

Let's start with a simple example of using a Groovy script to perform an inline payload transformation. The transformer in listing 14.5 will uppercase the string payload sent through it.

Listing 14.5 Uppercasing a string payload using a Groovy transformer

```

<service name="groovyTransformerService">
    <inbound>
        <vm:inbound-endpoint path="in" >
            <scripting:transformer>
                <scripting:script engine="groovy">
                    return payload.toUpperCase()
                </scripting:script>
            </scripting:transformer> </vm:inbound-endpoint>
```

```

</inbound>
<outbound>
    <pass-through-router>
        <vm:outbound-endpoint path="out" />
    </pass-through-router>
</outbound>
</service>

```

The syntax to declare a scripting transformer is similar to configuring a scripting component. We have the scripting transformer defined on ① along with an inline script that's defined on ②. As you can see, we've changed the engine to groovy from js, indicating that we want the Groovy scripting engine to execute the supplied script. The script itself is defined on ③; it simply uppercases the payload and returns it.

Defining global scripting transformers is also possible. This is done by declaring the scripting transformer before any model definition. Listing 14.6 shows how to declare the previous transformer globally.

Listing 14.6 Globally declaring a Groovy transformer

```

<scripting:transformer
    name="toUpperCase">
    <scripting:script engine="groovy">
        return payload.toUpperCase()
    </scripting:script>
</scripting:transformer>

<model name="groovyTransformerModel">
    <service name="groovyTransformerService">
        <inbound>
            <vm:inbound-endpoint path="in"
                transformer-refs="toUpperCase" />
        </inbound>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint system="out" />
            </pass-through-router>
        </outbound>
    </service>
</model>

```

We declare the transformer almost identically as before, but here placing it outside the model definition and supplying a name attribute. In this case, we're calling the transformer toUpperCase on ①. We then reference the transformer on ② by supplying the transformer-refs parameter to the VM inbound endpoint along with the previously declared name.

Let's now consider a more complex transformation scenario and see how it's simplified by Mule's scripting and Groovy support. In chapter 4, we saw a few examples of Cloud processing order data, as either XML documents or instances of an Order class. Cloud has recently decided to standardize on a canonical XML model to represent order data. The web application used by accounting to submit orders, though, is still using a legacy CSV representation. While the web development team works on

refactoring the web application to submit the order as XML data to a JMS queue, you've been tasked to find an interim solution. You decide to use a file inbound endpoint and implement a custom transformer to build an XML order representation from the CSV file. Once the document has been created, it'll be submitted to a JMS queue for further processing. Listing 14.7 shows an example order represented as XML.

Listing 14.7 Representing an order as XML

```
<orders>
  <order>
    <subscriberId>409</subscriberId>
    <productId>1234</productId>
    <status>PENDING</status>
  </order>
  <order>
    <subscriberId>410</subscriberId>
    <productId>1234</productId>
    <status>PENDING</status>
  </order>
  <order>
    <subscriberId>411</subscriberId>
    <productId>1235</productId>
    <status>PENDING</status>
  </order>
</orders>
```

Listing 14.6 demonstrates the legacy CSV format we need to convert from, using the same data as listing 14.8.

Listing 14.8 Representing an order as CSV

```
409,1234,PENDING
410,1234,PENDING
411,1235,PENDING
```

We'll start off by writing the script to transform the CSV to XML. We'll make use of Groovy's *builder* functionality to create the XML. This will most likely be too verbose to include directly in the Mule config, so we'll define it in an external file as detailed by listing 14.9.

Listing 14.9 Transforming a CSV payload to XML

```
import groovy.xml.MarkupBuilder
writer = new StringWriter()
builder = new MarkupBuilder(writer)

builder.orders() {
    payload.split('\n').each {line ->
        def fields = line.split(',')
        order() {
            subscriberId(fields[0])
            productId(fields[1])
        }
    }
}
```

The diagram illustrates the transformation logic:

- Split payload into individual lines**: An annotation pointing to the line `payload.split('\n').each {line ->`.
- Split each line into comma-separated values**: An annotation pointing to the line `def fields = line.split(',')`.
- Use Groovy builder to construct the XML response**: An annotation pointing to the line `builder = new MarkupBuilder(writer)`.

```

        status(fields[2])
    }
}
return writer.toString()      ← Return result

```

This short script simply iterates over each line of the payload and constructs the corresponding XML. It uses Groovy's builder syntax to concisely create and return the XML response. Listing 14.10 illustrates how to configure Mule to load CSV data from a file, transform it to XML, and publish it to a JMS queue.

Listing 14.10 Using a Groovy transform to transform CSV to XML

```

<model name="groovyTransformerModel">
  <service name="groovyTransformerService">
    <inbound>
      <file:inbound-endpoint path=".data">
        <byte-array-to-object-transformer/>
        <scripting:transformer>
          <scripting:script
            file="orderTransformer.groovy"/>
          </scripting:transformer>
        </file:inbound-endpoint>
      </inbound>
      <outbound>
        <pass-through-router>
          <jms:outbound-endpoint queue="orders"/>
        </pass-through-router>
      </outbound>
    </service>
  </model>

```

The diagram illustrates the flow of the Mule configuration. It starts with a file inbound endpoint (1) which triggers a byte-array-to-object-transformer (2). This leads to a scripting transformer (3), which executes a script defined by the file parameter (4).

The file inbound endpoint is configured to wait for files to appear on ①. The files are transformed to an object, in this case a string, by the byte-array-to-object transformer on ②. Our scripting transformer is defined on ③. The script element is defined on ④ and will execute `orderTransformer.groovy`, whose contents are listing 14.9, when invoked. Just as in the earlier Rhino example, we leave off explicitly specifying the engine and instead let Mule infer it from the file extension. The resultant XML is finally sent as the payload of a JMS message on the `orders` queue.

14.2.2 Using the Groovy evaluator

In chapter 4, you saw how various filters can be used in conjunction with a selective-consumer router to control what messages reach a component. One filter we looked at was the expression filter. This allowed us to use various expression evaluators to parse messages entering the filter. The full suite of expression evaluators is documented in appendix A of this book. One such evaluator we'll discuss now is the Groovy evaluator, which allows us to use Groovy scripts as expressions.

Let's see how to use Groovy as an expression filter. Listing 14.11 illustrates how to use a Groovy expression in a selective-consumer router to only accept messages whose payloads are ASCII printable.

Listing 14.11 Using a Groovy expression

```

<service name="asciiPrintableService">
  <inbound>
    <vm:inbound-endpoint path="in"/>
    <selective-consumer-router>
      <expression-filter evaluator="groovy"
        expression=
          "org.apache.commons.lang.StringUtils.isAsciiPrintable(payload)"/>
    </selective-consumer-router>
  </inbound>
  <test:component enableNotifications="true"/>
</service>
```

Use supplied Groovy fragment to evaluate message payload

To use the Groovy evaluator, we simply need to set the evaluator to groovy and supply Groovy code as the expression. In this case we’re using Groovy to invoke the `isAsciiPrintable` static method of the Apache Commons’ `StringUtils` class to ensure only messages whose payloads are ASCII printable will be consumed. As you can see, we’re using the `payload` variable to pass to the `isAsciiPrintable` method—the same variables that are available in the scripted component and transformer context are available for the Groovy evaluator as well.

The Groovy expression you’ll use for this will generally be fairly simple—we’ll see in the next section how we can implement more powerful filtering by using Groovy in conjunction with Spring. By now you should be comfortable using Mule’s scripting functionality. You saw how you can use Rhino and Groovy (or any other JSR-223-compliant scripting engine) to author Mule components and transformers.

Now that you’re comfortable using Mule’s scripting support using JSR-223 and its scripting namespace, let’s look at some additional features we can leverage. In the next section we’ll see how to use Spring’s `lang` namespace to offer additional flexibility when using scripting together with Mule.

14.3 Using Spring

In addition to the scripting functionality offered by the scripting namespace, Mule can also take advantage of Spring’s dynamic language support. This affords you the ability to use scripts in places where there isn’t explicit schema support, as with components and transformers. Spring-managed scripts also have the benefit of being refreshable, as we’ll soon see. Let’s see how Spring’s scripting support augments Mule’s.

14.3.1 Implementing custom Mule functionality using Spring

So far we’ve seen how Mule provides explicit configuration support for scripted components and transformers through use of its scripting namespace. But what happens if you want to implement a custom router or filter as a script? In these cases, there’s no direct schema support for scripted implementations. When this is the case, you can take advantage of Spring’s scripting support offered by its `lang` namespace. The `lang`

namespace allows you to define arbitrary Spring beans using JSR-223 scripts. You can then use these beans anywhere in Mule that takes a bean reference.

When discussing the Groovy evaluator in section 14.1, we demonstrated a one-line Groovy script that determined whether the payload of a message was ASCII printable. For simple filtering situations like this, the Groovy evaluator is more than sufficient. What happens, though, when you need more elaborate filtering? Perhaps you need to perform some sort of database or rule-based validation on a message. This is likely unfeasible with Mule's supplied filters and evaluators. In cases like this, a custom filter is usually in order.

In listing 14.11 from section 14.1, we showed how to use a Rhino script to add a header to a message containing an MD5 checksum of the message's payload. Let's implement a custom filter using Groovy that will verify whether this checksum is correct. When used in conjunction with a selective-consumer router, only messages whose payloads contain the proper checksum will get passed to a component. Listing 14.12 illustrates the Groovy script to accomplish this.

Listing 14.12 Using a Groovy filter

```
import org.mule.api.routing.filter.Filter
import org.mule.api.MuleMessage
import org.apache.commons.codec.digest.*

class MD5Filter implements Filter {           ← Declare filter
    static String PROPERTY_NAME = "MD5SUM";

    public boolean accept(MuleMessage muleMessage) {           ← Compare MD5SUM
        if (muleMessage.getProperty(PROPERTY_NAME))             ← message header
            return muleMessage.getProperty(PROPERTY_NAME) ==
                DigestUtils.md5Hex(muleMessage.getPayloadAsString())
        else return false
    }
}
```

Our Groovy script declares a class that implements the `org.mule.api.routing.filter.Filter` interface. The `Filter` interface declares one method to implement, `accept`, which returns a boolean value depending on whether the message passes the filter. In this case, we're comparing the `MD5SUM` header property to the computed MD5 checksum of the current message payload and return the result (we return a `false` if the header is missing).

Now that we've written our filter, we need to configure Mule to read it. At the time of writing this book, there's no schema extension to declare scripted filters. As such, we must configure the `MD5Filter` as a Spring bean and inject it into our selective-consumer router. We'll use the Spring namespaces to accomplish the former. This will let us load the script as a Spring bean. We'll then inject the bean into the selective-consumer router. Listing 14.13 shows how to do this.

Listing 14.13 Configure a Groovy filter using the Spring lang namespace**Declare spring namespace****1**

```
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:scripting="http://www.mulesource.org/schema/mule/scripting/2.2"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation=" http://www.mulesource.org/schema/mule/core/2.2
                           http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
                           http://www.mulesource.org/schema/mule/scripting/2.2
                           http://www.mulesource.org/schema/mule/scripting/2.2/mule-scripting.xsd
                           http://www.mulesource.org/schema/mule/vm/2.2
                           http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/lang
                           http://www.springframework.org/schema/lang/spring-lang-2.5.xsd
                           "
      >
```

3 Begin Spring bean configuration

```
<spring:beans>
    <lang:groovy id="md5Filter"
        script-source="classpath:script/md5Filter.groovy" />
</spring:beans>
```

Declare lang namespace**2****Define Groovy bean implementation****4****5 Inject filter reference**

```
<model name="springFilterModel">
    <service name="in">
        <inbound>
            <vm:inbound-endpoint path="in"/>
            <selective-consumer-router>
                <filter ref="md5Filter" />
            </selective-consumer-router>
        </inbound>
        <echo-component/>
    </service>
</model>
```

In order to make use of Spring's scripting support we need to make sure the `spring` and `lang` namespaces are defined—this is handled by ① and ②. We begin our bean definitions on ③. In this case we want to use the Groovy script for listing 14.12 as the bean. This is accomplished by ④, where we set the bean's ID and define the location of the script. We then inject the bean into the selective-consumer router on ⑤. The router will now invoke the Groovy script, only letting messages through whose payloads' MD5 checksum match that of the checksum header. Spring also supports the inlining of scripts in its configuration.

BEST PRACTICE Externalize your Spring configuration. When you begin to have more than a few bean definitions, scripting or otherwise, it makes sense to put them in their own file and import them into your Mule config using the `<spring:import...>` element you learned about in chapter 2.

14.3.2 Auto-reloading scripts

As we mentioned at the beginning of this chapter, one of the major benefits of working with a scripting language is the instant feedback it affords. This saves you the hassle of having to redeploy your code any time you make a change, and as such can have a dramatic impact on your development productivity. Although the JSR-223 spec doesn't supply any facilities for auto-reloading scripts, we can leverage Spring's lang namespace and achieve this functionality for Mule components. To do this, you just need to set the refresh-check-delay parameter on the Groovy bean, as illustrated in listing 14.14.

Listing 14.14 Auto-refreshing a script using the refresh-check-delay parameter

```
<spring:beans>
    <lang:groovy id="md5Filter"
        script-source="file:script/md5Filter.groovy"
        refresh-check-delay="2000"/>
</spring:beans>
```

Refresh script every 2 seconds

By setting the refresh-check-delay parameter to 2000, we're telling Spring we want it to check the script for changes every 2 seconds. You can now make changes to the md5Filter.groovy script without having to restart Mule. This functionality can also be leveraged in components and transformers. In these cases, you simply define your component or transformer as a Spring lang bean and then inject the reference as needed. Just note that in situations like this, Mule won't autopopulate the script context for you. You'll need to use the techniques from chapter 13 to manually obtain references to the EventContext, message and message payload. Listing 14.14 illustrates this by reimplementing the toUpperCase transformer from listing 14.15 as a Groovy script injected as a transformer.

Listing 14.15 Using Spring to declare a refreshable Transformer

```
<spring:beans>
    <lang:groovy id="toUpperTransformer"
        script-source="file:script/toUpper.groovy"
        refresh-check-delay="2000"/>
</spring:beans>
```

- 1 Define transformer with location of script
- 2 Declare refresh interval of 2 seconds
- 3 Inject transformer reference

```
<model name="springTransformerModel">
    <service name="springTransformerService">
        <inbound>
            <vm:inbound-endpoint path="in">
                <transformer ref="toUpperTransformer"/>
            </vm:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint path="out"/>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

We define the script file's location on ❶ along with the refresh interval on ❷. The transformer is then defined on ❸ to reference the appropriate bean ID. Since this bean is being configured “outside” of Mule, we don't benefit from any of the features of the scripting namespace. This means we must actually implement the org.mule.api.transformer.Transformer interface as discussed in chapter 5 for things to work properly. Listing 14.16 shows the contents of `toUpperCase.groovy`, which does just that.

Listing 14.16 Extending AbstractTransformer in a Groovy script

```
import org.mule.transformer.AbstractTransformer
class ToUpperTransformer extends AbstractTransformer {
    protected Object doTransform(Object o, String s) {
        return o.toUpperCase();
    }
}
```



The `toUpperCase.groovy` script will now be refreshed every 2 seconds. We can achieve similar functionality with Groovy components. In listing 14.1, we examined how to use an inline Rhino script to add a priority header to a message. We'll now port this script to Groovy and see how it can be dynamically refreshed by Spring and Mule. Listing 14.17 shows the same behavior implemented as a Groovy script with a class implementing the Callable interface.

Listing 14.17 Auto-refreshing a Groovy component

```
class MessageEnricher implements Callable {
    public Object onCall(MuleEventContext muleEventContext) {
        def message = muleEventContext.getMessage()
        if (message.payload =~ "STATUS: CRITICAL") {
            message.setProperty("PRIORITY", 'HIGH')
        } else {
            message.setProperty("PRIORITY", 'LOW')
        }
        return message
    }
}
```

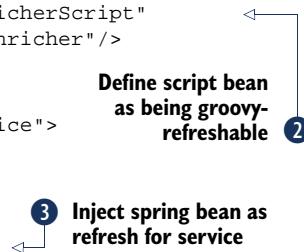
Since we don't have access to the scripting context prepopulated by Mule in Spring beans, we need to explicitly implement the Callable interface to get access to the `MuleEventContext`. We now need to configure Mule and Spring for the refresh. Listing 14.18 shows how to do this.

Listing 14.18 Configuring a refreshable Groovy refresh

```
<spring:beans>
    <lang:groovy id="messageEnricher"
        script-source="file:script/messageEnricher.groovy"
        refresh-check-delay="2000"/>
```



```
</spring:beans>  
  
<scripting:groovy-refreshable name="messageEnricherScript"  
refreshableBean-ref="messageEnricher" />  
  
<model name="springMessageEnrichmentModel">  
    <service name="springMessageEnrichmentService">  
        <inbound>  
            <vm:inbound-endpoint path="in"/>  
        </inbound>  
        <spring-object bean="messageEnricher" />  
        <outbound>  
            <pass-through-router>  
                <vm:outbound-endpoint path="out" />  
            </pass-through-router>  
        </outbound>  
    </service>  
</model>
```



We define the script bean as we've done before on ①. On ②, we're alerting Mule that the bean is refreshable. We finally inject the reference on ③, allowing us to modify the bean's implementation without restarting Mule.

Spring's scripting support nicely complements the scripting features of Mule. Being able to define arbitrary beans as scripts allows you to leverage the benefits of dynamic languages for Mule functionality, such as filters. This allows you to utilize scripts in Mule where there might not be explicit Mule schema support. The auto-refreshing of scripts furthers this point—permitting you to modify Mule's behavior dynamically without a restart.

14.4 Summary

Scripting languages can greatly benefit your development productivity. As we saw repeatedly in this chapter, Mule scripting support takes full advantage of this. By providing schema extensions for JSR-223 scripts, Mule allows you to directly implement components and transformers using any JSR-223 scripting engine. For areas of Mule where scripting support isn't explicit, Spring's lang namespace can be used to define beans using a scripting language. This additionally allows you to benefit from Spring's script auto-reloading capabilities in your Mule configurations. Finally, by extending its service interface binding functionality to scripts, Mule allows you to transparently invoke endpoint logic from your scripted components in a decoupled manner.

In this chapter we've seen how Mule's scripting support can simplify component and transformer development. Let's now see how Mule simplifies two other common integration challenges—business orchestration and event scheduling.

15

Business process management and scheduling with Mule

In this chapter

- Using jBPM for service orchestration
- Scheduling tasks with the Quartz transport
- Polling endpoints with the Quartz transport

Most of the examples we looked at while discussing transports and routing in chapters 3 and 4 were discrete—dealing with one scope of functionality at a time. For instance, we were interested in how to send JMS messages or how to group the responses from a collection of web services and aggregate their responses. In chapter 10, we saw how transactions allow us to perform these operations atomically. But what happens if the activity we need to coordinate takes longer than a typical transaction time-out, or needs to maintain state between multiple service invocations? Perhaps we have an endpoint that invokes a potentially long-running operation, such as waiting for a user to respond to a confirmation email. The receipt of this confirmation email may be one piece of an account-provisioning process—with events occurring before and subsequent dependent events occurring after the email is sent. While it might be possible to model such an activity using the routing

facilities introduced in chapter 4, it seems like a more straightforward solution should be at our disposal.

A job such as this is presumably invoked externally—some external user or system will initiate an account provisioning request. But what if we want to run a job at a specified time or interval? Maybe we have to perform a bulk load of data into a data warehouse. Such an operation will likely incur a large amount of overhead on the servers in question. As such, we probably want to start the bulk load after hours, when such an event wouldn't affect users. But how would we tell Mule to start such a job? So far we've only seen how to execute functionality when events occur on inbound endpoints—we don't (yet) have a way to initiate such an event from Mule.

In this chapter we'll look at solving the challenges above using Mule's process management support. We'll start off by seeing how Mule interacts with business process management (BPM) engines to coordinate long-running processes, such as the account provisioning process we just described. You'll see how Clood leverages this functionality to simplify its account provisioning processes. We'll then examine Mule's support for Quartz, a popular open source enterprise scheduling tool, to generate messages and trigger jobs at set times and intervals. Clood uses Mule and Quartz to perform various scheduling activities, such as triggering data loads and periodically polling JMS queues, as we'll also see in this chapter.

15.1 **Orchestrating services with Mule**

We saw in chapter 4 how to do ad hoc service composition with the chaining router. In listing 4.17, we used the chaining router to invoke multiple outbound synchronous endpoints, using the response of each endpoint as the input to the next. This allowed us to compose services together to form a single response. While the chaining router is useful in such a scenario, its limitations begins to become apparent as the complexity of the service orchestration grows. We already saw one such limitation at the beginning of this chapter with long-running operations. Another such limitation is branching—perhaps the response of one service dictates which subsequent service is invoked. This sort of functionality is unfeasible with naive orchestration approaches.

BPM engines provide an attractive option for solving such orchestration challenges. A BPM engine typically provides the functionality missing in a solution such as the chaining router. By supplying facilities such as process state management, job suspension, and service composition, complex business processes can be modeled and implemented. This is particularly important in a service-oriented environment, where disparate services often need to be tied together to accomplish some business need.

Before we delve into Mule's support for BPM engines, let's look at a challenge that might be solved by service orchestration. Clood, Inc.'s IT department has been using a manual method to provision employee accounts on various systems. This documented process was typically handled by the IT system administrators. As Clood has grown, though, this system has begun to be unmanageable—IT's sysadmins (SAs) are typically spending 1 to 2 hours a day dealing with account management issues. To provision an account, for instance, a Clood SA must do the following:

- 1 Use a command-line script to add the user's account data into an OpenLDAP directory.
- 2 Enter additional account data into an internal hours-tracking web application.
- 3 Create an account on a third-party ticketing system.
- 4 If the user is in sales, create an account on a third-party CRM system.
- 5 If the user is in operations, generate a Secure Shell (SSH) key pair for the user.
- 6 Create a mailbox for the user on the corporate IMAP server.
- 7 Send an email to the user's personal email account to confirm that he can access these systems.
- 8 Wait for the user's confirmation that all is well.

You've been tasked to work with the lead system administrator on automating this tedious process. After talking with the lead sysadmin, it becomes apparent that all eight steps can be performed programmatically in some way or another. As such, you decide to implement each of the steps as a Mule service, with a component implementing the business logic required to complete each step. Additionally, you and the lead system administrator decide to represent the provisioning request as XML. Such a request would resemble listing 15.1.

Listing 15.1 Representing account data as XML

```
<account>
    <name>John Doe</name>
    <group>Operations</group>
    <personalEmail>johndoe@gmail.com</personalEmail>
</account>
```

There are no current requirements to expose the provisioning process as a web service, so you decide to use JMS to accept the initial account XML from IT and coordinate the subsequent provisioning activities. This is illustrated in listing 15.2.

Listing 15.2 Exposing provisioning steps via JMS and VM endpoints

```
<spring:beans>
    <spring:import resource="spring-config.xml" />
</spring:beans>

<jms:activemq-connector
    name="jmsConnector"
    specification="1.1"
    brokerURL="tcp://mq.cloud.com:61616" />

<vm:endpoint name="SendEmailConfirmationEndpoint"
    path="it.provisioning.confirmation"/>

<model name="accountProvisioningModel">
    <service name="crmProvisioning">
        <inbound>
            <jms:inbound-endpoint queue="it.provisioning.crm" />
        </inbound>
    </service>
</model>
```



Accept CRM provisioning requests

```

<component>
    <spring-object bean="crmAccountService"/>
</component>
</service>
<service name="hourEntryProvisioning">
    <inbound>
        <jms:inbound-endpoint queue="it.provisioning.hour-entry"/>
    </inbound>
    <component>
        <spring-object bean="hourEntryAccountService"/>
    </component>
</service>
<service name="ldapAccountProvisioning">
    <inbound>
        <jms:inbound-endpoint queue="it.provisioning.directory"/>
    </inbound>
    <component>
        <spring-object bean="ldapAccountService"/>
    </component>
</service>
<service name="sshAccountProvisioning">
    <inbound>
        <jms:inbound-endpoint queue="it.provisioning.ssh"/>
    </inbound>
    <component>
        <spring-object bean="sshAccountService"/>
    </component>
</service>
<service name="ticketingAccountProvisioning">
    <inbound>
        <jms:inbound-endpoint queue="it.provisioning.ticketing"/>
    </inbound>
    <component>
        <spring-object bean="ticketingAccountService"/>
    </component>
</service>
<service name="imapAccountProvisioning">
    <inbound>
        <jms:inbound-endpoint queue="it.provisioning imap"/>
    </inbound>
    <component>
        <spring-object bean="imapAccountService"/>
    </component>
</service>
<service name="sendEmailConfirmation">
    <inbound>
        <vm:inbound-endpoint ref="SendEmailConfirmationEndpoint"/>
    </inbound>
    <spring:object bean="sendEmailConfirmationService"/>
    <outbound>
        <pass-through-router>

```

Accept hours entry provisioning requests

Accept LDAP provisioning requests

Accept SSH provisioning requests

Accept ticketing provisioning requests

Accept IMAP account provisioning requests

Send email confirmations

```

<smtp:outbound-endpoint
    host="${smtp.host}"
    from="${smtp.from}"
    subject="Accounting Invoice"
    to="${smtp.to}">
    <email:string-to-email-transformer/>
</smtp:outbound-endpoint>
</pass-through-router>
</outbound>
</service>
<service name="receiveEmailConfirmation">
    <inbound>
        <imap:inbound-endpoint
            host="${imap.host}"
            port="${imap.port}"
            user="${imap.user}"
            password="${imap.password}">
            <email:email-to-string-transformer/>
        </imap:inbound-endpoint>
    </inbound>
</service>
</model>
```



Receive email confirmation responses

We're already a long way from the previous approach—the prior manual steps can now be triggered programmatically via JMS messages containing XML account data. This could form the basis for a centralized administration console, with a single form submission driving JMS message submission to each enumerated queue. You decide to use JMS for any provisioning endpoint that needs to be invoked externally. This could allow operations to spin up mailboxes without provisioning a full account, for instance. The `sendEmailConfirmation` endpoint is finally set to use a VM endpoint, as you have no requirement (or intention) to allow entities external to Mule to send account confirmation emails.

There are some drawbacks to this approach. The administrative console is now tightly coupled to the queue submission sequence. If we modify the provisioning process, we'll need to modify the admin console. This drawback is further amplified if a requirement arises to support account provisions from outside the console. Perhaps Cloud, Inc., at some point merges with another company and needs to expose its account provisioning process. In this scenario, changes to the account provisioning procedure would require modifications on both the internal administrative console and the externally exposed service.

Let's see how we can solve this problem by encapsulating the provisioning process in a BPM engine.

15.1.1 Introducing jBPM

jBPM is an open source BPM product developed by JBoss, Inc. In addition to providing a BPM engine, *jBPM* also supplies an XML-based process definition language called *jPDL* as well as a graphical process designer called *GPD*. *jBPM* can be deployed as a

standalone server, deployed to an application server, or embedded into an existing Java application. In this section we'll be using jBPM embedded in a running Mule instance. Full documentation on jBPM, jPDL, GPD, and the various deployment options for jBPM are available from the jBPM web site: <http://www.jboss.org/jbosbjbpm/>.

DEFINING THE PROCESS WITH JPDL

Let's see how we can orchestrate the endpoints we defined in listing 15.2 with jBPM. Figure 15.1 illustrates the process definition as defined by Cloud's IT group.

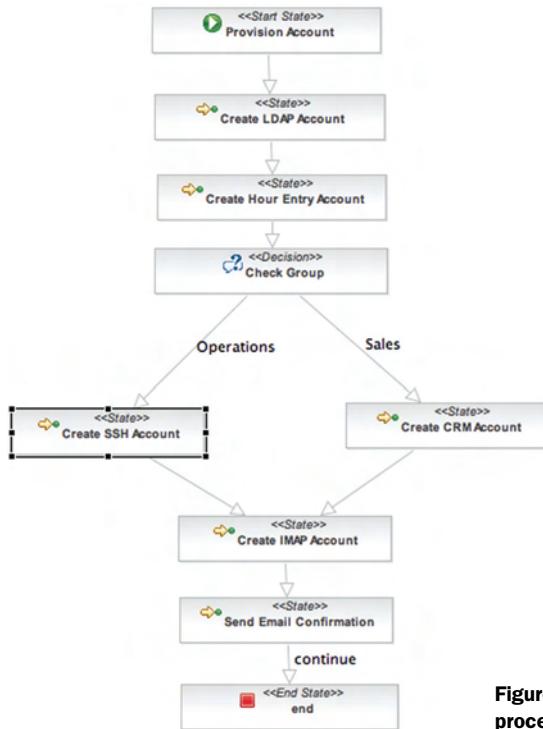


Figure 15.1 The account provisioning process visualized with GPD

You and the lead sysadmin came up with figure 15.1 using jBPM's GPD tool. This is an Eclipse plug-in that allows you to graphically lay out process definitions without worrying about XML or Java code. This diagram can be subsequently used to generate a skeleton business definition. Listing 15.3 is the skeleton definition generated from figure 15.1.

Listing 15.3 Representing the account provisioning process as a process definition

```

<process-definition name="AccountProvisioning">

  <start-state name="Provision Account">
    <transition to='Create LDAP Account' />
  </start-state>
  
```

```

<state name='Create LDAP Account'>
    <transition to='Create Hour Entry Account' />
</state>

<state name='Create Hour Entry Account'>
    <transition to='Check Group' />
</state>

<decision name="Check Group">
    <transition name="Operations" to="Create SSH Account" />
    <transition name="Sales" to="Create CRM Account" />
</decision>

<state name='Create SSH Account'>
    <transition to="Create IMAP Account" />
</state>

<state name='Create CRM Account'>
    <transition to="Create IMAP Account" />
</state>

<state name='Create IMAP Account'>
    <transition to='Send Email Confirmation' />
</state>

<state name='Send Email Confirmation'>
    <transition name="continue" to='end' />
</state>

<end-state name='end' />
</end-state>

```

The business definition in listing 15.3 should capture the account provisioning logic we're trying to orchestrate. After receiving the provisioning account, we need to create an LDAP account, and then create an hours entry account. After this, depending on whether the user is in operations or sales, we want to create either an SSH account or a CRM account. After this, we want to provision an IMAP account and then send a confirmation email to the user. When the user responds to the confirmation email, the process ends.

NOTE jBPM is currently the only BPM engine supported out of the box by Mule. The interface for the BPM support is provider-agnostic, and support for other BPM engines is possible through the `org.mule.transport.bpm.BPMS` interface.

15.1.2 Using jBPM with Mule

Now that we have our business logic exposed and our business process defined, we need to tie the two together. We'll do this by fleshing out the jPDL definition in listing 15.3. Generally, you'd do this by implementing jBPM *actions*. These are classes that perform the business logic for each step. In our case, we need to implement two pieces of business logic. One must send JMS messages to the appropriate JMS endpoint. The other must make some sort of determination of which group the user is in to determine whether to create SSH or CRM accounts. For the first case, Mule ships

with a set of jBPM action handlers to simplify common tasks, such as sending Mule messages. These actions are described in table 15.1.

Table 15.1 Mule supplies action handlers to accommodate common tasks.

| Action | Description |
|--------------------------|---|
| Continue | Advance to the next state |
| SendMuleEvent | Send a message to a Mule endpoint |
| SendMuleEventAndContinue | Send a message to a Mule endpoint and advance to the next state |
| StoreIncomingData | Store incoming data to a variable |
| ValidateMessageSource | Validate a message came from a certain source |
| ValidateMessageType | Validate a message is of a certain type |

Let's use these action handlers to coordinate with our Mule endpoints. This is illustrated in listing 15.4.

Listing 15.4 The finalized jPDL account provisioning definition

```

<start-state name="provisioningRequest">
    <transition to='Validate Account' />
</start-state>

<state name="Validate Account">
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.ValidateMessageSource" >Validate incoming messages 1
            <expectedSource>
                endpoint.jms.it.provisioning.requests
            </expectedSource>
        </action>
        <action
            class="org.mule.transport.bpm.jbpm.actions.StoreIncomingData" >Store payload in process variable 2
            <variable>incoming</variable>
        </action>
        <action
            class="org.mule.transport.bpm.jbpm.actions.Continue" />
        </action>
    </event>
    <transition to='Create LDAP Account' />
</state>

<state name='Create LDAP Account'>
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.SendMuleEventAndContinue" >Send message to LDAP queue 3
            <endpoint>jms://it.provisioning.ldap</endpoint>
            <payloadSource>incoming</payloadSource> <synchronous>false</synchronous>
        </action>
    </event>

```

```

<transition to='Create Hour Entry Account' />
</state>

<state name='Create Hour Entry Account'>
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.SendMuleEventAndContinue">
                <endpoint>jms://it.provisioning.hour-entry</endpoint>
                <payloadSource>incoming</payloadSource>
                <synchronous>false</synchronous>
        </action>
    </event>
    <transition to='Check Group' />
</state>

<decision name="Check Group">
    <handler class="com.clood.it.ResolveGroup"/>
    <transition name="Operations" to="Create SSH Account" />
    <transition name="Sales" to="Create CRM Account" />
</decision>

<state name='Create SSH Account'>
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.SendMuleEventAndContinue">
                <endpoint>jms://it.provisioning.ssh</endpoint>
                <payloadSource>incoming</payloadSource>
                <synchronous>false</synchronous>
        </action>
    </event>
    <transition to="Create IMAP Account" />
</state>

<state name='Create CRM Account'>
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.SendMuleEventAndContinue">
                <endpoint>jms://it.provisioning.crm</endpoint>
                <payloadSource>incoming</payloadSource>
                <synchronous>false</synchronous>
        </action>
    </event>
    <transition to="Create IMAP Account" />
</state>

<state name='Create IMAP Account'>
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.SendMuleEventAndContinue">
                <endpoint>jms://it.provisioning imap</endpoint>
                <payloadSource>incoming</payloadSource>
                <synchronous>false</synchronous>
        </action>
    </event>
    <transition to='Send Email Confirmation' />
</state>

```

Send message to hours entry provisioning queue 4

Branch based on whether user is in Operations or Sales 5

Send message to SSH queue 6

Send message to CRM queue 7

Send message to IMAP queue 8

```

<state name='Send Email Confirmation'>
    <event type="node-enter">
        <action
            class="org.mule.transport.bpm.jbpm.actions.SendMuleEvent">
                <endpoint>SendEmailConfirmationEndpoint</endpoint>
                <payloadSource>incoming</payloadSource>
                <synchronous>false</synchronous>
            </action>
        </event>
        <transition to="end"/>
    </state>

    <end-state name='end'>
        <event type="node-enter">
            <action
                class="org.mule.transport.bpm.jbpm.actions.SendMuleEvent">
                    <endpoint>jms://it.provisioning.completed</endpoint>
                    <payloadSource>incoming</payloadSource>
                    <synchronous>false</synchronous>
                </action>
            </event>
        </end-state>
    </process-definition>

```

We start off by validating the incoming provisioning request coming from the appropriate Mule endpoint in ① (you'll see shortly how this request is sent). We then store the message payload, in this case the account XML, in a process variable on ②. ③ and ④ send asynchronous provisioning requests to create the LDAP and hours accounts. If we were interested in the responses for these actions, we'd set synchronous to true. This would result in the response from the endpoint being stored in the incoming process variable. ⑤ is the decision node that will branch the process execution based on the group the account is associated with. ResolveGroup is used to accomplish this and is illustrated in listing 15.5.

Listing 15.5 A Java class to determine group membership

```

public class ResolveGroup implements DecisionHandler {

    private XPath groupExpression;

    public ResolveGroup() {
        try {
            groupExpression = XPath.newInstance("//group");
        } catch (JDOMException e) {
            throw new RuntimeException(e);
        }
    }

    public String decide(ExecutionContext executionContext) throws Exception {
        String account =
(String) executionContext.getContextInstance().getVariable("incoming");
        SAXBuilder builder = new SAXBuilder();

```

```

        Document doc =
builder.build(new ByteArrayInputStream(account.getBytes()));

        return (groupExpression.valueOf(doc));
    }
}

```

This class performs an XPath evaluation against the account request and returns the value of the group expression to jBPM. This is subsequently used to decide where to branch by the decision node in ⑤, ⑥, ⑦, and ⑧ in listing 15.4 send more provisioning requests to their appropriate endpoints asynchronously using `SendMuleEventAndContinue`. Finally, when we reach ⑨, the email confirmation request is sent. Instead of using `SendMuleEventAndContinue`, though, we're simply using `SendMuleEvent`. This causes the process to wait until advanced externally. In our case, we'll be advancing the process from Mule once the user responds to the email confirmation. Before we can do that, though, we need to embed jBPM into our Mule configuration.

BEST PRACTICE Use Mule's custom jBPM actions to send messages to Mule endpoints from jBPM.

RUNNING JBPM EMBEDDED IN MULE

We now need to configure a jBPM instance to start when we launch Mule. One way to do this is using Spring's jBPM module. Listing 15.6 illustrates how to augment the Spring configuration we're referencing in listing 15.6 to bootstrap jBPM.¹

Listing 15.6 Using Spring to bootstrap jBPM for a Mule instance

```

<bean id="jbpmConfig"
      class="org.springframework.workflow.jbpm31. \
LocalJbpmConfigurationFactoryBean">
    <property name="sessionFactory">
        <ref local="jbpmSessionFactory"/>
    </property>
    <property name="configuration">
        <value>jbpm.cfg.xml</value>
    </property>
    <property name="processDefinitions">
        <list>
            <bean id="accountProvisioning"
                  class=
"org.springframework.workflow.jbpm31.definition. \
ProcessDefinitionFactoryBean">
                <property
name="definitionLocation">
                    <value>account-provisioning-process.xml</value>
                </property>
            </bean>
        
```

Location of our
process definition

¹ As we mentioned before, this isn't the only, or necessarily the best, deployment strategy for jBPM and Mule. You can also deploy jBPM as a standalone web app or in an application container. Consult the jBPM documentation for further details.

```

</list>
</property>
<property name="createSchema">
    <value>false</value>
</property>
</bean>

<bean id="jbpmDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:mem:mule</value>
    </property>
</bean>

<bean id="jbpmTypes"
class="org.springframework.orm.hibernate3.TypeDefinitionBean">
    <property name="typeName">
        <value>string_max</value>
    </property>
    <property name="typeClass">
        <value>org.jbpm.db.hibernate.StringMax</value>
    </property>
</bean>

<bean id="jbpmSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource">
        <ref local="jbpmDataSource"/>
    </property>
    <property name="mappingLocations">
        <value>classpath*:org/jbpm/**/*.hbm.xml</value>
    </property>
    <property name="typeDefinitions">
        <ref local="jbpmTypes"/>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">
                org.hibernate.dialect.HSQLDialect
            </prop>
            <prop key="hibernate.cache.provider_class">
                org.hibernate.cache.HashtableCacheProvider
            </prop>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
        </props>
    </property>
</bean>
```

The code is annotated with three numbered callouts:

- Callout 1:** Points to the line `<value>classpath*:org/jbpm/**/*.hbm.xml</value>`. It contains the text "The location of our process definition is defined on ①. This will generally be somewhere on Mule's classpath."
- Callout 2:** Points to the line `<value>jdbc:hsqldb:mem:mule</value>`. It contains the text "Data source to persist process data to ②".
- Callout 3:** Points to the line `org.hibernate.dialect.HSQLDialect`. It contains the text "Dialect used by Hibernate ③".

The important bits of listing 15.6 are on ①, ②, and ③. The location of our process definition is defined on ①. This will generally be somewhere on Mule's classpath. The data source and corresponding Hibernate dialect jBPM will use are defined on ② and ③. For testing purposes, we're using an in-memory HSQL data source. Production

instances of this configuration will likely use an external database instance, such as MySQL running on a database server.

NOTE BPEL, or *Business Process Execution Language*, is a web-services-based business process management solution. BPEL engines are used to orchestrate SOAP endpoints and as such can interact with Mule endpoints exposed using the CXF and Axis transports.

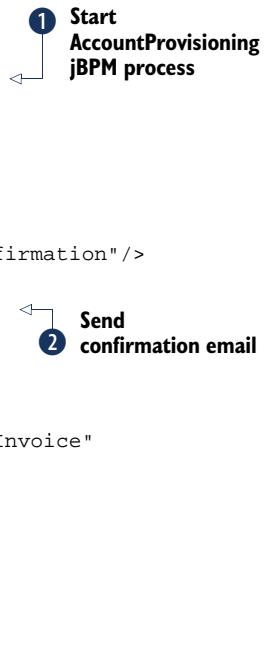
We now need to let Mule interact with the account provisioning process. In our case, we need to start the process when an account provisioning request arrives and stop the process once the email confirmation from the user is received. We also need to maintain some sort of state once the confirmation email leaves Mule, so we can correlate the response. Listing 15.7 shows the modified bits of listing 15.2 to accomplish this.

Listing 15.7 Using BPM outbound endpoints to start and advance a jBPM process

```
<service name="createAccount">
    <inbound>
        <jms:inbound-endpoint
            queue="it.provisioning.requests"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <bpm:outbound-endpoint
                name="startProvisionRequest"
                address="bpm://AccountProvisioning"/>
        </pass-through-router>
    </outbound>
</service>

<service name="sendEmailConfirmation">
    <inbound>
        <jms:inbound-endpoint queue="it.provisioning.confirmation"/>
    </inbound>
    <spring:object bean="sendEmailConfirmationService"/>
    <outbound>
        <pass-through-router>
            <smtplib:outbound-endpoint host="${smtp.host}"
                from="${smtp.from}"
                subject="Accounting Invoice"
                to="${smtp.to}">
                <email:string-to-email-transformer/>
            </smtplib:outbound-endpoint>
        </pass-through-router>
    </outbound>
</service>

<service name="receiveEmailConfirmation">
    <inbound>
        <imap:inbound-endpoint host="${imap.host}"
            port="${imap.port}"
            user="${imap.user}"
            password="${imap.password}">
```



```

<email:email-to-string-transformer/>
</imap:inbound-endpoint>
</inbound>
<spring-object bean="populateProcessId"/>
<outbound>
    <pass-through-router>
        <bpm:outbound-endpoint address="bpm://AccountProvisioning">
            <transformers>
                <message-properties-transformer>
                    <add-message-property
                        key="MULE_BPM_PROCESS_ID"
                        value="#{header:MULE_BPM_PROCESS_ID}" />
                </message-properties-transformer>
            </transformers>
        </bpm:outbound-endpoint>
    </pass-through-router>
</outbound>
</service>

```

The diagram illustrates the workflow. Step 1 shows the BPM outbound endpoint being invoked. Step 2 shows the email confirmation being received. Step 3 shows the IMAP inbound endpoint triggering a scheduled poll. Step 4 shows the association of the email with the jBPM process. Step 5 shows the advancement of the AccountProvisioning jBPM process.

The account provisioning process will begin when a JMS message is received on the `it.provisioning.request` queue. When this occurs, the BPM outbound endpoint is invoked on ①. When the address for the BPM outbound endpoint is specified without any trailing URI, a new process is started. In this case, an account provisioning process will be started by jBPM and an ID will be associated with it. This ID will be subsequently accessible as the `MULE_BPM_PROCESS_ID` property on the Mule message.

The business process will now continue as we described, until it hits the `Send Email Confirmation` state. At this point, the jBPM process will stop and won't continue until advanced externally. In order to advance the process, we must first receive an email confirmation from the user that her account has been created successfully. This email confirmation is sent by the SMTP outbound endpoint on ②. We have a problem, though. When the email response is received by ③, we need a way to associate the email with the process ID. We can do this by associating the email address with the process ID using business logic defined in `sendEmailConfirmationService` on ④. For the purposes of this example, let's assume the component will store the `MULE_BPM_PROCESS_ID` of the message in a database keyed by the email address.

When the user receives and replies to the email, it's picked up by the IMAP inbound endpoint defined on ③ during one of its scheduled polls of its account on Cloud's email server. Once this message is received, the `populateProcessId` component is invoked on ④. It queries the database updated by `sendEmailConfirmationService` on ④ and populates the `MULE_BPM_PROCESS_ID` of the message with the appropriate value. The BPM outbound endpoint is then subsequently invoked on ⑤. As you can see, the address is qualified with the process ID using an expression evaluator on the endpoint.² This will cause jBPM to advance the process, which will cause `Send Email Confirmation` to advance to the end state and our process to end. The

² See appendix A for more information about using expression evaluators on endpoints.

BPM transport also supplies a BPM inbound endpoint. This allows Mule to receive events directly from the BPM engine.

NOTE In this example we're ignoring the possibility of exceptions occurring in Mule or jBPM. Exceptions occurring in Mule services are handled as we discussed in chapter 8. In such a situation, you'll likely want to abort the process on the jBPM side. A BPM process can be aborted from Mule by specifying the abort action on an endpoint invocation as follows:

```
<bpm:outbound-endpoint address="bpm://AccountProvisioning">
    <transformers>
        <message-properties-transformer>
            <add-message-property
                key="MULE_BPM_PROCESS_ID"
                value="#[header:MULE_BPM_PROCESS_ID]"/>
            <add-message-property
                key="MULE_BPM_ACTION"
                value="abort"/>
        </message-properties-transformer>
    </transformers>
</bpm:outbound-endpoint>
```

Exceptions that occur in a Mule action handler within jBPM must be handled by jBPM's error handling facilities. This will most likely involve defining an exception handler for the process and is fully documented in the jPDL reference materials.

By now you should be comfortable with orchestrating processes with Mule. We saw how to use jBPM in conjunction with Mule for native process orchestration. This allowed us to perform a composite set of activities against a set of Mule endpoints. Let's turn our attention now to Mule's scheduling features and how they can be used to trigger orchestrations like the ones we just discussed.

15.2 Job scheduling with Mule

In the introduction to this chapter, we discussed a scenario in which we needed to trigger a bulk load into a data warehouse. To avoid adversely impacting users, we wanted to precisely schedule the load for a time after business hours. Situations like this are common in computing environments. You're no doubt familiar with the Unix scheduling commands cron and at or the Windows Task Scheduler. These facilities allow us to schedule events at certain or repeated intervals. In the Java ecosystem, there are commercial and open source tools that meet this need. One such tool is Quartz. Quartz is an open source job scheduling system that can be used for a variety of scheduling tasks. In this section we'll investigate Mule's support for Quartz. We'll start by seeing how we can execute jobs at specified times and intervals. We'll then see how messages can be used to trigger a scheduling job.

15.2.1 Using Quartz to schedule jobs

Let's consider the data warehousing loading problem we introduced earlier in the context of Clood, Inc. In listing 10.5 we saw how Clood was performing inserts into operational and warehouse databases. In this example, billing statistics were being sent across as the payload of JMS messages. The overhead of the XA transaction when inserting between both databases during high message volume has become problematic. The high load on the database servers during the inserts has begun to affect a front-end web portal that queries these databases. Since there isn't a current requirement to perform real-time inserts into the data warehouse, you and Clood's DBA have decided to schedule a bulk data load at 30 minutes past midnight every night. The bulk load is performed by a sequence of complex SQL queries, so rather than use a JDBC endpoint, you've decided to put this business logic into a component. Now we just need to trigger the data load at the appropriate time. Listing 15.8 illustrates how to use a Quartz inbound endpoint to accomplish this.

Listing 15.8 Triggering jobs with a cron expression

```
<model name="quartzCronModel">
    <service name="quartzCronService">
        <inbound>
            <quartz:inbound-endpoint jobName="cron-job"
                cronExpression="0 30 0 * * ?">
                <quartz:event-generator-job/>
            </quartz:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint path="data-warehouse-load" />
            </pass-through-router>
        </outbound>
    </service>
    <service name="warehouseService">
        <inbound>
            <vm:inbound-endpoint path="data-warehouse-load" />
        </inbound>
        <component class="com.clood.warehouse.DataWarehousingService">
            <method-entry-point-resolver acceptVoidMethods="true">
                <include-entry-point method="process" />
            </method-entry-point-resolver>
        </component>
    </service>
</model>
```

The diagram shows the flow of the Quartz cron job triggering process:

- 1 Define Quartz inbound endpoint**: Points to the `<quartz:inbound-endpoint ...>` configuration in the XML.
- 2 Generate message when job is triggered**: Points to the `cronExpression="0 30 0 * * ?"` and the `<quartz:event-generator-job/>` configuration.
- 3 Send message through VM outbound endpoint**: Points to the `<vm:outbound-endpoint path="data-warehouse-load" />` configuration.
- 4 Receive message on VM inbound endpoint**: Points to the `<vm:inbound-endpoint path="data-warehouse-load" />` configuration.

The Quartz inbound endpoint configured on ① will instantiate a Quartz scheduler and define a job with the supplied `jobName`. The scheduler will then send events to the Quartz inbound endpoint using the supplied `cronExpression`. If you've used the Unix cron command, the format in figure 15.2 should look familiar to you.³

³ With the exception of the extra field for seconds.

This cronExpression will trigger a job at 30 minutes past midnight every day. An asterisk on the field indicates that the job will be triggered for each value of that field. For instance, the asterisks on the *day of month* and *month* fields indicate that the job is to be run every day of the month and every month of the year. The question mark (?) on a field indicates ambivalence as to when the event should occur. In the example, for instance, we don't care what day of the week the event falls on. The *month* and *day of week* fields can take either numeric values or the appropriate abbreviation. For instance, you can specify FEB as a month value or WED as a day-of-week value. Ranges can also be used. An expression of 0 * 0-2 * * ? would cause a job to be triggered every minute from midnight to 2 a.m., for instance. This also works with month and day-of-week values, letting you specify ranges such as MON-FRI or JAN-MAR.

We now need to define a job on the Quartz inbound endpoint to specify what happens when a job is triggered. In this case, we have an event-generator-job specified on ❷. This indicates that we want a Mule message generated when the Quartz endpoint is triggered. This event will be routed to the VM outbound endpoint on ❸, which is sent to the VM inbound endpoint on ❹. At this point the data warehouse job will run.

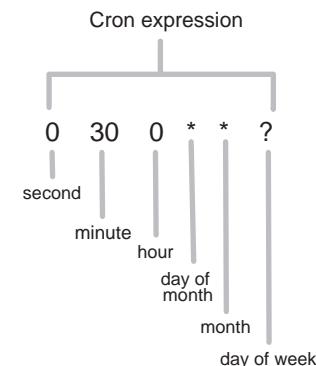


Figure 15.2 Using a Quartz cron expression to trigger a job every half hour and detail the meaning of each field

15.2.2 Polling endpoints

You might recall from our discussion of transports in chapter 3 that certain transports support polling behavior. The JDBC, HTTP, and IMAP inbound endpoints can all behave this way, periodically polling a remote resource for data. Certain transports, such as JMS, don't natively support this behavior. This could be useful in certain scenarios. Perhaps we have a JMS queue that we only want to consume messages off of periodically. This could be useful if we want to batch messages up and process them only at a specified interval. This sort of behavior is possible by using a Quartz job endpoint. Listing 15.9 illustrates how to consume messages off a specified JMS queue every 30 seconds.

Listing 15.9 Using a Quartz job endpoint to poll an endpoint

```

<model name="quartzPollingModel">
    <service name="quartzPollingService">
        <inbound>
            <quartz:inbound-endpoint jobName="poll-job"
                repeatInterval="30000">
                <quartz:endpoint-polling-job> ❷ Define endpoint-polling job
                    <quartz:job-endpoint address="jms://messages" />
  
```

❶ Define Quartz inbound endpoint

❷ Define endpoint-polling job

```

        </quartz:endpoint-polling-job>
    </quartz:inbound-endpoint>
</inbound>
<echo-component/>
</service>

</model>

```

Consume pending
messages off supplied
JMS endpoint 3

The Quartz inbound endpoint defined on ① will invoke the endpoint-polling job on ② every 30 seconds. The endpoint-polling job is configured with a job endpoint to invoke on ③. In this case we want to consume pending messages off the messages JMS queue. The payloads of these messages are then echoed to the console by the echo component.

BEST PRACTICE Use the Quartz endpoint-polling job to perform advanced periodic execution on endpoints that might otherwise not support it.

The endpoint-polling job is also useful if the polling mechanism provided by a transport isn't sophisticated enough for your needs. A file inbound endpoint, for instance, can be configured with a polling frequency to read a source directory for files. You might need something more granular than this. For instance, you might only want to read the source directory every Wednesday morning at 6:00 a.m. This can be accomplished by using the endpoint-polling job in conjunction with a cron expression, as illustrated in listing 15.10.

Listing 15.10 Polling a file endpoint using a Quartz cron expression

```

<model name="quartzPollingFileModel">
    <service name="quartzPollingFileService">
        <inbound>
            <quartz:inbound-endpoint jobName="cron-job"
                cronExpression="0 0 6 ? * WED">
                <quartz:endpoint-polling-job>
                    <quartz:job-endpoint address="file://./data"/>
                </quartz:endpoint-polling-job>
            </quartz:inbound-endpoint>
        </inbound>
        <echo-component/>
    </service>
</model>

```

1 Define Quartz
inbound endpoint
with given
cronExpression
2 Read files from
supplied file endpoint 2

We have a cronExpression configured on ① to start a job at 6:00 a.m. every Wednesday morning. This will trigger the job endpoint on ② to read from the ./data directory. The contents of the files will then be sent to the echo component and displayed on the console.

15.2.3 Dispatching jobs

So far we've seen how the Quartz transport can be used in conjunction with inbound endpoints for event scheduling. Scheduling can also be useful in conjunction with

message dispatch as well. A message is typically dispatched as soon as it reaches an outbound endpoint in a service. This might not be appropriate, though. You may want to receive a message and then send the message at a later date. Let's see how this could be useful for Cloud in the context of the account provisioning example we discussed at the beginning of this chapter. Cloud's IT team has recently requested that all account provisioning take place at 9:00 a.m. from Monday through Friday. This is so the IT team can deal with any provisioning errors during business hours. Currently the requests are being sent to the BPM engine as they arrive. Listing 15.11 shows how to use a Quartz outbound endpoint along with a scheduled-dispatch job to accomplish this.

Listing 15.11 Dispatching jobs at a specified time

```
<service name="quartzDispatchService">
    <inbound>
        <jms:inbound-endpoint queue="inbound">
            <jms:jmsmessage-to-object-transformer
                name="JmsMessageToString"
                returnClass="java.lang.String"/>
        </jms:inbound-endpoint>
    </inbound>
    <outbound>
        <pass-through-router>
            <quartz:outbound-endpoint jobName="dispatch-job"
                cronExpression="0 0 9 ? * MON-FRI" repeatCount="1">
                <quartz:scheduled-dispatch-job>
                    <quartz:job-endpoint
                        address="jms://it.provisioning.requests"/>
                </quartz:scheduled-dispatch-job>
            </quartz:outbound-endpoint>
        </pass-through-router>
    </outbound>
</service>
```

The IT provisioning requests are received on the JMS queue as before. Instead of getting immediately passed to the BPM endpoint, though, they're scheduled for dispatch at the specified time. In this case, the `cronExpression` defined on ① ensures that the message will be scheduled for dispatch on 9:00 a.m. on any day except Saturday or Sunday. A message received on Saturday or Sunday will be sent at 9:00 a.m. Monday morning. We've set `repeatCount` to 1 to indicate that the message should only be fired once.

NOTE You might be wondering what happens if Mule crashes while jobs are queued, as in listing 15.11. Since the Quartz connector uses an in-memory job store, by default the jobs will be lost. You can thankfully use Quartz's support for persistent job stores to overcome this limitation. This is accomplished by creating a `quartz.properties` file in Mule's classpath and setting the `org.quartz.jobStore.class` with the scheduling factory appropriate for your needs. Full documentation is available

on the Quartz web site, but you'll most likely be interested in the JDBC-JobStoreTX, which will allow you to store your jobs in a database.

Integrated job scheduling allows Mule to send timed events without using an external scheduling mechanism. As we saw in this section, this allows us to schedule events, poll endpoints, and dispatch messages all within a single Mule configuration—greatly easing our configuration and deployment burdens when using these facilities.

15.3 *Summary*

Business process management and scheduling mechanisms can be complex moving parts in integration endeavors. This functionality is traditionally supplied by complex (and often expensive) standalone solutions. The burden of such approaches is lifted by Mule's built-in support for jBPM and Quartz. These transports allow Mule to orchestrate its services and schedule events—all within a consolidated configuration.

By now you should be comfortable with these business process management and scheduling features. You saw how these powerful ingredients enable you to compose and schedule services. They also demonstrate the flexibility of Mule's transport architecture—allowing services exposed over disparate transports to be composed and scheduled in a common manner. We'll now turn our attention to tuning Mule, where you'll learn how to troubleshoot and improve the performance of your Mule instances.

16

Tuning Mule

In this chapter

- Mule's threading model
- Configuring threading profiles
- Profiling and performance tuning

Whether you have predetermined performance goals and want to be sure to reach them, or you've experienced issues with your existing configuration and want to solve them, the question of tuning Mule will occur to you sooner or later in the lifetime of your projects. Like any application, Mule is constrained by the limits of memory size and CPU performance. Tuning Mule is about finding the sweet spot where your business needs meet the reality of software and hardware constraints.

The same way a race car needs tuning to adapt to the altitude of the track or to the weather it will race in, Mule can require configuration changes to deliver its best performance in the particular context of your project. Up to this point of the book, we've relied on the default configuration of Mule's internal thread pools and haven't questioned the performance of the different moving parts, whether they're standard or custom. We'll now tackle these tough questions.

In short, the objective of this chapter is twofold: to give you a deep understanding of Mule's threading model and to offer you some hints on how to configure

Mule so it reaches your performance targets. This chapter will often make references to previous chapters, as the quest for performance isn't an isolated endeavor but the outcome of scattered but related activities. We'll also examine how Cloud, Inc., tuned one of its most intensive services.

Let's start by looking deeper into the architecture of Mule than ever before in order to learn how threading works behind the scenes.

16.1 Understanding thread pools

We've evoked Mule's core architecture in section 1.3.6, where we introduced the design principles of SEDA. Following this architecture, Mule is designed with numerous thread pools at its heart. Each thread pool handles a specific task, such as receiving or dispatching messages, or invoking component entry point methods. When a thread is done with a particular task, it hands it off to a thread in the next thread pool before coming back to its own pool. This naturally implies some context switching overhead, hence an impact on performance. Bear in mind that the SEDA architecture isn't about maximizing performance but ensuring a graceful degradation under load, which allows it to handle peaks of activity in a predictable manner.¹

Are these thread pools used all the time? Not at all. As we'll discover in this section, the synchronicity configuration influences the usage of thread pools. We'll also look at how transport particularities can impact Mule's threading model. Finally, we'll discuss your options in term of configuring thread pools.

WARNING Don't mix up thread pools and component pools! The upcoming figures and discussions will help you tell them apart.

Let's start by discovering the different thread pools that exist in Mule. Figure 16.1 represents the three thread pools that can be involved when a service handles a message event. It also represents the component pool that may or may not have been configured for the component (see discussion in section 6.3.3).

NOTE In the coming figures, pay attention to the thread pool box. If it's full (three arrows represented), this means that no thread from this pool is being used. If a thread is used, its arrow is filled in (black) and moved out of the thread pool box into the stage where it's used. If the thread spans stages, its arrow is stretched accordingly.

The receiver and dispatcher thread pools belong to a particular connector object, while the service thread pool is specific to the service. This is illustrated in figure 16.2. A corollary of this fact is that if you want to segregate receiver and dispatcher pools for certain services, you have to configure several connectors. We'll come back to this idea in section 16.2.

¹ To probe further on Mule and SEDA, you can turn to this highly imaged article: <http://www.infoq.com/articles/SEDA-Mule>.

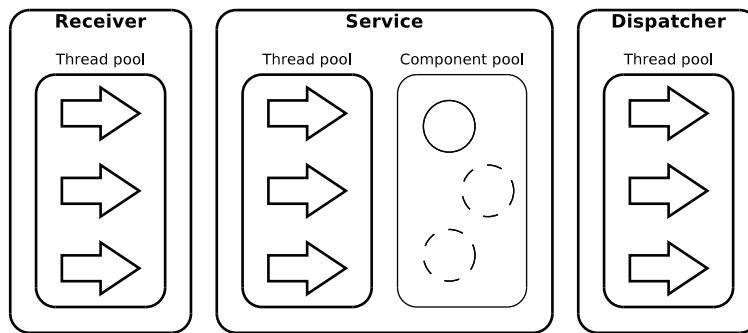


Figure 16.1 Mule relies on three thread pools to handle message events before, inside, and after each service.

If we “zoom” in on a thread pool, we discover a buffer alongside the threads themselves, as represented in figure 16.3. Each thread pool is associated with a buffer that can queue pending requests in case no thread is available for processing the message right away. It’s only when this buffer is full that the thread pool itself is considered exhausted. When this happens, Mule will react differently depending on your configuration: it can, for example, reject the latest or oldest request. We’ll see more options in section 16.1.3.

You now have a better understanding of the overall design of the Mule threading model. We’ll now look at how the synchronicity of a message event impacts the way thread pools are leveraged.

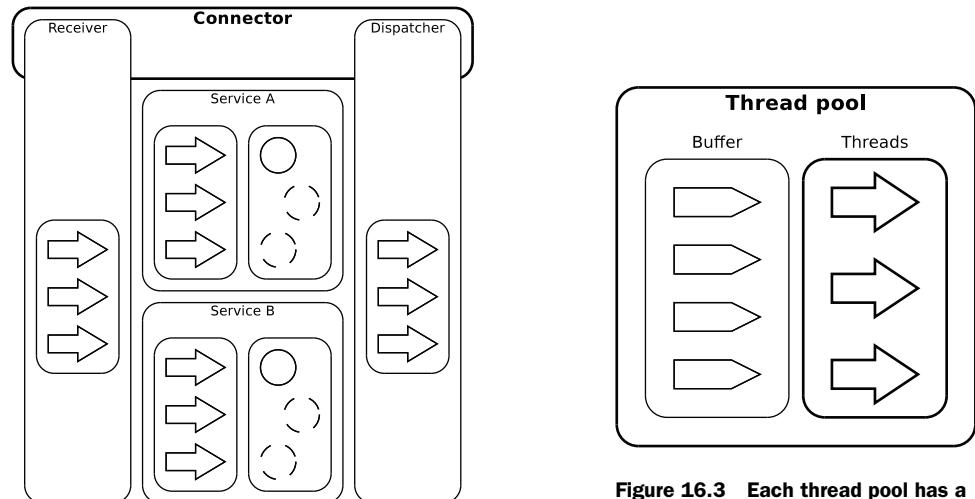


Figure 16.2 Two services using the same connector share the same receiver and dispatcher thread pools.

Figure 16.3 Each thread pool has a dedicated buffer that can accumulate pending requests if no thread is available to process the message.

16.1.1 Synchronicity aspects

At each stage of the message event processing, Mule decides whether it needs to borrow a thread from the corresponding pool. A thread is borrowed from the receiver or the dispatcher pool only if this stage is set to be asynchronous. And, by extension, the component thread pool is used only if the receiver was asynchronous. There are several factors that determine whether either the receiver or the dispatcher will be synchronous:

- *Configured synchronicity*—If you've set an inbound or outbound endpoint to be synchronous, the receiver or dispatcher will be synchronous, respectively.
- *Incoming event*—If the received message event is synchronous, a receiver can act synchronously even if its inbound endpoint is configured to be asynchronous.
- *Outbound router*—A router can enforce the dispatching stage to be synchronous. This is the case for the chaining router, for example.
- *Transaction*—If the service is transactional, its receiver and dispatcher will be synchronous no matter how the endpoints are configured.

If you look at figure 16.4, you'll notice that when both the receiver and the dispatcher are asynchronous, a thread is borrowed from each pool at each stage of the message processing. This configuration fully leverages the SEDA design and therefore must be preferred for services that handle heavy traffic or traffic subject to peaks. Of course, this configuration can't be used if a client is expecting a synchronous response from the service. Table 16.1 summarizes the pros and cons of this configuration model.

Table 16.1 Pro and con of the fully asynchronous mode

| Pro | Con |
|--|--|
| Leverages the SEDA model with three fully decoupled stages | Cannot return the component response to the caller |

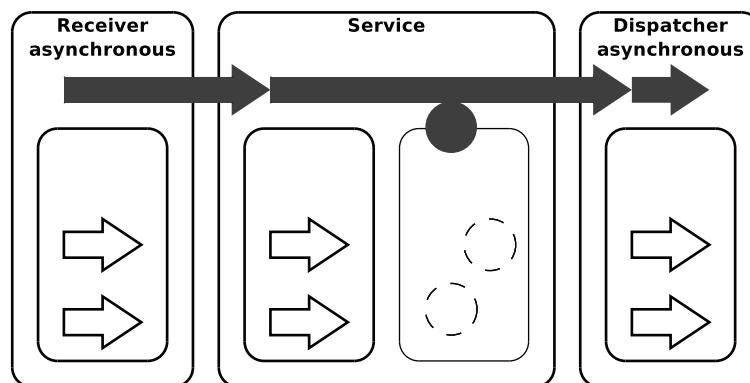


Figure 16.4 In fully asynchronous mode, a thread is borrowed from each pool at each stage of the message processing.

When a service needs to return a response to the caller, its receiver is configured to be synchronous. It is still possible to leverage the dispatcher thread pool by configuring it to be asynchronous. The threading model of this configuration is shown in figure 16.5. Notice how the receiver thread is used for calling the service component: this makes sense when you consider that it is the response of the method called on the component that is waited for synchronously. Table 16.2 presents the pros and cons of this approach.

Table 16.2 Pros and con of the synchronous-asynchronous mode

| Pros | Cons |
|---|---|
| Returns the component response to the caller. Dispatching is decoupled from receiving and servicing. | If a pool of components is used, it can be overwhelmed by the amount of receiver threads. |

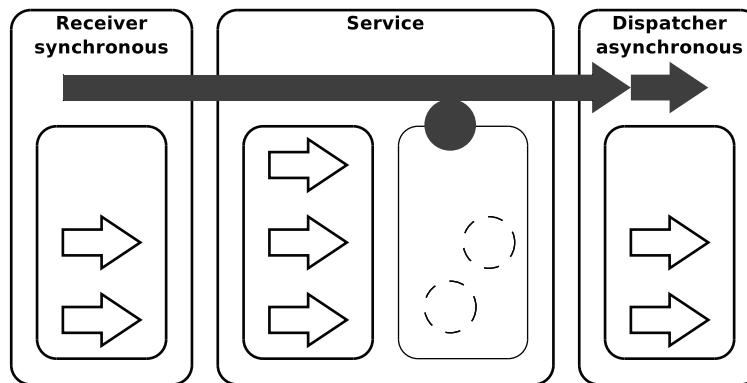


Figure 16.5 If only the receiver is synchronous, one thread from its pool will be used up to the component method invocation stage.

As discussed in the introduction, a transactional service will automatically run synchronously: the receiver thread will execute end-to-end and the dispatcher thread pool will not be used. This is obvious when you consider that, in Java, the transactional context is attached to the executing thread. Figure 16.6 demonstrates how the thread used in the receiver is piggybacked all along the processing path of the message event. The pros and cons of this configuration are listed in table 16.3.

Table 16.3 Pro and cons of the fully synchronous mode

| Pro | Cons |
|-----------------------------------|---|
| Supports transaction propagation. | All the load is handled by the connector's receiver threads. If a pool of components is used, it can be overwhelmed by the amount of receiver threads. |

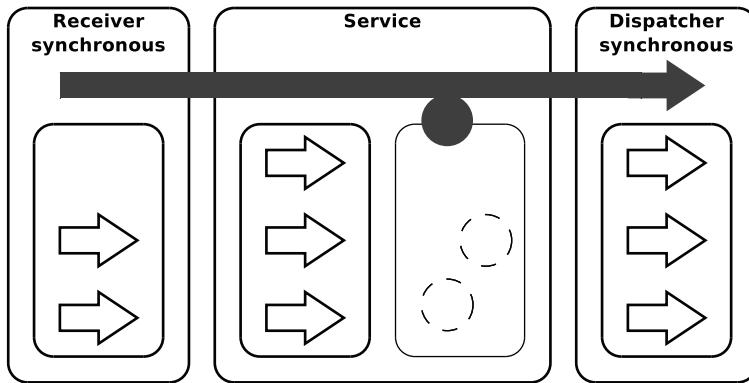


Figure 16.6 In fully synchronous mode, the receiver thread is used throughout all the stages of the message processing.

Like we said before, using a router such as the chaining router can make the dispatcher act synchronously. In that case, an asynchronously called service can end up dispatching synchronously, using a thread from the service pool for that end. This is illustrated in figure 16.7. This can also occur if the service component leverages the event context (see section 13.3) to perform a synchronous dispatch programmatically. Look at table 16.4 for a summary of the pros and cons of this threading model.

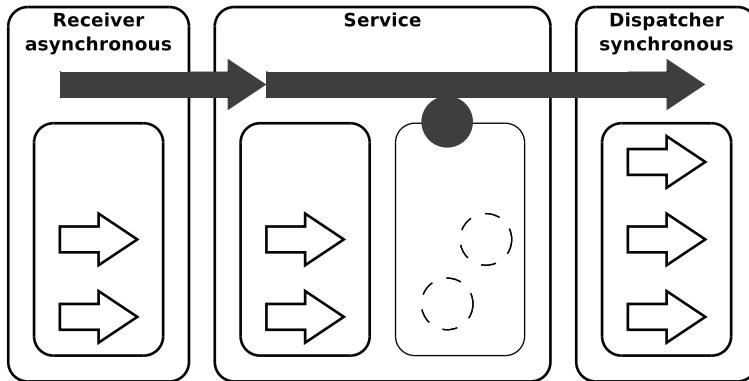


Figure 16.7 If only the dispatcher is synchronous, the receiver thread is used up to the service where a thread is taken from the pool and used throughout the dispatcher.

Table 16.4 Pros and cons of the asynchronous-synchronous mode

| Pros | Cons |
|--|--|
| <p>Allows synchronous routing in the dispatcher (like with a chaining router).</p> <p>Receiving is decoupled from dispatching and servicing.</p> | <p>Can't return the component response to the caller.</p> <p>The service thread pool size constrains the outbound dispatching load capacity.</p> |

BEST PRACTICE Use synchronous endpoints sparingly and only when it's justified.

As you've just learned, the usage of each of the three thread pools that can be involved in the processing of a message in a service depends heavily on the synchronicity context, whether it's inbound or outbound. We'll now look at how transports can also directly influence the thread pools' usage.

16.1.2 Transport peculiarities

Transports can influence the threading model mostly in the way they handle incoming messages. In this section, we'll detail some of these aspects for a few transports. This will give you a few hints about what to look for; we still recommend that you study the threading model of the transports you use in your projects if you decide you want to tune them.

The first peculiarity we'll look at is illustrated in figure 16.8. It's possible for a message event to be fully processed in service without using any thread from any of its existing pools. How come? This can happen when you use the VM transport because, since it's an in-memory protocol, Mule optimizes the event-processing flow for synchronous services by piggybacking the same thread across services. This allows, for example, a message to be transactionally processed across several services by the same thread.

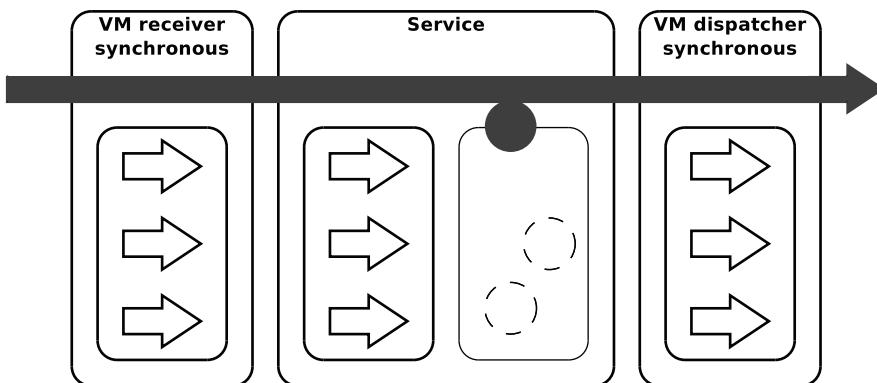


Figure 16.8 A fully synchronous service using the VM transport can piggyback an incoming thread for event processing.

Some transports, such as HTTP or File, support the notion of polling receivers. Keep in mind that if you activate this feature, the poller will permanently borrow a thread from the receiver pool of the connector, as illustrated in figure 16.9. Since you can configure several connectors for the same transport, it's a good idea to have a connector dedicated to polling and one or several other connectors for normal message processing.

Other transports don't rely on a thread pool directly handled by Mule, as shown in figure 16.10. This is the case for the servlet transport, where the receiver thread pool is fully handled by the web container that hosts your Mule instance. With the JMS transport, only the creation of receiver threads is handled by the JMS client infrastructure. In this case, Mule remains in control of these threads via its pool of JMS message listeners.²

In the introduction to this section, we mentioned the existence of a buffer for each thread pool. Similarly, certain transports natively support the notion of a request backlog that can accumulate requests when Mule isn't able to handle them immediately, as shown in figure 16.11. For example, the TCP and HTTP transports can handle this situation gracefully by stacking incoming requests in their specific backlog. Other transports, such as JMS or VM (with queuing activated), can also handle a pool-exhausted situation in a clean manner because they naturally support the notion of queues of messages.

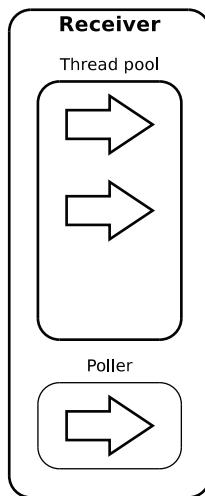


Figure 16.9 A polling receiver permanently borrows a thread from the transport's receiver pool.

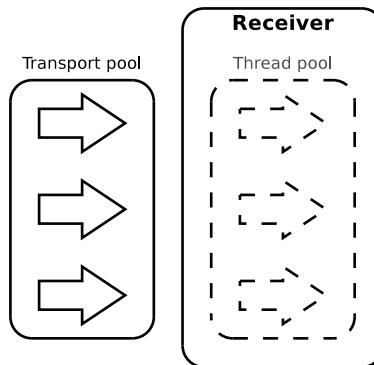


Figure 16.10 Some transports have their own receiver thread pool handled outside of Mule's infrastructure.

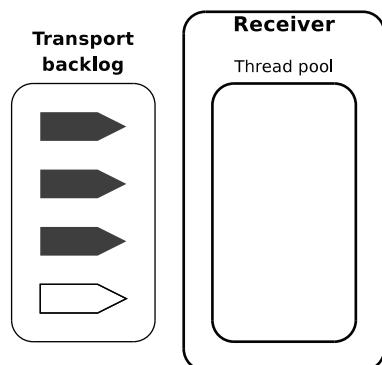


Figure 16.11 Some transports support the notion of a request backlog used when the receiver thread pool is exhausted.

² The pool of JMS dispatcher threads is still fully controlled by Mule.

As you can see, transports can influence the way thread pools are used. The fact of the matter is that transports matter as far as threading is concerned. Therefore, it's a good idea to spend some time understanding how the transports you use behave.

BEST PRACTICE Build a thorough understanding of the underlying protocols that you're using through Mule's transports.

Let's now look at the different options Mule gives you for configuring these thread pools.

16.1.3 Configuration options

Thread pools aren't configured directly but via the configuration of *threading profiles*. Threading profiles are organized in a hierarchy of configuration elements, whose scope varies from the most generic (Mule-level configuration) to the most specific (connector-level or service-level configuration). This hierarchy is represented in figure 16.12, where you can see the Mule-wide default threading profile at the top, then the profiles for the connector (receiver and dispatcher) on the left, and those for the service on the right.

In this hierarchy, a profile defined at a lower (more specific) level overrides one defined at a higher (more generic) level. For example, consider the following configuration fragment:

```
<configuration>
  <default-threading-profile
    maxBufferSize="100" maxThreadsActive="20"
    maxThreadsIdle="10" threadTTL="60000"
    poolExhaustedAction="ABORT" />
</configuration>
```

This fragment defines a global threading profile that sets all the thread pools (receiver, dispatcher, and service) to have by default a maximum number of active threads limited to 20 and a maximum number of idle threads limited to 10. It also defines that threads are deemed idle after a minute (60,000 milliseconds) of inactivity and that, in case of a pool exhaustion (which means that the 100 spots in the buffer are used), any new request will be aborted and an exception will be thrown.

But what if a critical service should never reject any request? We'd then override this Mule-wide default with a service-level thread pool configuration, as shown in the following excerpt:

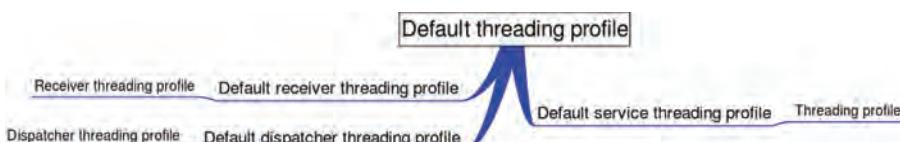


Figure 16.12 Thread pools are configured via a hierarchy of profiles, whose scope goes from the most generic to the most specific profile.

```

<service name="CriticalSection">
  ...
  <threading-profile
    maxBufferSize="100" maxThreadsActive="20"
    maxThreadsIdle="10" threadTTL="60000"
    poolExhaustedAction="RUN" />
</service>

```

With this setting, the service will never reject any incoming message even if its thread pool is exhausted. It will, in fact, piggyback the incoming thread to perform the work, as it can't hand it off to a thread from its own pool (this will tax the receiver's thread pool, potentially creating problems there, too). Note that we had to duplicate all the values defined in the global threading configuration, as there's no way to inherit individual setting values.

In addition to `RUN` and `ABORT`, the other supported exhausted actions are `WAIT`, which holds the incoming thread for a configurable amount of time until the pool accepts the event or a time-out occurs, and `DISCARD` and `DISCARD_OLDEST`, which silently drop the incoming event or the oldest one in the buffer, respectively.

TIP *Outside pool* If you use the servlet transport, you may be wondering where to configure the receiver threading profile, much like when you arrive in an unknown hotel, you know that there must be a pool but may have no clue where it is. In the case of the servlet transport, the answer is easy: as we said in the previous section, the receiver thread pool is handled by the servlet container that hosts your Mule instance. Therefore, you must configure it by using its specific configuration.

As an example, if you deploy Mule in an embedded web application on Tomcat, you'll need to configure the `maxThreads` and `backlog` parameters of the HTTP or AJP connectors.

You're now certainly burning with the following question: how do I size all these different thread pools? MuleSource provides a comprehensive methodology for calculating these sizes³ based on four main factors: expected number of concurrent user requests, desired processing time, target response time, and acceptable time-out time. Before you follow that path, we'd like to draw your attention to the fact that for many deployments, the default values provided by Mule are all that's needed. Unless you have specific requirements in one or several of these four factors, you'll most of the time be better off leaving the default values. We encourage you to load-test your configuration early on in your project, and decide to tweak the thread pools only if you have evidence that you need it.

NOTE An example of JMS transport tuning Coming back to our provisioning system at Clood, Inc., we have to deal with peaks of activity coming from batch processes happening in the existing systems of Billow, Inc. To

³ See <http://www.mulesource.org/display/MULE2USER/Tuning+Performance#TuningPerformance-CalculatingThreads>.

abide by our SLA, we have to ensure that we can process a batch of 10,000 messages within 30 minutes. Since these messages are sent over JMS and we use Mule's standard receiver (i.e. `org.mule.transport.jms.MultiConsumerJmsMessageReceiver`, not one of the polling message receivers that are also available) to consume them transactionally, we know that we don't need to configure a Mule thread pool. The working threads will directly come from the JMS provider and will be held all along the message processing path in Mule in order to maintain the transactional context. We need only configure the number of JMS concurrent consumers. A load test allowed us to measure that we can sustain an overall message process time of 4 seconds. A simple computation, similar to the ones explained in Mule's thread pool sizing methodology, gives us a minimum number of concurrent consumers of $10000 \times 4 / (30 \times 60) = 22.22$. Margin being one of the secrets of engineering, we've opted for 25 concurrent consumers and, since then, haven't failed our SLA.

Here are a few complementary tips:

- *Use separate thread pools for administrative channels*—For example, if you use TCP to remotely connect to Mule via a remote dispatcher agent (see section 13.1.2), use a specific connector for this TCP endpoint so it'll have dedicated thread pools. That way, in case the TCP transport gets overwhelmed with messages, you'll still be in the position to connect to Mule.
- *Don't forget the component pool*—If you use component pooling, the object pool size must be commensurate to the service thread pool size. It's easy to define a global default service thread pool size and forget to size a component pool accordingly.
- *Waiting is your worst enemy*—The best way to kill the scalability of an application is to mobilize threads in long-waiting cycles. The same applies with Mule: don't wait forever, and avoid waiting at all if it's acceptable business-wise to reject requests that can't be processed.

Mule offers total control over the thread pools it uses across the board for handling message events. You're now better equipped to understand the role played by these pools, when they come into play, and what configuration factors you can use to tune them to your needs.

When a thread is taken out of a pool, your main goal is to get this thread back in the pool as fast as possible. How can we achieve this? By shooting for the best possible performances at every stage of the message event processing. We'll now consider this aspect of tuning.

16.2 Increasing performance

Whereas threading profiles define the overall capacity of your Mule instance in term of scaling and capacity, the performance of each moving part involved in processing each request will also impact the global throughput of your application. If the time needed to process each request is longer than the speed at which these requests come

in or if this time degrades while the load increases, you can end up in a position where no matter how many threads and how big the buffer you use, your thread pools will end up exhausted.

Therefore, fine-grained performance matters. On the other hand, we all know Donald Knuth's words of caution:

We should forget about small efficiencies, say about 97 percent of the time: premature optimization is the root of all evil.⁴

So should performance optimization *always* be avoided? Of course not. Premature optimization is the issue. This encourages us to follow a pragmatic approach to the question of increasing performances: *build, measure, and correct*.

BEST PRACTICE Don't tune randomly; use a systematic approach to tuning.

Building has been the main focus of this book, so we consider this subject (hopefully) covered. In this section, we'll first talk about gathering metrics to pinpoint actual points of performance pain. Then we'll give you some advice on what you can act on based on these measures. This advice will be generic enough to guide you in the build phase, not in achieving premature optimization, but in making some choices that will reduce your exposure to performance issues.

Let's start by looking at measuring performance issues with a profiler.

16.2.1 Profiler-based investigation

The most convenient way to locate places in an application that are good candidates for optimization (aka *hot spots*) is to use a profiler. As a Java-based application that can run on the most recent JVMs, and because its source code is available, Mule is all but a black box when it comes to profiling. To make things even easier, you can download a free Profiler Pack from MuleSource.org that contains the libraries required to profile a Mule instance with YourKit (<http://yourkit.com>). You still need to own a valid license of YourKit in order to analyze the results of a profiling session.

After installing the pack, the way you activate the profiler depends on how you deploy and start Mule (see section 7.1). With a standalone deployment, adding the `-profile` parameter to the startup command does the trick. If you deploy Mule in a JEE container or bootstrap it from an IDE, you'll have to refer to YourKit's integration guidelines to activate the profiler.

To make the most of a profiling session, you'll have to exercise Mule in a way that simulates its usage for the considered configuration. This will allow hot spots to be detected easily. For this, you can use one of the test tools we mentioned in chapter 12 or create your own ad hoc activity generator, if you have very specific or trivial needs. Figure 16.13 shows YourKit's memory dashboard while profiling the LoanBroker sample application that ships with Mule. This demonstration application comes with a

⁴ Knuth, Donald. "Structured Programming with go to Statements." ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268.

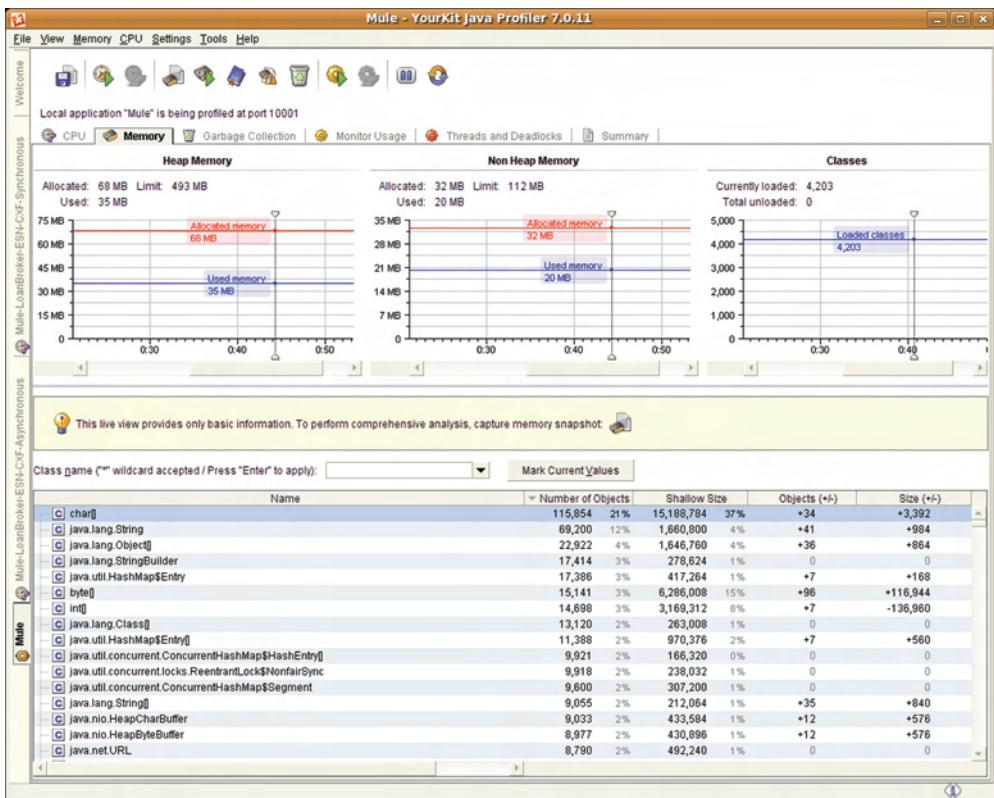


Figure 16.13 Profiling Mule’s memory with YourKit while messages are being processed

client that can generate a hundred fake requests: we used it as a convenient ad hoc load injector able to exercise Mule in a way that’s realistic for the configuration under test.

What can you expect from such a profiling session? Covering the subject of Java application profiling is beyond the scope of this book, but here are a few key findings you can expect:

- *Hot spots*—Methods that are excessively called or unexpectedly slow.
- *Noncollectable objects*—These lead to memory leaks.
- *Monitor locks*—An excess of these can create contentions between threads.
- *Deadlocks*—These can take down a complete application.
- *Excessive objects or threads creation*—This can lead to unexpected memory exhaustion.

Once you’ve identified areas of improvement, it’ll become possible to take corrective actions. As far as profiling is concerned, a good strategy is to perform differential and incremental profiling sessions. This consists in first capturing a snapshot of a profiling session before you make any change to your configuration or code. This snapshot will

act as a reference, to which you will compare snapshots taken after each change. Whenever the behavior of your application has improved, you'll use the snapshot of the successful session as the new reference. It's better to do one change at a time and reprofile after each change, or else you'll have a hard time telling what was the actual cause for an improvement or a degradation.

The LoanBroker sample application we've already profiled comes with several configuration flavors. We've compared the profiles of routing-related method invocations of this application running in asynchronous and synchronous modes. The differences, which are shown in figure 16.14, confirm the impact of the configuration change: the asynchronous version depends more on inbound routing and filtering, whereas the synchronous one leverages the response routers for quotes aggregation.

In chapter 11, we introduced the notion of agents that are available in the management module of Mule. This module also contains an agent specific for profiling that

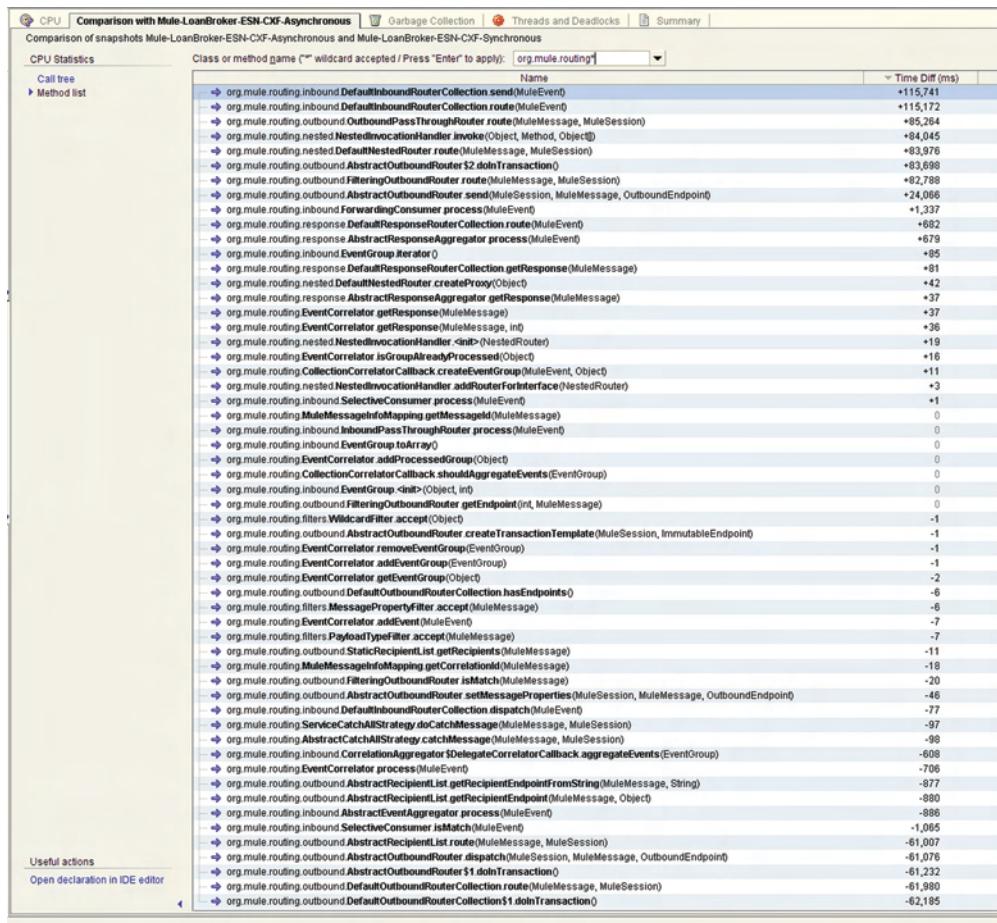


Figure 16.14 Comparing two profiling session snapshots helps determine the impact of configuration changes.

you can use in conjunction with the YourKit profiler pack. This agent doesn't need any more configuration than this single configuration element:

```
<management:yourkit-profiler />
```

This agent registers an MBean that gives you complete control of the YourKit profiler. This allows you to perform JMX actions that you'd usually perform from the YourKit control panel, such as starting and stopping monitor profiling. Figure 16.15 shows the operations exposed by this agent as seen from the standard JConsole.

TIP Many operations of the YourKit profiler MBean take magic numbers as parameters. You can find the details of these numbers and to what operations they apply in the JavaDoc of the `com.yourkit.api.ProfilingModes` class, available online at <http://www.yourkit.com/docs/80/api/com/yourkit/api/ProfilingModes.html>.

This agent comes in handy when network or firewall restrictions prevent you from remotely connecting the YourKit client to Mule, as you'll be able to use it over HTTP thanks to the MX4J HTTP agent (see section 11.1.3).

NOTE The Profiler Pack and the management agent are packaged for a particular version of YourKit. Be sure to check that they're compatible with the version you own. If you use a different version and don't want to upgrade or downgrade to the version supported by Mule's extensions, you'll have to use the libraries that shipped with your version of YourKit in lieu of the ones distributed by MuleSource.

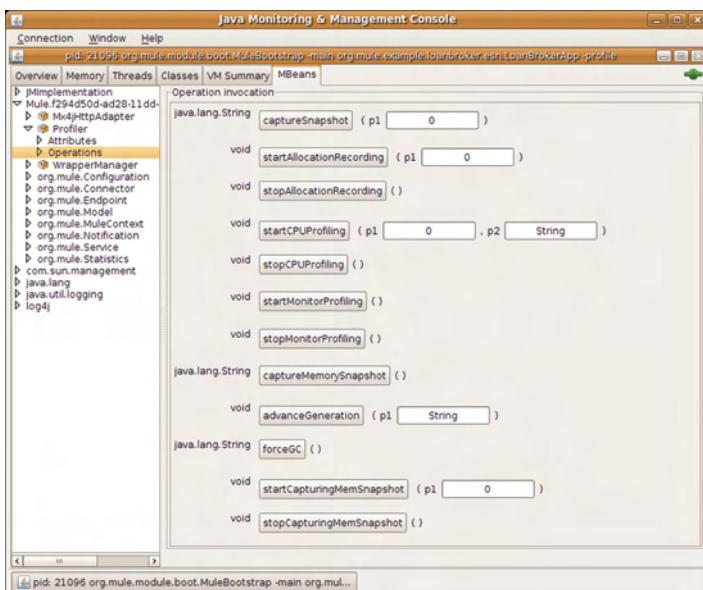


Figure 16.15 Mule's management module offers a YourKit agent that registers an MBean which controls the profiler.

Using a profiler is the best way to identify performance-challenged pieces of code. But, whether you use a profiler or not, there's some general advice that you can apply to your Mule projects. We'll go through that now.

16.2.2 Performance advice

Whether you've used a profiler before or not, this section will give you some advice about coding and configuring your Mule applications for better performance. You'll see that some of this advice is very generic while other parts are Mule-specific.

FOLLOW GOOD MIDDLEWARE CODING PRACTICES

Mule-based applications don't escape the rules that apply to middleware development. This may sound obvious, but it's not. It's all too easy to consider a Mule project as being different from, say, a standard JEE project because way less code is involved (or it's all in scripts). The reality is that the same best practices apply. Let's name a few:

- Use appropriate algorithms and data structures.
- Write sound concurrent code in your components or transformers (thread safety, concurrent collections, no excessive synchronization, and so on).
- Consider caching (for example, turn back to section 13.4.2, where Cloud used caching to get rid of a performance hot spot in their Md5FileHasher component).
- Avoid generating useless garbage (favor singleton-scoped beans instead of prototypes).

REDUCE BUSYWORK

In section 11.2, we talked about the possibility of leveraging log files and notifications to follow what's happening in a Mule instance. Be aware that inefficient logging configuration can severely harm performance. Reduce verbosity in production and activate only the relevant appenders (for example, if you only need FILE, there's no need to configure CONSOLE). In the same vein, don't go overboard with your usage of notifications. If you activate all the possible families of notifications (see section 13.4.2), a single message can fire multiple notifications while it's being processed in Mule, potentially flooding your message listening infrastructure. You certainly don't need to expose your application to a potential internal self-denial of service.

USE EFFICIENT MESSAGE ROUTING

Message routing is an important part of what keeps a Mule instance busy; therefore, it doesn't hurt to consider performance when configuring your routers and filters. For example, avoid delivering messages to a component if you know it'll ignore it: use a forwarding consumer (see section 4.3.2) to bypass the component altogether. Doing so will also save you the time spent in the component interceptor stack and the entry point resolvers.

BEST PRACTICE Investigate high-performance routers such as SXC or Smooks if you intend to perform content-based routing on large messages.

CARRY LIGHTER PAYLOADS

Carrying byte-heavy payloads creates a burden both in memory usage and in processing time. Mule offers different strategies to alleviate this problem. These strategies mostly depend on the transport you use. Here's an incomplete list of possible options in this matter:

- The File transport can receive `java.io.File` objects as the message payload instead of the whole file content, allowing you to carry a light object until the real processing needs to happen and the actual file content needs to be read.
- Some transports can receive incoming requests content as streams instead of arrays of bytes, either by explicitly configuring them for streaming or by virtue of the incoming request. This is a powerful option for dealing with huge payloads, but you'll have to be careful with the synchronicity of the different parts involved in the request processing (see section 16.1.1), as you can end up with a closed input stream being dispatched if the receiver wasn't synchronously bound to the termination of the processing phase.
- You can opt to return streams from your components that produce heavy payloads. Many transports can accept streams and serialize them to bytes at the latest possible stage of the message-dispatching process. In the worst case, you can always leverage an existing transformer to deal with the stream serialization (see section 5.3.1).
- Consider deactivating message level caches: see section 13.3.1 for more on this subject.

TUNE TRANSPORTS

As we discussed in section 16.1.2, transports have their own characteristics that can influence threading, and consequently performance. The general advice we can give is you should know the transports you use and how they behave. You should look for time-out parameters, buffer sizes, and delivery optimization parameters (such as *keep alive* and *send no delay* for TCP and HTTP, chunking for HTTP, or *DUPS_OK_ACKNOWLEDGE* for nontransactional JMS).

BEST PRACTICE For HTTP inbound endpoints, you should prefer the Jetty transport in standalone deployments and the servlet transport in embedded deployments, over the default HTTP transport.

TUNE MULE'S JVM

Finally, Mule being a Java application, tuning the JVM on which it runs can also contribute to increasing performance. Mule's memory footprint is influenced by parameters such as the number of threads running or the size of the payloads you carry as bytes (as opposed to streams). Right-sizing Mule's JVM, tuning its garbage collector (like by activating the brand new G1 collector) or some other advanced parameters, can effectively be achieved by running load tests and long-running tests that simulate the expected traffic (sustained and peak).

Performance tuning is the kingdom of YMMV:⁵ there's no one-size-fits-all solution for such a domain. In this section we've given you some hints on how to track down performance bottlenecks with a profiler and some common advice you can follow to remedy these issues.

16.3 Summary

In this chapter, we investigated the notions of thread pools and performance optimization. We've talked about how synchronicity deeply affects the way threading occurs in Mule. We left you with a lot of different options to tune the different thread pools to your needs. We also presented a pragmatic approach to performance optimization, with the help of a profiler, and have given you a handful of general tips for better utilizing the available resources of your system (memory and CPU).

We're now done with our journey into Mule, but we believe that, for you, this is just the beginning. From the first chapters, where you learned what Mule can achieve with just a few lines of configuration, to this final chapter where you learned to tune the inner parts of the system, you've gathered both breadth and depth of knowledge that'll allow you to succeed in your Mule-powered projects. Whether you use Mule as a lightweight messaging framework or as a highly distributed object broker, we wish you happy trails!

⁵ Your mileage may vary.

The expression evaluation framework

Mule offers a rich expression evaluation framework that allows you to hook advanced logic into different parts of your configuration. In chapters 4 and 5, we mentioned the possibility of using evaluators. In this appendix, we'll review the existing evaluators and present different examples, side by side, to help you realize how versatile and powerful they are.

In short, an expression evaluator evaluates an expression on an object. The nature of the expression and the type of object accepted actually depends on the type of the evaluator. Most of the time, the object consists of the instance of `MuleMessage` that's currently under processing.

Table A.1 presents an overview of the evaluators available in Mule at the time of this writing.¹ Note how each evaluator has an ID: this is the ID that you'll need to use in your configuration to tell Mule what evaluator to use.

Table A.1 Available expression evaluators in Mule 2.2

| Evaluator ID | Description |
|------------------|--|
| attachment | Gets an attachment payload by its name from the message. |
| attachments | Builds a map of attachment names and payloads from a comma-separated list of attachment names. |
| attachments-list | Builds a list of attachment payloads from a comma-separated list of attachment names. |

¹ Refer to the JavaDoc of your installed Mule for the most accurate evaluators list and description.

Table A.1 Available expression evaluators in Mule 2.2 (continued)

| Evaluator ID | Description |
|--------------|---|
| bean | Evaluates a bean expression, such as <code>object.propertyA</code> , <code>propertyB</code> , on the message payload. Behind the scenes, this evaluator uses the <code>jxpath</code> evaluator. |
| endpoint | Gets a property of a global endpoint. The expression is of the form <code>endpointName.propertyName</code> . The only supported property is <code>address</code> . |
| function | Executes predefined functions, such as <code>now</code> , <code>date</code> , <code>datestamp</code> , <code>datestamp:dd-MM-yyyy</code> , <code>uuid</code> , <code>hostname</code> , <code>ip</code> , <code>count</code> , <code>payloadClass</code> , <code>shortPayloadClass</code> . |
| groovy | Runs a specified Groovy script, with the following properties bound: <code>log</code> , <code>result</code> , <code>muleContext</code> , <code>payload</code> , and <code>src</code> (<code>src</code> is an alias of <code>payload</code>). If the expression is run against a <code>MuleMessage</code> , the following properties are also bound: <code>message</code> and all the message properties (headers). If the expression is run against <code>MuleEvent</code> , the following properties are also bound: <code>originalPayload</code> , <code>eventContext</code> , <code>id</code> , and <code>service</code> . |
| header | Gets the value of the message property (aka header) whose name is passed to the evaluator. |
| headers | Builds a map of message property names and values from a comma-separated list of property names. |
| headers-list | Builds a list of message property values from a comma-separated list of property names. |
| jxpath | Evaluates the specified expression with JXPath. |
| map-payload | Gets the value that corresponds to specified key out of the message payload. Works only with payloads that are instances of <code>java.util.Map</code> . |
| message | If no expression is given to this evaluator, it simply returns the message as is. Otherwise, the expression must be one of the following, which correspond to methods invoked on the <code>MuleMessage</code> object: <code>id</code> , <code>correlationId</code> , <code>correlationGroupSize</code> , <code>correlationSequence</code> , <code>replyTo</code> , <code>payload</code> , <code>encoding</code> , and <code>exception</code> . |
| mule | Evaluates the expression against the Mule event context. The supported expressions are <code>serviceName</code> , <code>modelName</code> , <code>inboundEndpoint</code> , <code>serverId</code> , <code>clusterId</code> , <code>domainId</code> , <code>workingDir</code> , and <code>homeDir</code> . |
| ognl | Evaluates an OGNL (Object-Graph Navigation Language) expression against the current message payload. |
| payload | If no expression is given to this evaluator (using <code>payload:</code>), it simply returns the payload as is. If an expression is defined, it must be the class FQN into which the payload must be transformed. This auto-transformation relies on the mechanism described in section 13.3.1. For byte arrays, use this fake FQN: <code>byte[]</code> . |

Table A.1 Available expression evaluators in Mule 2.2 (continued)

| Evaluator ID | Description |
|--------------|---|
| string | A string expression can be composed of text and optionally other nested expressions, such as #[string:ACK] or #[string:ID#[function:uuid]]. It's typically the evaluator to use when multiple evaluators are present within one composite expression. |
| xpath | Evaluates the specified expression with XPath on a DOM instance. |
| xpath-node | Same as xpath but returns the actual DOM node object instead of its content. |

A.1 Standard evaluators

Let's review some examples where expression evaluators have been used in different contexts.

A.1.1 In filters

Using expressions in message filters has been discussed in section 4.2.3. Listing A.1 demonstrates a filter that uses a tiny Groovy script to sort messages by size. Messages with more than one kilobyte of payload are dispatched to a different outbound endpoint than the smaller ones. The capacity to use Groovy and, from there, reach any helper class that's in the classpath of your application allows you to deploy advanced routing logic with no need for custom classes.

Listing A.1 Using a Groovy expression evaluator to filter messages

```
<service name="PayloadSizeFiltering">
    <inbound>
        <vm:inbound-endpoint path="PayloadSizeFiltering.IN" />
    </inbound>

    <outbound>
        <filtering-router>
            <outbound-endpoint ref="TargetChannel" />
            <expression-filter evaluator="groovy"
                expression="message.payloadAsBytes.length>1024" />
        </filtering-router>
        <forwarding-catch-all-strategy>
            <outbound-endpoint ref="LoggerChannel" />
        </forwarding-catch-all-strategy>
    </outbound>
</service>
```

NOTE On addition to those listed in table A.1, there are a few extra evaluators that can only be used in expression filters. They're payload-type, regex, and wildcard, which filter on the message payload, and exception-type, which filters on the exception payload of the message.

A.1.2 In transformers

In section 5.3.4, we demonstrated the usage of expression transformers. The obvious usage of a transformer is to alter the payload or properties of a message while it's processed. With a little imagination, we can do way more than this! Listing A.2 shows a bridge service that leverages a response transformer to provide a synchronous response to the caller, while dispatching the original message asynchronously to the outbound router. The synchronous acknowledgement returned to the caller is the correlation ID of the message it has sent, allowing it to ensure its message has been received correctly. We've achieved, thanks to the message expression evaluator, a disconnect between what's replied to the caller and what's dispatched without resorting to using the API, as we did in section A.2.

Listing A.2 An expression evaluator can return a simple request acknowledgement.

```
<service name="AckingAsyncDispatcher">
<inbound>
    <inbound-endpoint ref="AckingAsyncDispatcherChannel"
        synchronous="true">

        <response-transformers>
            <expression-transformer>
                <return-argument evaluator="message"
                    expression="correlationId" />
            </expression-transformer>
        </response-transformers>
    </inbound-endpoint>
</inbound>

<outbound>
    <pass-through-router>
        <outbound-endpoint ref="TargetChannel" synchronous="false" />
    </pass-through-router>
</outbound>
</service>
```

A.1.3 In endpoint URIs

In section 13.1.2 we looked at the client code used to call a Mule service that looks up stock information over the Internet. Listing A.3 is the configuration of this service. Notice how we use the payload expression evaluator: unlike the previous example where the evaluator ID and expressions were configured in two different attributes, we have to multiplex all this information in the URI. This is achieved by concatenating the evaluator ID and the expression itself, separating them with a colon. If you wonder what the %23 character stands for, it's the # sign escaped to be parsed correctly in the context of an HTTP endpoint. The normal expression placeholder syntax is #[evaluatorID:expression].

Listing A.3 Expressions in endpoint URIs can be resolved at runtime by evaluators.

```
<service name="TickerLookupService">
  <inbound>
    <inbound-endpoint ref="TickerLookupChannel">
      <response-transformers>
        <byte-array-to-string-transformer />
      </response-transformers>
    </inbound-endpoint>
  </inbound>

  <outbound>
    <pass-through-router>
      <http:outbound-endpoint synchronous="true"
        address="http://finance.google.com/finance/historical?
          ↪ q=%23[payload:java.lang.String]&histperiod=weekly
          ↪ &output=csv" />
    </pass-through-router>
  </outbound>
</service>
```

A.1.4 In custom code

Expression evaluators are also available from your own code, thanks to the `org.mule.api.expression.ExpressionManager` instance that's accessible from the Mule context (see section 13.2). The following code shows the logic inside a component that parses a configured String expression against the messages it receives:

```
public Object onCall(MuleEventContext eventContext) throws Exception {
    ExpressionManager expressionManager = eventContext
        .getMuleContext().getExpressionManager();

    return expressionManager.parse(expression, eventContext.getMessage(),
        true);
}
```

Here's the configuration for this component that makes it return a String containing the correlation ID of the incoming message and the date stamp when it was received, under a specific format:

```
<component>
  <singleton-object class="com.muleinaction.component.ExpressionParser">
    <property key="expression"
      value="#[message:correlationId]@#[function:datestamp:yyyy-dd-MM]" />
  </singleton-object>
</component>
```

NOTE You can easily perform what this custom component does with a transformer and the string expression evaluator:

```
<expression-transformer>
  <return-argument evaluator="string"
    expression="#[message:correlationId]@#[function:datestamp:
      ↪ yyyy-dd-MM]" />
</expression-transformer>
```

A.2 Custom evaluators

As always, should your needs not be covered by Mule's standard implementations, you have the possibility to roll out your own expression evaluator. If this is the case, you need to create an object that implements `org.mule.util.expression.ExpressionEvaluator`. If you use Spring to instantiate your custom evaluator, Mule will automatically discover it at startup time. You can also register it programmatically using the `register` method on the `org.mule.api.expression.ExpressionManager` instance that's accessible from the Mule context (see section 13.2).

Using a custom evaluator is transparent in string expressions: you simply need to refer to it by its name. For example, using a custom evaluator named `translator` in an expression is done the usual way: `#*[translator:an_expression]`.

In XML elements, you have to specify both that you use a custom evaluator and its name:

```
<return-argument evaluator="custom"
                  customEvaluator="translator"
                  expression="an_expression" />
```

Let's now look at a few samples that show you how the expression evaluators can allow you to achieve complex behaviors without coding your own classes.

The Mule community

B

This appendix is a short introduction to the Mule community. As suggested by figure B.1, this community is mainly organized around the MuleSource.org web site, with some activity on the MuleForge.org and MuleSource.com sites. The latter is the commercial and corporate site of MuleSource, Inc., where you can find Enterprise versions of the products and professional services. We won't discuss that site in this appendix.

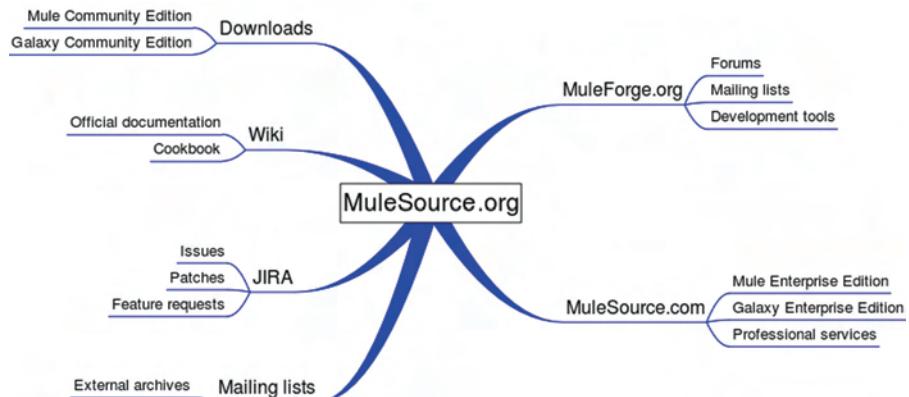


Figure B.1 The Mule community is mainly organized around the MuleSource.org web site.

B.1 MuleSource.org

As a Mule user, MuleSource.org will be your primary destination. Presented as the documentation and resource center of Mule, this is where you'll find the

downloadable versions of the Community Editions of Mule and Galaxy. You'll also find the complete user guides for Mule 1 and Mule 2. If you're looking for the latest details of a particular transport or an in-depth review of an advanced concept in Mule, this is where you're most likely to find it.

Another valuable resource of this site is the cookbook. Mostly built by user contributions, the cookbook contains technical solutions for common problems.

B.2 JIRA

JIRA is an issue tracker used by MuleSource for managing bugs, feature requests, and patch submissions. This is where you'll look first if you have an issue: a look at the roadmap or a search on open issues will give you a fair idea of what's cooking and what's burning.

By creating an account on this system, you'll be allowed to watch issues and track their progress but also vote for the ones you consider important. As a community member, we can only encourage you to exercise your voting power and cast your ballot for issues or feature requests you'd like to see solved or implemented first.

B.3 Mailing lists

Mule's mailing lists are notable for the non-trollish answers you can get. Out of respect for the persons who take time to respond to you, always search for an answer or an already-reported bug before posting in the list. Since all MuleSource web sites and related mail archives are indexed by the major Internet search engines, you don't need to manually search all these different places. Just use your favorite search engine and you'll get pretty decent results. If this isn't the case, then consider a targeted search in JIRA with selective criteria such as module or ticket state.

B.4 MuleForge.org

MuleForge.org is a complete development platform that MuleSource has opened to the community to support the creation of new transports, components, or any other Mule-related artifacts. As such, MuleForge is the workplace of the Mule community. It's also its factory outlet, as this is where you'll go to "shop" for a specific transport that you can't find in the standard transports supported by MuleSource.

It's also a great place to share your own stuff: you'll find there all the development tools of the moment such as Subversion, Confluence, JIRA, Bamboo, and of course, Maven, with Xircles as the one platform to bind them all in a consistent dashboard. On top of that, mailing lists and forums are created for each project, to encourage discussion and feedback. MuleForge allows you to develop, build, distribute, document, and support your Mule-related creations in an efficient and professional way.

Why would you share your own creations? Consider this selfish reason: to have a community of users debugging it for you! More seriously, as developers, we benefit a lot from open source in general and it's fair to give back whenever possible. Open

source communities thrive on public contributions, but have to maintain a gated access to committing on the core product. This is why MuleForge is attractive: it allows open contributions and treats them with all due respect. It's sure a good place to host your code and expose it to the world.

index

A

AbstractEntryPointResolver 153
AbstractMessageAwareTrans-
former 135, 137
AbstractPolicyTemplate 208
AbstractTransformer 132, 137
Acegi 221
ACID 234
ActiveMQ 209
 RetryPolicyTemplate 210
 use of the VM broker 290
adaptive retry policy 210
AdaptiveRetryPolicyTemplate-
Wrapper 211
Ant notation 30
AOL Instant Messenger 76
Apache ActiveMQ 64
 external instance 66
Apache Ant 274
Apache AXIS 148
Apache Chainsaw 197, 202,
 213, 215
 filtering 215
Apache Commons Lang,
 StringUtils 336
Apache Commons Logging 213
Apache CXF 55
 contract-first development 59
 wsdl2java 59
Apache Directory Server 223
Apache JMeter 294
 adjusting the Ramp-Up
 period 296
 custom samplers 298
 HTTP 294

JDBC 294
JMS 294
LDAP 294
load testing 297
setting up a JMS test 296
specifying a Test Plan 295
stopping a test 296
stopping a thread 295
Apache JXPath 87
Apache Maven 274
 archetypes 282
 artifactId 277
 artifacts 278
 automatic resolution of
 dependency graph 277
 building EARs 277
 building JARs 277
 building WARs 277
 dependency
 management 277
 groupId 277
 hierarchy of JAR files 275
JUnit 277
location of central
 repository 281
M2_HOME 275
 bin subdirectory 275
main source tree 275
Mule artifacts
 mule-transport-
 archetype:create 282
 TODO pointers 282
mule-core artifact 280
negating the need for custom
 build infrastructure 274
pom.xml 276

repository 281
specifying artifact id 275
specifying group Id 275
test source tree 275
use of dependency tree 280
use of versioning 277
API 299
array-entry-point-resolver 153
aspect-oriented programming
 (AOP). *See* interceptor
async-reply. *See* router
AuditComponent 262
auditing. *See* monitoring
automated integration
 testing 294
automation 274
auto-transformer 121, 137, 313
Axis transport, and use of with
 BPEL engines 354

B

bean, expression evaluator 121,
 156
binding 161
BitTorrent 70
boolean operations 89
BPM transport
 aborting a process 356
 advancing an existing
 process 355
 exceptions 356
 externally advancing a
 process 355
 interacting with from
 Mule 354

BPM transport (*continued*)
MULE_BPM_PROCESS_ID
 property 355
 process definition
 location 353
 SendMuleEvent 352
 SendMuleEventAndContinue 352
 starting a process 355
 supplied actions 349
 synchronicity 351
 use of the org.mule.trans-
 port.bpm.BPMS
 interface 348
 XPath evaluation 352
 bridge component 15, 142
 explicit 143
 implicit 23, 143
 bridge-component 143
 Business Process Execution
 Language (BPEL) 354
 business process management
 (BPM) 101, 343
 byte transformers 115
 byte-array-to-hex-string-
 transformer 117
 byte-array-to-object-
 transformer 115
 byte-array-to-serializable-
 transformer 117
 byte-array-to-string-
 transformer 13, 117

C

Cacti. *See* monitoring
 Callable 151, 308, 312
 callable-entry-point-resolver 153
 canonical data model 162
 logging 262
 catch-all strategy
 custom 84
 error handling 84
 logging 84
 Central Authentication Service
 (CAS) 218
 chaining-router 56
 Chainsaw 264
 console 265
 chainsaw-notifications 264
 client/server 179
 Clood, Inc. 40
 account provisioning 343, 360
 accounting data 54
 analysis of monitoring data 92

async-reply routing 105
 auditing 262, 264
 backup provider 61–62
 BPM 343
 business intelligence tool
 integration 47
 caching interceptor 321
 canonical data models 333
 custom logging levels 213
 custom Velocity
 transformer 131
 data coupling 239
 data warehousing 357
 duplicate order messages 95
 email generation 131, 146,
 155, 315
 exception-based routing 204
 FTP 72
 HTML dashboard 268
 IDE templates 284
 IMAP 45
 IMAP server 52
 initialization interceptor 324
 JBossTS 243
 JMS backup-report topic 65
 JMS provider 95
 LDAP 278
 managing their message-
 enricher project 275
 MD5 file hashing 158, 160,
 301, 315, 322
 message-enricher 278
 monitoring data 235
 monitoring database 73, 83
 moving data from an HTTP
 inbound-endpoint to a JMS
 topic 219
 multicasting-router 99
 OpenLDAP 223
 order processing 333
 order provisioning 95, 371
 order status via synchronous
 JMS 68
 partner reporting 206
 performance bottlenecks 294
 periodic data monitoring 267
 presentation 38
 publication application.
 See publication application
 publishing of analytics
 data 243
 Quartz 343
 real-time order statistics 96
 relational databases 235
 routing 83
 security concerns 219
 selective-consumer router 86
 sending IMAP messages to a
 database 45
 transactionally receiving bill-
 ing data 244
 transactions with outbound
 endpoints 239
 use of error channels 98
 use of the HTTP transport 61
 use of the static-recipient-list-
 router 98
 use of transactions 235
 using a RetryPolicy 209
 VM farms 104
 VM transport 79
 wire-tap-router 96
 XA transaction overhead 357
 XA transactions 249
 XML payload 87
 clustering 186
 session state 185
com.clood.monitoring.URL-
 AlertComponent 46
com.clood.Order 87
 command line 273
 profile parameter 373
 comma-separated value
 (CSV) 61, 75, 333
Common Retry Policies 210
 ForeverRetryPolicyTemplate
 211
 competing consumers 70
 component 15, 58
 binding 160, 331
 method 161
 bridge 23
 See also bridge component
 bypassing with a router 91
 configuring 156
 type mismatch 156
 controlling message process-
 ing. *See* message processing
 creation cost 158
 custom 151, 328
 echo. *See* echo component
 entry point resolution 93
 implementing Callable 151
 in service 140
 include-entry-point 69
 initialize. *See* lifecycle
 injecting Mule objects 159,
 269
 injecting Mule service 160
 instantiation policy 151
 interceptor 319
 log. *See* log component

component (*continued*)
 message enrichment 91, 237
 method called. *See* entry point
 method-entry-point-resolver 69
 Mule objects 156
 prototype 156
 singleton 156
 null. *See* null component
 pooled 158
 reflection message
 builder 145
 response 140
 REST service. *See* SOAP
 role 141
 SOAP wrapper. *See* SOAP
 Spring beans 156
 re-using 151, 157, 174
 spring-object 238
 stateless 157
 thread-safety 157–158
 Component (interface) 152
 compression transformers 117
 compressing 117
 uncompressing 118
 configuration
 builder 22, 44
 custom elements 27
 data types 30
 default values 30
 enumerated values 30
 environment dependent 34
 families of elements 26
 Groovy 37
 modularity 34
 advantages 34
 importing files 36
 independent files 34
 inheritance 35
 mixed format 36
 names and references 31
 properties 30
 reuse 34
 scripted 22
 specific elements 26
 Spring elements 28
 Spring XML 25
 advantages 22
 loading 173
 testability 34
 XML Schema 25
 core 25
 in JAR files 32
 location 32
 namespace handler 33
 transport 32

connector 12
 Transport Service
 Descriptor 41
 connectors, failed 207
 console, reading from
 stdin 22
 console, writing to
 stdout 22
 conversational state 188
 core concepts
 component 15
 connector 12
 endpoint 12
 event 16
 filters 14
 message 16
 model 10
 router 14
 service 11
 summarized 19
 transformer 13
 transport 12
 correlation ID, relevance to
 monitoring 262
 correlationId, relevance to business scenario 93
 Cryptix 230
 CSV 46
 custom-entry-point-resolver 153
 customer relationship management (CRM) 344, 348
 custom-interceptor 325
See also interceptor
 custom-protocol 28
 custom-transformer 28, 133
 CXF endpoint
 frontend 60
 wsdlLocation property 58
 CXF transport
 component 58
See also component
 default location 58
 generated WSDL 58
 hostname 58
 inbound endpoint 57
 JAXB. *See* Java Architecture for XML Binding
 JAX-WS
 front end 59
See also Java API for XML Web Services
 POJOs 57
 simple frontend 58
 use of with BPEL engines 354
 WSDL 57

D

dashboard. *See* monitoring
 data compression. *See* compression transformers
 data warehouse 343, 356
 database 40, 72
 database transport 40
 default-connector-exception-strategy 198–201
 default-dispatcher-threading-profile 370
 default-exception-strategy 204
 DefaultMessageAdapter 314
 DefaultMuleContextFactory 173
 DefaultMuleMessage 124, 314
 default-receiver-threading-profile 370
 default-service-exception-strategy 198, 200–201
 default-service-threading-profile 370
 default-threading-profile 370
 dependency management 274
 deployment
 as a JCA resource. *See* Java EE Connector Architecture
 challenges 189
 development tools 190
 in a Java application 173
 in a web application 174
 management 189
 NetBoot
 standalone server 169
 deployment topology.
See topology
 deserialization. *See* object deserialization
 development practices 191
 discoverable transformer 137
 DiscoverableTransformer 137
 dispatcher 12
 dispatching (asynchronous) 300
 sending (synchronous) 300
See also thread pool
 dispatcher-threading-profile 370
 Disposable (interface).
See lifecycle
 distributed transactions 187
 document exchange 55
 domain-specific language (DSL) 26

E

EAI. *See* enterprise application integration
 echo component 144
 echo world 22
 echo-component 144
 EchoService 145
 Eclipse 273, 347
 EJB. *See* Enterprise JavaBeans
 email transport 40, 51
 encoding 116, 133
 endpoint 12
 asynchronous messaging 56
 global 13
 inbound 13
 outbound 13
 response 13
 synchronicity 56
 synchronous messaging 56
 transformer 13, 112
 URI 12, 57
 specifying a connector 302
 with expression 384
 endpoints
 address attribute 44
 global 43
 Mule 1.x style URI 44
 enterprise application integration (EAI) 5, 10
 enterprise integration
 broker centric 5
 challenges 4
 message bus 6
 patterns 10
 spaghetti plate 4
Enterprise Integration Patterns 10
 Enterprise JavaBeans (EJB) 177
 enterprise service bus (ESB) 6
 as a mediator 10
 canonical deployment 183
 definition 7
 enterprise service network (ESN) 181
 entry point 152
 discovery 153
 exclude methods 154
 include methods 154
 resolver 152
 default set 154
 set 152
 transformer 153
 void methods 153
 EntryPointResolver 153
 EnvelopeInterceptor. *See* interceptor

error channel 77
 ESB. *See* enterprise service bus
 ESN. *See* enterprise service network
 evaluator, Groovy 335
 event context 16, 311
 adapting messages 313
 transforming messages 313
 event processing. *See* message processing
 eventual consistency 187
 exception 196
 exception strategies 11, 197, 202
 component 197
 configured on a per-model basis 198
 configured on a per-service basis 198
 connector 197
 default 197
 default-service-exception-strategy 248
 and transaction rollbacks with JMS 248
 transactions and 247
 used in conjunction with routing 202
 exception-based routing 202, 205
 exception-based-router 207, 209
 ExceptionPayload 312
 exceptions 199, 203
 connector 203
 routing 203–204
 service 203
 exception-type-filter 204
 ExhaustingRetryPolicy 208
 exhaustion, of component pool 158
 expression evaluation
 available evaluators 381
 custom evaluator 386
 registration 386
 framework 381
 function
 dateStamp 49
 uuid 49
 header, originalFilename 49
 in custom code 385
 in endpoint URIs 384
 in filters 88, 383
 in transformers 120, 384
 expression transformer 120, 156, 263
 output 121
 ExpressionEvaluator 386

expression-filename-parser 49
 ExpressionManager 385–386
 expression-transformer 120

F

failover 207
 fault tolerance. *See* topology
 file connector
 autoDelete property 48, 50
 fileAge property 48
 pollingFrequency property 50
 file endpoint
 inbound 63, 68
 outbound 72
 outputPattern 50
 File Transfer Protocol 54, 70
 file transport 40, 46
 expression-filename-parser 48
 moving files 48
 outbound-endpoint 63
 FileAdaptor 49
 filename-wildcard-filter 50
 files
 JSP 49
 XML 49
 filter 14, 49
 boolean evaluation 89
 xpath-filter 87
 nesting logical 89
 not filter 89
 or filter 89
 org.mule.api.routing.filter.Filter 337
 payload-type-filter 86
 regex-filter 238
 regular expression 87
 wildcard 87
 filtering-router 97
 filter-router, payload-type-filter 97
 firewall 62, 64
 See also topology
 ForeverRetryPolicyTemplate 211
 forwarding-router 204
 FTP 40
 FTP endpoint
 inbound 71
 outbound 72
 function, expression evaluator 385
 Functional Test Case
 mocking components 292
 testing a component and transformer 289

Functional Test Case (*continued*)
 use of the MuleClient 289,
 291
 functional testing 294

G

Galaxy 171, 192
 Maven 194
 NetBoot 193
 query language 194
 Spring configurations 195
GalaxyApplicationContext 195
GalaxyConfigurationBuilder 194
 global endpoint. *See* endpoint
 global property 27
 Gnu Privacy Guard (GPG) 230
 governance. *See* Galaxy
GreetingServiceImpl 57
 JAX-WS annotated 59
 Groovy 22, 174, 328, 332
 builders 334
 expression evaluator 383
 expression transformer 263
gzip-compress-transformer 117
gzip-uncompress-transformer 118

H

headers. *See* message properties
hex-string-to-byte-array-transformer 117
 Hibernate 72
 high availability. *See* topology
 hot deployment
 alternative to 184
 lack of 169
 HtmlDashboard 269
HTTP
 basic authentication 219, 225
 chunking 378
 methods
 GET 61
 POST 66
 transport 55, 61
 polling inbound
 endpoint 63
 polling-connector 61
 http connector configuration 41
HTTP endpoint
 accepting XML via POST 62
 inbound 66, 69
 outbound 61, 63
 polling thread model 369
 https 229

http-security-filter, securing
 HTTP inbound endpoints 225
hub-and-spoke 180, 183
HypericHQ 252
Hypertext Transfer Protocol 70

I

IDEA 273
idempotent 185
IMAP 40, 45, 344, 348
 transport 45, 51
IMAP connector 51, 53
 backupEnabled 53
 backupFolder 53
 checkFrequency 53
 deleteReadMessages 53
imaps 229
 inbound endpoint,
 introduction 43
Initialisable (interface).
 See lifecycle
instant messaging 76
 reception 77
integrated development environment (IDE) 273
 and simplifying XML
 authoring 284
 creating a Mule configuration
 template 284
 integration with 273
Mule IDE 285
 installing as Eclipse plugin 285
 running a project 288
SAXParserFactoryImpl and
 Xerces 289
 starting a new project 287
integration testing 301
integration. *See* enterprise integration
interceptor
 bridge component 322
 component 319
 envelope 320
 Spring AOP 320
 stack 320
 defining 321
 using 322
Internet Message Access Protocol (IMAP) 75
inter-service
 communications 162
Invocation (object). *See* interceptor
DUPS_OK_ACKNOWLEDGE 378

J

JAAS
 DefaultLoginModule 224
 jaas.conf 224
 LoginContext 224
 maintaining a static list of user data 224
jaas-security-filter
 JMS header
 authentication 227
 use with MULE_USER 227
Jabber 76, 203–204
JAR files 274
Java API for XML Web Services 59
Java application
 deployment model 173
 Mule context 173
 pros and cons 174
 Spring parent context 174
 starting Mule 173
 stopping Mule 173
Java Architecture for XML Binding (JAXB) 59
Java Authentication and Authorization Service (JAAS) 220, 224
Java Business Integration (JBI) 9
Java Cryptographic Extensions (JCE) 228
Java Development Kit (JDK) 328
Java EE Connector Architecture (JCA)
 asynchronous listener 177
 deployment model 177
 pros and cons 178
 synchronous client 177
Java Management Extension (JMX) 256
 HTTP console 259
 JConsole 256
Java Message Service (JMS) 40, 46, 64, 197, 342
 1.0.2b 64, 66
 1.1 64
 as a backbone 184
 benefits for application integration 64
 bodyless message 127
 brokers 66
 BytesMessage 66
 consuming messages 126
DUPS_OK_ACKNOWLEDGE 378

Java Message Service (JMS)
(continued)

- durable subscription 67
- header authentication 226
- highly available provider 187
- MapMessage 66
- message types 125–126
- ObjectMessage 66
- producing messages 125
- quality of service (QoS) 184
- queue 64, 77
- Reply-To 69
- selector 68
- session 126
- StreamMessage 66
- temporary queue 69
- TextMessage 66–67
- threading 369
- topic 64
- transacting message flows 241
- transformers 125
 - from JMS messages 126
 - to JMS messages 125
 - turning off 115
- Java runtime environment (JRE) 224
- Java service wrapper. *See* stand-alone server
- Java Transaction API (JTA) 243, 246
- java.net.URI 45
- java.util.concurrent, use of latches in tests 293
- java.util.logging 212
- javac 274
- JavaScript Object Notation 55, 61
- javax.mail.Message 46
- JAX-WS
 - @WebMethod annotation 60
 - @WebService annotation 59
- JBI. *See* Java Business Integration
- JBossAS 243
- JBossTS 243
- jBPM 346
 - action 348
 - embedded HSQL data source 353
 - GPD 346
 - Hibernate 353
 - jPDL 346, 348
 - MySQL 354
- JCA. *See* Java EE Connector Architecture
- JConsole 256

JDBC 40, 45

- datasource 45

JDBC endpoint

- and inbound querying 73
- inbound 72, 75
- outbound 73, 75
- query results as a Map 74

JDBC transport 45, 72

JEE application server 243

JMS

- broker failure 209
- broker unavailable 210
- message properties 46

JMS Connector, ActiveMQ 77

JMS endpoint

- disableTemporaryReplyTo-Destinations 70
- inbound 68, 77, 84
- outbound 65
- responseTimeout 69
- synchronous 68

JMS transport 40

- and account provisioning 346
- filtering 68
- filters 68
- lack of polling support 358
- Maven dependencies 282
- periodic consumption off a queue 358
- Transport Service Descriptor 42

jmsmessage-to-object-transformer 126

JMX. *See* Java Management Extension

jmx-default-config 256, 310

jmx-log4j 262

jmx-mx4j-adaptor 258

jmx-notifications 264

JRuby 328

JSON 150

JSR-223 328

- lack of facilities for auto-reloading of scripts 339

See also scripting

JVM, performance considerations 378

K

Kerberos 218

L

legacy-entry-point-resolver set 155

lifecycle

- custom adapters 319
- interfaces 318
- methods 317

Lightweight Directory Access Protocol (LDAP) 218–219, 221, 223, 348

load balancer 58

See also topology

load sharing 70

log component 145

log4j 197, 212, 262

- receiver framework 214

log4j.properties 212

log4j-notifications 264, 322

Logback 213

log-component 145

logging 197

- and monitoring 261
- Chainsaw console 265
- message 144, 262

logging levels 212, 261

- changing at runtime 262

performance

- considerations 377

logging-catch-all-strategy 98

logging-interceptor. *See* interceptor

LogService 145

M

management agent profiler 376

management agents 252, 256

marshalling. *See* object serialization

Maven

- Assembly Plugin 190
- Publishing Plugin. *See* Galaxy

MBean. *See* Java Management Extension

MDB. *See* Message Driven Beans

message 16

- adapter 13, 110, 312
- attachment 16
- attachments 110
- bytes payload cache 313
- detaching 116
- encoding 110, 116
- exception payload 16, 312
- expression evaluator 384–385

message (*continued*)
 immutable 312
 original payload 313
 parts 16
 payload 16, 312
`getPayloadAsString()` 306
 performance
 considerations 378
 properties. *See* message properties
 reflection builder 145
 message dispatcher. *See* dispatcher
 Message Driven Beans (MDB) 177
 message processing 15, 314
 arbitrary dispatch 315
 controlling 17, 314
 performance
 considerations 377
 response handling 17, 140
 standard 17
 stopping 314
 message properties 46, 110, 118
 adding 119
 Mule-specific 119
 removing 119
 renaming 120
 transformer 110, 119
 transport-specific 119
 user-defined 119
 message receiver. *See* receiver
 message requester. *See* requester
 message transformer 110
 message-properties-transformer 119
 metadata. *See* message properties
 META-INF/services/org/mule/transport 41
 method-entry-point-resolver 153
 Microsoft Active Directory 223
 model 10
 inheritance 35
 monitoring
 auditing 262
 dashboard 268
 data 267
 graphs 254
 JMX 256
 MBean names 258
 JVM 254, 257
 log files 261
 Mule's health 252
 networking 253
 services 257
 SNMP 254

Mule 22
 API. *See* API
 as a proxy. *See* topology
 client. *See* Mule client
 clustering 185
 Community Edition 18
 competition 9
 context. *See* Mule context
 core concepts 10
 development practices 191
 distinctive aspects 9
 domain-specific language 26
 Enterprise Edition 207
 Galaxy 192
 history 8
 installing 18
 issue tracking 388
 mailing list 388
 NetBoot 171
 Profiler Pack 373
 project 7
 remote controlling. *See* WrapperManager (MBean)
 version 1 8
 version 2 8
 welcome to 19
 why the name? 7
 Mule client 173, 175
 bootstrapping Mule 306
 disposing 304, 306
 in memory 301
 instantiating 301–303,
 305–306
 module 303
 remote dispatcher 302
 role in testing 301
 usage 302
 using transports directly 306
 Mule community 387
 Mule context 307
 how to get ahold of 308
 registry 310
 starting and disposing 309
 statistics 309
 system configuration 309
 MULE_properties 119, 262
 MuleClient 300
 use of send 293
 use of sendAsync 291
 MuleClientTestCase 300
 MuleConfiguration (class) 309
 MuleContextAware 308
 MuleContextAware (interface).
 See lifecycle

MuleContextNotificationListener. *See* notifications
 mule-core.jar file 33
 MuleEvent 110
 MuleEventContext. *See* event context
 MuleForge 210, 388
 connectors 41
 MULE_HOME 18, 170
 MuleHQ 252
 MuleMessage 110, 134, 137,
 312
 MULE_MESSAGE_ID 46
 mule-module-client 300, 303
 mule-module-spring-config.jar file 33
 MuleReceiverServlet 176
 MuleRESTReceiverServlet 176
 MuleServer (class) 308
 MuleSource 8, 387
 MULE_USER
 defining the password encryption strategy 227
 encrypting credentials 227
 format with embedded credentials 226
 plaintext passwords 227
 MultiConsumerJmsMessage-Receiver 372
 multipurpose internet mail extensions (MIME) 55
 MX4J Http adaptor 259
 MySQL 74

N

Nagios 40
 NamespaceHandler 33
 NetBoot
 and Galaxy 171
 command line 171, 193
 deployment model 171
 local cache 172
 pros and cons 172
 network address translation (NAT) 58
 network load balancer. *See* topology
 no-action-transformer 115
 no-arguments-entry-point-resolver 153
 notifications
 activating 324–325
 advanced features 325
 agents 264
 framework 322

notifications (*continued*)
 listener 323
 registering 324–325
 performance impact 265
 null component 147
 null-component 147

O

object deserialization
 from bytes 115
 from xml 124
 object serialization
 to bytes 116
 to xml 124
ObjectStore 186
 object-to-byte-array-
 transformer 116, 313
 object-to-jmsmessage-
 transformer 13, 118, 125
 object-to-string-transformer 27,
 313
 object-to-xml-transformer 124
 acceptUMOMessage 124
 Open Source ESBs in Action 9
 OpenLDAP 223, 344
 OpenMQ 64
 OpenPGP 230
 outbound endpoint 43
 message properties 46
 outbound router
 bypassing 314
 explicitly calling 315

P

packaging 274
 pass-through-router 23, 49
 password-encryption-
 strategy 220
 decoding MULE_USER
 values 228
 decrypt-transformer 229
 encrypting credentials 227
 encrypt-transformer 229
 message payloads 228
PBEWithMD5AndDES
 algorithm 228
 payload expression
 evaluator 384
 payload format transformer 110
 payload type transformer 110
 performance 372
 improving 377
 using a profiler 373

benefits 374
 management agent 376
PGP module
 credential accessor 232
 key alias 232
 key ring management 232
**MuleHeaderCredentials-
 Accessor** 232
 and MULE_USER 232
 secret password 232
 Unlimited Strength Jurisdiction Policy files 232
 ping service 253
Plain Old Java Object
 (POJO) 57
 plaintext 61
 pluggable authentication modules (PAM) 224
 policies. *See Galaxy*
 policy template 208
 policyOK 208
PolicyStatus 208
 polling-connector 27
 pollingHttp 63
 pooled component. *See component*
 pooling profile 158
 pop3s 229
 pretty good privacy (PGP) 220
Profiler Pack 373
 properties
 file 31
 global 31
 override 31
 system 31
 property placeholders 30
 property-entry-point-
 resolver 153
PropertyPlaceholderConfigurer
 31
 prototype-object 157
 proxy. *See topology*
 public key encryption
 encryption and signing 230
 importance of a robust
 PKI 230
 motivations 230
 use of key pairs 230
 publication application 10, 127,
 170, 258, 262, 266, 269
 configuration files 128
 monitoring 253
 running 130
 server ID 258
 used transformers 127

publish-notifications 264
Python 328

Q

quality assurance (QA) 191
Quartz 343, 356
 scheduler 267, 357
Quartz transport
 cronExpression 357, 359–360
 use of a question mark 358
 use of asterisks 358
 endpoint-polling-job 359
 inbound-endpoint 357
 job dispatch 360
 job endpoint 358
 jobName 357
 scheduled-dispatch-job 360

R

RandomIntegerGenerator 200–
 201, 203
 really simple syndication
 (RSS) 63
receiver 12
 idempotent 185
 polling 369
 servlet. *See servlet transport*
receiver. *See thread pool*
receiver-threading-profile 370
 redundancy 70
 reflection message builder
 component 145
**reflection-entry-point-
 resolver** 153, 156
regex-filter 86
registry 310
 looking up objects 311
 looking up services 310
 Spring application
 context 311
 storing objects 311
registry. *See Galaxy*
 regular expression 85
 remote controlling Mule. *See*
 WrapperManager (MBean)
 remote dispatcher
 agent 303
 security considerations 305
 usage 303
 wire format 303
 XML 304
 representational state
 transfer 63

request processing. *See* message processing
 RequestContext 312
 requester 12
 requesting
 (synchronous) 300
 response
 asynchronous 17
 none 17
 synchronous 17
 responseTransformer-refs 113
 response-transformers 113, 253,
 384
 REST 40
 JSON response 150
 service component 149
 rest-service-component 150
 retry policy 207
 simple 207
 retry policy template,
 multithreaded 210
 retryCounter 208
 RETRY_LIMIT 208
 RetryPolicy 207
 RetryPolicyTemplate 210
 RFC-2396 45
 Rhino 328
 rollbacks 198
 router 14, 82
 and static-recipient-list-
 router 244
 async request reply 83
 async-reply 104
 using VM queues 104
 chaining 82
 chaining-router 85, 97, 101,
 104
 ad-hoc service
 composition 343
 BPM as an alternative 343
 use with STDIO
 endpoint 101
 collection-aggregator-
 router 92–93
 correlationGroupSize 93
 correlationId 93
 extension of selective-con-
 sumer-router 93
 exception-based router 206
 filtering-router 86, 97
 filtering-xml-message-
 splitter 103
 filter-router
 payload-type-filter 97
 forwarding router 91

forwarding-consumer-
 router 237
 extension of selective-
 consumer-router 91
 idempotent-receiver-
 router 94
 canonical banking
 examples 94
 catch-all-strategy and dupli-
 cate IDs 94
 exception strategy and
 duplicate messages 95
 forwarding-catch-all-
 strategy 95
 ID generation 94
 idExpression for idempo-
 tent identification 95
 simple-text-file-store 95
 inbound 14
 list-message-splitter-router
 dispatch to different
 endpoints 103
 use with xpath-filter 102
 message collection 92
 message-splitter-router 102
 multicasting-router 99
 outbound 14
 pass-through 82
 pass-through-router 85, 238
 response 14
 selective consumer 85
 selective-consumer 89–90,
 328, 335
 selective-consumer-router
 87, 97
 catch-all-strategy 90
 custom script 337
 header filter 90
 supplying a filter 90
 unstructured data 87
 static-recipient-list-router 98
 use of spring:value
 element 99
 wire-tap-router 96
 billing applications 96
 extension of selective-con-
 sumer-router 97
 message logging 96
 Quality of Service 96
 xml-message-splitter-router
 splitExpression 104
 routers
 and filters 86
 inbound 41
 outbound 41
 response 41

routing
 catch-all strategy 84
 component 84
 inbound 83
 outbound 83, 85, 97
 catch-all-strategy 86
 selective-consumer-router
 xpath-filter 85
 structured data 87
 shared logic 85
 RoutingNotification. *See* notifica-
 tions
 RoutingNotificationListener. *See*
 notifications
 Ruby 327
 Ruby on Rails 327
 RuntimeException 208

S

schedulers 45
 scheduling 40
 schema, cxf-wsdl 57
 ScriptConfigurationBuilder 174
 scripting 22
 component
 implemented with
 Rhino 328
 implementing Callable 340
 component-binding java-
 interface-binding
 element 331–332
 externally stored script 329
 Groovy custom
 transformer 332
 inline script 328–329
 JSR-223-compliant
 engine 329
 refreshable scripts 330
 script context 329
 message variable 329
 the engine attribute 331
 the file attribute 331
 transformer
 accessing
 MuleEventContext 340
 defining globally 333
 name attribute 333
 org.mule.api.transformer.
 Transformer 340
 secure copy 70
 secure shell (SSH) 70, 344, 348
 secure socket layer (SSL) 197,
 229
 security filter 225

security manager
and password encryption strategy 227
authentication 226
centralized security mechanisms 220
default security manager 220
password-based encryption 220
secret key-based encryption 220
encryption 220
LDAP provider 220
manager, delegation 221
MULE_USER 226
org.mule.api.security.Security Manager 220
referencing jaas.conf 224
Spring Security 220
swapping out delegate references 226
security provider,
org.mule.api.security.SecurityProvide 220
security-filter, controlling access, authorization and encryption 225
SEDA. *See* staged event-driven architecture
SeededRandomIntegerGenerator 201, 203
serializable-to-byte-array-transformer 117
serialization 64
serialization. *See* object serialization
server ID 257
ServerNotification. *See* notifications
service 11
composition 160
not yet implemented 147
proxying 160
service component. *See* component
service composition 100
service registry. *See* Galaxy service.
See thread pool
ServiceAware (interface). *See* lifecycle
service-level agreement (SLA) 190
service-oriented architecture (SOA) 6
service-overrides 49

servlet container. *See* web application
servlet transport
receiver servlet 176
See also web application
ServletContextListener 175
session 11, 312
Simple Network Management Protocol (SNMP) 254
Simple Object Access Protocol (SOAP) 40, 46, 55, 354
message properties 46
wrapper component 148
WS-* 64
SimpleRetryPolicy 208–211
single sign-on (SSO) 218
singleton-object 157, 319
SLA. *See* service-level agreement
SLF4J 197, 212
Smooks transformer 162
for performance 377
SMTP transport 51
outbound endpoint 53, 55, 355
properties 53
smtps 229
SNMP. *See* Simple Network Management Protocol
SOA. *See* service-oriented architecture
SOAP connector 55
SOAP transport,
synchronous 68
SOAP. *See* Simple Object Access Protocol
SoapUI 58
SocketHubAppender 214
software engineering
and Mule projects 191
performance considerations 377
Spring
AOP. *See* interceptor
beans 28
configuration in Galaxy. *See* Galaxy
context schema 29
dependency injection 207, 269
elements in configuration 28
external configuration 75
import 29, 36
JDBC datasource 74
JDBC template 72
lang namespace 338
parent context 174
property 29
property placeholder resolver 31
resource resolver 32
role in Mule 8
scripting
manual population 339
refresh-check-delay 339
scripting namespace 336
spring namespace 338
util schema 29
XML configuration builder 25
Spring in Action 28
Spring Portfolio 221
Spring Security 219
and OpenLDAP 223
LDAP support 223
{} evaluation 224
and MULE_USER for user-name propagation 224
defining the rootDN 223
group-search-base 224
prepending of ROLE_ 224
user-dn-pattern 224
namespaces 221
user-service 222, 226
static map of data 222
spring-object 157
SpringXmlConfigurationBuilder 173
staged event-driven architecture (SEDA) 16, 363, 365
standalone server
as a service 169
command line 169
profile parameter 373
deployment model 169
directory structure 170
installation 18
passing parameters 169
patching 170
project deployment 170
pros and cons 171
shared libraries 170
shutdown sequence 25
startup sequence 24
stopping 25
wrapper script 25
standard input/output streams 50
Startable (interface). *See* lifecycle
statistics 256, 259, 269, 309
stdio connector configuration 41

STDIO transport 51
 inbound endpoint 56
 stdio transport 22, 46
 dispatcher 44
 Stoppable (interface). *See* life-cycle
 streaming, performance
 considerations 378
 string expression evaluator 253, 385
 string-to-byte-array-transformer 117–118
 Structured Query Language 76

T

Test Compatibility Kit (TCK)
 AbstractTransformerTestCase 292
 FunctionalTestCase 289, 291
 test-component 292
 exceptionToThrow attribute 294
 introducing delays in component processing 292
 returning arbitrary data 292, 294
 simulate exceptions 293
 testing 274
 Apache Derby 290
 GreenMail 290
 HSQL 290
 integration 301
 thread pool 363
 asynchronous
 synchronous 367
 buffer 364, 369
 configuration. *See* threading profile
 dispatcher 363
 exhausted 371
 fully asynchronous 365
 fully synchronous 366
 message processing 365
 not handled by Mule 369
 polling receiver 369
 receiver 363
 service 363
 synchronous
 asynchronous 366
 transactions 365
 tuning 371
 versus component pools 363
 VM transport 368
 thread, dedicated to retry attempts 210

threading profile
 configuring 370
 exhausted action 371
 hierarchy of profiles 370
 not configured in Mule 371
 service 370
 tuning 371
 threadingPolicyTemplate 211
 Tibco EMS 64
 timer-interceptor. *See* interceptor topology
 client/server 179
 clustering 185
 ESB 183
 ESN 181
 fault tolerance 187
 high availability 184
 hub-and-spoke 180, 183
 instance and network level 178
 JMS backbone 184
 load balancer 184, 253
 proxy 182
 transaction 198
 action
 ALWAYS_BEGIN 237, 241, 245
 ALWAYS_JOIN 239, 241, 245
 component failures 238
 rollback 238
 exception strategy
 committing transaction 249
 commit-transaction
 element 249
 exception-pattern 249
 influencing transactional behavior 249
 log instead of commit 249
 rollback 248
 JMS rollback 237
 multicasting-router
 rollback 242
 outbound endpoint 239
 rollback 314
 rollback with JMS 241
 timeout with JMS 241
 transactions 187, 233
 as synchronous operations 365
 atomic database updates 234
 atomicity 234
 consistency 234
 databases 235
 durability 234
 isolation 234
 JMS 237
 multiple resource 235
 MySQL, requirement for transactional database engine 236
 necessitated by distributed data and systems 234
 real-world examples 233
 rollback 237
 rolling back 234
 single resource 235
 JDBC 235
 JMS 235
 spanning more than one resource 242
 support for JBossTS 243
 XA 242
 data sources and JMS and JDBC 243
 HeuristicExceptions 243
 JDBC provider support 242
 JMS provider support 243
 LookupFactory 246
 requiring special drivers 243
 rollback 246
 specifying JTA location with JNDI 247
 two-phase commit 243
 use in an application container 246
 use with Resin JTA 246
 transformation, no action 115
 transformer 13, 109
 and entry point resolver 154
 anonymous 112
 auto. *See* auto-transformer behavior 109
 byte. *See* byte transformers byte-array-to-object-transformer 335
 compression. *See* compression transformers
 custom 131, 328, 332
 transforming message 134
 transforming payload 131
 discoverable 137, 313
 expression. *See* expression transformer
 global 111
 idempotent 131
 inbound 13, 112–113
 input type 112
 JMS 125

transformer (*continued*)
 local 113
 no action 43
 on endpoint 13, 112
 order of precedence 113
 outbound 13, 113
 properties. *See* message properties
 response 13, 113, 253, 384
 return class 109, 112, 121
 round-trip 110, 132
 scripting 333, 335
 VelocityTransformer 290
 versus message adapter 110
 XML. *See* XML transformers
 Transformer (interface) 132
 transformer-refs 112
 transformers 112
 byte-array-to-string 63
 byte-array-to-string-
 transformer 62
 file-to-string-transformer 54
 object-to-jmsmessage-
 transformer 75
 string-to-email-transformer 55
 transport 12
 default transformer 13, 113,
 118, 125
 performance
 considerations 378
 Transport Layer Security
 (TLS) 229
 Transport Service Descriptor
 (TSD)
 JMS 42
 overriding 43
 service-overrides element 43
 transports
 and polling support 358
 lack of sophisticated polling
 mechanisms 359
 tuning. *See* performance
 two-phase commit (2PC) 243

U

Universal Message Object
 (UMO) 8, 124
 Unix 224
 scheduling with cron and
 at 356
 unmarshalling. *See* object deserialization
 URI. *See* endpoint
 use of spring value element 99

V

Velocity 131, 332
 configuration 133
 engine 134
 template 134, 136
 VelocityMessageTransformer
 135
 VelocityPayloadTransformer
 132
 vi 273
 VM connector, queueEvents
 property 78, 80
 VM endpoint, inbound 84
 VM transport 78
 default connector 41
 enabling service
 decomposition 80
 messages implementing
 Serializable 81
 persisted queues 187
 similarity to JMS 79
 thread pool 368
 use of
 FilePersistenceStrategy 81
 use of QueuePersistence-
 Strategy 81
 using in a transaction 242

W

WAR file. *See* web application
 web application
 deployment model 174
 interacting with Mule 175
 pros and cons 176
 service URIs 176
 starting Mule 175
 stopping Mule 175
 web services
 document exchange 63
 Remote Procedure Call 63
 Web Services Description Lan-
 guage (WSDL) 56
 soap address 58
 Web Services specifications 64
 web.xml. *See* web application
 WebLogic 243
 Windows, scheduling with the
 Windows Task
 Scheduler 356
 wire format. *See* remote dis-
 patcher
 wiretap 262

wrapper (Java service). *See* stand-
 alone server
 wrapper.conf 213
 wrapper.logfile.maxfiles 213
 wrapper.logfile.maxsize 213
 wrapper-component 148
 WrapperManager (MBean) 258
 WSDL transport
 inbound endpoint 56
 outbound endpoint 56

X

XA. *See* distributed transactions
 XML
 and editing 283
 escape sequences 68, 74
 schema 55
 transformers 122
 web services 55
 XML configuration. *See* configu-
 ration
 XML Schema. *See* configuration
 xml-to-object-transformer 124
 xml-wire-format. *See* remote dis-
 patcher
 XMPP 76, 204
 endpoint
 inbound 77
 outbound 77
 transport 40
 xmpps 229
 XSL Transformation
 (XSLT) 122
 XSLT. *See* XSL Transformation
 xsl-transformer 27
 xslt-transformer 13, 122
 concurrency 123
 parameter, configured 123
 workers 123
 XStream
 aliases 124
 for message
 transformation 124

Y

Yahoo! Messaging 76
 YourKit 373
 MBean 376

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



“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

5 4 4 9 9



9 7 8 1 9 3 3 9 8 8 9 6 2