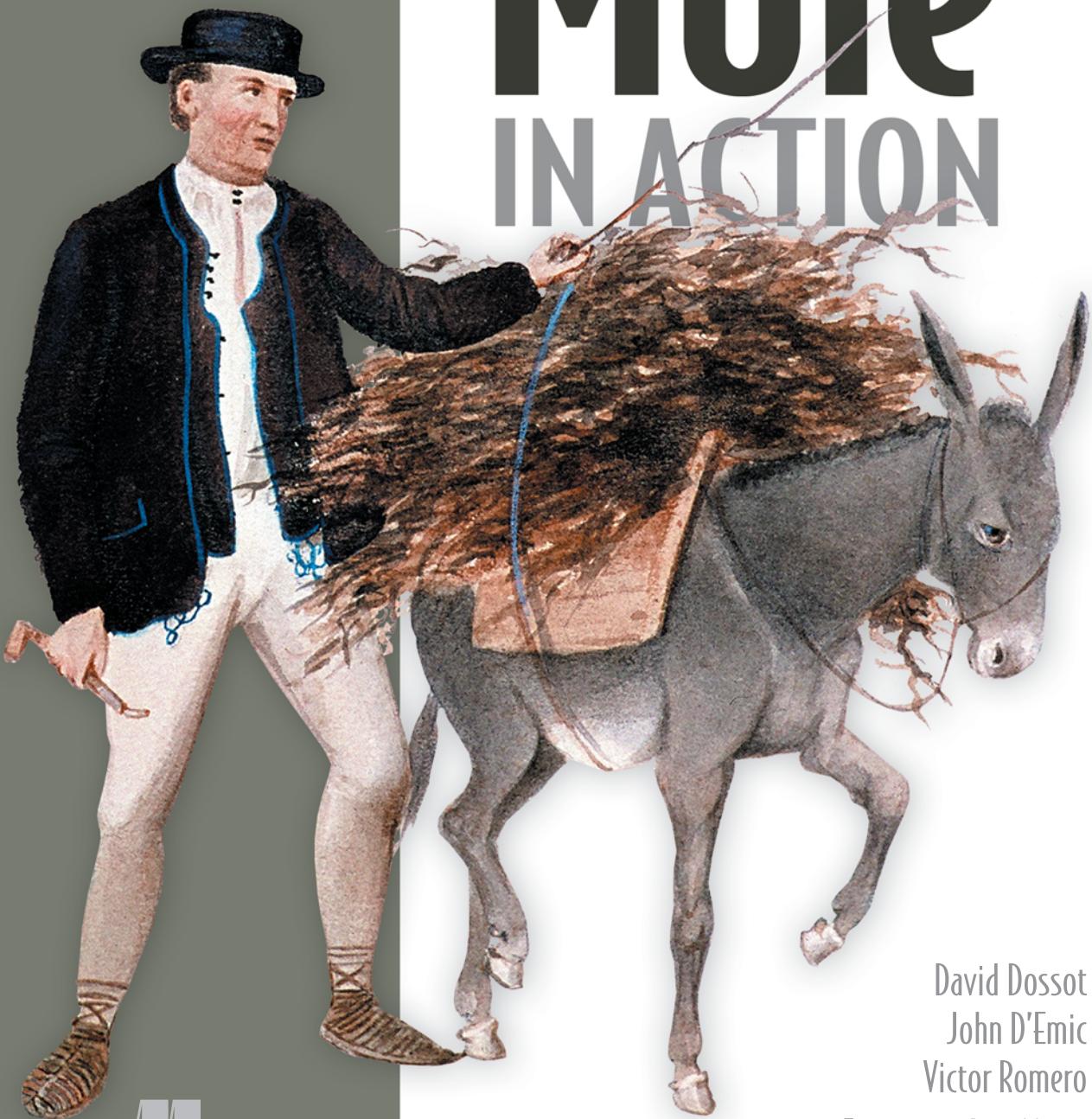


SECOND EDITION

Mule IN ACTION



David Dossot
John D'Emic
Victor Romero

FOREWORD BY Ross Mason



MANNING

Mule in Action
Second Edition

Mule in Action

SECOND EDITION

DAVID DOSSOT
JOHN D'EMIC
VICTOR ROMERO



MANNING
SHELTER ISLAND

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.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2014 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.

- ② 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.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editors: Jeff Bleiel, Nermina Miller
Copyeditor: Melinda Rankin
Proofreaders: Katie Tennant, Andy Carroll
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 9781617290824
Printed in the United States of America

brief contents

PART 1 CORE MULE.....1

- 1 ■ Discovering Mule 3
- 2 ■ Processing messages with Mule 27
- 3 ■ Working with connectors 50
- 4 ■ Transforming data with Mule 87
- 5 ■ Routing data with Mule 113
- 6 ■ Working with components and patterns 139

PART 2 RUNNING MULE.....173

- 7 ■ Integration architecture with Mule 175
- 8 ■ Deploying Mule 189
- 9 ■ Exception handling and transaction management with Mule 217
- 10 ■ Securing Mule 250
- 11 ■ Tuning Mule 264

PART 3 TRAVELING FURTHER WITH MULE.....285

- 12 ■ Developing with Mule 287
- 13 ■ Writing custom cloud connectors and processors 334
- 14 ■ Augmenting Mule with orthogonal technologies 358

contents

<i>foreword</i>	xv
<i>preface</i>	xvii
<i>acknowledgments</i>	xix
<i>about this book</i>	xxi
<i>about the authors</i>	xxiv
<i>about the cover illustration</i>	xxv

PART 1 CORE MULE 1

1	<i>Discovering Mule</i>	3
1.1	Enterprise Integration Patterns and service-oriented architecture	4
1.2	The Mule project	6
1.3	Competition	7
1.4	Mule: a quick tutorial	8
	<i>Installing Mule Studio</i>	9
	<i>Designing the flow</i>	13
1.5	Running, testing, and deploying the application	17
	<i>Running the application</i>	17
	<i>Testing the flow</i>	21
	<i>Working with the XML configuration</i>	22
	<i>Deploying to the Mule standalone server</i>	23
1.6	Summary	26

2	Processing messages with Mule 27
2.1	Going with the flow 28 <i>The response phase</i> 30 ▪ <i>Subflows</i> 31 ▪ <i>Private flows</i> 34
2.2	Interacting with messages 35 <i>Message sources</i> 36 ▪ <i>Message processors</i> 36 ▪ <i>Message exchange patterns</i> 37 ▪ <i>Endpoint URIs</i> 39
2.3	Exploring the Mule message 40 <i>Message properties</i> 42 ▪ <i>Understanding property scopes</i> 42 <i>Using message attachments</i> 45
2.4	Speaking the Mule Expression Language 46 <i>Using expressions</i> 47
2.5	Summary 49
3	Working with connectors 50
3.1	Understanding connectors 51 <i>Configuring connectors with XML</i> 52 ▪ <i>Configuring connectors with Mule Studio</i> 53
3.2	Using the file transport 56
3.3	Using the HTTP transport 58 <i>Sending and receiving data using HTTP</i> 58 ▪ <i>Using web services with Mule</i> 61
3.4	Using the JMS transport 67 <i>Sending JMS messages with the JMS outbound endpoint</i> 68 <i>Receiving JMS messages with the JMS inbound endpoint</i> 69 <i>Using selector filters on JMS endpoints</i> 70 ▪ <i>Using JMS synchronously</i> 70
3.5	Using email 71 <i>Receiving email with the IMAP transport</i> 71 ▪ <i>Sending mail using the SMTP transport</i> 73
3.6	Using the FTP transport 75 <i>Receiving files with inbound FTP endpoints</i> 76 ▪ <i>Sending files with outbound FTP endpoints</i> 77
3.7	Using databases 77 <i>Using a JDBC inbound endpoint to perform queries</i> 78 ▪ <i>Using a JDBC outbound endpoint to perform insertions</i> 79 ▪ <i>NoSQL with MongoDB</i> 80

3.8	Using the VM transport	82
	<i>Introducing reliability with the VM transport</i>	83
3.9	Using the Twitter cloud connector	85
	<i>Twitter</i>	85
3.10	Summary	86

4 **Transforming data with Mule** 87

4.1	Working with transformers	88
4.2	Configuring transformers	89
4.3	Using core transformers	91
	<i>Dealing with bytes</i>	92
	<i>Compressing data</i>	94
	<i>Modifying properties, flow variables, and session variables</i>	95
	<i>Transforming with expressions</i>	97
	<i>Enriching messages</i>	98
	<i>Automatic transformation</i>	99
4.4	Using XML transformers	100
	<i>XPath and Mule</i>	101
	<i>Transforming format with XSL</i>	102
	<i>XML object marshaling</i>	104
4.5	Transforming JSON with Mule	105
	<i>Querying JSON with MEL</i>	106
	<i>JSON object marshaling with Mule</i>	107
4.6	Scripting transformers	109
4.7	Summary	112

5 **Routing data with Mule** 113

5.1	Deciding how to route a message	115
	<i>Using the choice router</i>	115
5.2	Using filters	118
	<i>Filtering by payload type and header</i>	119
	<i>Filtering text and XML</i>	120
	<i>Filtering with expressions</i>	122
	<i>Logical filtering</i>	124
	<i>Ensuring atomic delivery with the idempotent filter</i>	125
	<i>Using the message filter</i>	127
5.3	Routing to multiple recipients	128
	<i>Dispatching messages with the all router</i>	128
	<i>Scatter/gather IO with the all router</i>	129
	<i>Going async with the async processor</i>	130
5.4	Routing and processing groups of messages	131
	<i>Splitting up messages</i>	132
	<i>Aggregating messages</i>	133
	<i>Routing collections</i>	134

5.5 Guaranteed routing 136

Resiliency for unreliable transports 136 ▪ Defining failure expressions 137

5.6 Summary 138

6 *Working with components and patterns 139*

6.1 Using Mule components 140

*Executing business logic 141 ▪ Resolving the entry point 143
Configuring the component 146 ▪ Annotating components 148
Handling workload with a pool 149 ▪ Scripting components 151 ▪ Component lifecycle 155*

6.2 Simplifying configuration with configuration patterns 157

*Using the simple service pattern 158 ▪ Using the bridge 162
Using the validator 164 ▪ Using the HTTP proxy 167
Using the WS proxy 168 ▪ Reusing common configuration elements 170*

6.3 Summary 172

PART 2 RUNNING MULE 173

7 *Integration architecture with Mule 175*

7.1 Structuring integration applications 176

Guerrilla SOA with hub and spoke 176 ▪ Mule as the enterprise service bus 177 ▪ Mule as a mediation layer 178

7.2 Mule implementation patterns 180

Using a canonical data model 180 ▪ Reliability patterns with asynchronous messaging 182 ▪ Proxying SOAP requests with CXF 185

7.3 Summary 188

8 *Deploying Mule 189*

8.1 Deploying standalone Mule applications 190

Packaging a standalone Mule app 194 ▪ Deploying applications to Mule 196 ▪ Configuring logs 198 ▪ Inter-application communication with Mule 200 ▪ Embedding web applications in Mule 202

8.2 Deploying Mule to a web container 204

8.3 Deploying applications to CloudHub 206

8.4	Embedding Mule into an existing application	208
8.5	Deploying Mule for high availability	210
	<i>High availability via fault tolerance</i>	213
8.6	Summary	216

9 *Exception handling and transaction management with Mule* 217

9.1	Dealing with errors	219
	<i>Using reconnection strategies</i>	219
	<i>Creating reconnection strategies</i>	221
	<i>Handling exceptions</i>	225
	<i>Using exception strategies</i>	227
9.2	Using transactions with Mule	230
	<i>Single-resource transaction</i>	232
	<i>Transactions against multiple resources</i>	241
	<i>Transaction demarcation</i>	247
9.3	Summary	249

10 *Securing Mule* 250

10.1	Spring Security 3.0 and Mule	251
	<i>User security with an in-memory user service</i>	252
	<i>User security with LDAP</i>	253
	<i>Securing endpoints with security filters</i>	254
10.2	Securing HTTP using SSL	255
	<i>Setting up an HTTPS server</i>	255
	<i>Setting up an HTTPS client</i>	256
10.3	Securing SOAP with Mule	258
10.4	Message encryption with Mule	259
	<i>Using password-based payload encryption</i>	260
	<i>Decrypting message payloads with PGP</i>	261
10.5	Summary	263

11 *Tuning Mule* 264

11.1	Staged event-driven architecture	265
	<i>Roll your own SEDA</i>	266
11.2	Understanding thread pools and processing strategies	267
	<i>Processing strategies and synchronicity</i>	269
	<i>Transport peculiarities</i>	272
	<i>Tuning thread pools</i>	274
	<i>Tuning processing strategies</i>	276

11.3	Identifying performance bottlenecks	278
	<i>Profiler-based investigation</i>	279
	<i>Performance guidelines</i>	281
11.4	Summary	284

PART 3 TRAVELING FURTHER WITH MULE285

12

Developing with Mule 287

12.1	Understanding the Mule context	288
	<i>Accessing the Mule context</i>	289
	<i>Using the Mule context</i>	290
12.2	Connecting to Mule	293
	<i>Reaching a local Mule application</i>	294
	<i>Reaching a remote Mule application</i>	295
	<i>Reaching out with transports</i>	298
12.3	Using the Mule API	300
	<i>Being lifecycle aware</i>	301
	<i>Intercepting messages</i>	303
	<i>Listening to notifications</i>	307
	<i>Configuring Mule data persistence</i>	310
12.4	Testing with Mule	315
	<i>Functional testing</i>	315
	<i>Behavior stubbing</i>	319
	<i>Load testing</i>	323
12.5	Debugging with Mule	328
	<i>Logging messages</i>	328
	<i>Step debugging a Mule application</i>	331
12.6	Summary	333

13

Writing custom cloud connectors and processors 334

13.1	Simplifying Mule development with the DevKit	335
13.2	Introduction to authoring cloud connectors	337
	<i>Rendering extensions configurable</i>	339
	<i>Managing connections</i>	340
	<i>Creating message processors</i>	343
	<i>Creating intercepting message processors</i>	344
	<i>Creating simple REST consumers</i>	345
	<i>Creating transformers</i>	347
	<i>Creating message sources</i>	348
	<i>Integrating Mule extensions with Mule Studio</i>	349
13.3	Creating a REST connector	350
13.4	Summary	357

14 Augmenting Mule with orthogonal technologies 358

- 14.1 Augmenting Mule flows with business process management 359
 - 14.2 Complex event processing 361
 - Using CEP to monitor event-driven systems* 361 • *Sentiment analysis using Esper and Twitter* 363
 - 14.3 Using a rules engine with Mule 365
 - Using Drools for selective message enrichment* 366 • *Message routing with Drools* 368
 - 14.4 Polling and scheduling 370
 - Using the poll message processor* 370 • *Scheduling with the Quartz transport* 371
 - 14.5 Summary 372
-
- appendix A Mule Expression Language* 373
 - appendix B Component and transformer annotations quick reference* 380
 - appendix C Mule Enterprise Edition* 383
 - appendix D A sample Mule application* 385
 - index* 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 this book would not exist.

Like thousands of developers before me, I was continually struggling with the complexities of systems and application 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 far too many assumptions about the environment and architecture, masking the ugliness with heavyweight tools, slick demo applications, and suave salesmanship. I used to spend 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.

Ten years on and Mule is the most widely used integration platform to connect any application, data service or API, across the cloud and on-premises. As SaaS, mobile, and big data converge, enterprises face a choice: become overwhelmed by the resulting explosion of endpoints or seize the opportunity to gain competitive advantage. Companies can no longer compete using only the assets, technology, and talent within their four walls. In the era of the new enterprise, companies must combine a surge of applications, data, partners, and customers into a single, high-performing entity.

Integration, services, and APIs have become critically important parts of application developers' lives in the new enterprise. No application is an island, and increasingly applications will use data from 10 or more data sources and services from within and outside the company. Couple this with the rise of SaaS, mobile, 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 the firewalls.

Since the first revision of *Mule in Action*, MuleSoft has launched the Anypoint Platform to address a broader set of enterprise needs that includes SOA use cases such as *legacy modernization* and *web services*; SaaS integration to connect cloud and on-premises applications for *data sync*, *batch*, and *process automation* use cases; and API creation and publishing support for *mobile APIs*, *B2B gateways*, and *device APIs*. The product offering has been expanded with the introduction of RAML support and the API Designer, APIkit, and API Management. Some of these capabilities are outside the scope of this book, but at the core of the Anypoint Platform is Mule.

This book provides thorough coverage of all aspects of the Mule core runtime. It provides great examples of fundamental things you'll most likely need to do with Mule, from creating and consuming services to working with various technologies such as JMS, web services, FTP, relational databases, and NoSQL. Importantly, it covers how to test, deploy, monitor, and tune Mule applications, topics that are critical for deploying Mule to production.

This book is also a great guide for anyone using CloudHub, the leading integration platform as a service (iPaaS). CloudHub has Mule at its core, so integration applications can run on Mule or CloudHub.

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 enterprise or hybrid architecture over time is compelling. Each chapter explains the tools provided by Mule for building service-oriented applications. The chapters cover the spectrum: configuration basics, message routing, data transformation, publishing services, and creating RESTful APIs.

This publication marks a significant milestone for Mule. It demonstrates that the ideals of open source and community building really do work. The authors—David Dossot, John D'Emic, and Victor Romero—are long-time community members and have made many contributions to the project; this book is a significant and lasting addition. This is the must-have guide for all current and prospective Mule users; it explains all aspects of Mule without going into unnecessary detail, focusing on the concepts most important for building integration applications. Read on to learn how to unlock the power of Mule.

ROSS MASON
FOUNDER, MULESOFT
CREATOR OF THE MULE PROJECT

preface

The integration and IT landscapes have dramatically evolved since the first edition of this book was released in 2009. Both API and mobile platform adoption have exploded, changing the way IT thinks about application integration. Advances in virtualization technology and the ever-decreasing price of storage have led to massive, horizontally scalable computation and data storage approaches. The broader acceptance of polyglot application development has led to cross-platform messaging solutions. Businesses are beginning to realize the value of the convergence of these as big data and are extracting real value from them.

In many ways, the early promises of service-oriented architecture are being realized, albeit in ways very different than originally intended. The world has largely moved away from SOAP and XML and their associated standards, in favor of RESTful, JSON-based APIs. UDDI, never widely adopted, is being replaced by lighter-weight mechanisms for service discovery that look very much like “App Stores” for APIs. Messaging solutions are more lightweight and decentralized than their previously monolithic predecessors. Finally, top-down-driven integration and mediation solutions have been supplanted with bottom-up, agile frameworks.

As anyone who has been around the block a few times knows, however, the old stuff never really goes away. An insurance company’s mainframe that has been processing claim data without a hiccup for years isn’t going to be suddenly replaced, nor is the full stack of SOAP services carefully implemented by a financial institution before the emergence of REST. Nobody is going to flip the switch overnight on a production, multimillion-row, geographically distributed database simply because a newer technology exists.

Mule is a platform to tie all of this together. This book will show you how to use Mule to develop, deploy, manage, and extend integration applications.

The authors have used Mule extensively for years, successfully delivering implementations to both startups and established enterprises, including insurance companies, financial institutions, and governments. In these contexts, they've used Mule in a variety of capacities, from a lightweight mediation layer to full-blown ESB implementations.

acknowledgments

We'd like to thank our development editors at Manning, Jeff Bleiel and Nermina Miller, who have patiently supported us during the whole process. We also want to thank Katie Tennant, our awesome proofreader at Manning.

We want to extend further thanks to our reviewers, whose insights helped us build a better book: Amjad Mogal, Andrew Johnson, Brad Johnson, Chris Mathews, Dan Barber, Davide Piazza, Frank Crow, Jesus de Oliveira, Joan Picanyol i Puig, Keith McAlister, Lee Dyson, Magnus Larsson, Nicolas Mondada, Ramiro Rinaudo, and Wayne Ellis.

Special thanks to Ross Mason for writing the foreword to our book, and to our technical proofreaders, Alberto Aresca, Felix Manuel Jerez, German Solis, Juan Alberto Lopez Cavallotti, and Sebastian Beltramini, from MuleSoft. We're also very grateful to Daniel Feist and the MuleSoft Engineering Team for their deep and extended feedback on the manuscript.

DAVID

I would like to thank my family for their support during this book update, which evolved into something more like a rewrite! I'm also grateful to our readers for their continuous feedback and sustained interest in this book.

JOHN

I would like to thank my wife, Catherine, and my son, Johnny, for putting up with me for two years while we "updated" this book for Mule 3.x. I also want to thank everyone at MuleSoft for their guidance and support throughout the entire process.

VICTOR

I would like to thank my mom for instilling a love of knowledge in me, my grandmother for teaching me the value of hard work, and the rest of my family for being such an inspiration. I would also like thank my friends and colleagues for their unconditional support during the creation of this book.

about this book

Mule, as the preeminent, open source integration platform, provides a framework for implementing integration solutions. The 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 in action in real-world contexts. After a little bit of history and some fundamentals of configuring Mule, we'll walk you through the 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, tuning, and working with complementary technologies such as BPM and CEP.

Who should read this book

This book is primarily targeted at developers who want to solve integration challenges using the Mule platform. It's also useful for architects and managers who are evaluating Mule as an integration platform or ESB solution. Additionally, system administrators tasked with supporting Mule instances will find this book, part 2 in particular, of value.

How to use this book

Each chapter in this book builds on the previous chapter. Readers new to Mule are encouraged to read the book with this in mind. Readers familiar with Mule 2.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 and screenshots of Mule Studio. More importantly, providing such a book would duplicate the content already available at www.mulesource.org, the Mule Java-docs, and the XSD documentation—sources from which complete reference documentation is available. We hope that reading *Mule in Action* gives you the skills to use these resources effectively.

Roadmap

The first part of this book will take you through a journey of discovery of the core aspects of Mule and will provide the fundamental knowledge you need to get started building integration applications with this ESB. After explaining the origin of the field of enterprise application integration, the first chapter sets the stage for the rest of the book by getting you started with your very first Mule project. Chapter 2 will make your head spin with the introduction of numerous essential concepts related to message processing in Mule, which will be referred to throughout the book. But fear not, chapter 3 will start building on this new knowledge by introducing you to transports and connectors, the basic building blocks required to start integrating systems. Chapter 4 will expand your knowledge by adding the notion of message transformation, one of the main activities performed by Mule. Going further, chapter 5 will delve into the crucial aspect of routing messages, while chapter 6 will detail how to handle these messages with components and configuration patterns.

The second part is not only focused on applying all the knowledge previously acquired but also on learning how to build and run production-grade Mule applications. With all the building blocks in hand, where does one need to start and what must be considered to succeed with Mule? These are the important questions answered in chapter 7. Chapter 8 covers the different deployment strategies supported by Mule, so you can decide on the best approach for your needs and particular environment. Chapter 9 digs into exception handling and transaction management so you can learn how to increase the reliability of your applications. Securing Mule is a must in publicly exposed applications or anytime sensitive data is processed; chapter 10 will tell you all about Mule’s security model. Finally, chapter 11 details strategies for testing and improving the performance of your applications.

The third part of the book will expand your horizon by giving you the tools and knowledge to go further with Mule. Chapter 12 will dive deep in Mule’s internal API, opening the door to complex and rich interactions with the integration framework in ways that the XML configuration alone can’t do. Chapter 13 explains in detail how DevKit, a free tool provided by MuleSoft, can be used to create custom extensions to Mule. Finally, chapter 14 presents how Mule can be made even smarter with the integration of business rules and complex event processor engines.

In order to preserve the reading flow of the book, we’ve extracted reference material in appendixes. The Mule Expression Language is detailed in appendix A, while appendix B covers Mule’s specific annotations that you can use in your custom code.

Appendix C introduces the features available in the Enterprise Edition of Mule. Finally, a complete Mule application is described in appendix D.

Code conventions and downloads

The code examples in this book are abbreviated in the interest of space. In particular, namespace declarations in the XML configurations and package imports 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 these cases, the \ or ➔ markers are used to indicate that a line has been wrapped for formatting. Code annotations highlight certain lines of code in some of the listings.

The source code of the book is available from the publisher's website at www.manning.com/MuleinActionSecondEdition and from GitHub here: <https://github.com/ddossot/mule-in-action-2e>. The required configuration to run these examples is the following:

- JDK 1.6 with unlimited JCE cryptography
- Maven 3
- Mule 3.4.x Community Release

Author Online

Purchase of Mule in Action, Second Edition, 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 www.manning.com/MuleinActionSecondEdition. This page provides information on 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 dialog 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 forum remains voluntary (and unpaid). We suggest you try asking the authors 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 authors

DAVID DOSSOT has worked as a software engineer and architect for more than 18 years. He's been using Mule since 2005 in a variety of different contexts and has contributed many extensions to the project. His focus is on building distributed and scalable server-side applications for the JVM and the Erlang VM. David is a member of IEEE, the Computer Society, and AOPA, and holds a production systems engineering diploma from ESSTIN.

JOHN D'EMIC has worked in various capacities as a system administrator, software engineer, and enterprise architect for more than 15 years. He has been working with Mule in a variety of capacities since 2006 and is currently principal solutions architect at MuleSoft. John holds a BS in Computer Science from St. John's University.

VICTOR ROMERO currently works as a solutions architect at MuleSoft in London. He started his career in the dot-com era and has been a regular contributor to open source software ever since. Originally from the sunny city of Malaga, Spain, his international achievements include integrating to the cloud from a skyscraper in New York City and creating networks for an Italian government licensee in Rome.

about the cover illustration

The figure on the cover of *Mule in Action, Second Edition* is captioned “A man and his mule from the village of Bgrud in Istria, Croatia.” The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian’s retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell the inhabitant of one continent from another and today the residents of the picturesque towns and villages on the Istrian Peninsula on the Adriatic coast of Croatia are not readily distinguishable from the residents of other parts of Europe and the world. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Part 1

Core Mule

M

ule is a lightweight, event-driven enterprise service bus and an integration platform and broker. As such, it resembles more a rich and diverse toolbox than a shrink-wrapped application. In the first chapter of this book, we'll introduce you to the history of its origins and the competing projects that exist on the market, and then we'll immediately dive into a quick tutorial that will get your feet wet and your mouth watering!

In chapter 2, we'll go through an extensive review of the principles involved in processing messages with Mule. You'll learn the notions of flows, message sources, and processors and will look deep into the structure of the Mule message. The Mule Expression Language (MEL) will be introduced too.

Chapter 3 will be the first one dedicated to one of the major moving parts of Mule: transports and connectors. You'll discover the most prominent protocols that the platform supports in the context of actual configuration samples. You'll also learn how connectors can help you interact with the cloud by looking at a simple Twitter example.

A second important feature of Mule is message transformation. Chapter 4 will show you how to take advantage of Mule transformers and how to create new ones.

Message routing is a crucial facet of enterprise message buses. We'll explore the advanced capacities of Mule in this domain in chapter 5.

Finally, we'll close this first part with chapter 6, which will focus on components and patterns, the places where message and business logic happens in Mule.

Discovering Mule



This chapter covers

- An introduction to enterprise integration
- Building, testing, and deploying your first Mule application

All it takes is a simple request: send this to Salesforce, publish that to Twitter, connect to the inventory mainframe. 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 speaks 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, protocols, and connectors. Welcome to the world of integration!

The IT landscape has been shaped by years of software evolution, business mergers, and third-party API integration, which 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 careers. As Michael Nygard, author of *Release It!* (Pragmatic Bookshelf, March 2007) states, "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.”

Developing integration applications encompasses a variety of difficulties:

- *Protocol*—Applications can accept input from a variety of means, ranging from a local filesystem to a RESTful API.
- *Data format*—Speaking the right protocol is only part of the solution, since applications can use almost any form of representation for the data they exchange.
- *Invocation styles*—Synchronous, asynchronous, RPC, messaging, and batch call semantics entail very different integration strategies.
- *Lifecycle and management*—Applications of different origins that serve varied purposes tend to have disparate development, maintenance, and operational lifecycles.
- *Error handling*—Error handling is crucial in any application and is amplified with applications that are forced to integrate with remote, and often unreliable, systems.
- *Monitoring*—Integration applications often have more esoteric monitoring requirements than a traditional web or server-side application. These include the monitoring of transactions per second, awareness of the latency of remote servers, and the absence of events, to name a few.

This book is about Mule, the leading open source enterprise integration platform, which will help you tackle these difficulties and much more. Mule frees you from much of the plumbing associated with enterprise application integration, allowing you to focus on your application’s core requirements.

In this chapter, you’ll gain a high-level understanding of Mule before we dive head first into building a real, complete Mule application. This will prepare you for the rest of the book, in which you’ll learn how Mule lets you focus your development effort on solving business problems instead of fighting low-level integration “donkey work.”

1.1 **Enterprise Integration Patterns and service-oriented architecture**

The last decade or so has seen a renaissance of application integration. Gone are the days of proprietary messaging stacks and closed APIs. Open platforms, protocols, and services dominate the landscape. Nothing is more evidence of this than the “API Explosion” of recent years as companies, organizations, and governments race to expose their data. REST, JSON, and lightweight message brokers lead the charge but, as always, don’t allow you to throw out what’s already in place (or more importantly, what already works). These nimble new technologies also don’t solve what is ultimately the bigger problem: how these services are composed into distributed applications.

Until Hohpe and Woolf's seminal publication of *Enterprise Integration Patterns* (Addison-Wesley, November 2003), there was little in the way of prescribed solutions to solve these, and many other, integration challenges. When developers of integration applications finally had a catalog of patterns, they were still left with little in the way of implementations. This is how Mule and many other open source and commercial integration frameworks received their cue. The integration developer was now freed from having to implement the patterns and could once again focus on the solutions.

A parallel phenomena to the publishing of *Enterprise Integration Patterns* was the emergence of service-oriented architecture. Service-oriented architecture, or SOA, is a software architecture style that acknowledges the need for integration up front by providing well-defined, programmatic means for interacting with an application. Initially embodied by the heavyweight SOAP specification and more recently refined by the widespread adoption of REST and JSON, SOA has become pervasive in the modern software development landscape.

Guerrilla SOA

Early SOA adoption was usually done with heavyweight integration technologies like SOAP, verbose XML, and the complicated infrastructures and tooling that come along with these technologies. Compounding this complexity was an unfortunately common “waterfall” approach to integration development, in which existing infrastructures were “converted” to SOA over a long period of time. The fate of many such projects is unfortunately obvious to anyone reading this book.

Guerrilla SOA, a concept introduced by Jim Webber (www.infoq.com/interviews/jim-webber-qcon-london), is the idea that service-oriented architecture can be introduced in a lean, incremental, and agile manner. We'll see in this book how Mule, along with lightweight messaging patterns, alleviates the pain of introducing SOA to your applications.

We'll cover Guerrilla SOA, as well as other architectural approaches, in chapter 7.

There is a natural, but unfortunate, tendency to integrate applications informally. This often leads to “spaghetti integration,” as illustrated by figure 1.1. Applications in such implementations are connected directly to each other in a “point-to-point” manner. On a small scale this might be OK. Pain quickly becomes apparent, however, as the number of integration points grows. The application becomes mission critical, and your remote systems begin to change.

A pattern described in *Enterprise Integration Patterns* that solves this problem received attention from the industry and community: the message bus, commonly

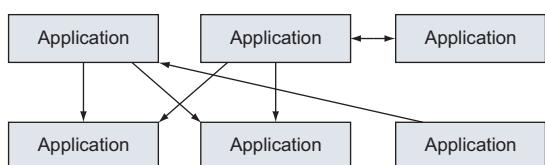


Figure 1.1 Point-to-point,
or spaghetti, integration

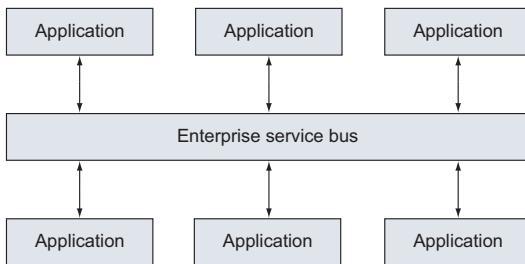


Figure 1.2 Application integration with an ESB

called an *enterprise service bus*, or ESB, when implemented. The ESB, depicted in figure 1.2, provides a solution to the problem of point-to-point integration. An ESB architecture prescribes placing a dedicated integration application, called a bus, in between all of your integration points. Your previous point-to-point integrations now all talk to the bus, which decouples them from the remote applications.

This decoupling is achieved by protocol adaptation and a canonical data format. Protocol adaptation means the bus can communicate over different transport protocols, like HTTP or FTP. A canonical data format is a common format all messages are transformed to, usually a common Java domain model or XML schema. This allows you to centralize concerns like security, auditing, and routing onto the ESB framework. It also means your client applications are insulated from the volatility typically present in integrating with remote applications. This provides the flexibility to do things such as swap out one vendor API for another without having to modify every downstream application.

Although Mule is often billed as an ESB, it's important to note that an ESB is an architecture and not a product. We'll discuss using Mule as an ESB, as well as in many other ways, in chapter 7.

1.2 **The Mule project**

The Mule project was started with the motivation to make life simpler for developers of integration applications. A major driver for the project was the need to build a lightweight, modular integration solution that could scale from an application-level messaging framework to an enterprise-wide, highly distributable enterprise services bus.

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, cofounder of MuleSoft

Mule's core is an event-driven framework combined with a unified representation of messages, expandable with pluggable extensions. These extensions provide support

for a wide range of transports or add extra features, such as distributed transactions, security, and management. Mule's developer-friendly framework offers programmers the means to graft on additional behavior such as specific message processing or custom data transformation. This philosophy has allowed Mule to quickly adapt to and support emergent trends in enterprise computing, such as NoSQL, distributed memory grids, and lightweight messaging protocols like AMQP and ZeroMQ.

This orientation toward software developers helps Mule to remain focused on its core goals and to carefully avoid entering the philosophical debate about the role of an ESB in an integration scenario. Although Mule is often billed as an ESB, and can be used in such a fashion, the framework makes no dictation on the architecture of your integration applications. 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 it into a solid and proven platform.

Mule 3, released in 2010, represented a significant departure from Mule 2 (on which the first edition of this book was based). The most noticeable differences reside in new configuration mechanisms that aim to simplify Mule configuration. Most specifically, the introduction of the “flow” construct frees the user from the rigid service-based configuration model of Mule 2. Flows allow the free composition of message processors, contributing to the prodigious simplification of the often-verbose XML configurations in Mule 2.

Mule Studio, introduced in 2012, further simplifies integration application development with Mule. Mule Studio is a graphical, Eclipse-based development environment. Among its features are drag-and-drop composition of flows, full XML round-tripping, and the ability to run Mule applications directly in the IDE or deploy them to a server or to the cloud.

Extending and developing for Mule has also been greatly simplified. Mule DevKit, which we'll cover in depth in chapter 13, makes it easy to write custom Mule components that fully integrate with the Mule ecosystem. Annotations have also been widely adopted by the framework, simplifying the development and testing of these components. Cloud connectors streamline integration with remote APIs and platforms. A new deployment model trivializes packaging and deploying Mule applications.

The configuration and developer simplifications in Mule 3 are complemented by incremental changes to the framework. Expression evaluation has been standardized by the Mule Expression Language. REST support is now native and is coupled with support for JSON.

1.3 **Competition**

The large enterprise players (IBM, Oracle, Red Hat, and so on) all have an ESB in their catalog. They are typically based on their middleware technologies and are 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 Java EE application servers, like the ones from Progress Software and Software AG.

MULE ENTERPRISE EDITION A commercially supported version of Mule with additional features and support options is supplied by MuleSoft. More details about Mule EE can be found in appendix C.

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 8)
- 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 capabilities as an integration framework open it to all sorts of other protocols

Mule is not the only available open source ESB. To name a few, major OSS actors such as Red Hat and Apache provide their own solutions. Spring also provides an integration framework built on their dependency injection container. Although most of these products use proprietary architectures and configurations, the integration products from the Apache Software Foundation are notably standards-focused: ServiceMix was previously 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 extensive support for WS-* standards.

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. This chapter will provide that. Now let's dive in head first and create a real, working Mule application.

1.4 **Mule: a quick tutorial**

To frame the examples in this book, we hereby introduce you to Prancing Donkey Maltworks, Inc. Prancing Donkey is a rapidly expanding, medium-sized, US-based microbrewery. Its small but competent development group has selected Mule to ease the integration pains as they grow.

Our tour of Mule will begin with a tutorial. You'll build an application to allow third parties to register products for sale on www.theprancingdonkey.com, Prancing Donkey's online store. This application will allow Prancing Donkey's partners to post product data, formatted as JSON, to an HTTP URL. Once the data is accepted, it will be transformed from its original format, a stream of bytes, into a String and placed in a JMS queue from which subsequent processing can take place. This tutorial will demonstrate common tasks you'll perform when building applications with Mule.

You'll start by creating a new Mule project for your application using Mule Studio. You'll then author an integration flow to process product data and test it with an embedded version of the Mule Server in Mule Studio. You'll formalize this test by writing a functional test to programmatically assert that your flow behaves the way you expect. Finally, you'll download the Mule Server and deploy your packaged application to it, demonstrating a typical lifecycle of building, testing, and deploying a Mule application.

1.4.1 **Installing Mule Studio**

Mule Studio can be downloaded from www.mulesoft.org. Once you download it, uncompress the archive and double-click on the Mule icon, and you'll be presented with a screen like the one in figure 1.3.

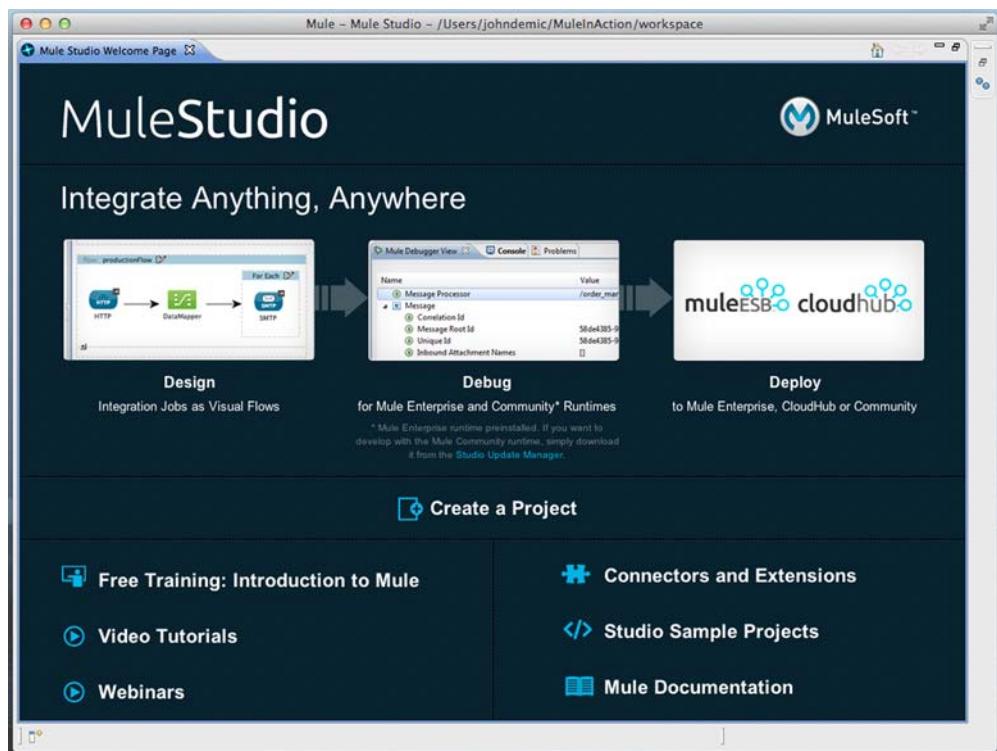


Figure 1.3 Launching Mule Studio

Before you can create a project, you'll need to install the Mule Community Runtime. To do this, click Help and then Install New Software. In the screen that follows, expand the drop-down list prefixed with Work With: and select Mule ESB Runtimes for Studio. Finally, select Mule ESB Server Runtime 3.4.0 CE, as illustrated in figure 1.4.

You can now click Create a Project to get started (figure 1.5).

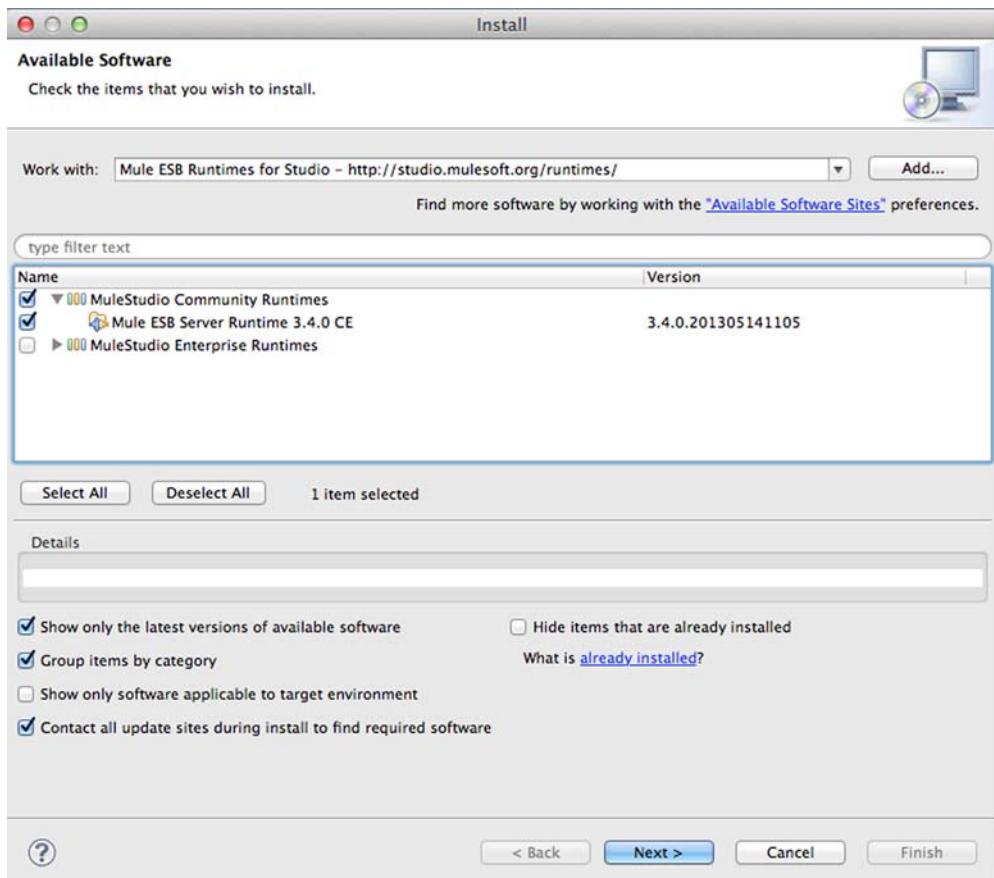


Figure 1.4 Installing the community runtime

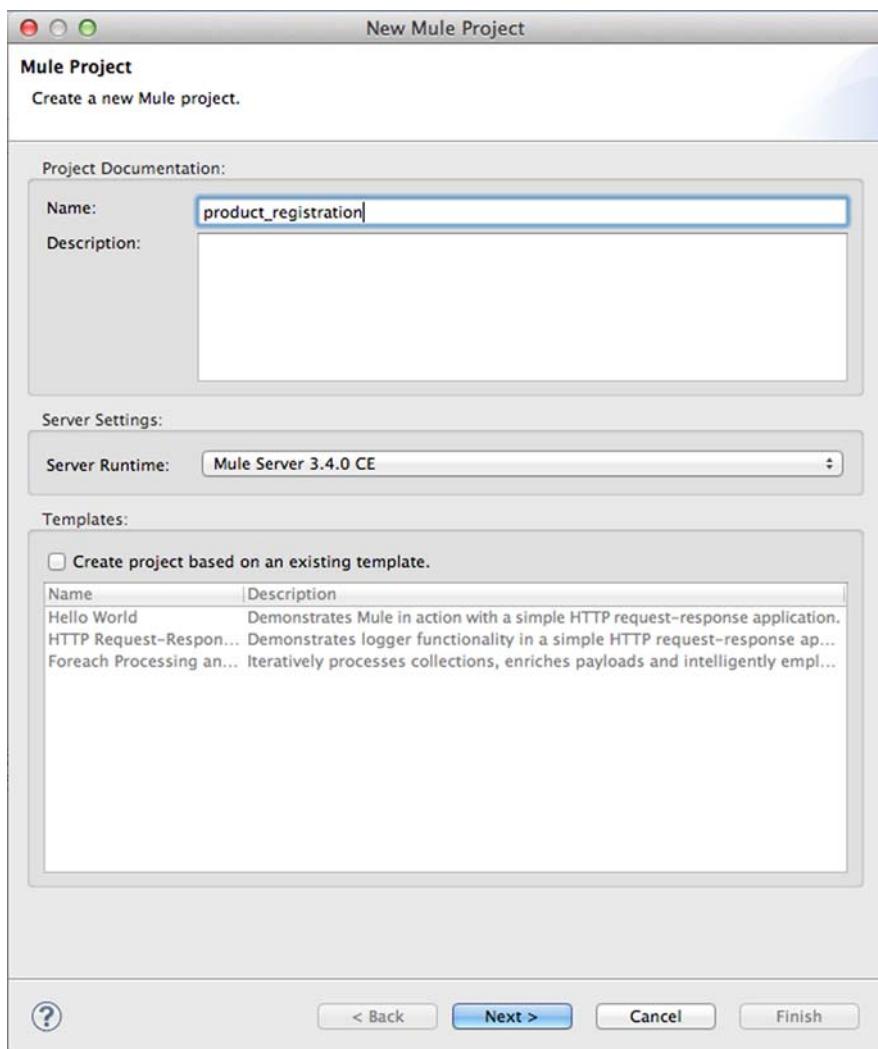


Figure 1.5 Creating a new Mule project

Here you set the project's name, give it a description, and select the Mule runtime—in this case, Mule 3.4.0 Community Edition, which is the most recent as of this writing.

Clicking Next again gives you the opportunity to enable Maven for this project. Skip this part, and the next few steps, by clicking Next, and then you'll click Finish to start authoring your flow (figure 1.6).

After you set the name and description of the flow, you can dive in and use Mule Studio to graphically define your integration.

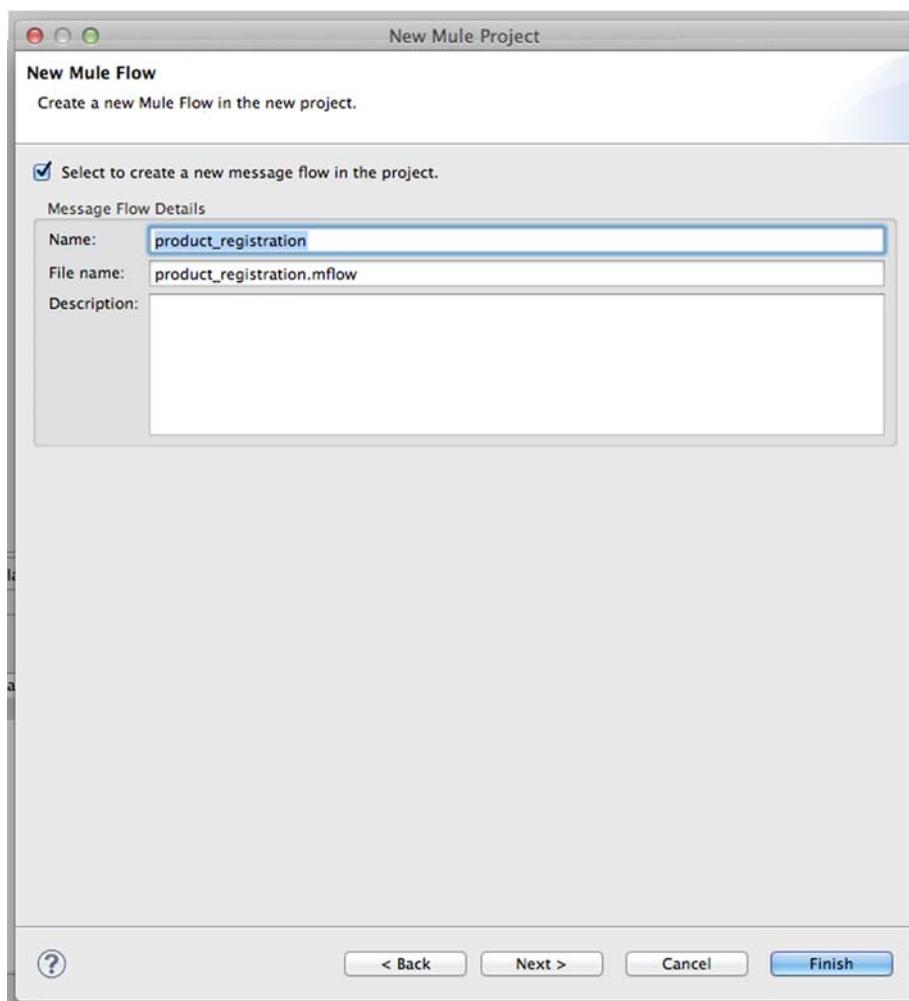


Figure 1.6 Defining the initial flow

1.4.2 Designing the flow

Flows are the primary mechanism for building integration applications with Mule. A flow consists of a source of data followed by a series of message processors. A message begins its life from an inbound endpoint, which could be an HTTP POST or the scheduled polling of a database table, and is processed by the subsequent processors in the flow in the order in which they are defined. Flows support multiple invocation styles as defined by their *exchange pattern*. A *one-way* exchange pattern typically means the flow is asynchronous, for instance. The *request-response* exchange pattern means the flow will return a result. A flow can optionally end with an outbound endpoint, which sends the message to another flow or server.

The palette on the right-hand side of the screen contains the library of endpoints and message processors you'll use to build your flows. Use the filter search box to find the HTTP endpoint and drag it into your flow. Your screen should now look something like figure 1.7.

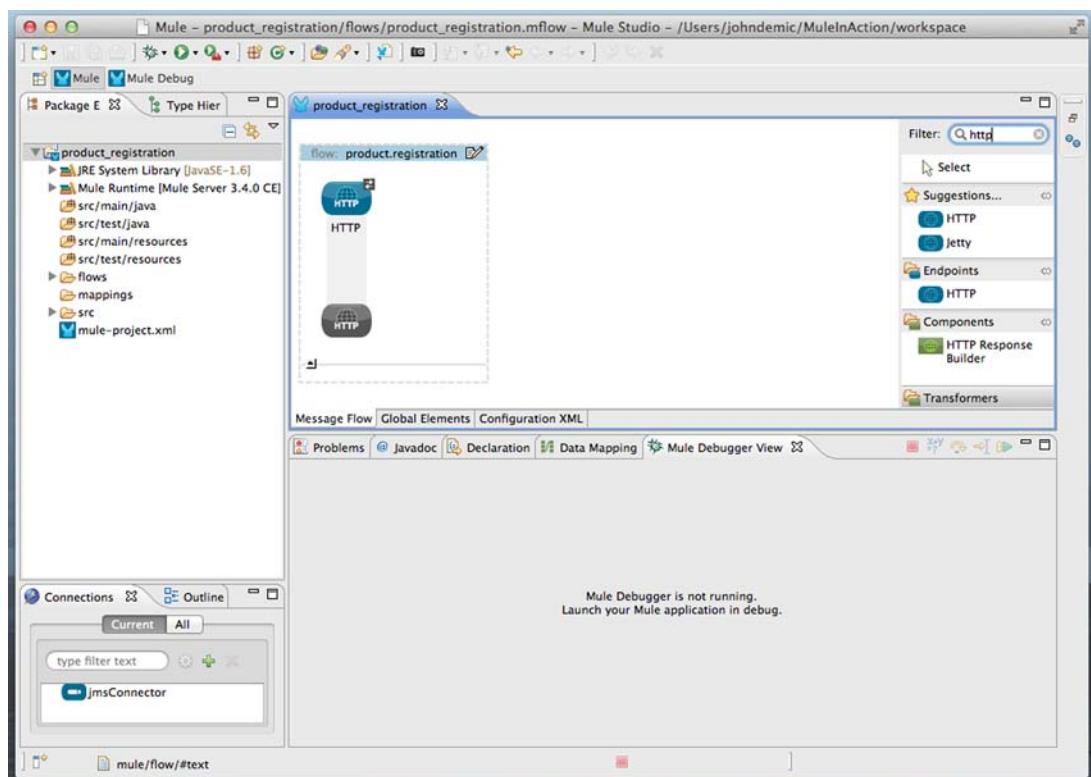


Figure 1.7 Dragging the HTTP inbound endpoint to the flow

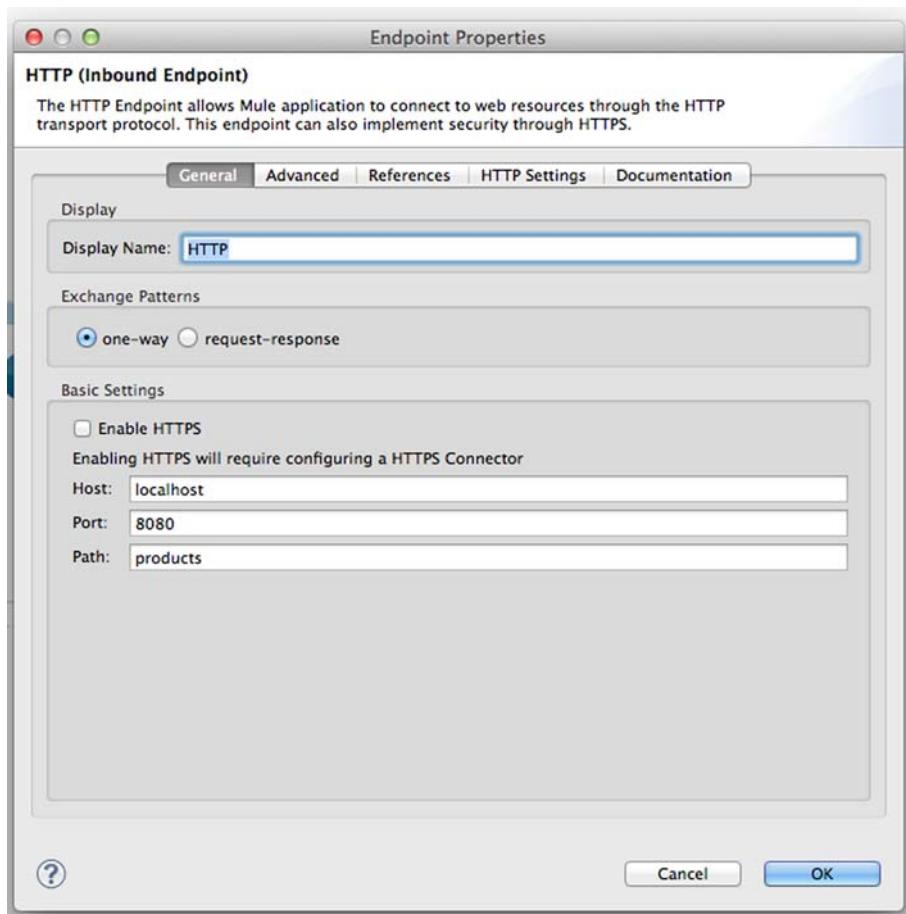


Figure 1.8 Configuring the HTTP endpoint's properties

The vertical orientation of the HTTP endpoint indicates that the flow's exchange pattern is request-response. Exchange patterns indicate if a flow returns a result or not. Let's configure the flow's exchange pattern to one-way, along with the host, port, and path the HTTP server will be listening on. You can do this by right-clicking on the HTTP endpoint, which will show you something that looks like figure 1.8.

You now need to add two more message processors to this flow: the byte-array-to-string transformer and the JMS outbound endpoint. The byte-array-to-string transformer is needed to transform the HTTP inbound endpoint's default payload type, a stream of bytes, to an instance of a String. You'll once again drag these from the library of processors on the right into your flow, leaving you with something that looks like figure 1.9.



Figure 1.9 Adding the byte-array-to-string transformer and JMS outbound endpoint to the flow

Now right-click on the JMS endpoint and define the queue to dispatch to (figure 1.10).

You might notice that the JMS endpoint has a red X on it. This is because you haven't configured a JMS broker for it to connect to. Let's configure it to use an instance of ActiveMQ, an Apache-licensed, open source messaging broker that supports JMS, running on localhost (you'll install ActiveMQ in a second). To do this you'll

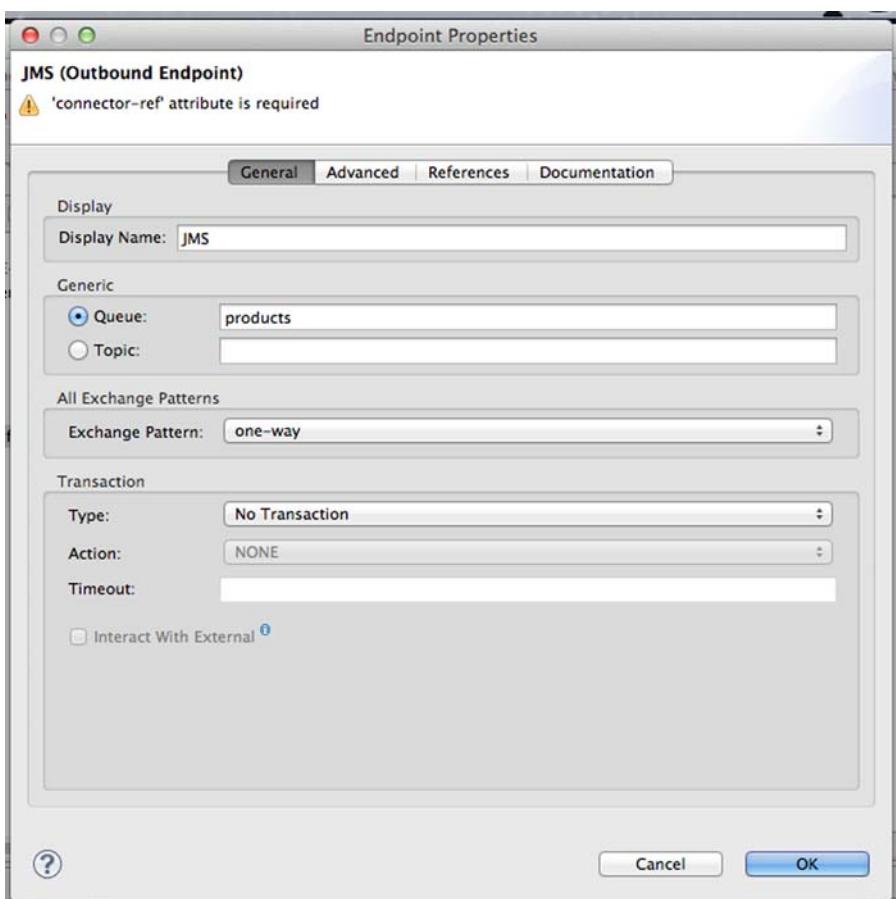


Figure 1.10 Configuring the JMS endpoint's properties

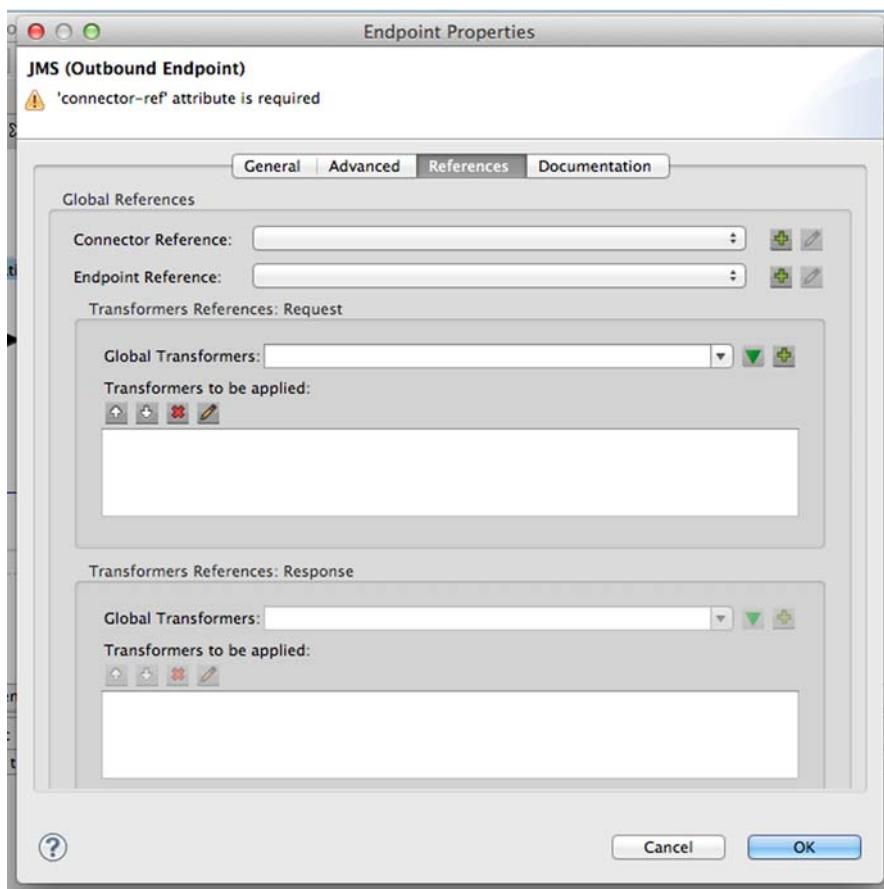


Figure 1.11 Configuring the JMS connector

need to right-click on the endpoint, select the References tab and click on the Plus icon, as illustrated in figures 1.11 and 1.12. You need to change the JMS spec to 1.1.

Now let's set up a local ActiveMQ instance that you can use to test with. Download ActiveMQ (<http://activemq.apache.org/download-archives.html>; we tested with version 5.5.1), uncompress the file, navigate into the bin directory, and then run activemq start.

Let's take a step back and consider what you've done. Products will be posted to the HTTP endpoint as JSON. The byte-array-to-string transformer converts the raw bytes of the HTTP POST to a String. We'll discuss transformers in depth in chapter 4. The JMS outbound endpoint then finally dispatches the String to the specified JMS queue. Endpoints are Mule's mechanism for getting data into and out of flows. We'll discuss both in detail in chapter 3.

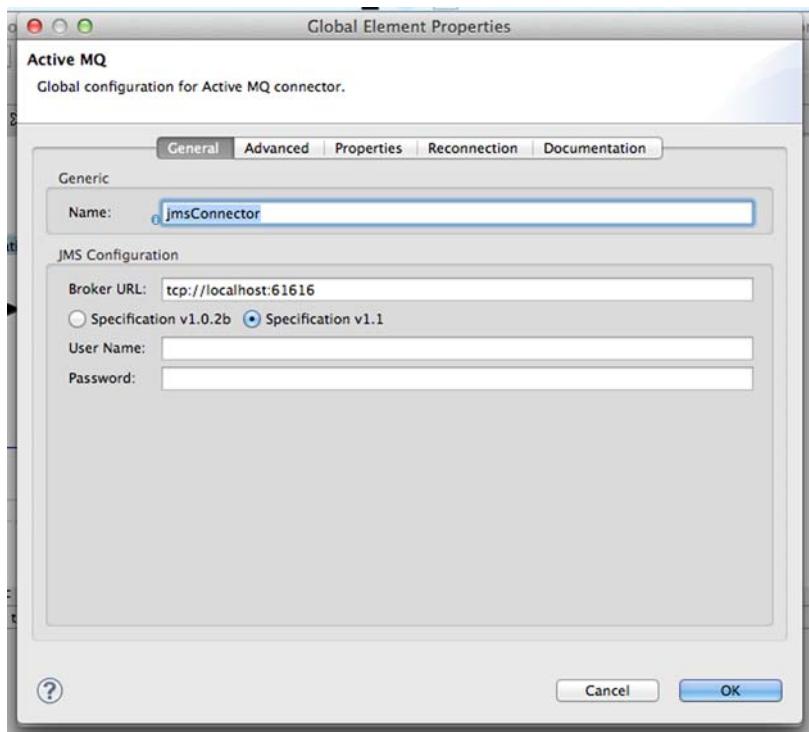


Figure 1.12 Configuring the ActiveMQ connector

1.5 **Running, testing, and deploying the application**

We just finished developing our first Mule application as well as setting up a messaging broker for it to interact with. Now let's see how we can run, test, and deploy the application.

1.5.1 **Running the application**

Before you can run the application, you'll need to add the JAR for ActiveMQ to your project. The procedure is identical to adding a JAR to any Eclipse project. Right-click on Mule Runtime in the Project Explorer pane on the left, select Build Path, and then select Configure Build Path. At this point, you'll be presented with a screen like that in figure 1.13, in which you can add the ActiveMQ JAR to the project.

Now you're ready to run your application. Before you do that, however, let's modify your flow to log some output to the console. This will give you some visual feedback that things are behaving properly.

You can do this by selecting a logger and dragging it into the flow after the byte-array-to-string transformer and before the JMS outbound endpoint. You'll set the message attribute to print a String followed by the payload of the message, using the Mule Expression Language. The Mule Expression Language, or MEL, is a lightweight scripting language that's evaluated at runtime. The payload, in this case, will be the JSON content of the data being sent to Mule. This is illustrated in figure 1.14.

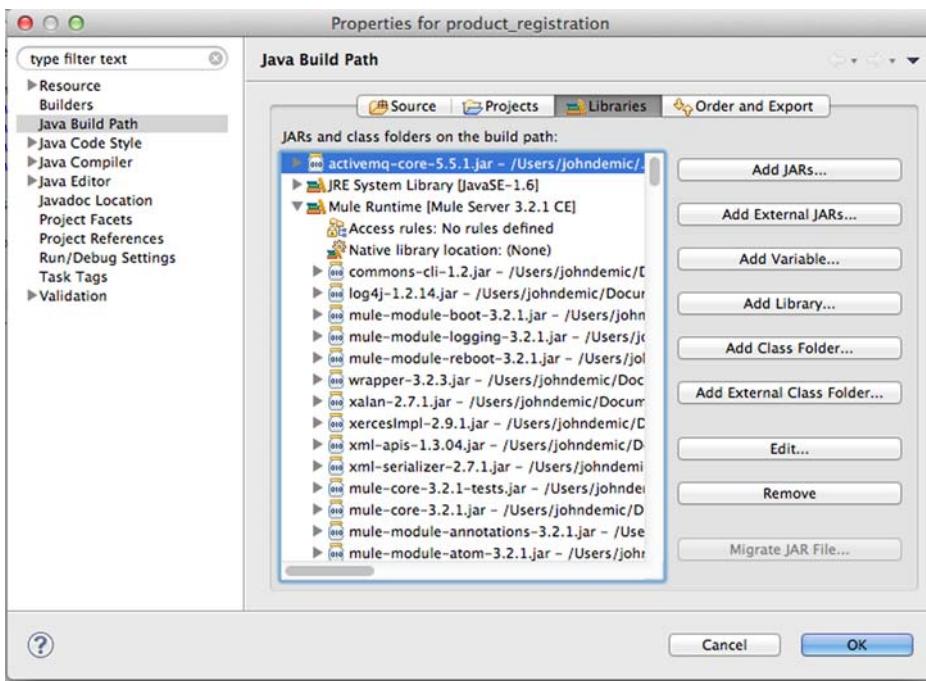


Figure 1.13 Adding a JAR

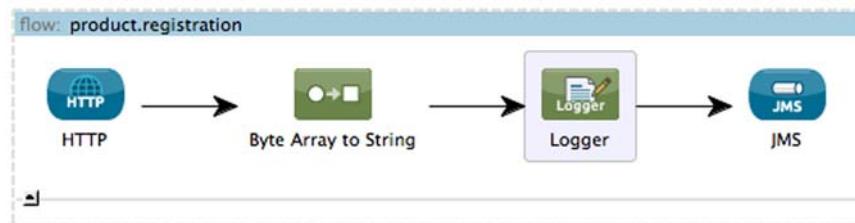


Figure 1.14 Adding a logger to the flow

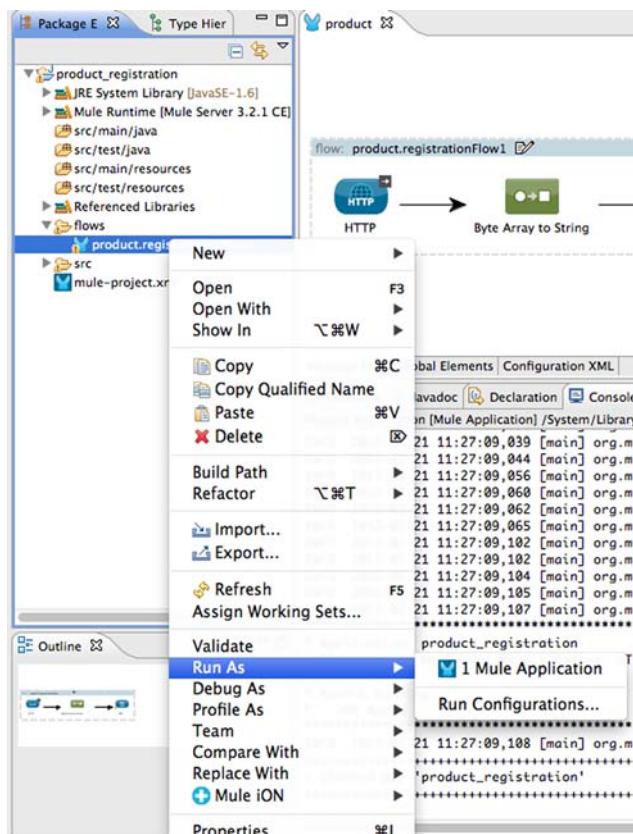


Figure 1.15 Running the application

Right-clicking the project on the Project Explorer page and selecting Run As and then Mule Application, as illustrated in figure 1.15, will launch the app in an embedded Mule instance. You should see something like figure 1.16 in your console, illustrating the app is running.

The screenshot shows the Eclipse Console view with the title bar "Mule Properties View Problems Console". The main area displays the following log output:

```

chapter01 [Mule Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (Nov 18, 2013 10:05:36 PM)
INFO 2013-11-18 22:05:39,629 [main] org.mule.module.management.agent.JmxAgent: Registered Endpoint Service with no
Ichapter01 [Mule Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/ org.mule.module.management.agent.JmxAgent: Registered Connector Service with r
IJava/JavaVirtualMachines/1.6.0.jdk/Contents/ org.mule.module.management.agent.JmxAgent: Registered Connector Service with r
1Home/bin/java (Nov 18, 2013 10:05:36 PM) org.mule.DefaultMuleContext:
*****
* Application: chapter01 *
* OS encoding: MacRoman, Mule encoding: UTF-8 *
*
* Agents Running:
*   Clustering Agent *
*   JMX Agent *
*****
INFO 2013-11-18 22:05:39,633 [main] org.mule.module.launcher.MuleDeploymentService:
+-----+
+ Started app 'chapter01' +
+-----+

```

Figure 1.16 Examining the console output

After posting some JSON data using a tool such as curl or Rest Console to `http://localhost:8080/products`, you should see something like the following logged to the console:

```
INFO 2013-07-09 08:17:43,468
[[chapter01].connector.http.mule.default.receiver.02]
org.mule.api.processor.LoggerMessageProcessor:
We received a message: {"name": "Widget",
    "price": 9.99,
    "weight": 1.0,
    "sku": "abcd-12345"}
```

The logger shows that your byte-array-to-string transformer has successfully transformed the `InputStream` to a String. Now let's check ActiveMQ's console to make sure the message is in the queue. Point a browser at `http://localhost:8161/admin/queues.jsp`, and you should see one message in the products queue, similar to what's shown in figure 1.17.

Now that you've manually verified that the flow works, let's write a functional test so that you can programmatically verify that it works.

The screenshot shows a web browser window titled "localhost : Queues". The URL in the address bar is "localhost:8161/admin/queues.jsp". The page header includes the ActiveMQ logo and the Apache Software Foundation logo. The main content area is titled "Queues" and displays two rows of queue information:

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
incoming.widgets	0	0	0	0	Browse Active Consumers Atom RSS	Send To Purge Delete
products	1	0	1	0	Browse Active Consumers Atom RSS	Send To Purge Delete

On the right side of the page, there are three sidebar sections: "Queue Views" (Graph, XML), "Topic Views" (XML), and "Useful Links" (Documentation, FAQ, Downloads, Forums). At the bottom of the page, there is a copyright notice: "Copyright 2005-2011 The Apache Software Foundation. ([printable version](#))" and "Graphic Design By Hiram".

Figure 1.17 Looking at the products queue

1.5.2 Testing the flow

You're now ready to write a test for the flow you've written. Create a class called `ProductServiceFunctionalTestCase`. You can do this by right-clicking on `src/test/java` in the Package Explorer on the right side of the screen, then selecting New, and then Class. Modify the newly created class to look like the following listing.

Listing 1.1 Extending FunctionalTestCase to ensure that your configuration works

```
public class ProductRegistrationFunctionalTestCase
    extends FunctionalTestCase {
    protected String getConfigResources() {
        return "./src/main/app/product_registration.xml";
    }
    @Test
    public void testCanRegisterProducts() throws Exception {
        MuleClient client = muleContext.getClient();

        String productAsJson = "{\"name\":\"Widget\", \"price\": 9.99,
            \"weight\": 1.0, \"sku\":\"abcd-12345\"}";
        client.dispatch("http://localhost:8080/products",
            productAsJson, null);

        MuleMessage result = client.request("jms://products",
            RECEIVE_TIMEOUT);
        assertNotNull(result);
        assertNull(result.getExceptionPayload());
        assertFalse(result.getPayload() instanceof NullPayload);
        assertEquals(productAsJson, result.getPayloadAsString());
    }
}
```

Sample product data represented as JSON **1**

POST the JSON to <http://localhost:8080/products> **2**

Wait for JMS message to arrive on products queue **3**

Assert that response isn't null **4**

Assert that payload of JMS message matches payload of HTTP POST request **5**

We'll discuss testing in detail in chapter 12, but let's do a quick rundown of the previous listing to get a preview. The provided `TestCase` extends `FunctionalTestCase`, a base class provided by Mule that abstracts the details of starting and stopping a Mule instance from your tests. The method's abstract `getConfigResources()` method points the test case at the configuration to use to bootstrap Mule.

Mule's testing framework uses JUnit 4, as you can probably tell by the `@Test` annotation on `testCanRegisterProducts`. You'll use the `MuleClient`, a facility described in depth in chapter 12, to interact with the Mule flow programmatically. Your test data, in this case some simple JSON, is defined at **1**. You POST the JSON to the HTTP endpoint **2**, and then wait for it to appear on the JMS queue **3**. Once you either receive the message or `RECEIVE_TIMEOUT` is met, your assertions, starting with **4**, are run. You first do a series of assertions that ensure your response or payload isn't null and that an exception wasn't thrown during processing. You then assert that the payload on the JMS queue matched the data you posted to the HTTP endpoint **5**.

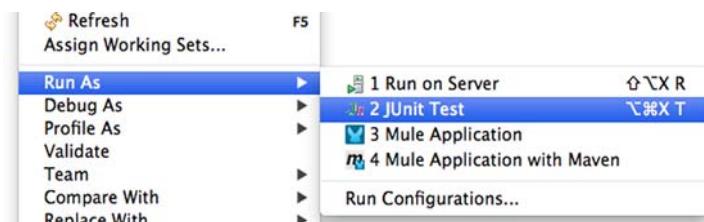


Figure 1.18 Running the FunctionalTestCase

Go ahead and run the test by right-clicking on the test case and selecting Run As and then JUnit Test. This will start an instance of Mule that your test will be executed against. If everything went well, you should see a green bar indicating that the test passed (see figure 1.18).

1.5.3 Working with the XML configuration

It's possible to develop extremely sophisticated integration applications using the graphical editor of Mule Studio. Eventually, however, you'll probably need to at least look at, if not edit, the Mule XML configuration that's generated by Mule Studio. This not only enables you to understand what your flows are doing behind the scenes, but also gives you the ability to fully harness Mule's power as an integration framework.

STUDIO AND XML EXAMPLES The examples in this book will mostly focus on the XML configuration of flows. We will, however, show the corresponding screenshots of the flows in Mule Studio where it makes sense. It's also important to note that no functionality in the CE version of Mule is dependent on Mule Studio. Your choice of IDE, or your choice to use no IDE, is largely irrelevant when working with Mule applications.

Clicking on the Configuration XML tab below the flow will show you the Mule configuration XML that corresponds to this application. The next listing shows what the XML looks like for the flow you developed.

Listing 1.2 The product registration flow

```
<mule xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http="http://www.mulesoft.org/schema/mule/http"
      xmlns:jms="http://www.mulesoft.org/schema/mule/jms"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/jms
          http://www.mulesoft.org/schema/mule/jms/current/mule-jms.xsd
          http://www.mulesoft.org/schema/mule/http
          http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
          http://www.mulesoft.org/schema/mule/http
          http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
          http://www.mulesoft.org/schema/mule/jms
          http://www.mulesoft.org/schema/mule/jms/current/mule-jms.xsd"
```

1

Namespace definitions

```

http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-current.xsd
http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/current/mule.xsd "
version="CE-3.4.0">

<description>
    Mule Application to REST-fully Accept Product Data
</description>

<jms:activemq-connector name="jmsConnector" doc:name="Active MQ" /> Configure a JMS connector to use ActiveMQ

Name of the flow <flow name="product.registration" doc:name="product.registration">
    <http:inbound-endpoint host="localhost"
        port="8080"
        path="products" doc:name="HTTP" />
    <byte-array-to-string-transformer
        doc:name="Byte Array to String"/>
    <jms:outbound-endpoint queue="products"
        connector-ref="jmsConnector"
        doc:name="JMS" /> Transforms the POST data from a byte array to a String
An HTTP inbound endpoint to receive a POST request with product data </flow>
</mule> Dispatch JMS messages to a queue called products

```

The first thing to notice about this configuration file is the declaration of namespaces ①. These namespaces implement the XML domain-specific language used by Mule's XML configuration. The flow and its message processors' configurations follow. These map one-to-one to the elements you dragged in the graphical view.

AUTOMATIC NAMESPACE IMPORTS Users of Mule 2 will be happy to note that Studio automatically imports namespace declarations for you, freeing you from having to manually add them every time you introduce a new transport or module into your Mule application.

1.5.4 Deploying to the Mule standalone server

Confident that your application is in good shape thanks to your testing, you're now ready to deploy. Before that, however, you need a running Mule server. You can download the Mule standalone server from www.mulesoft.org/download-mule-esb-community-edition. You'll want to download the Mule ESB standalone runtime (without Mule Studio). Uncompress the file, navigate into the bin directory, and run the mule executable to start a standalone instance of the Mule server.

Now you'll use Mule Studio to build your deployment artifact. First you need to export the application as a Mule deployable archive. You can do this by right-clicking on the project, selecting Export, selecting Mule Studio Project to Mule Deployable Archive (includes Studio metadata), and then setting the path to where you want the ZIP file, Mule's deployment artifact, saved. This is illustrated in figures 1.19, 1.20, and 1.21.

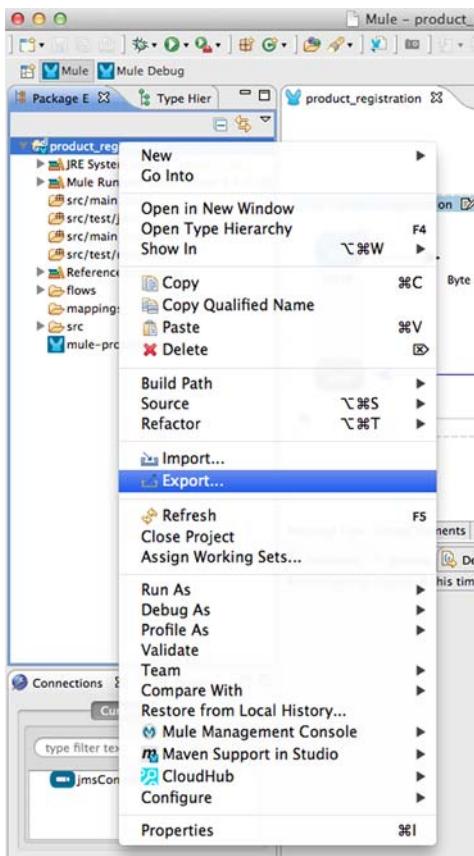


Figure 1.19 Choose to export the application.

Now that you've exported the Mule application as a ZIP file, you can deploy it.¹ This is accomplished by copying the ZIP file to the apps directory of wherever you installed the Mule standalone server. After a few seconds, you should see something like the following appear on the console on which Mule is running:

```
INFO 2011-12-19 10:01:07,741 [Mule.app.deployer.monitor.1.thread.1]
org.mule.module.launcher.DeploymentService:
+++++
+ Started app 'productservice-1.0-SNAPSHOT'
+
+++++
```

¹ We'll discuss Mule deployment options in detail in chapter 8.

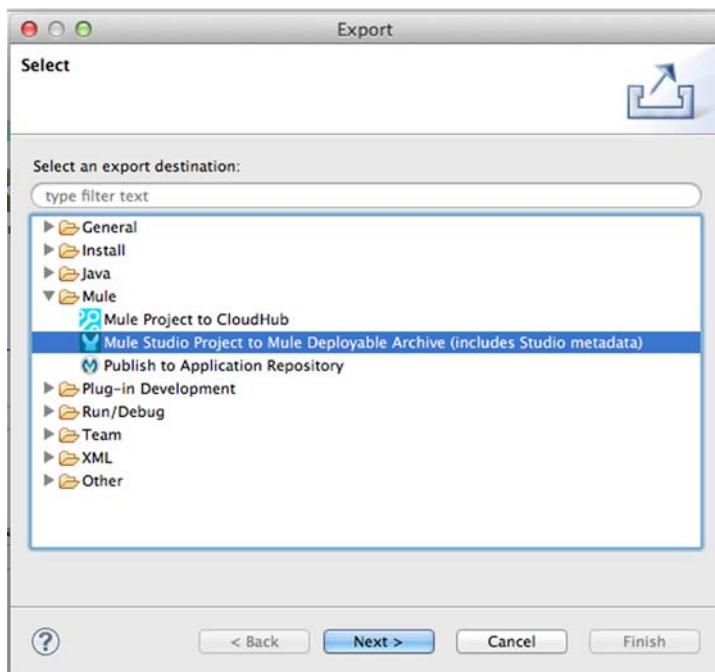


Figure 1.20 Choose the format to export to.

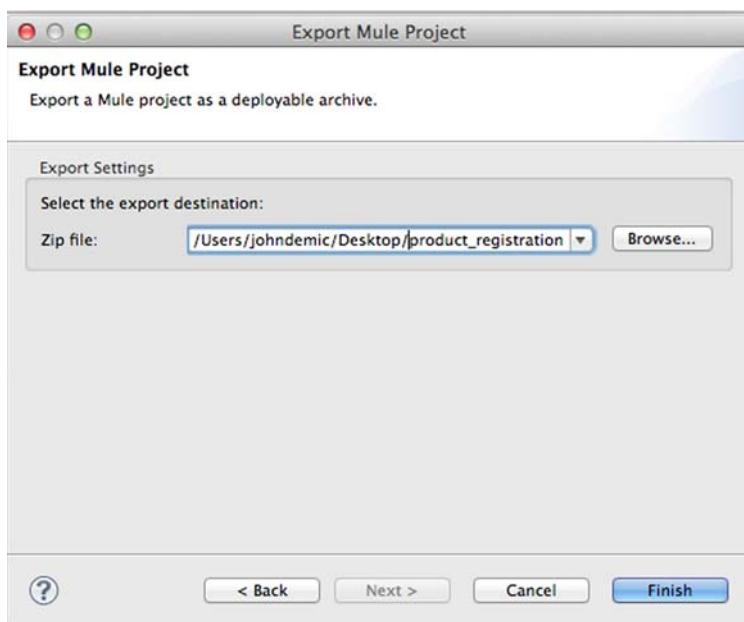


Figure 1.21 Choose where to save the resulting ZIP file.

That's it! You've developed a functional, well-tested, integration application with around 20 lines of XML and a small amount of Java. Hopefully you can appreciate the relative difficulty of implementing a similar approach without Mule; you would need to do the following:

- Bootstrap a web server to accept HTTP requests
- Configure and manage the JMS connection factories, sessions, and so on
- Figure out a way to functionally test the application
- Decide how to package and deploy the application

That's a lot of time being wasted writing code that isn't solving your goal: bridging an HTTP request to JMS. Throughout this book, you'll see numerous examples of how Mule simplifies and speeds up common integration tasks.

1.6 **Summary**

By this point you've received a primer on enterprise integration, learned about the philosophy and features of Mule 3, and written, tested, and deployed a working Mule app. We're now ready to formally begin our discussion of Mule; next, we'll discuss processing messages with Mule.



Processing messages with Mule

This chapter covers

- What role flows play in Mule
- How messages are created
- The structure of Mule messages
- The Mule Expression Language

Mule is a workhorse whose sole purpose in life is to move your messages around. It actually does way more than just moving messages: it's also able to transform, enrich, and smartly route them. Picture a mail delivery service that would automatically rewrite letters in the preferred language of the recipient, while decorating them with illustrations that appeal to the culture of the addressee.

Where does such capacity come from? The answer is two words: *control* and *abstraction*.

Mule gives you complete control over the way messages flow through it. We said the word: Mule indeed uses *flows* as the main control structure in its configuration files. You've already seen a flow in the very first chapter of this book; it was used by Prancing Donkey to accept product registration data. In this chapter, you'll learn

more about flows and how they work, and the kind of control they give you when processing messages.

Mule also uses a set of abstractions that allows users to deal with complex message processing scenarios in a unified manner using consistent and generic artifacts. In this chapter, we'll look into the abstractions that sit at the core of Mule, namely:

- The Mule message
- Message sources and processors
- Exchange patterns
- Endpoint URIs

We'll take the time to delve deep into these abstractions, how they operate, and why they're important. We'll also look at how these abstractions are put in motion inside flows and how the Mule Expression Language dynamizes configurations. For this, we'll look at more examples from Prancing Donkey's systems, including their email order handler and part of their invoicing chain.

The journey may feel arid at times, but it's a necessary voyage, so grab a fresh bottle of Prancing Donkey Pale Ale, relax, and get ready for the ride!

In Mule, things start moving when they get assembled in a flow. So let's just go with it...

2.1 Going with the flow

Flows are the foremost elements of a Mule configuration. They typically start with a message source (discussed in section 2.2.1) followed by message processors (discussed in section 2.2.2), all chained together by the sole virtue of being encompassed by the flow element. Flows impose virtually no limit to the type of message processors that can be added in them or in what order they can be added.

Let's circle back to the product registration flow you've already seen in section 1.4.2. Listing 2.1 shows this flow with a slight modification that you, astute reader, have certainly already spotted: a logger element has been added to allow Prancing Donkey to perform some activity auditing.

Listing 2.1 The product registration flow, now with logging

```
<flow name="product-registration">
    <http:inbound-endpoint
        address="http://api.prancingdonkey.com/products"
        method="POST" />
    <byte-array-to-string-transformer />
    <logger level="INFO" category="products.registration" />
    <jms:outbound-endpoint queue="products" />
</flow>
```

The same flow represented in Mule Studio would look like figure 2.1.

So where's the message source? And what are the message processors? Let's represent this flow slightly differently and make the difference between message sources and



Figure 2.1 The product registration flow in Mule Studio

processors more apparent. Figure 2.2 uses ovals for message sources and rectangles for message processors. We'll use the same representation in upcoming similar diagrams.

In a flow, messages are produced by the message source and then processed along the top-down path, going from message processor to message processor. Following the message processors along this path gives you a good idea about how requests will be handled in a flow. Later in this chapter, and further in the book, you'll see that this path can be altered by message processors. For example, routing message processors (discussed in chapter 5) allows you to exert even more control on the paths followed by messages.

EXCEPTIONAL CIRCUMSTANCES Another important aspect of controlling message flows resides in exception handling. You'll learn how to define alternate flow paths when exceptions occur in chapter 9.

If a message processor in a flow returns `null`, processing will be stopped right away. The response phase, discussed in the next section, won't even be fired. Note that regular transformers or components can't return `null`, but instead return a `NullPayload`,¹ which doesn't affect the flow execution.

Each flow represents a meaningful unit of work in your overall message processing needs. For example, receiving messages from a source, transforming them to a canonical form, and then sending them to another flow via an outbound endpoint for subsequent processing is a typical unit of work you'll naturally roll out in one flow.

Mule offers specialized flow elements, named configuration patterns; these patterns are preconfigured flows designed to perform very specific tasks. Configuration patterns will be discussed in section 6.2.

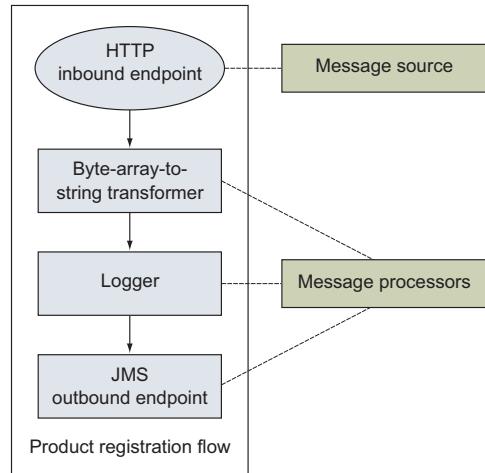


Figure 2.2 A symbolic representation of the product registration flow that highlights its message source and processors

¹ Mule takes care of returning a `NullPayload` even if the transformer or component has returned `null`. This doesn't apply to custom message processors.

SERVICES = FLOWS + CONSTRAINTS Mule 2-style services are flows with extra constraints added that severely limit the types of message processors that can fit in them. Services have been kept in Mule 3 for backward compatibility. They are now legacy and shouldn't be used anymore.

So far we've considered flows in one direction only: the top-down path. This path is the one taken by a message that comes into Mule and gets processed. But what about the opposite way? Let's look at how flows behave when it comes to building and delivering responses.

2.1.1 The response phase

Understanding the response phase of flows is crucial to mastering the art of moving messages around Mule. We'll take our time and do a deep dive into this concept so that you feel confident with what's happening inside your flows.

The first thing to realize is that the response phase is implicit; it's always present and, by default, consists of a simple echo of the message as it is at the end of the request phase, which is the top-down path in the flow. Though not apparent, the flow shown in listing 2.2 (used by Prancing Donkey for exposing their Accounts REST resource) has a response phase.

Listing 2.2 A flow with an implicit response phase

```
<flow name="accountService">
    <http:inbound-endpoint exchange-pattern="request-response"
        host="localhost"
        port="8080"
        path="services" />
    <jersey:resources>
        <component class="com.prancingdonkey.resource.Accounts" />
    </jersey:resources>
</flow>
```

Mule Studio makes this response phase very apparent. Look at figure 2.3; the bottom part of the flow where arrows are pointed left (back to the HTTP-inbound endpoint) represents the response phase.

The payload and properties used for the response are what they were in the Mule message when it hit the end of the flow (in top-bottom order). When a flow contains branching (like with a choice router; see section 5.1), it actually ends up having not a single but several possible endings, one at the end of each routing branch.

With this in mind, the second thing to consider is that Mule gives you the capacity to hook into the response phase with a specific configuration element appropriately named *response*. This allows you to add in a flow message processor that will only be called during the response phase. What can

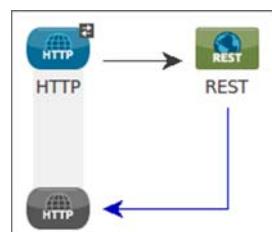


Figure 2.3 Mule Studio clearly shows the response phase.

be a little confusing about elements in the response phase is that they kick-in in reverse order, from the bottom of the flow to its top (or more accurately, from where the flow ended back to where it started).

Take a look at the configuration in listing 2.3; can you guess what the response will be if we send a message to it with a payload of “hello”?

Listing 2.3 A flow with elements explicitly in the response phase

```
<flow name="explicitResponse">
    <vm:inbound-endpoint path="input"
        exchange-pattern="request-response">
        <response>
            <append-string-transformer message=" web" />
        </response>
    </vm:inbound-endpoint>
    <response>
        <append-string-transformer message=" wide" />
    </response>
    <append-string-transformer message=" world" />
</flow>
```

The right answer is “hello world wide web.” As expected, the string appenders have been hit in the following order: “world” during the request phase, then “wide” and “web” during the response phase. Notice how we’ve placed response elements in the main flow and in an endpoint.

READABILITY FIRST Locate response elements where it makes the most sense when reading the flow.

Response elements allow you to perform all the necessary processing to ensure that the message that will be returned to the caller of the flow has the expected content. You’ll typically perform message transformations in the response phase in order to prepare for delivery the payload that has been created by components or outbound endpoint interactions.

Finally, the last thing to consider is that the outcome of the response phase is ignored if the flow response is returned via reply-to routing. In that case, it’s the state of the message at the end of the request phase that’s used as the flow response. The message processors of the response phase are executed, but their outcome will be silently discarded.

That was quite a chunk of information to swallow! Now that you’re starting to grok the dynamics of flows, let’s take a look at how flows can be modularized.

2.1.2 Subflows

Copy-pasting is the bane of software development. If you reach the point where you start to have a lot of commonality between your flows, you should quickly feel the need to extract and reuse common sequences of message processors across your flows. Enter subflows, a configuration construct that has been created for this very purpose.

A subflow behaves like a standard flow, but without a message source. It receives messages to process only when explicitly invoked by name via a `flow-ref` element. You can think of a subflow as a macro that contains a predefined set of message processors and which you can invoke on demand.

When a flow invokes a subflow, the whole Mule message (payload, properties, attachments) and its context (session, transactions) are passed to the subflow. Similarly, the complete context of the result of the subflow is passed back to the main flow. Actually, everything behaves as if the message processors in the subflow were in the main flow. It's also important to note that the caller thread is used to execute the subflow.

A CHAIN WITH A NAME Conceptually, a subflow is a processor-chain that's named, can live outside of a main flow, and can be invoked by any flow. It's very similar to a macro in your favorite office productivity suite.

Listing 2.4 illustrates this by showing how Prancing Donkey leveraged subflows to share a common message transformation chain in the context of a migration strategy, which impacts many flows. In the listing, you can see how they placed a set of transformers in a subflow and reused them in two different flows. As expected, messages going through these flows will be transformed by the set of transformers as if they were directly configured in each of them.

Listing 2.4 Using a subflow to share a common set of transformers

```
<sub-flow name="legacyAdapterSubFlow">
    <mulexml:xslt-transformer xsl-file="v1_to_v2.xsl" />
    <mulexml:xml-to-object-transformer />
</sub-flow>

<flow name="mainFlow1">
    <http:inbound-endpoint host="localhost"
                           port="8080"
                           path="v1/email" />
    <flow-ref name="legacyAdapterSubFlow" />
    <component class="com.prancingdonkey.service.v2.EmailGateway" />
</flow>

<flow name="mainFlow2">
    <jms:inbound-endpoint queue="v1.email" />
    <flow-ref name="legacyAdapterSubFlow" />
    <component class="com.prancingdonkey.service.v2.EmailGateway" />
</flow>
```

The same code is shown as represented by Mule Studio in figure 2.4.

This is pretty straightforward. Oftentimes, you'll face a situation where reuse is needed, but there isn't a clear-cut set of processors you can extract and share. Bear in mind that flow variables (a.k.a. invocation-scoped message properties, discussed in section 2.3.1) can help enable reuse because they can act as flow-control variables, introducing the dynamic part that may be missing in your original configuration.

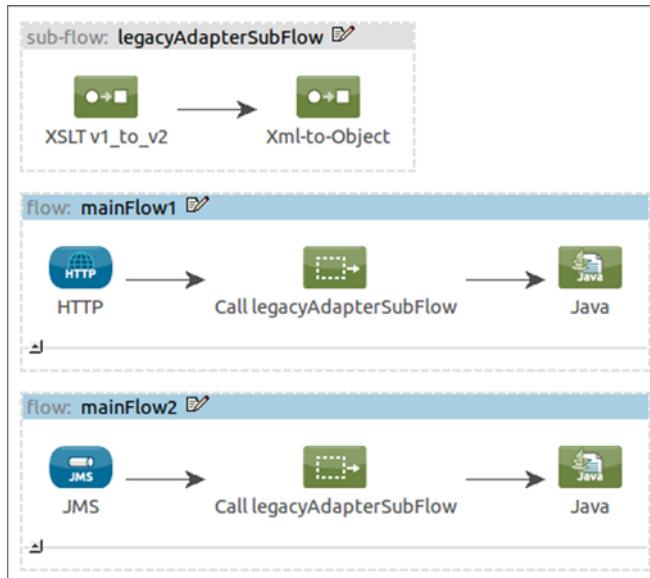


Figure 2.4 Two flows and a subflow shown in Mule Studio

Let's illustrate this with an example. Consider the subflow in listing 2.5. It relies on the preexistence of a custom flow variable named `valid` to direct the message to one path or another in the choice router. It's the responsibility of each parent flow to set this message property based on the specific validity rules they enforce. Everything else has been put in common in the shown subflow.

Listing 2.5 Flow variables can be used to parameterize a subflow.

```

<sub-flow name="requestDispatcher">
    <choice>
        <when expression="#[valid]">
            <vm:outbound-endpoint path="valid.request.handler"
                exchange-pattern="request-response" />
        </when>
        <otherwise>
            <vm:outbound-endpoint path="invalid.request.handler"
                exchange-pattern="request-response" />
        </otherwise>
    </choice>
</sub-flow>

```

If you have experience with Mule before version 3, you have certainly used a lot of VM endpoints to tie services together (the VM transport will be discussed in section 3.8). Should subflows be used all the time, or does tying together flows with VM endpoints still make sense? A big differentiator between the two approaches is that a subflow invocation carries all the message properties along while a VM endpoint message exchange requires copying properties across scopes. On the other hand, using a VM queue to connect flows introduces decoupling that can be useful in some situations.

Mule offers another option for sharing flows called *private flows*. Let's see how they differ from subflows and when it's a good time to use them.

2.1.3 Private flows

Private flows are another type of reusable flows, very similar to subflows, but with a different behavior in terms of threading and exception handling. The primary reason for using a private flow instead of a subflow is to define in it a different exception strategy from the calling flow (something that is impossible with a subflow). Another reason is that subflows aren't materialized at runtime and, as such, don't have specific statistics or debug properties attached to them and can't be controlled or monitored independently. Private flows offer all this.

As we said in the previous section, when a subflow is invoked, the execution behaves as if the message processors of the subflow were actually located in the calling flow. With a private flow, the execution behaves as if a message was passed from the calling flow to the private flow. This decoupling allows for defining processing and error handling strategies that are local to the private flow.

So what does a private flow look like? Basically it's just a regular flow, but without any message source. To illustrate this, let's revisit the subflow used by Prancing Donkey to transform messages from an old format to a new one (see listing 2.4). It happened that for a small subset of messages, the transformation was failing; they then decided to configure an exception strategy local to the transformation in order to send failed messages to a JMS queue for later analysis and reprocessing. For this, they had to convert their subflow into a private flow as shown in the next listing.

Listing 2.6 Using a private flow to define a local exception strategy

```
<flow name="legacyAdapterPrivateFlow" processingStrategy="synchronous">
    <mulexml:xslt-transformer xsl-file="v1_to_v2.xsl" />
    <mulexml:xml-to-object-transformer />
    <catch-exception-strategy>
        <jms:outbound-endpoint queue="legacyAdapter.failures" />
    </catch-exception-strategy>
</flow>
```

Processing strategy

In the previous example, you may have noticed that we've configured an attribute named `processingStrategy` on the private flow. Though not compulsory, we strongly recommend that you do so for all your private flows, selecting either a synchronous or an asynchronous strategy, depending on the expectations of the calling flow (use synchronous if the calling flow expects a response from the private flow).

If you don't specify the processing strategy, Mule will use the synchronicity of the inflow event to select one. Events generated from one-way endpoints will be executed asynchronously even if the calling flow is expecting a response from the private flow. We'll have an in-depth discussion of flows' processing strategies in chapter 11.

Similarly, we strongly recommend that you wrap your `flow-ref` invocations towards *asynchronously processed private flows* with `<async>` blocks (see section 5.3.3). Strictly defining the expected invocation and processing behavior with private flows will save you headaches.

In Mule Studio, the only notable difference with a subflow is in the grey label before the private flow name, which says *flow*, as shown in figure 2.5.

So far, we've focused on flows and the control they give you over how messages transit within Mule. We'll now look in detail into the message interactions Mule supports and the set of abstractions that enables these interactions.

2.2 Interacting with messages

Mule supports a handful of interactions that lead to either creating a new message or processing an existing one. It's important you become knowledgeable about the lingo used for these interactions, so let's review it in detail:

- *Receiving* happens in message sources when an external event occurs (for example, a new HTTP request or an incoming JMS message), generating a new Mule message.
- *Polling* happens in message sources too, but this time at the initiative of Mule, at a particular frequency, also generating a new Mule message.
- *Dispatching* happens in message processors when Mule sends out a message, but doesn't wait for a response.
- *Sending* happens in message processors when Mule sends out a message and waits for a synchronous response. This response generates a new Mule message.
- *Requesting* happens programmatically and thus whenever decided, generating a new Mule message by fetching data from a particular source (for example, reading a particular file content or consuming one message out of a JMS queue). Requesting can be done by code using the Mule client (see section 12.2).

Don't be alarmed if your head spins a little; we'll come back to these interactions many times in the remainder of the book. For now, it's good enough that you've been exposed to this terminology.

To support these interactions, Mule relies on a set of core abstractions that act as a foundation for all the other aspects of message processing. In this section, we'll detail the following abstractions, a grasp of which is fundamental to successfully using Mule:

- Message sources
- Message processors
- Message exchange patterns
- Endpoint URIs

First things first, so let's start with message sources.

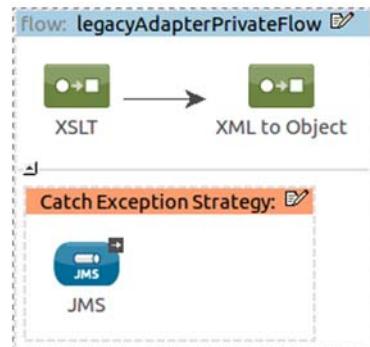


Figure 2.5 A private flow shown in Mule Studio

2.2.1 Message sources

In a Mule configuration, message sources usually manifest themselves as inbound endpoints. Pollers (discussed in section 14.4.1) are also commonly used message sources. Cloud connectors can also provide message sources that can be used in a configuration (more on this in section 13.2.7).

Only a single message source is allowed in a flow (discussed in section 2.1). To allow multiple inbound endpoints to feed messages in the same flow, a composite message source must be used. As you've guessed, this composite message source is also a message source! The following listing shows a few valid message sources, including a composite one.

Listing 2.7 Simple and composite message sources

```
<vm:inbound-endpoint path="payment-processor" />
<composite-source>
    <jms:inbound-endpoint queue="payment-processor" />
    <http:inbound-endpoint host="localhost"
        port="8080"
        path="payment-processor" />
</composite-source>
```

COMPOSITE MESSAGE SOURCES Each endpoint in a composite message source will start a new flow execution when it receives a message. There's no constraint on the transports, nor even on the exchange patterns of the inbound endpoints in a composite source. You mix and match at will!

That wasn't too hard, so let's immediately jump into the next abstraction: message processors.

2.2.2 Message processors

Message processors are the basic building blocks of a Mule configuration; as you've seen in the previous section, besides message sources, flows are mostly composed of message processors. Message processors take care of performing all the message handling operations in Mule and as such manifest themselves in the configuration under a diversity of elements.

Here's a list of the main features provided by message processors:

- As *outbound endpoints* (discussed in section 3.1), they take care of dispatching messages to whatever destination you want.
- As *transformers* (discussed in chapter 4), they're able to modify messages.
- As *routers* (discussed in chapter 5), they ensure messages are distributed to the right destinations.
- As *components* (discussed in chapter 6), they perform business operations on the messages.

Most of the time, you'll be configuring message processors without knowing it; all you'll see in your configuration files will be transformers, components, or endpoints. The fact that all these diverse entities are message processors is the key to Mule's versatility; it allows you to freely tie them together in flows, since behind the scene they're all of the same nature and thus interchangeable.

SHOEHORN MESSAGE PROCESSORS WITH A CHAIN When only a single message processor is allowed at a configuration location, use a processor-chain element, which is a message processor that can encapsulate several message processors that will be called one after the other (as if they were chained).

It's also possible to create custom message processors by implementing the `org.mule.api.processor.MessageProcessor` in a custom class and referencing it in the configuration with a custom processor element. This is extremely powerful, but at the cost of a direct exposure to Mule internals (something components shield you from).

The next abstraction we'll explore deals with the time aspect of message interactions.

2.2.3 Message exchange patterns

Message exchange patterns (MEPs) define the timely coupling that will occur at a particular inbound or outbound endpoint. By defining if an endpoint interaction is synchronous or asynchronous, a MEP influences the way both sides of the endpoint (sender and receiver or, said differently, client and server) interact with each other.

Currently, Mule supports only two MEPs:

- *One-way*, where no synchronous response is expected from the interaction
- *Request-response*, where a synchronous response is expected

Both MEPs can be used on inbound and outbound endpoints, but some transports can apply restrictions to the MEPs they actually support (for example, POP3 is one-way). On inbound endpoints, one-way means Mule won't return a response to the caller, while request-response means it will. On outbound endpoints, one-way means Mule won't wait for a response from the callee, disregarding it if there is one, while request-response means it will.

SAY SOMETHING! If a client calls a one-way service and blocks expecting a response, it will receive an empty one. For example, performing a GET on an in-only HTTP inbound endpoint will return an empty response (content length of size 0) and a 200 OK status. The notable exception to this is the VM transport; both client and server must be in agreement for the MEP they use, otherwise the message exchange will fail.

Don't equate one-way with fire-and-forget; Mule ensures that the messages have been correctly delivered over a one-way endpoint. For example, when dispatching to a one-way JMS endpoint, Mule will ensure that the message has been reliably delivered and will raise an exception if that's not the case. The same will apply to a one-way HTTP

endpoint, whereas the biggest difference with request-response endpoints is that Mule will stop caring for the HTTP interaction as soon as it gets a response status code.

Let's illustrate this with an example. Look at listing 2.8 to see two different HTTP endpoints, each configured with a different MEP (configured by the exchange-pattern attribute). The inbound endpoint uses the request-response MEP, so it will provide synchronous responses to clients that send HTTP requests to it. On the other hand, the outbound endpoint is one-way, which means that Mule (acting as client in this case) won't wait for any response from the remote server, whether there is one or not.

Listing 2.8 HTTP endpoints with different exchange patterns

```
<http:inbound-endpoint host="localhost"
                        port="8080"
                        path="payment-processor"
                        exchange-pattern="request-response" />
<http:outbound-endpoint host="localhost"
                        port="8081"
                        path="notifier"
                        exchange-pattern="one-way" />
```

MEPs affect the timeline of message interactions; request-response creates a timely coupling, while one-way avoids it. MEPs have a direct impact on the parallelization of tasks in Mule, and when this happens, flows may behave in mysterious ways.

To illustrate this, let's take a look at the configuration in the next listing.

Listing 2.9 A simple flow in which execution doesn't happen as expected

```
<flow name="acmeApiBridge">
    <vm:inbound-endpoint path="invokeAcmeAmi" />
    <jdbc:outbound-endpoint queryKey="storeData" />
    <http:outbound-endpoint address="http://acme.com/api" />
</flow>
```

Prancing Donkey uses this configuration to keep track of the state of HTTP conversations in a database. This flow is used when a new conversation with the Acme Corp. API is initiated. In another flow, out-of-band HTTP responses further change the state of the conversation. Surprisingly, issues have arisen because in this flow it happens from time to time that the HTTP request gets dispatched before the JDBC insert, though the latter is positioned before the former in the configuration. How is that possible?

To answer this question, let's look at the log file recorded while a message goes through the flow of listing 2.9:

```
DEBUG acmeApiBridge.stage1.02 [HttpConnector]
Borrowing a dispatcher for endpoint: http://acme.com/api
INFO jdbcConnector.dispatcher.01 [SimpleUpdateSqlStatementStrategy]
Executing SQL statement: 1 row(s) updated
DEBUG acmeApiBridge.stage1.02 [HttpClientMessageDispatcher]
Connecting: HttpClientMessageDispatcher{this=1e492d8,
endpoint=http://acme.com/api, disposed=false}
```

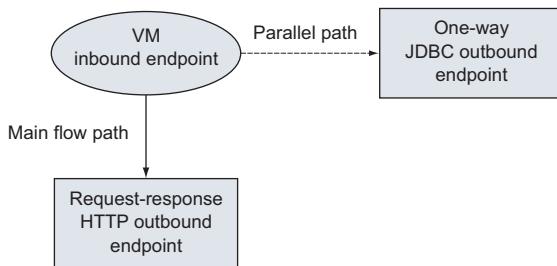


Figure 2.6 Message exchange patterns influence a flow's threading model.

Pay attention to the thread names (recorded between the log level and the class name in square brackets). What do you see? Bingo! The JDBC insert is performed by a different thread than the main one that goes through the flow (notice how it's named after the flow name). Why does Mule perform the JDBC operation in a separate thread?

The answer to this excellent question lies in the message exchange pattern used by each endpoint. Notice how in this previous flow no exchange pattern is specified; the default ones are thus used, meaning one-way for JDBC and request-response for HTTP. Because it's one-way, Mule knows no response is expected in the flow and consequently detaches the processing of this particular interaction in a separate thread!

Figure 2.6 illustrates how a one-way MEP initiates parallel processing of the message interaction. In section 5.3.3, we'll look at other ways to parallelize message processing.

NO GUESSING GAME Make your flows predictable; don't rely on transport defaults, and use explicit MEPs on your endpoints.

The last abstraction we'll look at is not the least: endpoint URIs are indeed the means by which Mule is able to represent all the resources it interacts with (message sources and destinations) in a unified manner.

2.2.4 Endpoint URIs

Mule uses Uniform Resource Identifiers (URIs) as the unified representation for all the resources it exposes or accesses via its endpoints. In other words, inbound and outbound endpoints are internally configured by a URI. Externally, in the XML configuration, they're configured by specific elements with transport-specific attributes. Behind the scenes, all this information is folded into an endpoint URI.

Don't skip this section thinking that you'll never have to deal with endpoint URIs since they're internal. In fact, you'll need to understand endpoint URIs no later than in the last section of this chapter. Indeed, the Mule client uses endpoint URIs to interact with Mule or through Mule (more about this in section 12.2).

Take a look at the following examples. Though the URI representation for HTTP resources is obviously feeling familiar, note how a JMS topic and a TCP socket are also very naturally represented with this scheme. Notice how query parameters, like a connector name, are used to provide extra information to Mule:

```
http://localhost:8080/products
jms:topic://news
tcp://localhost:51000?connector=pollingTcpConnector
```

Now compare that with listing 2.10, which shows a Mule configuration exposing resources at the URIs listed in the preceding code block; the external/internal correspondence we were talking about previously should feel much clearer.

Listing 2.10 Mule endpoints exposing different types of resources

```
<http:inbound-endpoint host="localhost"
                        port="8080"
                        path="products" />
<jms:inbound-endpoint topic="news" />
<tcp:inbound-endpoint host="localhost"
                        port="51000"
                        connector-ref="pollingTcpConnector" />
```

That's it, you've discovered the most important abstractions at work within the very core of Mule.

NO URIS FOR DEVKIT CONNECTORS Connectors built with DevKit (see section 13.1) don't expose standard Mule endpoints, so they aren't accessible via endpoint URIs.

We've been talking about messages moving around in Mule. But what is the exact nature of these messages? It's now time to take our magnifying glass and explore the Mule message.

2.3 Exploring the Mule message

When a message transits in Mule, it is in fact an event (for example, an instance of `org.mule.api.MuleEvent`) that's moved around. This event carries not only a reference to the actual message itself (for example, an instance of `org.mule.api.MuleMessage`), but also the context in which this message is processed. 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 Mule are accessible. This is illustrated in figure 2.7.

In this section, we'll focus exclusively on the structure of the Mule message and leave the discussion of the encapsulating event to chapter 12. This will be more than enough to cover all your needs in the vast majority of Mule use cases.

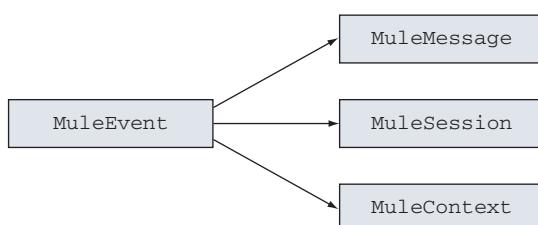


Figure 2.7 The MuleEvent and its main objects

A Mule message is composed of different parts, as illustrated in figure 2.8.

- The payload, which is the main data content carried by the message
- The properties, which contain the meta information of the 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

Messages are normally created by message sources (discussed in section 2.2.1), whether because they've received an incoming event (like a new HTTP request or an incoming JMS message) or because they've successfully polled a resource (like a filesystem or a database). Messages can also be created programmatically (see section 12.3.2).

THREAD SAFETY By default, Mule ensures that only a single thread can modify a message at any point in time. Message copying is a common way to pass messages between threads.

Figure 2.9 illustrates messages that have been created by HTTP (left) and email (right) sources. Notice how the Mule message structure is versatile enough to carry all the relevant information from these two very different transports.

As illustrated in figure 2.9, the Java class of the message payload depends on the origin of the message. Byte arrays, implementations of `java.io.InputStream`, or `String` are payload types commonly created by standard Mule transports. More transport-specific payload types, like `java.io.File` for the File transport, can be also be carried by Mule messages.

The payload wouldn't be enough to ensure that the whole context of an event has been captured and is being propagated in Mule. This is when message properties come into play.

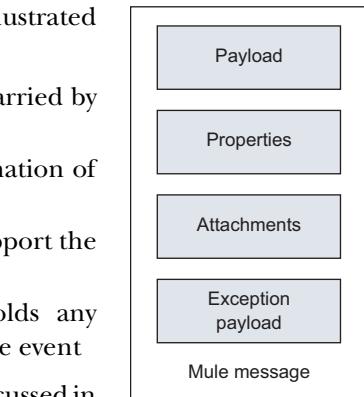
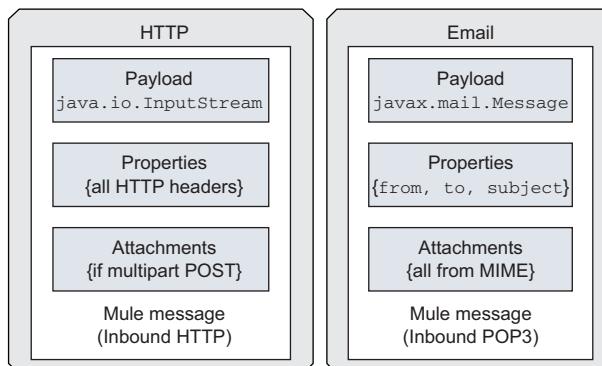


Figure 2.8 Structure of the Mule message

Figure 2.9 Mule messages can contain data from any source.

2.3.1 Message properties

Message properties are meta information that provides contextual data around the payload that's being carried by Mule. This meta information is created automatically when a new message is generated (typically by an inbound endpoint). Message properties are also read by Mule when it needs to dispatch a message to a particular destination (typically via an outbound endpoint). On top of that, message properties are often used when processing or routing messages.

Concretely, message properties are defined by a name (string), a value (object), and a scope (enumeration). While name and value are pretty much self-explanatory, we'll discuss scope in the next section.

Usually, message properties represent the meta information coming from the transport that's underlying the source they originate from. More precisely, message properties encompass the following:

- *Generic transport meta information*, which is stored as-is in message properties. Think about the headers of an HTTP request or the properties of a JMS message. Some transports, like TCP, don't create any generic meta information.
- *Mule-specific transport meta information*, used by the transport to store extra contextual information in the message. For example, the request path of all HTTP requests received by Mule is stored in a property named `http.request.path`.

Most Mule transports have outbound endpoints that are intrinsically sensitive to message properties. For example, the SMTP transport automatically recognizes properties relevant to all the meta information required to send an email, like `subject` or `toAddresses` (see section 3.5.2).

TRANSPORT PROPERTIES Some transports have requirements regarding property names. For example, the JMS transport will remind you loud and clear when you're trying to make it use properties that aren't compatible with the JMS spec!

Unfortunately, no authoritative list of all the properties transports create or recognize exists; you'll need to consult the online reference documentation for each transport you're interested in. Moreover, we suggest using the `logger` message processor in development mode to discover what properties are actually created by a transport.

Message properties are strictly scoped in Mule. Let's see what this means.

2.3.2 Understanding property scopes

As the name *scope* suggests, properties in one scope aren't available in another. This means that, unlike payloads, message properties won't automatically be carried across Mule. Scopes are boundaries that properties won't propagate through. But how are these boundaries defined?

Flows define the boundaries that property scopes abide by. More precisely, flows' inbound and outbound endpoints are the boundaries between which message

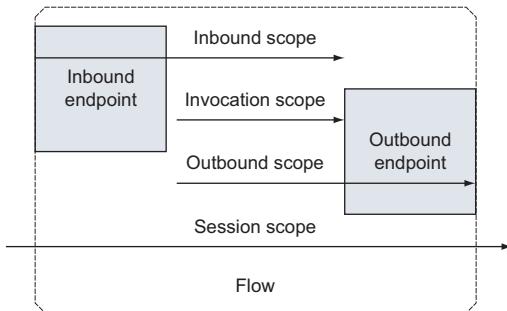


Figure 2.10 Properties' scopes determine their travel boundaries.

properties can travel. Figure 2.10 illustrates the different property scopes and how they relate to these boundaries.

To recapitulate, Mule fully supports four property scopes:

- *Inbound*—Created by message sources like inbound endpoints, these properties are read-only for the end user. Inbound properties can be wiped out midflow by outbound endpoints; if this happens to you, message enrichment is your friend (see section 4.3.5).
- *Outbound*—Created by `set-property`, `message-properties-transformer` with outbound scope, or code and recognized by outbound endpoints.
- *Invocation*—Created by `set-variable`, `message-properties-transformer` with invocation scope, or code, mainly known as *flow variables* because they are strictly bound to a flow.
- *Session*—Created by message sources, `set-session-variable`, `message-properties-transformer` with session scope, or code, this scope is bound to the in-flight message² and can therefore span multiple flows, as it follows the message through them (think of Java's `ThreadLocal` mechanism but with a `MessageLocal` twist).

CROSSING BOUNDARIES Use the `copy-properties` element to copy properties across scopes, for example, to propagate an inbound property to the outbound scope or to store it in the session scope.

So far, we've focused on properties in the context of the request phase. What happens to them in the response phase? As you might expect, during the response phase (discussed in section 2.1.1) things are reversed. Inbound endpoints look for properties in the *outbound* scope to propagate to their callers. Request-response outbound endpoints create *inbound* properties. Only the flow and session variables behave the same, provided they've been preserved and have remained unaffected by the interactions that happen in message processors.

² And not to the conversation with the remote client, like with a web container session.

Let's make this concrete. Take a look at listing 2.11, which illustrates how to set the HTTP content-type header in the response phase; see how we've created the property so that the inbound endpoint uses it when it responds to its caller? The trick here is that `set-property` creates properties in the outbound scope.

Listing 2.11 Adding a header to the outbound scope adds it to the response

```
<flow name="responseHeader">
    <http:inbound-endpoint exchange-pattern="request-response"
        host="localhost"
        port="8080"
        path="json/data" />
    <component class="com.prancingdonkey.data.JsonDataFetcher" />
    <response>
        <set-property propertyName="Content-Type"
            value="application/json" />
    </response>
</flow>
```

RESPONSE PHASE PROPERTY PROPAGATION Propagating a message property from the response of an outbound endpoint back to the caller of an inbound endpoint requires copying it from the inbound to the outbound scopes in the response phase.

Let's look at another example, with more scopes involved. Listing 2.12 shows a flow that receives an XML user representation. It extracts its ID with an XPath expression and stores it in a flow variable named `userId`. This flow variable is subsequently used in two places: in an HTTP outbound endpoint in order to build a REST resource URI, and in a response block to add a property named `X-User-ID`. As you can see, the invocation scope is still active in the response phase; the flow variable that was set in the inbound phase is still present in the response phase.

Listing 2.12 A flow that sets a variable and uses it in two places

```
<flow name="flowVarPersistence">
    <vm:inbound-endpoint path="user.fetch"
        exchange-pattern="request-response" />

    <response>
        <set-property propertyName="X-User-ID"
            value="#{userId}" />
    </response>

    <set-variable variableName="userId"
        value="#{xpath('/user/@id').value}" />

    <http:outbound-endpoint
        address="http://${internalApiBaseUri}/api/users/#/{userId}"
        method="GET"
        exchange-pattern="request-response" />
</flow>
```

ACCESSIBLE BUT NOT OWNED Though the flow and session variables are both accessible via the properties accessor of a Mule message, they are, behind the scene, respectively owned by the Mule Event and Session objects. For convenience, the Mule message shares references to the maps that back these two scopes.

Though less frequently used, attachments are also important members of a Mule message. Let's look into them.

2.3.3 Using message attachments

As briefly presented in the introduction, message attachments allow for supporting extra chunks of data that aren't meta information, semantically speaking, but that are more like alternative or complementary chunks of payload.

An attachment is defined by a name, a source of data (an instance of `javax.activation.DataHandler`), and a scope. Attachments' scopes are limited to inbound and outbound, with the same flow-bound boundaries as the properties' scopes.

Attachments are mainly used to support the multiple parts of inbound email messages (created with the POP or IMAP transport) and outbound email messages (sent with the SMTP transport). Inbound attachments are created by the email endpoint directly, while outbound endpoints require attachments to be created programmatically.

The following listing shows a flow that's used by Prancing Donkey to accept email orders.

Listing 2.13 Extracting message attachments with an expression transformer

```
<flow name="email-order-processor">
    <inbound-endpoint ref="email-orders" />
    <expression-transformer
        expression="#[message.inboundAttachments.values()]" />
    <collection-splitter />
    <outbound-endpoint ref="pdf-orders" />
</flow>
```

Splits attachment list into individual messages

1 Receives email over a global endpoint configured elsewhere

2 Extracts all attachments from the current message

3

4 Sends an individual order message for processing

Notice at ② how an expression transformer is used to extract all the attachments and replace the current message payload with the extracted attachments list (`inboundAttachments` is a `java.util.Map` of attachment names and values). Though Prancing Donkey likes a little chitchat from time to time, in this particular case, the email message body that was stored in the message payload after ① is abandoned in favor of the attachments. Because customers are allowed to attach several invoices to one email, a message splitter ③ is used to split the attachment list into individual messages, each of them processed downstream by a service behind the endpoint called at ④.

This wraps up our in-depth discussion about the structure and hidden goodness of Mule messages. Payload, properties, scopes, and attachments should by now have lost their mystery for you.

We've alluded to it in many places, so it's now time to look into the Mule Expression Language.

2.4 Speaking the Mule Expression Language

As we said in the opening of this chapter, expressions are the way to bring dynamism to your configuration. Indeed, Mule offers a rich expression language that allows you to hook advanced logic into different parts of your configuration.

EXPRESSIONS RELOADED The Mule expression evaluation framework has gone through a complete overhaul in version 3.3. The old syntax, which was of the form `# [evaluator:expression]` and required two attributes (`expression` and `evaluator`), has been deprecated and replaced by a unified syntax. We'll cover only this new syntax in the book.

The Mule Expression Language, a.k.a. MEL, is based on the MVFLEX Expression Language (MVEL), which gives access to a complete range of programmatic features, enriched with Mule-specific variables and functions. MEL's thorough documentation is available online, so we'll only discuss here its general principles before delving into a few examples (you can also refer to appendix A for a more detailed reference).

With MEL, you can do the following:

- Use any Java class available on the classpath
- Easily navigate lists, maps, arrays, and bean properties
- Express complex logic, including ternary expressions
- Define local variables and use them in statements

On top of that, Mule binds a set of top-level objects that act as entry points to Mule's overall execution context, but also to the context of the message currently processed. Currently, four context objects are available:

- `server`—Properties of the hardware, operating system, user, and network interface, for example, `server.ip`
- `mule`—Properties of the Mule instance, for example, `mule.version`
- `app`—Properties of the Mule application into which the expression is evaluated, for example, `app.name`
- `message`—The in-flight Mule message, for example, `message.correlationId` or `message.payload`

Mule also binds helper methods as top-level members of the evaluation context. As of this writing, two helpers are available:

- `xpath`—To easily extract values from XML payloads or any other MEL expressions, for example, `xpath('/orders/order[0]')`

- `regex`—To return an array of matches for payloads or any other MEL expressions, for example, `regex('^(To|From|Cc):')`

EXTENDING MEL It's possible to extend MEL in order to define custom imports, reusable functions, and even global aliases. Refer to appendix A, section A.2, for more information.

It's valid to place several statements in the same expression, as long as they're separated by semicolons. The value of the last statement is used as the value of the whole expression. For example, evaluating the following expression

```
targetDir = new java.io.File(server.tmpDir, 'target');  
targetDir.mkdir();  
targetDir
```

returns `targetDir` as a `java.io.File` (and also ensures that it exists).

In the XML configuration, it's recommended to wrap the expression with a hash-square `#[]` in any attribute where you use an expression (not using `#[]` works in many places, but the behavior is not consistent enough to be relied on). For example, though the following no-hash-square expression works,

```
<when expression="message.inboundProperties['valid']">
```

we recommend you use

```
<when expression="#[message.inboundProperties['valid']]">
```

This doesn't apply to expression components (discussed in section 6.1.6), in which the `#[]` can be omitted.

Let's now look at a few examples using expressions.

2.4.1 Using expressions

Expressions can be used almost anywhere a string attribute is expected in the XML configuration. Of course, because they're evaluated at runtime when messages are flying through Mule, you can't use expressions that require an in-flight message to work in values that are read by Mule on startup (say, for example, a port to which Mule must bind).

ALMOST EVERYWHERE Expressions are well supported across all Mule modules and transports. This said, you may still encounter in your configurations XML attributes in which expressions are not supported. If that's the case, raise the issue with MuleSoft (www.mulesoft.org/jira/browse/MULE).

Listing 2.14 shows how the `java.util.UUID` class is used to dynamically create a transaction ID and pass it to an XSLT transformer (discussed in section 6.1.6). Notice how, because an expression-specific XML attribute wasn't available, the expression needed to be hash-square wrapped as discussed previously.

Listing 2.14 Using an expression to create a dynamic XSLT parameter

```
<mulexml:xslt-transformer xsl-file="to_payment_processor_call.xsl">
    <mulexml:context-property key="transactionId"
        value="#{java.util.UUID.randomUUID().toString()}" />
</mulexml:xslt-transformer>
```

Message processors make fair usage of expressions too. In section 4.3.5, we'll discuss the *message enricher*, a message processor that relies heavily on expressions. Another message processor that relies on expressions is the logger.

Take a look at listing 2.15 to see how the `xpath` helper is used by Prancing Donkey for logging the invoice ID out of an in-flight message with an XML payload. Notice also how the `getValue()` method is called on the `org.dom4j.Attribute` returned by the `xpath` evaluation via the `value` short bean property access.

Listing 2.15 Logging a value extracted with XPath

```
<logger message="#{xpath('/invoice/@id').value}"
    category="com.prancingdonkey.service.Invoice.id"
    level="INFO" />
```

Another way in which expressions are used a lot is to express the decision logic of conditional routers, like the choice router and other similar filtering routers. These routers will be discussed in detail in section 5.2.

TRUE OR FALSE MEL expressions that return Boolean values can be used in conditional logic. String values that can be evaluated as Booleans can be used too, but we prefer the stricter approach that results in returning true Booleans.

Configuration patterns, discussed in section 6.2, also rely a lot on expressions in order to offer the most concise configuration possible. Listing 2.16 shows a validator pattern that uses simple string literals for its acknowledgement and rejection messages. It also uses MEL for checking if the in-flight message has any attachment.

Listing 2.16 The validator configuration pattern heavily relies on expressions.

```
<pattern:validator name="ensureAttached"
    inboundAddress="vm://ensure.attached"
    outboundAddress="vm://valid.request.handler"
    ackExpression="#['OK']"
    nackExpression="#['ERROR: no attachment!']">
    <expression-filter
        expression="#{!(message.inboundAttachments.empty)}" />
</pattern:validator>
```

Expressions can also be used in the configuration of endpoints. Consider, for example, listing 2.17, which shows a flow that allows for dynamically fetching stock market history. This flow receives the ticker identifier as its main payload, but doesn't assume that it will be a `java.lang.String` (it could very well be `byte[]`). Instead, it uses a

payload evaluator to fetch the current payload and transform it into a `java.lang.String` before building the HTTP path with it.

Listing 2.17 Expressions in endpoint URIs can be resolved at runtime by evaluators.

```
<flow name="tickerFetcher">
    <vm:inbound-endpoint path="ticker.fetcher"
        exchange-pattern="request-response" />
    <http:outbound-endpoint
        exchange-pattern="request-response"
        host="www.google.com"
        port="80"
        path="finance/historical?q=# [message.payloadAs(java.lang.String)]" />
</flow>
```

It's possible to use the Mule expression evaluation framework from your own code. The only thing you need is a reference to the Mule context (discussed in section 12.1), which allows you to use the expression language evaluator:

```
ExpressionLanguage mel = muleContext.getExpressionLanguage();
String applicationName = mel.evaluate("app.name");
```

You should find that the evaluation methods on the evaluator are pretty self-explanatory; if not, refer to the Mule API SDK (www.mulesoft.org/docs/site/current3/apidocs/).

In annotations too

Expressions can be used in annotations too (see section 6.1.4). For that, the `@Mule` annotation must be used, as shown here:

```
@Mule("#[app.name]")
String applicationName
```

We can't possibly cover all the possibilities of MEL in this section. But throughout the remainder of the book, you'll have plenty of opportunities to see the power of MEL in action!

2.5 Summary

We're done with our tour of Mule message processing capacities. You've learned about controlling message paths with flows. You've also discovered the abstractions that sit at Mule's core, the true nature of messages that move around in Mule, and how expressions can help you solve advanced problems that static configurations can't.

It's now time that you expand your horizon by looking into the different ways that Mule reaches the outside world. The next chapter will cover Mule transports and cloud connectors and how you can use them to transfer data using popular protocols and APIs.

Working with connectors



This chapter covers

- Using Mule connectors to integrate with disparate protocols
- Consuming APIs with Mule cloud connectors
- Integrating with SaaS APIs

Moving data around is the crux of integration. Precious time spent authoring WSDLs, writing JMS consumers, or struggling with a vendor's obtuse API paradoxically contributes little value to an application. It instead has the opposite effect: burning you out writing code that doesn't contribute to solving the business issue at hand. Even when such code is trivially developed, implementations will still differ across projects and between teams. This increases the difficulty when bringing developers up to speed or when developers move between teams. Developers must be careful that their transport code is thread-safe, handles errors correctly, and updates as protocols and APIs change.

Let's revisit the concrete, but relatively simple, example of accepting an HTTP request and sending the content of the request to a JMS queue, which we implemented using Mule in chapter 1 of this book. This is conceptually a fairly simple scenario, but manually implementing the same application without an integration framework like Mule would require the following steps:

- Setting up a web server to accept the request
- Transforming the HTTP request into a JMS message
- Obtaining and properly managing all the resources to send the JMS messages (connection factories, sessions, and so on)
- Properly dealing with concurrency issues, logging, scaling, security, and so on

Writing this code may or may not be difficult, depending on your skill and how well you know some third-party frameworks that simplify the implementation. Regardless, is this code you (or your team) is willing to develop, test, document, and maintain?

As moving data around is such an integral piece 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!

Arnor Accounting and Frozen North Freezing, Inc.

The examples in this chapter will focus on Prancing Donkey's integrations with Arnor Accounting and Frozen North Freezing, Inc. Arnor Accounting is a small, forward-thinking startup with an SaaS-based accounting application.

Frozen North Freezing is an established provider of smart cooling products. Many of their products feature integration points to interact with other systems, typically for monitoring, as you'll see later on.

3.1 Understanding connectors

Connectors are Mule's abstraction for sending data, receiving data, and interacting with different APIs. The most common form of connector you'll use in Mule is called a *transport*. Transports provide an adaptation layer for a protocol, like HTTP or JMS. Transports provide sources and sinks for data. They're used to get data into and out of flows. A source of data for a flow is called an *inbound endpoint*. A sink, or destination for data, is called an *outbound endpoint*. The first part of this chapter will cover some of Mule's more prominent transports.

The other kind of connector is called a *cloud connector*. Cloud connectors are typically used to interact with an API. Initially these were cloud-based APIs (hence the name) such as those provided by Twitter or Salesforce. The definition has been muddied a bit, as you'll see later in this chapter, to include other sorts of APIs like MongoDB. Cloud connectors don't typically offer endpoints, but rather have message processors that map to operations defined in the API. Cloud connectors encapsulate the operations of an API, greatly reducing the initial friction for a developer to come up to speed when using it.

This distinction between transports and cloud connectors is admittedly blurry. Typically, however, transports provide protocol support while cloud connectors facilitate interaction with an API.

3.1.1 Configuring connectors with XML

Your Mule configuration will very often contain one or several connector configuration elements. Each transport will contribute its own connector element with specific attributes. For example, here's the configuration for a file connector:

```
<file:connector name="fileConnector" autoDelete="false"/>
```

And here's a secure proxied HTTP connector configuration:

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

Connector elements contain a name attribute that you'll use to uniquely identify the connector in endpoints. This allows you to configure multiple connectors for a single transport, with different configuring parameters for each.

It's absolutely possible, and even common, to have a configuration that doesn't contain any connector definitions. Why is that? If Mule figures out that one of your endpoints needs a particular connector, it will 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 or the HTTP one.

Multiple connectors for the same protocol

When you have only one connector for a particular protocol, whether it's a default connector automatically created by Mule or one that you have 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 two 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" />
```

For transports, a connector provides the infrastructure to communicate to a particular protocol. An endpoint sends and receives messages on a specific channel using the infrastructure provided by the connector. A JMS connector, for example, is responsible

for managing the connection to the JMS broker, handling the authentication and reconnecting in the event of a communication failure. A JMS endpoint, on the other hand, is responsible for sending and receiving messages on queues and topics. Endpoints are the cornerstone of IO with transports in Mule; they're the tools you'll use to get data into and out of your flows.

Endpoints come in two different flavors, inbound and outbound. Inbound endpoints are used as message sources; outbound endpoints are used to send data. An inbound endpoint can do things like receive JMS messages, read file streams, and pull down email messages from a mail server. An outbound endpoint can do things like dispatch JMS messages, insert rows into a database table, and send a file to an FTP server. You'll use inbound and outbound endpoints to communicate between components and flows inside Mule as well as with the outside world.

You'll quickly notice that all endpoints offer an address attribute. Why is that? This allows for the configuration of a generic endpoint using a URI-based destination address instead of the dedicated attributes of the specific endpoint element. There are three cases where this is useful:

- An address can be used in place of individual attributes on an endpoint. Endpoints using the HTTP transport, for instance, can be configured explicitly with the full address to listen on as opposed to individually setting the host, port, and path.
- When using `MuleClient`, which you'll see in chapter 12, addresses are used to send messages to and receive them from Mule applications.
- While not common, generic endpoints that aren't prefixed with a transport's namespace can be used. This might be useful if you want to use a different transport depending on the environment (for example, using the VM transport instead of the JMS transport depending on where the app is deployed).

3.1.2 Configuring connectors with Mule Studio

Configuring connectors and endpoints is very similar, and much easier in Mule Studio than with XML. Simply right-click on an endpoint and select Properties, as illustrated in figure 3.1.

Figure 3.2 shows the configuration panel for an HTTP endpoint. You can define an HTTP connector by clicking on the References tab and then clicking on the Plus button next to Connector Reference. After you add the connector, it will be available on the drop-down list in the References tab, as illustrated by figures 3.3 and 3.4.

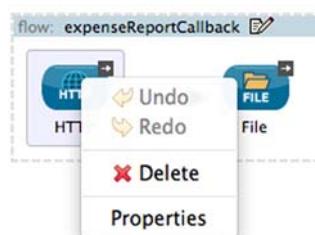


Figure 3.1 Selecting endpoint properties

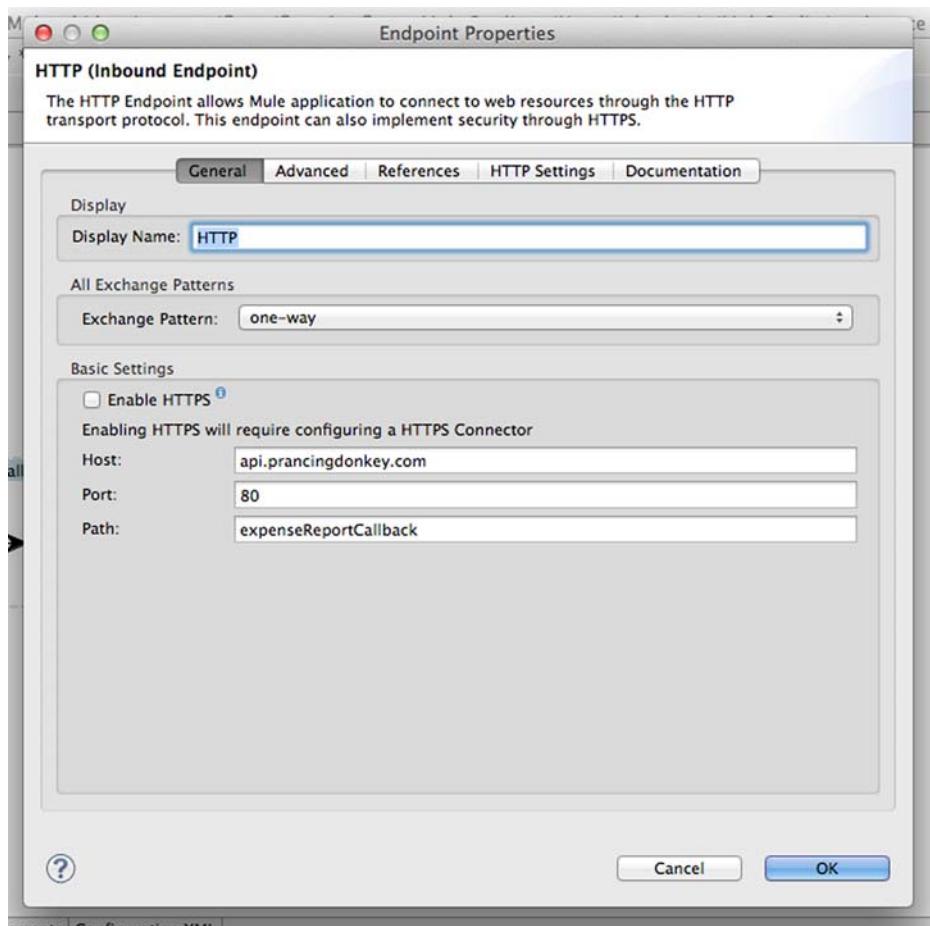


Figure 3.2 Selecting the Endpoint Properties panel

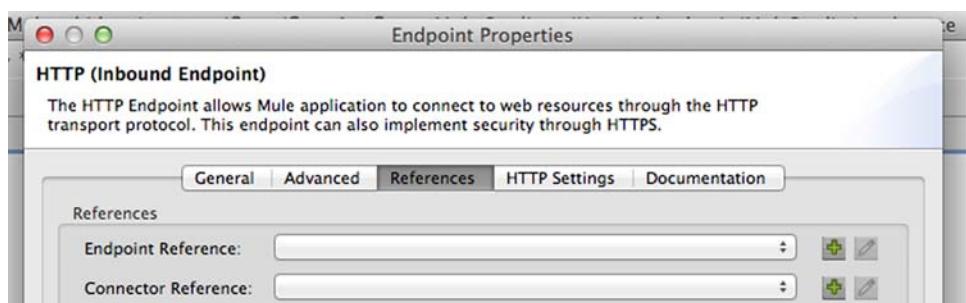


Figure 3.3 Explicitly adding a connector

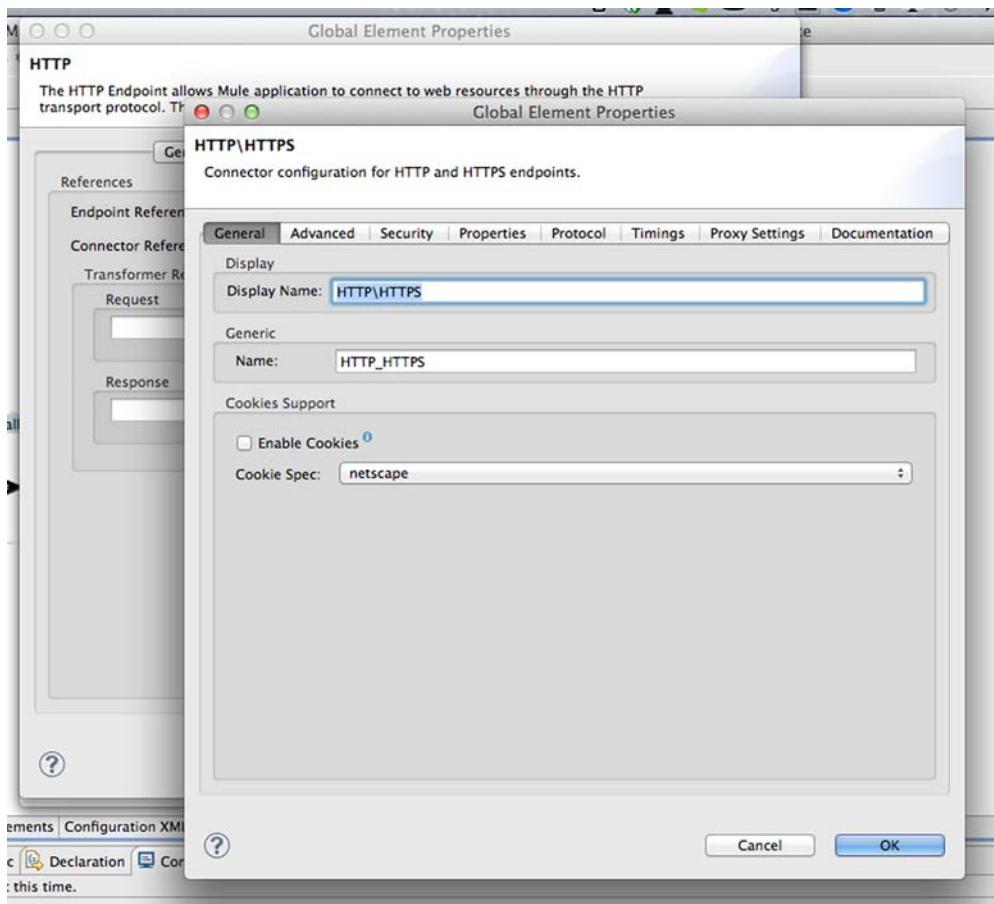


Figure 3.4 Editing the connector

Now that you're comfortable configuring connectors and endpoints both with XML and in Mule Studio, let's dive in and start looking at some of Mule's built-in transports. Here are the transports and cloud connectors we'll look at in this chapter:

- *File transport*—Read and write data to and from local and network filesystems.
- *HTTP transport*—Send and receive HTTP requests, and expose and consume web services.
- *JMS transport*—Dispatch and receive JMS messages on queues and topics.
- *Email transport*—Receive email from an IMAP server, and send emails with SMTP.
- *FTP transport*—Poll and send files to and from an FTP server.
- *JDBC transport*—Read and write to and from relational databases with JDBC.
- *MongoDB cloud connector*—Write and read data to and from MongoDB.
- *VM transport*—Use the virtual memory transport to implement reliability patterns.
- *Twitter cloud connector*—Interact with the Twitter API.

We'll demonstrate these transports in the context of Prancing Donkey's integration efforts with Arnor Accounting and Frozen North Freezing, starting off with the file transport.

3.2 Using the file transport

Reading and writing file data is often the easiest way get data into and out of applications. In this section, you'll see how to use Mule's file transport to read, write, move, and delete files.

Configuration properties

During the course of this book, you'll see certain configuration properties that look like this:

```
host="${imap.host}"
```

These are property placeholders that are used to avoid hardcoded, environment-specific variables in Mule configurations. The location of the properties files is configured as follows in the Mule configuration:

```
<context:property-placeholder location="/mule.properties"/>
```

Some of the configuration properties for the file transport 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.

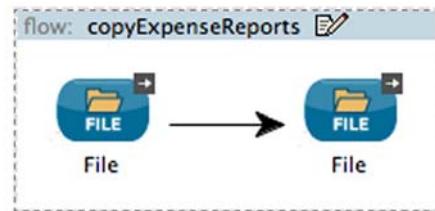
Table 3.1 Configuring the file transport

Property	Type	Target	Required?	Default	Description
writeToDirectory	String	Connector, outbound endpoint	No		The target directory for file output
readFromDirectory	String	Connector, inbound endpoint	No		The source directory for file input
autoDelete	Boolean	Connector, inbound endpoint	No	true	Whether or not to delete the source file after it has been read
outputAppend	Boolean	Connector, outbound endpoint	No	false	Specifies that the output is appended to a single file instead of being written to a new file
pollingFrequency	long	Connector, inbound endpoint	No	0	Specifies the interval in milliseconds at which the source directory should be polled

Table 3.1 Configuring the file transport (continued)

Property	Type	Target	Required?	Default	Description
moveToDirectory	String	Connector, inbound endpoint	No		The directory to move a file to once it has been read—if this and autoDelete are unset, the file is deleted
moveToPattern	String	Connector, inbound endpoint	No		The pattern used to indicate what files should be moved to the moveToDirectory—files not matching this pattern are deleted
outputPattern	String	Connector, outbound endpoint	No		The pattern used when creating new files on outbound endpoints—discussed in detail later

Some of Prancing Donkey's employees still use Excel spreadsheets to track their expenses. This is particularly frustrating to Prancing Donkey's developers, who want to take full advantage of the REST API provided by Arnor Accounting. To make matters worse, employees have been emailing the expense sheet directly to the accounting department. The accounting staff have taken matters into their own hands, tired of sifting through emails to find expense report attachments, and have created a network share for reports to be dropped into. This has been somewhat successful, but naming of the expense report files has been inconsistent. Let's look at a flow that can poll an incoming folder for new expense reports and copy them to another folder while normalizing the name along the way (see figure 3.5).

**Figure 3.5** `copyExpenseReports` flow in Mule Studio**Listing 3.1 Using the file transport to move files from one directory to another**

```

<flow name="copyExpenseReports">
    <file:inbound-endpoint path=".~/data/expenses/1/in"
        pollingFrequency="60000">
        <file:filename-regex-filter pattern=".xls$"
            caseSensitive="false"/>
    </file:inbound-endpoint>
    <file:outbound-endpoint path=".~/data/expenses/out"
        outputPattern="EXPENSE-REPORT-
#[org.mule.util.DateUtils.getTimeStamp('dd-MM-yy_HH-mm-ss.SSS')]"
-#[inboundProperties['originalFilename']]"/>
</flow>

```

Only accept files with .xls extension (2) → **Poll .~/data/expenses/1/in directory every minute for new files** (1)

Write file to .~/data/expenses/out, naming new file with the timestamp and original filename of the input file (3)

The message source of this flow is the file inbound endpoint, configured at ① to poll the /data/expenses/1/in directory once a minute for new files. The filename-regex-filter defined at ② instructs the endpoint to only accept files that end in .xls and to ignore the case. Finally, the outbound endpoint ③ uses the Mule Expression Language to normalize the filename with the timestamp and the file's original name. To obtain the latter, you reference the `originalFilename` header populated by the file inbound endpoint.

A file inbound endpoint will, by default, remove the file from the source directory it's read from when it's passed to an outbound endpoint. You can override this behavior by setting the `autoDelete` property on the file connector to `false`. Be careful when doing this, however, as it will cause the file to be repeatedly read by a file inbound endpoint until it's removed from the source directory.

The `fileAge` parameter allows you to read files that are older than a certain age in milliseconds. Setting this value to 5000, for instance, would cause a file inbound endpoint to only process files older than 5 seconds.

Multiple Mule nodes polling a shared filesystem

You'll see strategies in chapter 8 for deploying the same Mule application to multiple Mule instances. Care must be taken in such scenarios when an application on each instance attempts to poll the same directory on a network share, such as SMB or NFS. Typically state between the Mule nodes needs to be coordinated to ensure the same file isn't being read, deleted, or moved at the same time (usually more is required than the locking semantics offered by the underlying protocol).

Mule EE's clustering feature will automatically coordinate and distribute activity between Mule applications using the file transport.

In this section, you saw how the Mule file transport can be used to selectively move file data through Mule flows. You saw how Prancing Donkey uses this functionality to normalize the submission of expense reports to a shared folder.

Now let's look at something a little more interesting: Mule's support for HTTP and web services.

3.3 Using the HTTP transport

In this section, we'll look at Mule's support for HTTP and web services. We'll start off by looking at the HTTP transport, which makes it easy to send and receive data from websites and applications. We'll then look at Mule's support for JAX-RS and JAX-WS, which provide support for REST and SOAP APIs over HTTP.

3.3.1 Sending and receiving data using HTTP

The HTTP transport allows you to send and receive data using the HTTP protocol. You can use HTTP's POST method to send data through an outbound endpoint or the GET method to return data from a request-response inbound endpoint. Table 3.2

enumerates some of the common configuration elements of the HTTP connector and endpoints.

Table 3.2 The HTTP transport lets you specify typical client-side properties.

Property	Type	Target	Required?	Default	Description
host	String	Inbound endpoint, outbound endpoint	No	false	The host to either receive requests (inbound) or send requests (outbound)
port	int	Inbound endpoint, outbound endpoint	No	80	The port to bind to for inbound endpoints, and the port to send to for outbound endpoints
method	String	Inbound endpoint, outbound endpoint	No	false	HTTP method to use (GET, POST, PUT, DELETE, OPTIONS, HEAD, PATCH, TRACE, or CONNECT)
path	String	Inbound endpoint, outbound endpoint	No	false	The URI path
proxyHostname	String	Connector	No	false	The proxy hostname—this lets you use a web proxy for requests
proxyPort	String	Connector	No	false	The proxy port
proxyUsername	String	Connector	No	false	The proxy username
proxyPassword	String	Connector	No	false	The proxy password
enableCookies	Boolean	Connector	No	false	Enable cookies
user	String	Outbound endpoint	No	false	The username for the remote URL
password	String	Outbound endpoint	No	false	The password for the remote URL

Let's see how you can use an HTTP outbound endpoint to POST data to a URL. The developers at Arnor Accounting have graciously agreed to allow Prancing Donkey to upload expense reports that are XLS files. If they weren't already using Mule, this might be a big problem. Their developers would need to rip out the piece of code that copies the file to the accountant's directory and replace it with code to POST the file contents to the remote URL. Using Mule, they simply need to change the outbound endpoint, as illustrated in the next listing, and illustrated in figure 3.6.

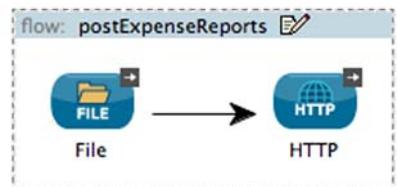


Figure 3.6 `postExpenseReports` flow in Mule Studio

Listing 3.2 Posting data with the HTTP transport

```
<flow name="postExpenseReports">
    <file:inbound-endpoint path="../data/expenses/2/in"
        pollingFrequency="60000">
        <file:filename-regex-filter pattern=".*xlsx$"
            caseSensitive="false"/>
    </file:inbound-endpoint>
    <http:outbound-endpoint host="${http.host}"
        port="${http.port}"
        path="expenseReports"
        method="POST" />
</flow>
```

POST contents of expense report using an HTTP outbound endpoint

Replace the `file:outbound-endpoint` with an `http:outbound-endpoint` and voilà! The data is no longer being written to a file, but is instead posted to the remote web application.

Using HTTP as a message source is just as easy. Since parsing and processing the expense spreadsheets takes some time, the SaaS application processes the spreadsheets asynchronously. The ever-accommodating developers at Arnor Accounting have added a feature that will POST a notification to a specified URL after the report has been processed. For now, Prancing Donkey wants to receive these notifications and write them to the filesystem, where they can be parsed by a monitoring system for errors (you'll see how this can be improved when we discuss JMS in section 3.4.) The next listing illustrates how to do this, and figure 3.7 shows it.

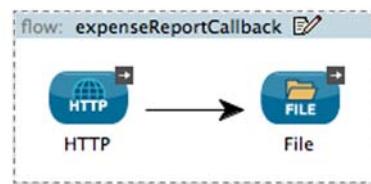


Figure 3.7 `expenseReportCallback` flow in Mule Studio

Listing 3.3 Using an HTTP inbound endpoint to POST data to a file

```
<flow name="expenseReportCallback">
    <http:inbound-endpoint host="${http.host}"
        port="${http.port}"
        path="expenseReportCallback"
        method="POST"
        exchange-pattern="one-way" />

    <file:outbound-endpoint path="../data/expenses/status"
        outputPattern="#{java.util.UUID.randomUUID().toString()}"
        -#{org.mule.util.DateUtils.getTimeStamp
        ('dd-MM-yy_HH-mm-ss.SSS')}
        .xml" />
</flow>
```

1 Accept HTTP requests on `http://${http.host}/expenses/status`

Write expense report processing statuses to `./data/expenses/status` directory

The HTTP inbound endpoint configured at ① accepts HTTP POST requests that will contain the status notifications. The payload of these posts will be written to the configured directory. Note that the previous flow and this flow both use one-way exchange patterns. This allows you to simulate asynchronous requests over HTTP, which is a synchronous protocol. In this case, Mule will return a 200 OK response to the client unless there's an exception thrown in the flow. You'll see how to handle exceptions in chapter 9.

HTTPS All the HTTP examples you'll see will make use of the default, unencrypted HTTP transport. We'll cover HTTPS and SSL in chapter 10.

The ability to asynchronously handle HTTP requests is a useful tool in your arsenal of integration techniques. It allows you to implement a messaging API over HTTP. More often than not you'll want to deal with HTTP in a request-response fashion. Let's see how to do that using Mule's support for web services.

3.3.2 Using web services with Mule

Prancing Donkey has a service class, called `BrewService`, that they'd like to expose for external clients to consume. This service currently has a single method, `getBrews()`, that returns a list of the current beers in their catalog. This method returns a `List` of `Brew` instances, a part of their domain model. Listings 3.4, 3.5, and 3.6 detail the `BrewService` interface and implementation and the domain model class for `Brew`.

JAX-RS AND JAX-WS The examples in this section will make use of the JAX-RS and JAX-WS standards, a full discussion of which is out of the scope of this book. For a detailed treatment of each, including how to annotate your domain models to control serialization, please see <http://jersey.java.net> and <http://cxf.apache.org/>.

Listing 3.4 The BrewService interface

```
package com.prancingdonkey.service;

public interface BrewService {
    List<Brew> getBrews();
}
```

Listing 3.5 The BrewService implementation

```
package com.prancingdonkey.service;

public class BrewServiceImpl implements BrewService {
    List<Brew> getBrews() {
        return Brew.findAll();
    }
}
```

Listing 3.6 The Brew domain model

```
package com.prancingdonkey.model;
public class Brew implements Serializable {
    String name;
    String description;
    public Brew(String name, String description) {
        this.name = name;
        this.description = description;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    static public List<Brew> findAll()
    {
        // Returns a List of Brews
        return ...
    }
}
```

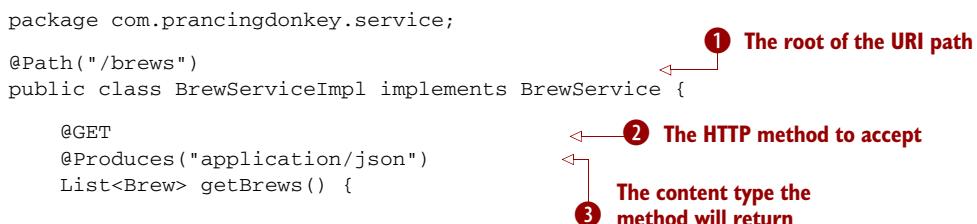
We'll now look at how to expose the `getBrews()` method to external users using REST and SOAP.

REST WITH JAX-RS

JAX-RS is the Java API for RESTful web services. Mule's support for Jersey, the JAX-RS reference implementation, allows you to develop RESTful web services using JAX-RS annotated classes in your Mule flows. Let's annotate `BrewServiceImpl` to return a JSON representation of the brew catalog.

Listing 3.7 The JAX-RS annotated BrewService

```
package com.prancingdonkey.service;
@Path("/brews")
public class BrewServiceImpl implements BrewService {
    @GET
    @Produces("application/json")
    List<Brew> getBrews() {
```



The code shows a `@Path` annotation with the value `/brews`, which is annotated with a red circle labeled **1 The root of the URI path**. Below it is a `@GET` annotation, which is annotated with a red circle labeled **2 The HTTP method to accept**. Below that is a `@Produces` annotation with the value `"application/json"`, which is annotated with a red circle labeled **3 The content type the method will return**.

```

        return Brew.findAll();
    }
}

```

The `@Path` annotation ① specifies the root URI of all methods you expose in this class (you can also specify the `@Path` annotations on methods to further qualify the path). The `@GET` annotation ② specifies that this method will only respond to HTTP GET requests. The `@Produces` annotation ③ indicates that you'll be returning JSON as your response.

Now that you have `BrewServiceImpl` properly annotated, let's wire it up in a flow. The following listing illustrates how to do this (see figure 3.8).



Figure 3.8 `brewRestService` flow in Mule Studio

Listing 3.8 Expose the brew listing over REST using JAX-RS

```

<flow name="brewRestService">
    <http:inbound-endpoint address="http://localhost:8091/rest"
                           exchange-pattern="request-response"/>

    <jersey:resources>
        <component class="com.prancingdonkey.service.BrewServiceImpl"/>
    </jersey:resources>
</flow>

```

1 HTTP inbound endpoint to receive REST requests
 2 Class to expose as a Jersey resource

The `http:inbound-endpoint` ① sets up the HTTP plumbing around the RESTful request. The `jersey:resources` and `component class` ② expose the JAX-RS class annotated in listing 3.7. You can now start Mule and execute your request. Let's invoke the service with curl and see what happens.

Listing 3.9 Using curl to invoke the web service

```
% curl
http://api.prancingdonkey.com/rest/brews/list
[ {"name": "Frodo's IPA", "description": "Indian Pale Ale" }, {
"name": "Bilbo's Lager", "description": "Indian Pale Ale" }, {
"name": "Gandalf's Barley Wine", "description": "Indian Pale Ale" } ]
```

The curl request against the `rest/brews/list` URI returns the method's response, a List of `Brew` classes, serialized as JSON. You can customize how the JSON is generated by annotating your domain model with JAXB. Now let's see how you can use Mule's support for JAX-WS to expose the same service using SOAP.

TOOLS TO CONSUME REST SERVICES Curl is a popular command-line tool that can be used to interact with RESTful services. It can be downloaded from <http://curl.haxx.se/>. Another great tool for consuming RESTful services is

REST Console for Google's Chrome web browser. It can be installed by pointing Chrome at <http://restconsole.com>.

SOAP WITH JAX-WS AND APACHE CXF

JAX-WS is the Java API for XML web services. JAX-WS provides a set of annotations to simplify the development of SOAP-driven web services. Mule supports JAX-WS via Apache CXF. To expose BrewService via SOAP, you'll need to annotate the implementation class, BrewServiceImpl, just as you did with JAX-RS. You'll also need to extract and annotate an interface, BrewService.

APACHE CXF Apache CXF is an open source framework for building web services. Mule delegates much of its SOAP support to Apache CXF. CXF is a very powerful, but complex, piece of software whose complete coverage is beyond the scope of this book. The reader is encouraged to consult the Apache CXF User Guide here for detailed coverage: <http://cxf.apache.org/docs/index.html>.

The interface is used by CXF to generate the WSDL for the web service. This is the contract that consumers will adhere to.

Listings 3.10 and 3.11 show how you'll annotate BrewService and BrewServiceImpl to generate the WSDL along with the corresponding annotation on the implementation class.

Listing 3.10 The JAX-WS annotated BrewService interface

```
package com.prancingdonkey.service;

@WebService
public interface BrewService {
    List<Brew> getBrews();
}
```

Listing 3.11 The JAX-WS annotated BrewService implementation

```
package com.prancingdonkey.service;

@WebService(endpointInterface =
    "com.prancingdonkey.service.BrewService",
    serviceName = "BrewService")
public class BrewServiceImpl implements BrewService {

    List<Brew> getBrews() {
        return Brew.findAll();
    }
}
```

You're now ready to wire up the SOAP service to Mule. As you can see in the following listing, this is similar to the configuration with Jersey. See figure 3.9.

Listing 3.12 Expose the brew listing over SOAP using CXF

```

<flow name="brewSoapService">
    <http:inbound-endpoint address="http://localhost:8090/soap"
        exchange-pattern="request-response"/>
    <cxf:simple-service
        serviceClass="com.prancingdonkey.service.BrewService"/>
    <component class="com.prancingdonkey.service.BrewServiceImpl"/>
</flow>

```

Service interface to expose → **HTTP inbound endpoint to receive SOAP requests**

← **Implementation class**

The WSDL for this service will be available on <http://api.prancingdonkey.com/soap?wsdl>. Let's point SoapUI, a popular SOAP client, at the WSDL, generate a request, and look at the response in figure 3.10.

The pane on the left shows the operations from the WSDL, which in this case only consists of the `getBrews` operation. Thankfully, SOAP UI generates sample operations for you from the schema, so simply clicking on `getBrews` will generate the sample request in the middle column. Clicking on the green Run icon will send the request to Mule and show the response on the right-hand side.¹

JAXB You can control how your domain model is serialized by CXF and Jersey by using JAXB annotations. More details are here: <http://jaxb.java.net/tutorial/>.

CXF can also be used to consume web services. Let's assume that Arnor Accounting offers a WSDL describing their SOAP API. You can modify the flow from listing 3.1 to submit expense reports via this mechanism, as illustrated in the next listing.

Listing 3.13 Submit expense reports using SOAP

```

<flow name="brewListingOverSOAP">
    <file:inbound-endpoint path="/data/expenses/in"
        pollingFrequency="60000" />
    <cxf:jaxws-client
        clientClass="com.arnor.api.AccountingService"
        wsdlPort="SoapPort"
        wsdlLocation="classpath:/wsdl/services.wsdl"
        operation="submitExpenses" />
</flow>

```

WSDL port → **1 CXF-generated client class**

← **3 Location of the WSDL**

← **4 Operation to invoke**



Figure 3.9 `brewSoapService` flow in Mule Studio

¹ SOAP UI is an invaluable tool for testing web services.

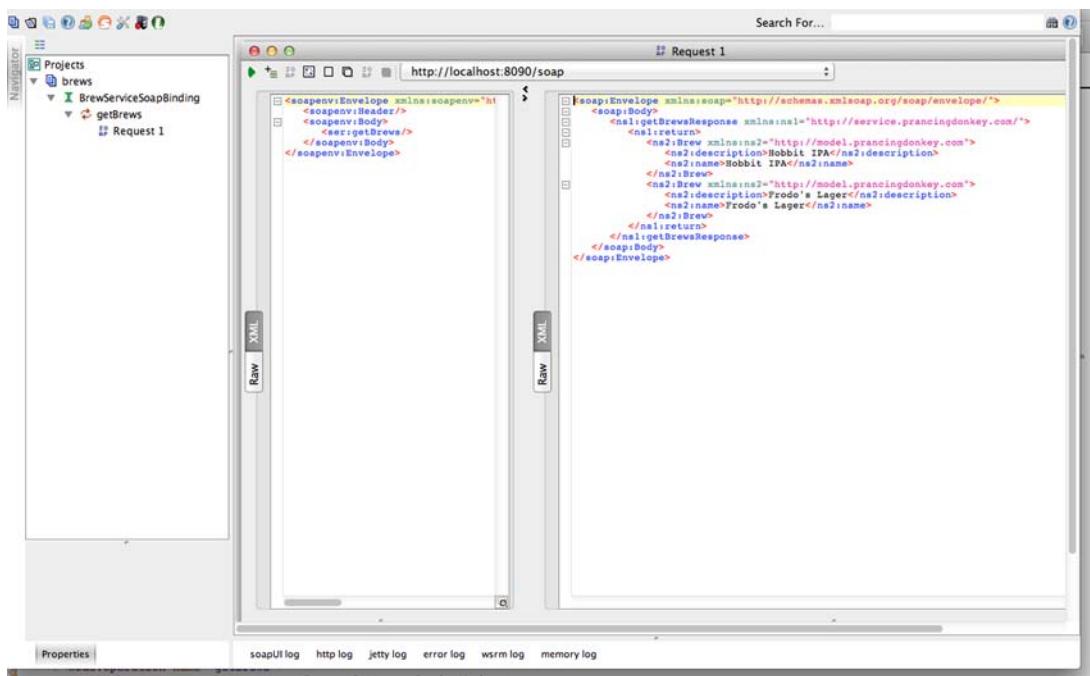


Figure 3.5 Using SOAP UI to invoke the SOAP service

Once you generate a client class using CXF's wsdl2java, you can consume the web service from Mule. This generated class is configured at ①. The WSDL port and location are defined at ② and ③. The operation to invoke is set at ④.

In this section, you learned how to use HTTP to exchange data. You saw how Mule's HTTP transport enables you to send and receive messages over HTTP, as well as how to asynchronously handle HTTP requests. You looked at using Mule's support for JAX-RS and JAX-WS to allow you to expose your data using REST and SOAP. You also saw how to use WSDLs to consume SOAP services on endpoints.

THE SIMPLE SERVICE PATTERN You'll see in chapter 6 how the simple service pattern can be used to expose some POJOs via SOAP and REST with just one line of configuration. This approach might make more sense if your services are simple, or you don't need to standardize them with JAX-WS and JAX-RS.

HTTP is an extremely popular way to do data exchange, but there are more robust options available, especially when you have control over the applications in question. In the next section, we'll consider how you can use JMS to do reliable messaging.

3.4 Using the JMS transport

The prevalence of HTTP makes web services an attractive alternative for integration outside of the firewall. Unfortunately, HTTP wasn't designed as an integration mechanism, and it fails to provide guarantees about delivery time, reliability, and security. The WS-* specifications attempt to make headway in these areas, but they're complicated, and their domain is limited to SOAP. Fortunately, the JMS protocol is an attractive alternative for integration inside the firewall. Let's take a look at Mule's JMS support.

If you're working in a Java environment and have control over the network between your applications, using JMS can make 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, and you can even pass around serialized objects between JVMs.

In this section, we'll explore Mule's support for JMS, starting off by seeing how you can send messages to and from queues and topics. We'll then look at how you can use filters to be selective about the JMS messages you receive and send. Finally, you'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 like ActiveMQ, HornetQ, or Tibco EMS.

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 3.3 lists some of these.

Table 3.3 Common configuration properties for the JMS transport

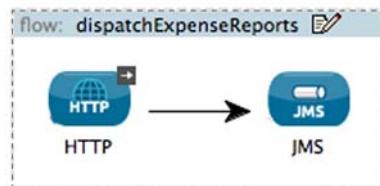
Property	Type	Target	Required?	Default	Description
queue	String	Inbound endpoint, outbound endpoint	Either queue or topic must be set		The queue to send to; can't be used in conjunction with topic
topic	String	Inbound endpoint, outbound endpoint	Either queue or topic must be set		The topic to send to; cannot be used in conjunction with queue
persistent-Delivery	Boolean	Connector	No	false	Toggle persistent delivery for messages
acknowledgement Mode	String	Connector	No	AUTO_ACKNOWLEDGE	Set the acknowledgement mode: AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, or DUPS_OK_ACKNOWLEDGE
durable	Boolean	Connector	No	false	Toggle durability for topics

Table 3.3 Common configuration properties for the JMS transport (continued)

Property	Type	Target	Required?	Default	Description
specification	String	Connector	No	1.0.2b	Specify which JMS specification to use, either 1.1 or 1.0.2b
honorQosHeaders	String	Connector	No	false	Specify whether or not to honor quality-of-service headers

3.4.1 Sending JMS messages with the JMS outbound endpoint

Let's send some messages to a queue using a JMS outbound endpoint. Recall that listing 3.3 set up a flow to accept notifications from Arnor Accounting when an expense report's processing has been completed. That flow accepts the notifications and saves them to a file. A more realistic use case is to take the notifications and dispatch them to a JMS topic to which interested parties can subscribe and be notified as expense reports are finalized, as illustrated in the following listing (see figure 3.11).

**Figure 3.11** `dispatchExpenseReports` flow in Mule Studio

Listing 3.14 Publish expense reports to a JMS topic

```

<jms:activemq-connector name="jmsConnector"
                        specification="1.1"
                        brokerURL="${jms.url}" />

<flow name="dispatchExpenseReports">
    <http:inbound-endpoint exchange-pattern="one-way"
        host="${http.host}"
        path="/expenses/status"
        method="POST"
        port="${http.port}"
        connector-ref="httpConnector"/>
    <jms:outbound-endpoint topic="expenses.status"/>
</flow>

```

① Configures the activemq-connector
② The HTTP inbound-endpoint
③ Send JMS messages to the topic

JMS brokers typically require slightly different configuration options. As such, you need to explicitly configure the connector for your broker. You're doing this for an external ActiveMQ instance at ①. The broker's URL is being set to the value of the \${jms.url} property. The JMS specification is declared here as well. If you wanted to use the 1.0.2b spec, you'd simply change 1.1 to 1.0.2b. The HTTP inbound endpoint configured at ② remains unchanged. The JMS outbound endpoint is defined at ③. The string from the inbound endpoint will be sent to the expenses.status 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 becomes a MapMessage, an input stream becomes a StreamMessage, an object becomes an ObjectMessage, and a String becomes a TextMessage, as you've just seen.

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

```
<jms:outbound-endpoint queue="expenses.status" />
```

Messages now sent through this endpoint will be placed in a queue called `expenses.status`.

SPECIFYING JMS CONNECTORS ON ENDPOINTS If you're specifying the JMS destinations using the URI notation (for example, `jms://expenses.status`), you can identify the connector to use for the endpoint by appending `?connector-ref=connector-name`. For instance, the outbound endpoint's URI for the previous example would look like this: `jms://expenses.status?connector=jmsConnector`. This syntax is necessary when more than one connector is configured for a transport.

3.4.2 Receiving JMS messages with the JMS inbound endpoint

Let's look at receiving JMS messages on an inbound endpoint. Listing 3.15 subscribes to the topic you're publishing to (see previous listings) and logs that an expense report has been processed (see figure 3.12).

Listing 3.15 Logging JMS messages published to a topic

```
<flow name="logExpenseReports">
    <jms:inbound-endpoint topic="expenses.status"
                           exchange-pattern="one-way" />
    <logger level="INFO"
           message="Expense Report Processed:
#[org.mule.util.DateUtils.getTimeStamp('dd-MM-yy_HH-mm-ss.SSS')]" />
</flow>
```

The inbound endpoint is configured at ①; it will consume messages off the `expenses.status` topic. The logger message processor is used to log messages via Mule's logging facilities. We'll talk more about logging in chapter 8.

You also have the ability to make a topic subscription durable 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://mq.prancingdonkey.com:61616" durable="true" />
```

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

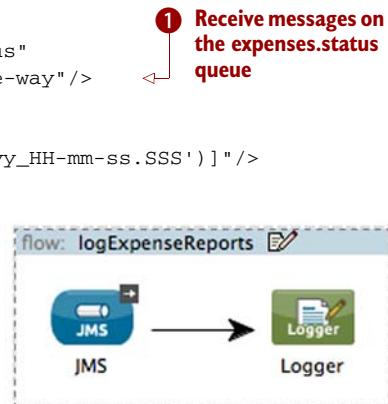


Figure 3.12 `logExpenseReports` flow in Mule Studio

3.4.3 Using selector filters on JMS endpoints

Filters can be used on JMS endpoints to be selective about the messages they consume. JMS inbound endpoint filters use the JMS selector facility to accomplish this. Let's modify the JMS inbound endpoint from listing 3.15 to only accept notifications created after midnight on January 1, 2012. You can see this in the following listing.

Listing 3.16 Using a JMS selector filter to choose the messages an endpoint receives

```
<jms:inbound-endpoint topic="expenses.status">
    <jms:selector expression="JMSTimestamp > 1325376000000" /> ① Define JMS selector
</jms:inbound-endpoint>
```

You can use JMS selectors in this manner on any header property. You might be wondering what the `>` characters are all about at ①. This is the XML escape sequence for `>`, the greater-than character. Failing to escape characters like `>` in your Mule configurations will produce XML parsing errors when Mule starts.

3.4.4 Using JMS synchronously

As JMS is inherently asynchronous in nature, you'll usually use JMS inbound endpoints with one-way message-exchange patterns—sending messages and not waiting around for a response. Sometimes, however, you'll want to wait for a response from a message you're sending. You can accomplish this by setting the exchange pattern on a JMS inbound endpoint to request-response.

Let's demonstrate this by invoking BrewServiceImpl's `getBrews()` method using JMS in the following listing (see figure 3.13).

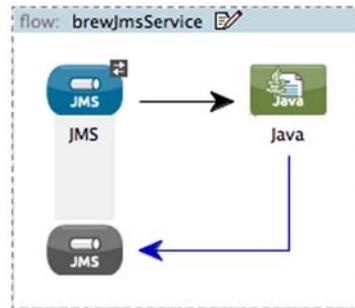


Figure 3.13 `brewJmsService` flow in Mule Studio

Listing 3.17 Synchronously invoke a component using JMS

```
<flow name="brewJmsService">
    <jms:inbound-endpoint queue="brews.list"
        exchange-pattern="request-response" /> Use JMS inbound endpoint instead of HTTP
    <component class="com.prancingdonkey.service.BrewServiceImpl" />
    <mule-xml:object-to-xml-transformer />
</flow>
```

The request-response exchange pattern on the JMS inbound endpoint indicates that a response is expected. To facilitate this, the JMS transport will create a temporary queue for the response data and set the queue name as the `Reply-To` property of the responding JMS message.

SYNCHRONOUS JMS AND NON-MULE CLIENTS Synchronous JMS may seem deceptively useful; note that it has its disadvantages. It will work out of the box with other Mule instances as well as `MuleClient`, but will require manually dealing with the `reply-to` header and temporary queue when implementing non-Mule

callers. In general, you should avoid using JMS in this manner when integrating with non-Mule clients unless you have the ability to manually set the reply-to header on the non-Mule end.

The ability to use JMS with Mule is an important part of your integration tool belt. You just learned how to send and receive messages with JMS endpoints using queues and topics.

JMS AND RECONNECTION STRATEGIES An important facet of working with JMS is dealing with situations in which the JMS servers are unavailable. In chapter 9 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.

You saw how to use filters on JMS endpoints to be picky about the messages you send and receive. Finally, we looked at how to use Mule's request-response support with JMS endpoints to perform synchronous operations over asynchronous queues. Let's take a further look at Mule's transport options, continuing with sending and receiving email.

3.5 **Using email**

While it's tempting to think of email as primarily for person-to-person communication, it's often used for more than a mechanism of conversation. Email messages relay monitoring alerts, send order receipts, and coordinate scheduling. This is evident from the automated emails you receive when you sign up with a website, order a book from Amazon, or confirm 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.

3.5.1 **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 is 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.

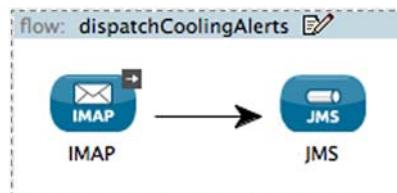
Table 3.4 lists some of the more common configuration properties of the IMAP transport.²

² POP3 support is also available and functions similarly to the IMAP transport.

Table 3.4 Common configuration properties of the IMAP transport

Property	Type	Target	Required	Default	Description
mailboxFolder	String	Connector	No		The IMAP folder to read messages from
backupFolder	String	Connector, inbound endpoint	No		If backupEnabled is true, the directory to back up mail messages to
backupEnabled	Boolean	Connector, inbound endpoint	No	false	Whether or not to store copies of read email messages
deleteReadMessages	Boolean	Connector	No	true	Whether or not to delete messages from the mail server once they have been read; if false, the messages are marked as SEEN on the mail server
checkFrequency (ms)	Integer	Connector	No	60000 ms (1 minute)	Interval at which to poll the server for new messages

Prancing Donkey recently purchased a new cooling system, manufactured by Frozen North Freezing, Inc., for their brewery. The cooling system has a feature that enables it to send an email to a specified address when certain environmental thresholds are exceeded. Prancing Donkey wants to use these emails to drive some ancillary functionality. For instance, they want their maintenance staff to receive SMS messages when these events occur. They also want to push the data into a complex event processing system so they can possibly infer when a catastrophic failure might be imminent. To accomplish this, they're going to use the IMAP transport in conjunction with a JMS topic and the file transport, as shown in the next listing (see figure 3.14).

**Figure 3.14** `dispatchCoolingAlerts` flow in Mule Studio**Listing 3.18 Dispatch cooling emails to a JMS topic**

```

<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="${jms.url}"/>

<imap:connector name="imapConnector"
    deleteReadMessages="true"

```

```

        checkFrequency="60000" />

<flow name="dispatchCoolingAlerts">
    <imap:inbound-endpoint
        host="${imap.host}"
        port="${imap.port}"
        user="cooling"
        password="password">
    </imap:inbound-endpoint>
    <all>
        <file:outbound-endpoint path=".//data/cooling/reports">
            <expression-transformer
                expression="#{message.inboundAttachments.report}" />
        </file:outbound-endpoint>
        <jms:outbound-endpoint topic="cooling.alerts">
            <email:email-to-string-transformer/>
        </jms:outbound-endpoint>
    </all>
</flow>

```

All router dispatches to multiple endpoints

Expression transformer is used to extract email attachment from MuleMessage

Configures IMAP connector

Configures IMAP inbound endpoint with mail server details

Convert the email to a string

The all router lets you dispatch the same message to multiple endpoints and will be considered in depth in chapter 5. For now you'll use it to save an attachment on the email (in this case a PDF of the cooling report) to an archive directory, and then send the content of the email to a JMS topic. Certain transports, like SOAP and HTTP with SOAP, support attachments. When this is the case, you can use the attachments evaluator, like we do in listing 3.18, to gain access to the attachments.

As the requested changes are only allowed on an IMAP connector, you need to explicitly define one ①. Set the checkFrequency to an hour (in milliseconds). Since Mule is the only client of this folder, you're also setting the deleteReadMessages property to true. This will remove messages from the server as they're read, conserving disk space and making the mail provider happy. Note the use of the email namespace ②. The email namespace contains message processors, like the email-to-string transformer you see here, to simplify working with mail messages. You'll see more about transformers in chapter 4.

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

3.5.2 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. You've already seen how Prancing Donkey's cooling

system uses SMTP to send alert data. You've also no doubt received automated emails confirming purchases, forum subscriptions, and so on. 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 3.5 lists the common properties for configuring SMTP. In particular, the properties allow you to configure how the headers of the outbound emails are generated.

Table 3.5 Common configuration properties of the SMTP transport

Property	Type	Target	Required?	Description
subject	String	Connector, outbound endpoint	No	The default message subject
from	String	Connector, outbound endpoint	No	The from address
replyToAddresses	String	Connector, outbound endpoint	No	A comma-separated list of reply-to addresses
ccAddress	String	Connector, outbound endpoint	No	A comma-separated list of bcc addresses

Let's revisit the Arnor Accounting example from earlier in this chapter. If you recall, you modified listing 3.14 to dispatch a JMS message whenever an HTTP callback was received from the Arnor Accounting application. In listing 3.15, you added a subscriber to this topic that would log these events. Let's add an additional subscriber that sends an email to Prancing Donkey's accountant to give her a heads-up when expense reports are processed (see figure 3.15).

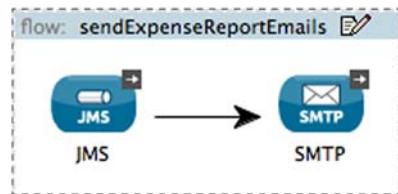


Figure 3.15 `sendExpenseReport-Emails` flow in Mule Studio

Listing 3.19 Using an SMTP endpoint to send an email

```

<flow name="sendExpenseReportEmails">
    <jms:inbound-endpoint topic="expenses.status"
        exchange-pattern="one-way" />
    <smtp:outbound-endpoint from="mule@prancingdonkey.com"
        host="${smtp.host}"
        port="${smtp.port}"
        subject="Expense Report Processed"
        to="accounting@prancingdonkey.com">
        <email:string-to-email-transformer/>
    </smtp:outbound-endpoint>
</flow>
  
```

Transform message payload to an email

Subscribe to expenses.status topic

Configure SMTP outbound endpoint

The JMS inbound endpoint configured at ① will pick up the notifications as they're published to the `expenses.status` topic. The payloads of these messages will be converted to emails and sent through the SMTP outbound endpoint to `accounting@prancingdonkey.com`.³

We just looked at how to receive email messages with the IMAP transport and send email messages with the SMTP transport. You saw how to augment the example from the beginning of this chapter to control how messages are downloaded from an IMAP server. You 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.

Now that you're comfortable sending and receiving emails with Mule, let's turn our attention to a old but venerable transport protocol: FTP.

3.6 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 take a look at using the FTP transport to send and receive data. First we'll look at how you can poll a remote FTP directory. We'll then show how you can send data to a remote FTP site using an outbound endpoint.

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

Table 3.6 The FTP transport's configuration, very similar to that of a typical FTP client

Property	Type	Target	Required?	Default	Description
<code>pollingFrequency</code> (ms)	long	Connector, inbound endpoint	No	0	For inbound endpoints, the frequency at which remote directory should be read
<code>outputPattern</code>	String	Connector, outbound endpoint	No		Specify format of files written by outbound endpoints
<code>binary</code>	Boolean	Connector, inbound endpoint, outbound endpoint	No	true	Use binary mode when transferring files

³ Since you're using topics, messages published to `expenses.status` will arrive for all listeners, including the logging flow from listing 3.15. If you were using queues, then the inbound endpoints would compete for messages. This can be used to horizontally load-balance messages across consumers.

Table 3.6 The FTP transport's configuration, very similar to that of a typical FTP client (continued)

Property	Type	Target	Required?	Default	Description
passive	Boolean	Connector, inbound endpoint, outbound endpoint	No	true	Use passive mode when transferring files
user	String	Connector, inbound endpoint, outbound endpoint	No		The username to use when connecting to remote FTP server
password	String	Inbound endpoint, outbound endpoint	No		The password to use when connecting to remote FTP server
host	String	Inbound endpoint, outbound endpoint	No		The host of remote FTP server
port	int	Inbound endpoint, outbound endpoint	No	21	The port of remote FTP server

3.6.1 Receiving files with inbound FTP endpoints

Certain merchants only support FTP as an integration mechanism to supply sales data to Prancing Donkey. To receive this data, Prancing Donkey is using Mule to poll the remote FTP server and save the data to a filesystem for manual processing later on. The next listing illustrates how they've accomplished this (see figure 3.16).

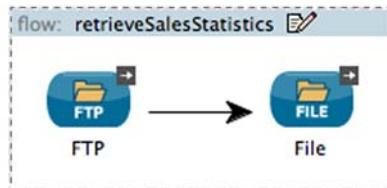


Figure 3.16 `retrieveSalesStatistics` flow in Mule Studio

Listing 3.20 Polling a remote FTP directory every hour for new files

```

<flow name="retrieveSalesStatistics">
    <ftp:inbound-endpoint user="joe" password="123456"
        host="${ftp.host}"
        port="${ftp.port}"
        path="/ftp/incoming"
        pollingFrequency="3600000" />
    <file:outbound-endpoint path="./data/sales/statistics"/>
</flow>

```

Configures **ftp inbound endpoint** ①

② **Saves transferred files to ./data/sales/statistics**

The inbound endpoint is configured at ①. You specify 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 at ②. The outbound endpoint will write the file to the `./data/sales/statistics` directory using the same filename as on the server.

BEST PRACTICE Always consider using the file transport in conjunction with FTP inbound endpoints. While you can stream FTP data through a flow, issues can arise if the files are large. Some FTP servers, for instance, have aggressive timeouts that can bite you if you’re processing many files as you download them. A better option is to send the FTP files to a file outbound endpoint, and perform your processing from there.

3.6.2 Sending files with outbound FTP endpoints

Sometimes you’ll need to send a file to a remote FTP server. The same retailer as in the last example has requested that Prancing Donkey send a subset of their product catalog periodically via FTP. To facilitate this, Prancing Donkey has set up a shared folder to drop these files into that Mule will poll and periodically send over FTP, illustrated in the next listing (see figure 3.17).

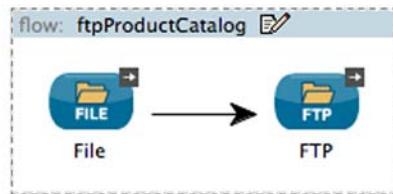


Figure 3.17 `ftpProductCatalog` flow in Mule Studio

Listing 3.21 Sending a file to a remote FTP server

```

<flow name="ftpProductCatalog">
    <file:inbound-endpoint path=".//data/in"/>
    <ftp:outbound-endpoint user="joe" password="123456"
                           host="${ftp.host}" port="${ftp.port}"
                           path="/data/prancingdonkey/catalog"/>
</flow>
  
```

① Poll the folder for files
② Send the files to the FTP server

The file inbound endpoint is configured at ①. As files are placed into the `./data/in` directory, they’ll be passed to the FTP outbound endpoint ②. This will place the file into the `/data/prancingdonkey/catalog` directory of the FTP server.

You just saw how to configure Mule to poll and submit files via FTP. We’ll now take a look at how you can use the JDBC transport to get data into and out of a database.

3.7 Using databases

Databases are, often unfortunately, 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 use 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.

A recent trend in database technology dubbed NoSQL offers alternative data solutions that are nonrelational. Such technologies often make use of the prevalence of cloud and distributed computing infrastructures to facilitate easy horizontal scaling and the ability to work with tremendous sets of data.

In this section, we'll start off by looking at the JDBC transport. First you'll see how to use a JDBC inbound endpoint to perform queries. Then we'll take a look at using JDBC outbound endpoints to perform insertions. We'll then take a look at MongoDB: a document-oriented, NoSQL database.

3.7.1 Using a JDBC inbound endpoint to perform queries

Let's look at how to configure the JDBC transport. Table 3.7 shows some common configuration properties. The `dataSource-ref` in particular is important; this is the reference to the configured data source 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 transport's `dataSource` reference, `pollingFrequency`, and `queryKey`

Property	Type	Target	Required?	Description
<code>dataSource-ref</code>	String	Connector	Yes	The JNDI or bean reference of the <code>dataSource</code>
<code>pollingFrequency</code>	long	Connector, inbound endpoint	No	How often the query is executed
<code>queryKey</code>	String	Connector, inbound endpoint, outbound endpoint	No	Specify the query to use (see how to define queries ahead)

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.

Prancing Donkey uses MySQL as the database for their web store. Web orders fulfillment goes through a third-party fulfillment service. Prancing Donkey periodically polls this service's API and updates their MySQL database with the status of the order. Sometimes orders don't get fulfilled in a timely fashion. When this happens, someone from Prancing Donkey needs to get in touch with the fulfillment service and figure out why the order is stuck. To alert Prancing Donkey operations when this happens, they've implemented a Mule flow that polls the database for orders that have been unfulfilled for more than a day, and then sends an alert to a JMS topic, as illustrated in this listing (see figure 3.18).

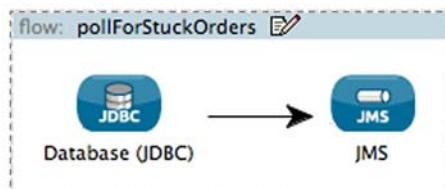


Figure 3.18 `pollForStuckOrders` flow in Mule Studio

Listing 3.22 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="${jdbc.driver}"/>
  <spring:property name="url"
    value="${jdbc.url}"/>
  <spring:property name="username" value="${jdbc.username}"/>
  <spring:property name="password" value="${jdbc.password}"/>
</spring:bean>

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
  <jdbc:query key="stuckOrderQuery"
  value="SELECT id FROM orders WHERE TIMESTAMP(timestamp)
  < { fn TIMESTAMPADD( SQL_TSI_DAY, -1, CURRENT_TIMESTAMP ) } "/>
</jdbc:connector>

<flow name="pollForStuckOrders">
  <jdbc:inbound-endpoint pollingFrequency="3600000"
    queryKey="stuckOrderQuery"
    connector-ref="jdbcConnector"/>
  <jms:outbound-endpoint topic="orders.status.stuck"/>
</flow>

```

The diagram illustrates the configuration flow for Listing 3.22. It starts with step 1, 'Configure JDBC datasource', which points to the first part of the XML code where a Spring datasource is defined. Step 2, 'Configure JDBC connector', points to the definition of the JDBC connector that references the datasource. Step 3, 'Define SQL query', points to the SQL query defined within the JDBC connector. Finally, step 4, 'Execute query once an hour', points to the polling frequency and endpoint definitions in the flow.

We'll start by examining ①. This configures a Spring datasource with the details of the MySQL database read from a properties file. You reference this datasource on the JDBC connector defined at ②. The actual query used is defined at ③. If you look closely, you'll notice you're escaping out the less-than sign in the query to avoid XML parsing issues when Mule starts up. This query will return a row for every stuck order inserted into the database in the last hour. Each member of the result set will be sent as an individual message to the JMS topic.

The payloads of the messages are `java.util.Map` instances. The `Map` representations can be transformed to other object types, like POJOs or XML, using transformers that we'll cover in chapter 4. The map's keys are the column names of the result set. The key values are the row values for each result.

Let's see how you can insert rows into a database with 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. Recall that listing 3.21 used an FTP outbound endpoint to send product data to a retailer. Let's modify this flow to additionally insert the product data into the database. This will give Prancing Donkey a single point of entry from which to load product data from the filesystem.

Listing 3.23 Using a JDBC outbound endpoint to insert rows into a table

```

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="productInsert"
        value="INSERT INTO products VALUES
        (0, #[payload.name], #[payload.acv],
        #[payload.cost], #[payload.description])"/>
</jdbc:connector>

<flow name="ftpProductsFromDatabase">
    <file:inbound-endpoint path=".//data"/>
    <byte-array-to-string-transformer/>
    <all>
        <processor-chain>
            <custom-transformer name="csvToMapTransformer"
            class="com.prancingdonkey.transformer.CSVToListOfMapsTransformer"/>
            <foreach>
                <jdbc:outbound-endpoint queryKey="productInsert"
                    connector-ref="jdbcConnector"/>
            </foreach>
        </processor-chain>
        <ftp:outbound-endpoint
            user="admin"
            password="123456"
            host="#{ftp.host}"
            port="#{ftp.port}"
            path="/data/prancingdonkey/catalog"/>
    </all>
</flow>

```

The diagram illustrates the flow of data from CSV to database. It shows five numbered steps: 1. Define JDBC connector with insert statement (points to the JDBC connector definition); 2. Dispatch to both JDBC and FTP (points to the all router); 3. Transform CSV data into a List of Maps (points to the custom transformer); 4. Iterate over elements in the List (points to the foreach message processor); 5. Insert each Map into database (points to the JDBC outbound endpoint).

You define your insert statement at ①. This is expecting a Map payload, which you reference by key with MEL. An all router is being used at ② to dispatch the product data to both the JDBC outbound endpoint and the FTP outbound endpoint. We'll discuss routing further in chapter 5. The product data is in CSV format, so you have to reference a transformer ③ to convert the CSV to a List of Map instances. We'll discuss transformers in depth in chapter 4. The foreach message processor, which we'll discuss in depth in chapter 5, iterates over each element of the list ④ and invokes the JDBC outbound endpoint to insert each Map using the query defined at ⑤.

BATCH INSERTS Mule Enterprise Edition allows you to batch database operations. If you were using the EE version of the JDBC transport, then you could simply pass the List of Maps to the JDBC endpoint. The entire list would be inserted in a single batch, and you could avoid having to use the foreach processor. This also allows you to perform the insert within a single transaction.

Now that you're comfortable retrieving and inserting data with a relational database, let's take a look at MongoDB.

3.7.3 NoSQL with MongoDB

MongoDB is a leading NoSQL database. It exposes data as JSON documents in *collections*, and also uses JSON for querying. MongoDB makes it easy to achieve horizontal scalability on commodity or cloud infrastructures. In addition to its facilities as a

document store, MongoDB also offers a solution for distributed file storage via GridFS. In this section, we'll look at the MongoDB connector for Mule. You'll see how to insert documents into collections as well as how to execute a query against a collection over HTTP.

THE MULESOFT CONNECTORS LIBRARY A wealth of community-contributed extensions for Mule are available from the MuleSoft Connectors Library: www.mulesoft.org/connectors. In chapter 13, you'll see how Mule's DevKit will enable you to author your own extensions to Mule.

Let's start by implementing another subscriber to the `cooling.alerts` topic. The following listing illustrates how Prancing Donkey stores their cooling alert data in a MongoDB collection (see figure 3.19).

Listing 3.24 Save cooling alerts to a MongoDB collection

```
<mongo:config name="mongoDB"
    database="prancingdonkey"
    username="${mongo.user}"
    password="${mongo.password}"
    host="${mongo.host}"/>
<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="${jms.url}"/>

<flow name="saveCoolingAlerts">
    <jms:inbound-endpoint topic="cooling.alerts"
        connector-ref="jmsConnector" />
    <mongo:json-to-dboobject/>
    <mongo:insert-object collection="cooling_alerts"
        config-ref="mongoDB" />
</flow>
```

Transform alerts → **Configure MongoDB transport to connect to MongoDB**

→ **Save alert to the cooling_alerts collection**

You might notice that the MongoDB operations look a lot different from the inbound/outbound endpoint configurations we've been working with so far in this chapter. This is because MongoDB support in Mule is implemented as a cloud connector, which we discussed at the beginning of this chapter. These differ from transports in that they expose the individual methods of an API rather than a messaging channel to get data into and out of flows.

Because the MongoDB cloud connector exposes the methods of the MongoDB Java driver, you need to ensure your message payload is in a format that API expects. For this case, you need to use the `json-to-dboobject` transformer to convert the JSON payload to an instance of `com.mongodb.DBObject` before you can invoke the `insert-object` message processor. Now let's see how you can query documents in a MongoDB collection (see figure 3.20).

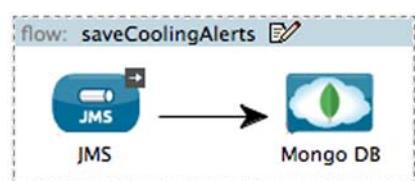


Figure 3.19 `saveCoolingAlerts` flow in Mule Studio

The following listing defines a flow that allows a user to supply a JSON query in a URL to receive back the matching documents from the configured MongoDB collection (see figure 3.20).

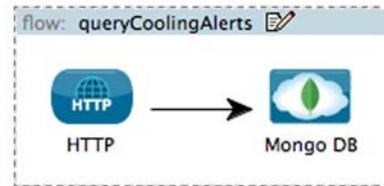
Listing 3.25 Query a MongoDB collection over HTTP

```
<flow name="queryCoolingAlerts" >
    <http:inbound-endpoint host="${http.host}"
                           port="${http.port}"
                           path="alerts/cooling"/>
    <byte-array-to-string-transformer/>
    <mongo:find-objects config-ref="mongoDB"
                         collection="cooling_alerts" />
    <mongo:mongo-collection-to-json/>
</flow>
```

Transform collection to JSON

1 Define http:inbound-endpoint to accept the query

Execute query against the collection



The HTTP inbound endpoint ① allows a client to submit queries to return a set of documents from the `cooling_alerts` collection encoded as JSON. A wealth of other operations are available from the Mule MongoDB module. The full listing can be found here: www.mulesoft.org/muleforge/mongodb-connector.

You saw in this section how databases, both relational and NoSQL, can be used with Mule. We demonstrated how the JDBC transport can be used as a message source, allowing you to repeatedly generate Mule messages from database queries. You also saw how to insert data into tables using a JDBC outbound endpoint. Finally, we looked at how to insert and query documents in MongoDB, an attractive NoSQL contender.

Let's now switch gears and take a look at a slightly different sort of transport: the VM transport.

3.8 Using the VM transport

The VM transport is a special kind of transport that you'll use to send messages inside the JVM in which the Mule instance is running. Messages sent over VM endpoints can be made transactional and persisted to disk. This enables you to layer-in reliability and decouple your flows without the need for an external messaging broker. We'll take a look at how to do that in this section. We'll revisit implementing reliability patterns with the VM transport in chapter 9, when we discuss Mule's support for transactions.

Table 3.8 shows some of the properties you can set for the VM transport.

Table 3.8 Configuring the VM transport

Property	Type	Target	Required?	Description
path	String	Endpoint	Yes	The name of the VM queue
queueTimeout	long	Connector	No	How often queued messages are expunged

3.8.1 Introducing reliability with the VM transport

You saw in listing 3.2 how Prancing Donkey posts expense reports to a URL provided by Arnor Accounting. The approach they came up with read expense reports off a shared directory in the filesystem and subsequently posted the spreadsheet to the URL. Since neither the filesystem nor HTTP are transactional resources, however, a failure in either operation could result in a lost expense report. As such, let's take the first step in making this flow reliable by decoupling each endpoint with VM endpoints, illustrated in the following listing (see figure 3.21).

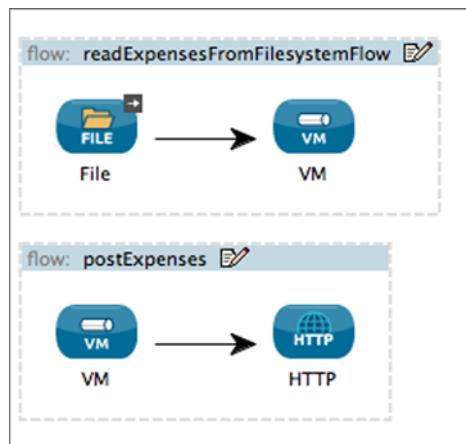


Figure 3.21 VM flows in Mule Studio

Listing 3.26 Using VM endpoints to implement reliability patterns

```

<flow name="readExpensesFromFilesystemFlow">
    <file:inbound-endpoint path=".//data/expenses/in"
                           pollingFrequency="60000"
                           >
        <file:filename-regex-filter pattern=".*xls$"
                                   caseSensitive="false" />
    </file:inbound-endpoint>
    <vm:outbound-endpoint path="expenses" />
</flow>

<flow name="postExpenses">
    <vm:inbound-endpoint path="expenses" />
    <http:outbound-endpoint host="${http.host}"
                           port="${http.port}"
                           method="POST"
                           path="expenses/prancingdonkey" />
</flow>

```

← Dispatch messages to the expenses VM queue

← Accept messages off the expenses VM queue

This approach is analogous to using a JMS queue to decouple the flows. The benefit here is that you don't incur the administrative and performance overheads of running a JMS broker to reap the benefits of durable, asynchronous messaging. The message being passed over the expenses queue will be persisted to disk, so in the event of a Mule shutdown or failure it will still be delivered to the HTTP outbound endpoint. In chapter 9 you'll see how to make the dispatch and reception of the message transaction, facilitating true reliability between both flows.

VM QUEUES AND FLOWS Using VM queues was a common way to compose services together in Mule 2. The introduction of flows in Mule 3 and their supporting functionality, like flow refs and subflows, makes the use of the VM

transport less useful in this context. They do, however, provide a foundation for reliability patterns, which we'll discuss in chapter 7.

The characteristics of how the VM transport stores its messages can be configured by defining the queue profile as a child of a VM connector. Some properties of the queue store are shown in table 3.9.

Table 3.9 Configuring a queue store for the VM transport

Property	Type	Target	Description
persistent	Boolean	queue-profile	Whether or not messages are persisted to disk; default is true
maxOutstandingMessages	long	queue-profile	The maximum number of undelivered messages to keep on the queue

Listing 3.27 demonstrates a queue profile configuration that turns off persistence (all messages kept in memory) and limits the number of messages to 1000. Turning off persistence can be useful if you can tolerate message loss and are interested in maximizing throughput over the VM queues.

VM TRANSPORT EXCHANGE PATTERNS The VM transport supports the request-response exchange pattern and behaves much like the JMS transport in this respect.

Listing 3.27 Explicitly configuring a VM queue profile

```
<vm:connector name="fastVMQueue">
    <vm:queue-profile persistent="false" maxOutstandingMessages="1000" />
</vm:connector>
```

Define nonpersistent queue profile
that buffers up to 1000 messages


The VM transport will deliver messages as fast as it gets them, which can be considerably faster than JMS. Keep this in mind when you tune your flows and components; it's very easy to overload a flow with VM queues. The `maxOutstandingMessages` attribute ① tells the VM transport to buffer up to 1000 messages if they're arriving too quickly to be processed by the flow on demand. Messages arriving after the buffer has been filled will be lost. We'll discuss tuning in detail in chapter 11.

Using the VM transport gives you some of the benefits of decoupling middleware without the overhead of administering a broker. You saw in this section how the VM transport facilitates the implementation of reliability patterns—techniques you can use to make unreliable transports, like file or HTTP, reliable. You also saw how to configure VM queue stores, which let you override the default queuing and persistence behaviors of the VM transport.

Now that you've seen a solid selection of Mule's transports and connectors, we'll take a look at a special kind of connector, the cloud connector, which facilitates API integration.

3.9 Using the Twitter cloud connector

The API explosion of recent years is probably evident to most readers of this book. It's become the exception for applications, of almost every variety, to not offer some kind of public API to facilitate external integration. This is great in principle, but in practice it introduces the same challenges we discussed at the beginning of this chapter. Even when strict standards are adopted, like SOAP, client code still needs to be either generated or written before the APIs can be consumed. Then, of course, the developers are forced to deal with the inevitable bugs, tuning, peculiarities, and so on that come with each API.

Mule's cloud connectors attempt to solve these problems by offering prepackaged integration with various APIs that integrate fully with Mule's XML configuration as well as Mule Studio. As of this writing, cloud connectors currently exist for Facebook, SAP, Twitter, Salesforce, Twilio, and eBay. Let's take a look at how Prancing Donkey uses the Twitter cloud connector to help in its marketing activities.

3.9.1 Twitter

It's hard for anyone to be unaware of the success of Twitter, the immensely popular microblogging service. Prancing Donkey, like most companies, makes extensive use of Twitter as part of its marketing strategy. They've gone as far as automating some of their status updates, including publishing their new brews as they're added to the system. The next listing illustrates how they do this (see figure 3.22).

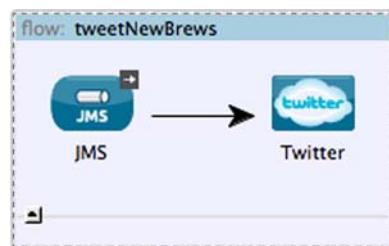


Figure 3.22 `tweetNewBrews` flow in Mule Studio

GETTING CONSUMER KEYS Before you can use the Twitter connector, you'll need to register your forthcoming Mule application to obtain consumer keys. Apps can be registered via <https://dev.twitter.com>. Note that you'll need a Twitter account to register an application.

Listing 3.28 Publishing Twitter status updates from a JMS topic

```
<twitter:config consumerKey="12345abcdef" consumerSecret="12345abcdef" />
<flow name="tweetNewBrews">
    <jms:inbound-endpoint topic="brews.new" />
    <twitter:update-status status="Check out #[payload.name] and
        #[payload.type],
        our brand new #[map-payload:type]!" />
</flow>
```

Configure Twitter authorization information

Update Twitter status using the given Mule expression

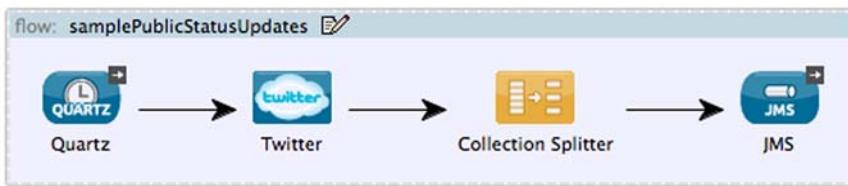


Figure 3.23 `samplePublicStatusUpdates` flow in Mule Studio

This flow subscribes to the JMS topic `brews.new` and uses the message payload, which is a Map in this case, to update prancingdonkey's Twitter status. The Twitter connector exposes most methods of the Twitter API. The following example illustrates how to subscribe to a subset of the Twitter public status updates (see figure 3.23). Some of Prancing Donkey's more ambitious marketing personnel might use this to gauge public sentiment prior to launching a new brew.

Listing 3.29 Sampling the Twitter public status update stream

```

<flow name="samplePublicStatusUpdates">
    <quartz:inbound-endpoint repeatInterval="60000"
        startDelay="0" jobName="tweetPoller" >
        <quartz:event-generator-job/>
    </quartz:inbound-endpoint>
    <twitter:get-public-timeline/>
    <collection-splitter/>
    <jms:outbound-endpoint topic="tweets" />
</flow>
  
```

Send each message to tweets JMS topic

Use Quartz transport to generate a trigger once a minute

Retrieve sample of the Twitter public timeline

Split collection result into individual messages

This listing uses the Quartz transport, which we'll cover in depth in chapter 14, to generate an event every minute. This event will trigger the Twitter `get-public-timeline` message processor, which will return a collection of status updates. You pass this to a collection splitter, which we'll cover in chapter 5, which splits the collection into individual messages. Each of these is subsequently sent to the JMS topic `tweets` for consumption by interested parties.

3.10 Summary

In this chapter, you've seen how you (and Prancing Donkey) can use Mule's transports and cloud connectors 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 letting 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. In the next chapter, you'll see how to use Mule's transformation capabilities to convert data coming from endpoints to different formats.



Transforming data with Mule

This chapter covers

- How transformers behave and how you can work with them
- General-purpose transformers from Mule's core library
- Specialized transformers from the XML module
- Transformers for JSON from the JSON module
- Custom transformers made with JVM scripting languages

Every application nowadays understands XML or JSON 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 developers, 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 requires 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 bestrewn with data transformation challenges.

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

4.1 Working with transformers

A Mule transformer has a simple behavior, as illustrated by the flow chart in figure 4.1. As this diagram suggests, a transformer strictly enforces the types of data it receives and outputs. This can be relaxed by configuration; 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.

A transformer can alter a message in different ways:

- *Payload type transformation*—The data type of the message payload is transformed from one binary 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 byte array containing an XML document instance is transformed into a byte array containing a plain-text instance for logging. It's also possible to transform from one format to the same format, such as when transforming from noncanonical to canonical XML formats.
- *Properties transformation*—The properties of the message are modified, whether by adding, removing, or modifying existing properties. For example, a message needs a particular property to be set before being sent to a JMS destination.

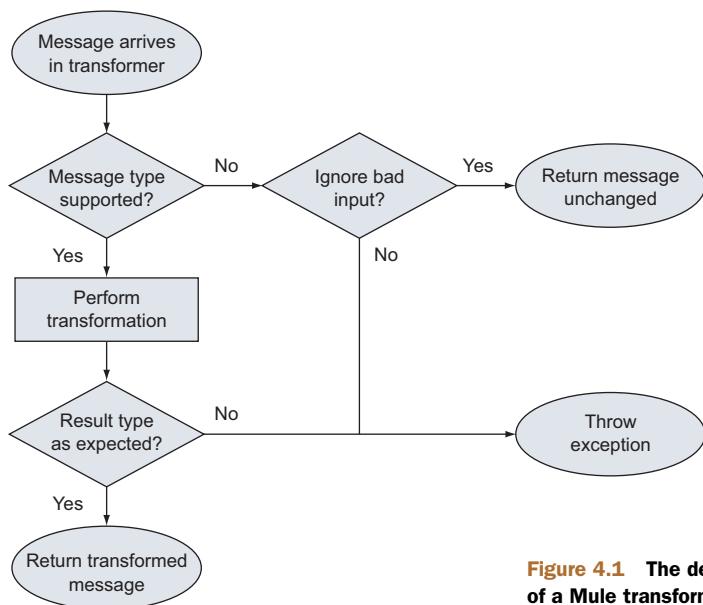


Figure 4.1 The default behavior of a Mule transformer

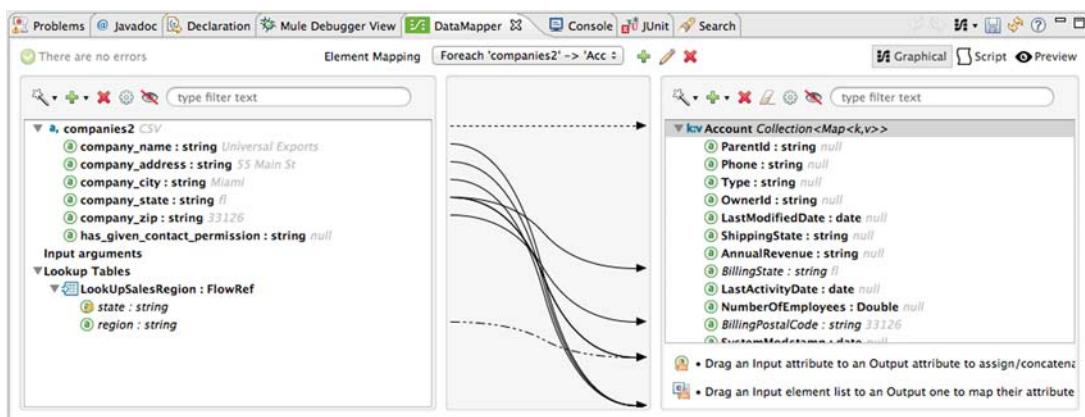
Transformers often come in pairs, with the second transformer able to undo what the first one has done. For example, compression and decompression, which are payload-format transformations, are handled by two different transformers. A transformer able to reverse the action of another one is called a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

Mule is extremely rich in 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 4.3.
- Modules can also contain transformers; in sections 4.4 and 4.5, we'll take a look at some of those that come with the XML module and JSON module, respectively.
- Transports and cloud connectors may provide transformers as well.

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

TRANSFORMING AND MULE ENTERPRISE EDITION Mule Enterprise Edition comes with a simple yet powerful graphical data-mapping and transformation system: DataMapper. It provides a graphical facility to define message and payload transformation that currently supports XML, JSON, CSV, POJO, Excel, and fixed-width files. It provides a simpler alternative to implementing either programmatic Mule transformers or complex XSLT transformations.



4.2 Configuring transformers

Before we discuss how to use transformers, we should discuss their scopes and their common configuration attributes. As with endpoints, which we discussed in chapter 3, transformers can be configured to be local or global. For instance, in listing 4.1 you can see a flow with a VM inbound endpoint declared as a local endpoint that will encode its input in Base64.

Listing 4.1 Encoding a payload in Base64

```
<flow name="base64-flow">
    <vm:inbound-endpoint path="base64.in"
        exchange-pattern="request-response" />
    <base64-encoder-transformer />
</flow>
```

This transformer is deemed local because it doesn't have a name attribute and can't be referenced. This kind of declaration can be a viable option for extremely short configurations. As soon as your configuration won't fit fully on one or two screens, or whenever you have a transformer that's shared by several flows, you might risk losing track of the different transformers you use and might miss opportunities to reuse them in different places. When your configuration starts to grow, prefer using global transformers and reference them from your endpoints. For the sake of brevity, we'll often use local transformers in the upcoming listings.

On the other hand, the following configuration fragment shows the declaration of a global object to a byte-array transformer (discussed in section 4.3.1) named `ObjectToByteArray`.

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

Now you know how to declare a basic transformer. The next natural step is to configure it. When configuring, you should consider that transformers support four, usually optional, common configuration attributes:

- `ignoreBadInput`—The purpose of this Boolean attribute has been demonstrated in figure 4.1. 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 enforce a stricter type than the default one of a transformer (for example, a transformer might target `java.lang.Object`, whereas you want to enforce that it produces only `java.util.Map` objects).
- `encoding`—This represents the encoding that should be used for the transformer result. Setting the encoding could be useful to deal with old, non-Unicode systems. This is only useful in transformers such as the `string-to-byte-array-transformer` for which the output is encoded.
- `mimeType`—Sometimes return class and encoding might not be enough; in those cases, it might be useful to configure an exact MIME type. For instance, a `String` `UTF-8` result potentially could be `text/plain`, `text/tab-separated-values`, or something else.

It's worth emphasizing that those aren't the only configuration attributes a transformer supports; there could be other attributes specific to each kind of transformer. For example, the `xslt-transformer` accepts an `xsl-file` attribute, and the `jaxb-object-to-xml-transformer` needs a `jaxbContext-ref`. We'll cover these transformers in sections 4.4.2 and 4.4.3, respectively.

Now that you know how to configure a transformer, let's have some fun chaining a few of them. You have two choices to chain transformers: you can put them in a flow in the desired order, or you can use the `transformer-refs` element that will accept a whitespace-separated list of transformer references. Let's see both styles in action in the next listing.

Listing 4.2 Two styles of references to declare a chain of three transformers

```
<object-to-byte-array-transformer name="objectToByteArray" />
<object-to-string-transformer name="objectToString" />
<mulexml:xslt-transformer name="prancingToBM"
    xsl-file="xsl/prancing-to-gondor-bm.xsl" />
<flow name="transformerRefsFlow">
    <http:inbound-endpoint
        host="localhost" port="8080"
        transformer-refs=
            "objectToByteArray prancingToBM objectToString" />
    <logger message="Message received!" />
</flow>
<flow name="transformerFlow">
    <http:inbound-endpoint
        host="localhost" port="8081" />
    <transformer ref="objectToByteArray" />
    <transformer ref="prancingToBM" />
    <transformer ref="objectToString" />
    <logger message="Message received!" />
</flow>
```

The code listing shows two ways to declare a chain of three transformers. The first approach uses the `transformer-refs` attribute on the `flow` element, separated by spaces. A red callout points to this line with the text "Whitespace-separated list of transformer references". The second approach uses the `ref` attribute on each `transformer` element within the `flow`. A red callout points to this section with the text "List of 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 will try to load the configuration.

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.

4.3 Using core transformers

There are dozens of 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 six of them that illustrate common payload type and property 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 a payload
- *Modifying properties*—Working with the transformer that fiddles with the properties of a message
- *Using the expression language*—Creating a new payload with a transformer that can evaluate expressions
- *Enriching messages*—Adding more information to your message without completely transforming it
- *Automagic transformation*—Automatic selection of the best transformer available to get the desired result

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

4.3.1 **Dealing with bytes**

Bytes are the finer grain of data unit that's 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 are able to 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 build a `String` from the bytes. The following code 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.

U+2764 Unicode

Have you ever entered your accented first, last, or street name in a website and received a series of question marks in the confirmation screen in return? If so, you've been the 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 will 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 the encoding. Note that

UTF-8 is nowadays a common encoding, as it's backwards 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 defined in the transformer, and then at the encoding of the message. If none is specified, the encoding defined on the inbound endpoint at which 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 3, 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 you'd expect, 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 to copy all the bytes of an incoming input stream into an array:

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

HINT Sometimes you might need to have a human-readable description of an object, for example, for debugging purposes. Instead of using object-to-byte-array and obtaining bytes you need to interpret correctly, you can use instead the object-to-string transformer that will take care of the encoding automatically.

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. Why would you 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 will 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.

BEST PRACTICE When dealing with transient data that may not be available for the time needed to process the whole flow (for example, FTP data), consider using the file, VM, or JMS transports to send the message to a reliable outbound endpoint and perform your processing from there.

If you list all the byte-related transformers in Mule 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 other ones. 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.

Now you have a good idea of the transformers you can use to transform your data from bytes to other forms. Let's cover how to deflate message payloads.

4.3.2 Compressing data

Under their byte representations, messages can become big—to the point that they aren't sent practically 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.

How do you 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 are in such a situation, the `gzip-compress-transformer` is the one you're looking for. In the next listing, you'll see how to use it.

Listing 4.3 Compressing a payload using gzip

```
<string-to-byte-array-transformer />
<gzip-compress-transformer />
<jms:outbound-endpoint
    queue="compressedDataQueue"
    connector-ref="dataJmsConnector"/>
```

Why do you use two transformers? Why can't you just apply the `gzip-compress-transformer`? The reason why you use a `string-to-byte-array-transformer` before the `gzip-compress-transformer` is subtle. Because the endpoint receives a `java.lang.String` payload and because `String` implements `Serializable`, the natural behavior of the compressor would be to serialize the string first, and then compress it. But what you want to send to the JMS queue are the bytes that constitute the string in a compressed manner. This is why you use the `string-to-byte-array-transformer` first.

Conversely, if the receiving consumer was a Mule JMS inbound endpoint, you'd have to use several transformers—in fact, the round-trip twins of the ones on the outbound endpoint demonstrated previously—but in reverse order. This is demonstrated in the following listing.

Listing 4.4 Uncompressing a payload using gzip

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

The transformers we've shown so far were all performing payload-type transformations. Let's now look at a transformer able to modify message properties.

4.3.3 Modifying properties, flow variables, and session variables

In sections 2.3.1 and 2.3.2, we discussed message properties and the different scopes they can have. Whether they were called message properties, flow variables, or session variables, you should've already been exposed to the notion of extra chunks of data carried alongside the main payload of a message. For example, the headers that you send with data in an HTTP POST action are the inbound properties of a message whose payload would be the body of the HTTP operation.

Adding, copying between scopes, and modifying or removing properties or variables is therefore an important aspect of dealing with messages in Mule. This is where the property transformer, the variable transformer, and the session variable transformer (as they're known in Mule Studio) come in handy. Even if they're shown in Mule Studio as a single transformer for simplicity (you can see the three of them chained in figure 4.2), you should know that each of them offers a set of message processors if you're writing your Mule applications in plain XML. In table 4.1, you can find the corresponding message processor to use depending on the desired scope and operation.

Table 4.1 Transformers for properties, flow variables, and session variables

Operation	Property	Flow variable	Session variable
Set	set-property	set-variable	set-session-variable
Remove	remove-property	remove-variable	remove-session-variable
Copy	copy-properties		

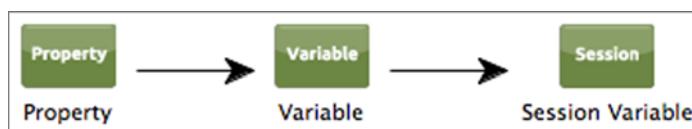


Figure 4.2 A chain of a property transformer, a flow variable transformer, and a session variable transformer

The set and remove operations are self-descriptive. They add or remove an item in the corresponding type. But what's that `copy-properties` element that's available for properties and not for variables or session variables? You may remember from section 2.3.2 that properties have inbound and outbound scopes, but variables and session variables don't. The `copy-properties` element will copy a property present in the inbound scope to the outbound one. The inbound properties aren't sent to the next endpoint, and they get lost. That's why `copy-properties` can be useful if you plan to pass inbound properties to the next endpoint.

Let's consider its different operations while looking at a few examples.

Removing properties that you don't need is a common operation. Listing 4.5 shows a transformer that removes some properties that aren't supposed to arrive at the next endpoint.

Listing 4.5 Removing properties

```
<remove-property propertyName="temp.productId" />
<remove-property propertyName="temp.customerId" />
<remove-property propertyName="prancing.totalValue" />
```

If the keys of the properties follow a name convention, it might be useful to delete groups of properties. The `remove-property` operation lets you do it easily with a wildcard or a regular expression as key, as you can see in the next listing.

Listing 4.6 Removing multiple properties

```
<remove-property propertyName="temp.*" />
<remove-property propertyName="prancing*" />
```

Annotations for Listing 4.6:

- A red bracket on the left side of the first line is labeled "Regular expression to delete any header starting with temp."
- A red bracket on the left side of the second line is labeled "Wildcard expression to delete any header starting with prancing"

If you need to set properties that might arrive at the next endpoint, you need to use the `set-property` element. You can see how simple it is in this listing.

Listing 4.7 Adding properties with set-property

```
<set-property propertyName="Content-Type"
    value="application/vnd.ms-excel" />
<set-property propertyName="Content-Disposition"
    value="attachment; filename=stats.csv" />
```

You may also need to rename an existing property. Listing 4.8 demonstrates two transformers that rename a property `prancing.isbn` to `prancing.productid`.

Listing 4.8 Renaming properties

```
<set-property propertyName="prancing.isbn"
    value="#[message.outboundProperties['prancing.productid']]" />
<remove-property propertyName="prancing.productid" />
```

What happens when you need a property from one scope to be present in a different scope? For instance, you might need to copy the property `PrancingDonkeyRequestId` from inbound to outbound so that the requestor can have it as a response property. The solution in the following listing uses the `copy-properties` transformer.

Listing 4.9 Copying a property from inbound to outbound scope

```
<copy-properties propertyName="PrancingDonkeyRequestId" />
```

The elements for variables and session variables have been intentionally designed to be similar, almost identical, to the ones you learned before. In the next listing, you can see how to add a variable in the flow.

Listing 4.10 Setting a variable

```
<set-variable variableName="prancingVariable"  
value="Value or expression" />
```

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 it's to satisfy an internal need in your Mule instance or for an external transport or remote system. Let's now explore a last core transformer that's a valuable resource.

4.3.4 Transforming with expressions

In section 2.4, you learned about the support for expressions that exists in Mule. The `expression-transformer` is able to use these expressions to transform the payload of the message it processes. This transformer can be configured to evaluate one or several expressions. Depending on the configuration, the resulting message payload will be an object (single expression) or an array of objects (multiple expressions).

Internally, Prancing Donkey has to deal with a lot of internet addresses for such things as monitoring the activity of their clients for security and statistics. They use instances of `java.net.InetAddress` as the payload of administrative messages that run around in their Mule instances. Unfortunately, one of their statistics applications needs to receive only the host IP of the client, whereas the security system needs the host IP and to know whether it's multicast or not.

To feed them the right information, you use an expression transformer returning a single value in listing 4.11 for the statistics system, whereas you return two values in listing 4.12 to extract the relevant bits for the security system.

Listing 4.11 Expression transformer returning a single element

```
<expression-transformer expression="message.payload.hostAddress" />
```

The output of the transformer present in listing 4.11, intended for the statistics system, is a string representing the host address. Notice the use of the Mule Expression Language in the next listing.

Listing 4.12 Expression transformer returning an array

```
<expression-transformer expression=
"#{message.payload.hostAddress, message.payload.multicastAddress}"/>
```

To feed the security system with an array of objects, you take advantage of the capacity to return arrays of the Mule Expression Language; more on this in appendix A.

The expression transformer is so powerful that it can sometimes replace a trivial component. Therefore, before writing any code, check first if you can achieve your goal with the expression transformer.

4.3.5 Enriching messages

It's not always about completely transforming a message of one kind into a message of another kind; what about when you want to enrich a message with a piece of additional information? Let's see how the message enricher can help you to salt your food a bit without converting fruits into vegetables.

When a flow contains a request-response outbound endpoint, the message that comes back from this endpoint replaces whatever message was under processing in the flow before calling the endpoint. This isn't always desirable when you have subsequent message processors that need some or all of the original message context to work properly.

When faced with this kind of situation, what you need to use is a message enricher instead of an outbound endpoint. A message enricher is indeed a message processor that can perform a call to an outbound endpoint (or any message processor) and merge the response back to the message currently in flow. Message enrichers rely heavily on Mule expressions for handling the source data and its destination; the Mule Expression Language will be discussed in appendix A.

Figure 4.3 illustrates a flow used by Prancing Donkey for processing client invoices in which a message enricher is used to fetch the preferred currency code of a client and store it in an inbound header before calling a processor component.

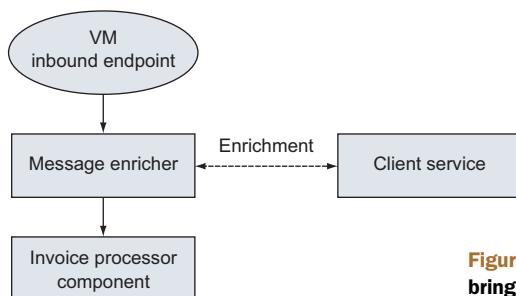


Figure 4.3 A message enricher brings extra data to the main flow.

Listing 4.13 shows the configuration for this enrichment scenario. The service called by the enricher returns the client information in a hash map, where `currencyCode` is the key under which the client currency code is stored. See how the enricher uses two expressions (discussed in section 2.4): one for extracting the desired value from the map data it fetched from the outbound endpoint call and one for setting the value on the current message in an inbound header (that is, a message property).

Listing 4.13 Enriching a message

```
<flow name="invoice-processor">
    <vm:inbound-endpoint path="invoice-processor"
        exchange-pattern="request-response" />
    <enricher source="#[message.payload.currencyCode]"
        target="#[flowVars['currencyCode']]">
        <vm:outbound-endpoint path="client.service"
            exchange-pattern="request-response" />
    </enricher>
    <component class="com.prancingdonkey.service.InvoiceProcessor" />
</flow>
```

LOOK IT UP Combined with a caching message processor, an enricher can be a efficient way of fetching lookup data.

The message enricher is versatile enough to be able to perform multiple enrichment operations using the data it fetched on its message processor. Listing 4.14 shows a new version of the previous enricher that now copies over both the `currencyCode` and `promoCode` fields from the client data hash map.

Listing 4.14 The message enricher can perform multiple enrichments at once.

```
<enricher>
    <vm:outbound-endpoint path="client.service"
        exchange-pattern="request-response" />
    <enrich source="#[message.payload.currencyCode]"
        target="#[flowVars['currencyCode']]"/>
    <enrich source="#[message.payload.promoCode]"
        target="#[flowVars['promoCode']]"/>
</enricher>
```

We've focused so far on flows and the control they give you over how messages transit within Mule. We'll now look in detail into the message interactions Mule supports and the set of abstractions that enables these interactions.

4.3.6 Automagic transformation

Before we close this section on core transformers, let's look at the auto-transformer. As its name suggests, this transformer is able to apply the 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.prancingdonkey.statistics.ActivityReport" />
```

The auto-transformer can only select transformers that are *discoverable*. It works better with custom objects, as shown in the example, instead of generic ones like strings or byte arrays. For the latter, there are far too many choices available for the auto-transformer to pick the right one.

Discoverable transformers

You already know that Mule provides several transformers to convert between common types. Those transformers implement the interface `org.mule.api.transformer.DiscoverableTransformer`. That lets Mule know the possible source and destination formats the transformers support and the priority of each of them to perform transformations. With this information, Mule is capable of choosing the most appropriate transformer for an input.

In chapter 13, you'll see how to write your own autodiscoverable transformers and how to register them in Mule for *magical* transformations.

When is using this transformer advisable? 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 use the power of expressions. Your bag of transformation tricks already allows you to perform all sorts of message manipulations. That said, none of the transformers we've looked at were performing 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.

4.4 Using XML transformers

The XML module provides several transformers, while MEL offers a global XPath function. We'll look at the XPath function and the two most significant transformers:

- *XPath function*—To query XML documents for selected parts or computed values
- *XSL transformer*—To transform an XML payload into another format thanks to XSLT
- *XML-marshaling transformers*—To marshal and unmarshal objects to and from XML

The other transformers of the XML module, not covered 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 note about the namespace prefix used for the XML module configuration elements: You'll notice that it's `mulexml`. *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 you won't be surprised by this detail, let's proceed.

4.4.1 XPath and Mule

Before we jump into the XML transformers, there's one outstanding tool in the plethora of XML tools available: the XML Path Language, or XPath. XPath is a query language used to select nodes from an XML document. It can even do simple computations, such as summing or checking the number of children of an element. The XPath expressions are present in some of the more relevant XML tools, and they're also useful by themselves.

The Mule approach to XPath is a Mule Expression Language function: `xpath`. We've already visited expression evaluators in section 2.4, so you should be familiar with their mechanism; we'll put them in action one more time. Let's say that you have this product-listing XML:

```
<products>
  <product>
    <id>1234</id>
    <type>Imported Beer</type>
    <name>Mordor's Pale Lager</name>
    <price>10.90</price>
  </product>
</products>
```

Now let's propose a popular usage scenario; you might want to take the ID and put it into a message property for later use with a message-property transformer. In listing 4.15, you see how this can be accomplished using an XPath expression.

Listing 4.15 Using a XPath expression to add a property

```
<set-property propertyName="productId"
  value="#[xpath('/products/product[1]/id').text]" />
```

① **XPath used in a Mule expression**

The XPath expression in ① will extract the value of the element `id` inside the element `product` and, thanks to the message-properties transformer that we discussed in section 4.3.3, it'll go straight to an outbound message property called `productId`. As you might already suspect, the XPath expression is using a Mule Expression Language function named `xpath`.

Sometimes you might be interested in replacing the payload in the node you selected with an XPath expression. To accomplish this, you might think of an expression transformer or an XSL transformation (which we'll visit soon), but Mule has a more configurable and faster solution for this—the `xpath-extractor-transformer`:

```
<mulexml:xpath-extractor-transformer
  expression="/products/product[1]"
  resultType="NODESET" />
```

Here you instruct the `xpath-extractor-transformer` to replace your payload. First you set an expression with the `expression` attribute to extract the first product in `products`, and then you request it to return it as a node set with the `resultType` attribute. The `resultType` could also be configured to return `Node`, `NodeSet`, `Boolean`, or `Number`.

4.4.2 **Transforming format with XSL**

XSL transformation (XSLT; see 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. What does it take to use this transformer? Here's the definition of the XSL transformer that performs a transformation to convert from Prancing Donkey's product XML to the product format required to appear in the *Gondor's Beer Magazine* search engine:

```
<mulexml:xslt-transformer  
    xsl-file="xsl/prancing-to-gondor-bm.xsl" />
```

This is nothing exciting; it creates a transformer that loads a specified file from the classpath. *Gondor's Beer Magazine* has a bit more complex format; it might require modularized XSLTs that rely on import statements. This is 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 into the filesystem 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.

The input type of an XSL transformer

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

XSL templates can receive parameters when they start processing an XML source. How do you do this in Mule? Going back to the previous example, let's pretend that you want to change the discount offered to the magazine readers from the default value to 10%. This can be achieved by passing the appropriate parameter to the XSL transformer, as shown in the following listing.

Listing 4.16 An XSLT transformation with a parameter

```
<mulexml:xslt-transformer  
    xsl-file="xsl/prancing-to-gondor-bm.xsl">  
    <mulexml:context-property  
        key="discount" value="10" />  
</mulexml:xslt-transformer>
```

Notice how you use the repeatable `mulexml:context-property` element. It's 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 best way to inject dynamic values into the XSL transformer; no, 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 `discount`. The best way to pass this value to the XSL is to use the expression language that we talked about in section 4.3.4. You can see how this works in the next listing.

Listing 4.17 Using an expression to pass a parameter to XSLT

```
<mulexml:xslt-transformer  
    xsl-file="xsl/prancing-to-gondor-bm.xsl">  
    <mulexml:context-property key="discount"  
        value="#[message.inboundProperties['discount']]"/>  
</mulexml: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:

```
<mulexml:xslt-transformer maxActiveTransformers="5"  
    maxIdleTransformers="5"  
    xsl-file="xsl/prancing-to-gondor-bm.xsl" />
```

Note that because this particular XSL is expensive to load in memory, you don't want to dereference any transformer once it's been created. This is why you have set the number of idle transformers to be the same as the maximum number of transformers. A lower number would have caused the potential destruction and re-creation of transformers.

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 will 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 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 transformation work.

You've learned how to transform the payload format with XSL, so let's see another transformer from the XML module that can deal with the payload type itself.

4.4.3 **XML object marshaling**

If you've done more than trivial tasks with Java serialization, you've realized that it's challenging at best...and challenged at worst. Alternative marshaling techniques have been developed, including creating XML representations of objects. The paired XML marshaler and unmarshaler from the XML module uses ThoughtWorks' XStream (<http://xstream.codehaus.org/>), for that matter.

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. Notice also how this transformer is able to marshal any object, unlike the byte array ones you've seen before, which were relying on Java's serialization mechanism. The following declares an XML object marshaler:

```
<mulexml:object-to-xml-transformer />
```

Nothing spectacular, but a lot is going on behind the scenes. As a variant of the previous example, the following declares a marshaler that serializes the full `MuleMessage` instead of the payload:

```
<mulexml:object-to-xml-transformer acceptMuleMessage="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 in which you want to send a `MuleMessage` over the wire without resorting to standard Java serialization.

Similarly, the declaration of the round-trip XML object unmarshaler is as trivial as this:

```
<mulexml:xml-to-object-transformer />
```

Mule also supports XStream's concept of custom aliases. XStream will, by default, name the tag elements with the fully qualified name of the classes (for example, `org.mule.DefaultMuleMessage`), and this can affect the readability of the XML. With custom aliases, you can change the element name used to marshal/unmarshal objects. Let's see custom aliases in action:

```
<mulexml:object-to-xml-transformer>
    <mulexml:alias
        class="org.mule.DefaultMuleMessage"
        name="MuleMessage" />
</mulexml:object-to-xml-transformer>
```

When marshaling `org.mule.DefaultMuleMessage`, this transformer will output XML whose root element will be named `MuleMessage`.

As you've seen, the XML module contains transformers that can be useful in different scenarios even if you don't use XML extensively.

JAXB and Mule

The out-of-the-box features of XStream (to marshal and unmarshal objects with almost no configuration) and the fact that XStream is one of the oldest players in the XML-marshaling game gave it a privileged place in the Mule XML module, where in fact it's still the main XML marshaler.

But over the past few years, the Java Architecture for XML Binding, or JAXB (<http://jaxb.java.net/>), has gained some momentum thanks to its more holistic approach. Mule has similar support for JAXB as for XStream; in fact, you'll probably find it almost identical. Let's start discussing it with an example. The following shows an XML object marshaler that will transform its input to XML:

```
<mulexml:jaxb-object-to-xml-transformer  
    jaxbContext-ref="myJaxbContext" />
```

① **JAXB transformer using a context reference**

Here you can see a similar statement to the equivalent for XStream marshaling of an object, and the behavior is also similar. You can spot one difference at ①: the reference to a `jaxbContext`. Although XStream figures out most of the required information, JAXB will require a context to be able to do its binding framework operations. A JAXB context is a collection of Java packages that holds all the information and meta-information required to marshal and unmarshal objects.

THE JAXB CONTEXT Usually the JAXB context packages are generated from an XML schema document; this requires a certain level of integration between your build tool and JAXB that goes beyond the scope of this book. For more information about creating the JAXB context, visit the JAXB tutorial at <http://jaxb.java.net/tutorial/>.

To define a JAXB context in Mule, you need to pass a list of one or more package names, separated by commas, that contain classes generated by the JAXB binding compiler or manually annotated classes. Here you can see how to define a JAXB context:

```
<mulexml:jaxb-context name="myJaxbContext"  
    packageNames="com.prancingdonkey.model.jaxb" />
```

On the other hand, the round-trip transformer to convert from XML to objects using JAXB could be easily predicted and is shown here:

```
<mulexml:jaxb-xml-to-object-transformer  
    jaxbContext-ref="myJaxbContext" />
```

Here we conclude the visit to the XML features of Mule, in which we reviewed data selection with XPath, transformation with XSLT, and marshaling/unmarshaling with XStream and JAXB. Now let's study another common format: JSON.

4.5 **Transforming JSON with Mule**

XML has been around for more than a decade. Its predecessor, SGML, originated in the 1980s. XML has an established and well-documented toolset capable of navigation, selection, validation, and marshaling. But as XML users and Peter Parker surely know:

“With great power comes great responsibility.” In this case, with great power comes great CPU consumption.

Although XML works well for a number of application scenarios, JSON has been gaining ground over the last few years. In those situations for which XML is too much, JSON is a favorable alternative. It’s not uncommon now to see big players in the cloud, such as Google or Yahoo, using JSON in some of their key services.

JSON is an open, lightweight, text-based, data-exchange format based in JavaScript. It’s human-readable, platform-independent, and enjoys a wide availability of implementations for almost all possible platforms. It’s popular in AJAX because it can easily be parsed by JavaScript, but it’s commonly used in all kinds of services and document-oriented databases.

As you might expect, Mule has first-class support for JSON. Mule is therefore able to query a JSON document and marshal between JSON and POJO. In this section, we’ll discuss these features and how they’ll help you to consume and produce JSON. Let’s start with a method to query JSON documents.

4.5.1 **Querying JSON with MEL**

In section 4.4.1 you saw how to query XML using the XPath language. Unfortunately, there’s no standard equivalent language for JSON. Mule fills the gap using a combination of the `json-to-object-transformer`, to convert the JSON to a hierarchy of objects, and the Mule Expression Language, to query it. This mechanism doesn’t have all the potential that XPath has, but it’ll probably be enough for most JSON-querying use cases.

To visit these capabilities with examples, let’s say that Prancing Donkey needs to accept stock availability requests from third parties, and as they usually use JavaScript, they want to deliver the requests in JSON.

Listing 4.18 JSON request to query stock availability

```
{
  "requestType": "availability",
  "products": [
    { "productId": "100345", "requestedUnits": "10" }
  ]
}
```

Now let’s see in listing 4.19 how to query the document for the request type and the productIDs and put it in message headers using the `message-properties transformer` we discussed in section 4.3.3.

Listing 4.19 Querying a JSON document and putting the result in headers

```
<json:json-to-object-transformer
  returnClass="java.util.HashMap" />
<set-property propertyName="requestType"
  value="#[message.payload.requestType]" />
<set-property propertyName="productId"
  value="#[message.payload.products[0].productId]" />
```

1 Extract simple item

2 Extract value inside an array

The syntax couldn't be simpler; if an item of the path is an array, use [n] to index the array. Following this simple format, you can see how you select the `requestType` value at ①, and how at ② you select the `productId` value of the first child of the `products` array.

BEST PRACTICE This JSON query method is a simplistic tool that doesn't have all the features that XPath has. As it doesn't have filtering or aggregate functions, it'll only work well with simple scenarios. For the cases in which it doesn't fit, you can try Groovy or JSON marshaling.

4.5.2 JSON object marshaling with Mule

In the same manner that you marshal to and from XML using the Mule transformers in section 4.4.3, you can marshal and unmarshal objects using JSON. This time, instead of XStream or JAXB, Mule is supported by the Jackson processor, a popular, high-performance, and open source JSON parser and generator (<http://jackson.codehaus.org/>).

To walk through the characteristics of this feature, let's try marshaling and unmarshaling a stock item. In order to do so, you need some annotated Java classes and the JSON format. First let's take a look in the next listing at the JSON entry that represents a provider for the Prancing Donkey business.

Listing 4.20 JSON representing a business provider of the Prancing Donkey

```
{
  "name": "sandyman's Mill",
  "phonenumbers": [
    { "name": "pbx", "number": "555-3456-2342" },
    { "name": "office", "email": "sandyman@hobbiton-online.local" },
    { "name": "corporative", "email": "info@sandymansmill.local" }
  ]
}
```

Now you need some Java classes that represent this JSON. You'll need a class for the provider itself that you can find in the next listing, a class for a phone number, and finally a class for an email address.

Listing 4.21 Provider.java

```
@JsonAutoDetect
public class Provider {
    private String name;
    private List<PhoneNumber> phoneNumbers;
    private List<EmailAddress> emailAddresses;
    // getters and setters here
}
```

The code is annotated with three red callouts:

- Annotation ①: `@JsonAutoDetect` is annotated with the comment: **Sets class to be inspected for accessor methods**.
- Annotation ②: `private List<PhoneNumber> phoneNumbers;` is annotated with the comment: **List containing POJOs of a different type**.
- Annotation ③: `private List<EmailAddress> emailAddresses;` is annotated with the comment: **List containing POJOs of a different type**.

You can see here a perfectly regular POJO with the exception of the annotation at ①. This is a Jackson annotation that will instruct the processor to autodetect getters and

setters in this class. Jackson also supports most of the JAXB annotations; as a result, you can use the same model to generate JSON and XML as you've already followed in section 3.3.2. There are also other, more fine-grained annotations available in Jackson. For more information, please refer to the Jackson documentation (<http://wiki.fasterxml.com/JacksonAnnotations>).

With this POJO, you almost represent a Prancing Donkey provider, but not quite, because of the items in ② and ③. You can see that there are some subelements that need to be addressed as POJOs too. As with the provider, they're going to be simple. You can find them in listings 4.22 and 4.23.

Listing 4.22 PhoneNumber.java

```
@JsonAutoDetect  
public class PhoneNumber  
{  
    private String name;  
  
    private String email;  
  
    // getters and setters here
```

Listing 4.23 EmailAddress.java

```
@JsonAutoDetect  
public class EmailAddress  
{  
    private String name;  
  
    private String number;  
  
    // getters and setters here
```

Once you have the classes that represent the JSON data, you can apply a transformer to unmarshal from a JSON document to a proper Java object. In order to unleash all this magic, you'll need to use a JSON unmarshaler transformer:

```
<json:json-to-object-transformer  
    returnClass="com.prancingdonkey.model.json.Provider" />
```

After declaring this, the name of the round-trip transformer to marshal from objects to a JSON document won't be a surprise:

```
<json:object-to-json-transformer />
```

Now you can marshal and unmarshal between JSON and annotated Java classes. But what happens when you want to use classes that can't be annotated directly? This might happen with classes from external dependencies or with other special cases that won't let you access the class source code. The solution is to use a mixin. A mixin is an interface or abstract class that defines methods with exactly the same signature as an original class so that they can be annotated. Later, a mixin map will be able to mix the annotations and the original class. You can see how you can define a mixin map in listing 4.25 that will use the class found in listing 4.24, and then finally reference it in a

JSON transformer, as you can see in listing 4.26. Note that mixins aren't available out of the box for XML.

Listing 4.24 Creating a mixin

```
@JsonAutoDetect
public abstract class ExternalItemMixin
{
    public abstract String getItemNumber();

    public abstract void setItemNumber(String itemNumber);

    @JsonIgnore
    public abstract String getUnwantedValue();

    @JsonIgnore
    public abstract void setUnwantedValue(String unwantedValue);
}
```

Listing 4.25 Definition of a mixin map

```
<json:mapper name="myMixinMap">
    <json:mixin
        mixinClass="com.prancingdonkey.model.json.ExternalItemMixin"
        targetClass="org.external.ExternalItem"/>
</json:mapper>
```

Listing 4.26 Referencing a mixin map

```
<json:json-to-object-transformer
    name="jsonToExternalItem"
    returnClass="org.external.ExternalItem"
    mapper-ref="myMixinMap" />
```

You learned how to deal with the two most common formats: XML and JSON. These are best used for small data retrievals or for big transformations such as marshaling and stylesheet templates. That will cover quite a few of the use cases for Prancing Donkey, but there are still areas that aren't covered: small, special-case transformations that we'll cover next as scripting transformers, and medium- to big-size transformation cases that will be covered in chapter 13.

4.6 **Scripting transformers**

In chapter 13, you'll see how to write a custom transformer in Java for transformations that Mule doesn't supply in its distribution, but that's not the only way to go. Mule lets you use any JSR 223-compliant script engine such as Groovy, JavaScript, Jython, or JRuby to implement custom transformations.

Let's start with a simple example of using a Groovy script to perform an inline payload transformation. The transformer in the following listing will uppercase the String payload sent through it.

Listing 4.27 Uppercasing a String payload using a Groovy transformer

```

<scripting:transformer name="groovyTransformer">
    <scripting:script engine="groovy">
        <scripting:text><! [CDATA[
            return payload.toUpperCase();
        ]]>
    </scripting:text>
</scripting:script>
</scripting:transformer>

```

The code is annotated with three numbered callouts:

- ① Define scripting transformer**: Points to the opening tag `<scripting:transformer name="groovyTransformer">`.
- ② Define groovy script**: Points to the opening tag `<scripting:script engine="groovy">`.
- ③ Uppercase message payload and return it**: Points to the Groovy script body `return payload.toUpperCase();`.

The scripting transformer is defined at ① along with an inline script that's defined at ②. As you can see, the engine is set to groovy; all JSR 223 scripting languages have an engine name, such as js, ruby, or python. You set this to indicate that you want the Groovy scripting engine to execute the supplied script. The script itself is defined at ③, and it's uppercasing the payload and returning it.

Let's now consider a more complex transformation scenario and see how it's simplified by Mule's scripting and Groovy support. Prancing Donkey has recently decided to standardize on a canonical XML model to represent order data. The web application used to submit orders, however, is still using a legacy CSV representation. Although 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. The next listing shows how an example order is represented as XML.

Listing 4.28 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>

```

The next listing demonstrates the legacy CSV format you need to convert from, using the same data as listing 4.28.

Listing 4.29 Representing an order as CSV

```
409,1234,PENDING
410,1234,PENDING
411,1235,PENDING
```

Start off by writing the script to transform the CSV to XML. You'll make use of Groovy's builder functionality to create the XML. This will most likely be a bit too verbose to include directly in the Mule config, so you'll define it in an external file, as detailed in the next listing.

Listing 4.30 Transforming a CSV payload to XML

```
port 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])
            status(fields[2])
        }
    }
}
return writer.toString()
```

The code is annotated with several red arrows and text boxes:

- A vertical arrow on the left points to the line `payload.split('n').each {line -> //}` with the text "Split payload into individual lines".
- A vertical arrow on the right points to the line `def fields = line.split(',') //` with the text "Split each line into its comma-separated values".
- A horizontal arrow points from the line `payload.split('n').each {line -> //}` to the line `builder.orders()` with the text "Use Groovy builder to construct the XML response".
- A horizontal arrow points from the line `return writer.toString()` to the text "Return result" below it.

This short script iterates over each line of the payload and constructs the corresponding XML. It makes use of Groovy's builder syntax to concisely create and return the XML response. Listing 4.31 illustrates how to configure Mule to load CSV data from a file, transform it to XML, and publish it to a JMS queue.

Listing 4.31 Using a Groovy transform to transform CSV to XML

```
<flow name="groovyCsvTransformer">
    <file:inbound-endpoint path=".//data" />
    <byte-array-to-object-transformer/>

    <scripting:transformer>
        <scripting:script
            file="orderTransformer.groovy" />
    </scripting:transformer>
    <jms:outbound-endpoint queue="orders" />
</flow>
```

The flow configuration is annotated with four numbered steps:

- Step 1: A red circle with the number 1 and the text "Wait for files to appear in supplied directory" points to the `<file:inbound-endpoint` element.
- Step 2: A red circle with the number 2 and the text "Transform byte array from the file to an Object" points to the `<byte-array-to-object-transformer/>` element.
- Step 3: A red circle with the number 3 and the text "Declare scripting transformer" points to the `<scripting:transformer>` element.
- Step 4: A red circle with the number 4 and the text "Execute script defined by the file parameter" points to the `file="orderTransformer.groovy"` attribute.

The file inbound endpoint configured at ① is configured to wait for files to appear. The files are transformed to an Object, in this case a String, by the byte-array-to-object-transformer at ②. Your scripting transformer is defined at ③. The script element is defined at ④ and will execute `orderTransformer.groovy`, whose contents are shown in listing 4.30 when invoked. This time, you 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.

4.7 **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, you've learned the way transformation occurs in Mule, what it's good for, and how to use it in your integration projects.

You've discovered some of the existing Mule transformers. Some of them came from the core library, and others came from specific modules or transports. Though they had different purposes, they were 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 in which Mule shines by its simplicity and extensibility. The several lines of code and configuration required to roll your own custom transformers hopefully convinced you of this.

In the next chapter, you'll learn how to invoke business logic with Mule components and how to simplify your projects using configuration patterns.



Routing data with Mule

This chapter covers

- Content-based routing with Mule
- Filtering messages
- Scatter/gather
- Reliability routing

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 a 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. You'll see in this chapter how to route data using Mule.

You've already seen an example of routing in chapter 3, where you saw how the choice router uses a message's payload to determine which execution path it takes. Let's see how to use Mule's routers to solve an integration problem.

In listing 3.18 in chapter 3, you saw how Prancing Donkey was dispatching alert notifications to file and JMS endpoints. As it turns out, the majority of the notifications produced by the cooling system are innocuous and don't require immediate

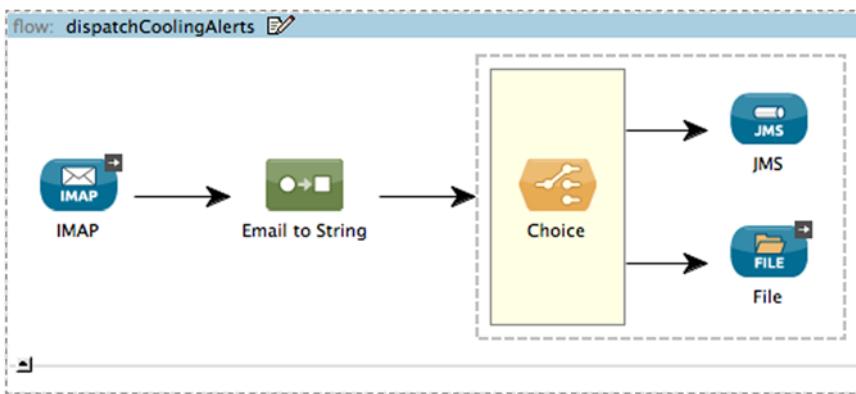


Figure 5.1 Dispatch cooling alerts

action. Let's modify the example to only route messages to the JMS queue if they're of critical importance (see figure 5.1).

Listing 5.1 Routing messages using expressions

```

<flow name="dispatchCoolingAlerts">
    <imap:inbound-endpoint
        host="${imap.host}"
        port="${imap.port}"
        user="cooling"
        password="password"/>
    <email:email-to-string-transformer/>
    <choice>
        <when expression="payload.contains('SEVERE') ">
            <jms:outbound-endpoint
                topic="cooling.alerts"
                connector-ref="Active_MQ"/>
        </when>
        <otherwise>
            <file:outbound-endpoint
                path=".//data//cooling//reports"
                responseTimeout="10000"/>
        </otherwise>
    </choice>
</flow>
  
```

❶ Use choice router
to selectively
route message

The choice router defined at ❶ evaluates the supplied expression and routes the messages to either the JMS or file endpoint. If the message's payload contains the string SEVERE, then the message is routed to JMS. Otherwise it's saved as a file to the /data/cooling/reports directory.

In this chapter, we'll investigate Mule data-routing capabilities:

- You'll learn how the Mule Expression Language (MEL) is used to decide how to route a message.

- We'll discuss Mule's filtering capabilities, examining some of Mule's more useful out-of-the-box filters.
- You'll see how Mule supports the splitting, aggregation, and collection processing of messages.
- You'll see how reliability routers introduce reliability for potentially unreliable transports like HTTP.

Framing these examples will be the use cases of Prancing Donkey. We'll revisit some of Prancing Donkey's use cases from the previous chapters and demonstrate how they can be improved or augmented with Mule's routing capabilities. We'll also expand on the domain model introduced in chapter 3, expanding its scope to include some of Prancing Donkey's operational concerns.

By the end of this chapter, you'll hopefully be able to look at your integration challenges in terms of Mule's routing concepts, making it easier for you to identify and solve integration problems.

5.1 Deciding how to route a message

The crux of any routing implementation is the logic that determines where the data goes. Mule greatly simplifies the implementation of this logic by letting you use the Mule Expression Language to determine how a message is routed.

In listing 3.3 in chapter 3, you saw how Prancing Donkey built a simple web service that allowed them to receive callback notifications from Arnor Accounting. Arnor Accounting added a new feature in which rejected expense reports would be returned and annotated where corrections needed to be made. Let's see how you can route messages with the annotated spreadsheets as payloads to a different directory than the status reports.

5.1.1 Using the choice router

The following listing demonstrates how the Content-Type header of an HTTP POST request can determine which directory an expense report is saved in (see figure 5.2).

Listing 5.2 Routing expense reports based on Content-Type

```
<flow name="expenseReportCallback">

    <http:inbound-endpoint
        exchange-pattern="request-response"
        host="localhost"
        port="8081"
        path="expenses"/>

    <choice>
        <when
            expression="message.inboundProperties['Content-Type']"
            == 'application/vnd.ms-excel'>
            <file:outbound-endpoint
                path=".//data/expenses/status"
```

Route message based
on its content type

1

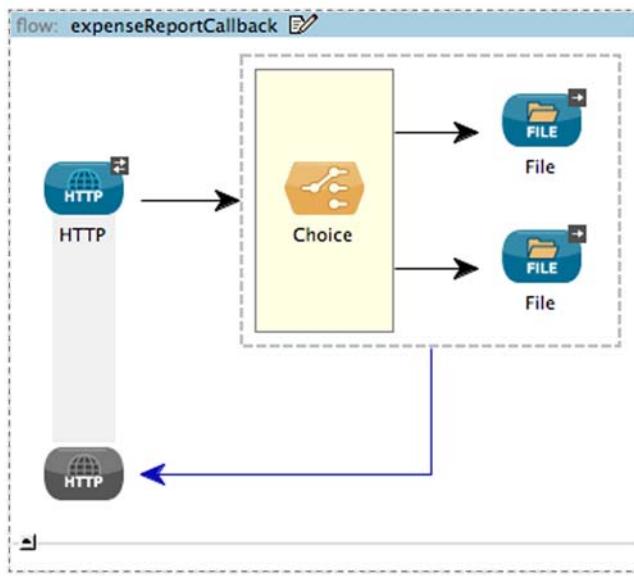


Figure 5.2 Routing expense reports based on Content-Type

```

outputPattern=
#[java.util.UUID.randomUUID()-
#[java.lang.System.currentTimeMillis()].xls"/>
</when>
<otherwise>
    <file:outbound-endpoint
path=".//data/expenses/rejected"
outputPattern=
#[java.util.UUID.randomUUID()-
#[java.lang.System.currentTimeMillis()].xls"/>
    </otherwise>
</choice>
</flow>

```

The MEL expression ① evaluates the MIME type of the message's payload and routes it to `./data/expenses/rejected` if it's an Excel spreadsheet. Another common use case is to route a message based on its Java type. This is illustrated in the next listing, where you route a message to a JMS queue depending on the class of the payload (see figure 5.3).

Listing 5.3 Routing a message based on the type of the payload

```

<flow name="expenseReportCallbackWithDLQ">
    <vm:inbound-endpoint path="orders" />
    <choice >
        <when

```

Selectively route message based on the payload's class

```

expression="payload is com.prancingdonkey.domain.Brew">
    <jms:outbound-endpoint
        queue="brews.definitions"
        connector-ref="Active_MQ" />
</when>
<when
expression="payload is com.prancingdonkey.domain.Order">
    <jms:outbound-endpoint queue="brews.orders"
        connector-ref="Active_MQ" />
</when>
<otherwise>
    <jms:outbound-endpoint queue="DLQ"
        connector-ref="Active_MQ" />
</otherwise>
</choice>
</flow>

```

This example will route a message to either the brews or orders queue based on its type. Messages whose payloads aren't of either type are routed to the dead-letter queue.

The payload itself can also be used to route a message. XML payloads, for instance, can be routed using XPath expressions. If you recall from listing 3.3 in chapter 3, Prancing Donkey implemented a Mule flow to log expense report statuses received on a JMS queue. Let's modify that example to print a different log message based on the XML content of the payload, shown in the next listing (see figure 5.4).

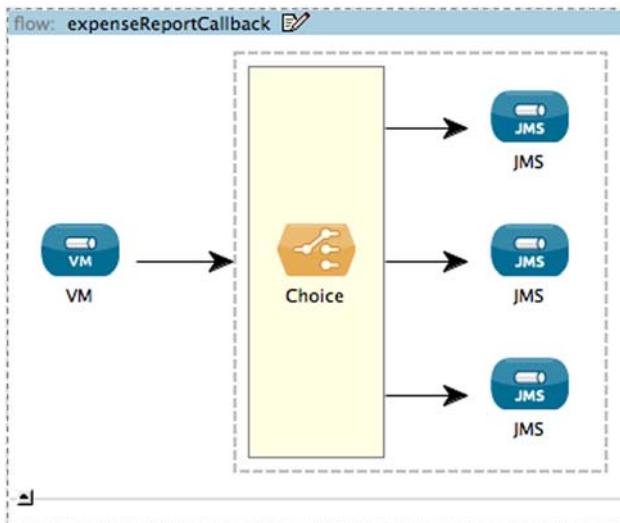


Figure 5.3 Routing a message based on the type of the payload

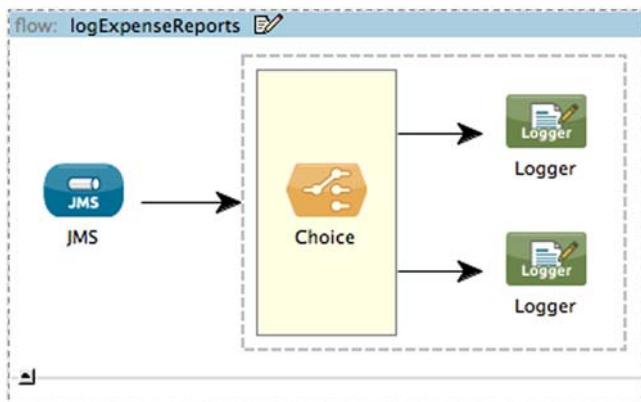


Figure 5.4 Routing a message using an XPath expression

Listing 5.4 Routing a message using an XPath expression

```

<flow name="logExpenseReports" >
    <jms:inbound-endpoint topic="expenses.status"
    connector-ref="Active_MQ" />
    <choice >
        <when
            expression="xpath('/expense/status/').text == 'PROCESSED'" >
                <logger
                    message=
                    "#[xpath('/expense/id').text] processed"
                    level="INFO"/>
            </when>
            <when expression=
                    "xpath('/expense/status').text =='FAILED'" >
                <logger message=
                    "Error processing expense report:#[xpath('/expense/id/text())]"
                    level="ERROR" />
            </when>
        </choice>
    </flow>
  
```

Route message based on result of an XPath evaluation

In this case, messages whose `status` element equals `FAILED` will be logged differently than messages that have been successfully processed.

Mule's message processors combined with the Mule Expression Language lay the foundation of data routing with Mule. You'll use MEL over the course of this chapter to decide where Mule routes messages.

5.2 Using filters

Mule's filters determine what continues processing through a flow. Filtering is typically used for policy enforcement. You might want to ensure only a certain type of JMS message is consumed from a particular queue. Mule provides dedicated routers for these common use cases. More elaborate filtering is also made easy with the Mule Expression Language, as you'll see shortly.

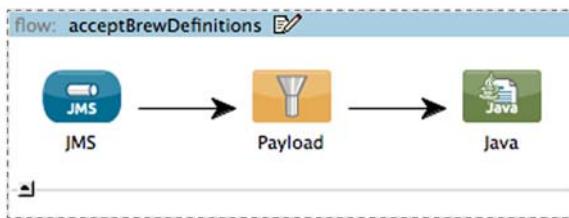


Figure 5.5 Filter messages by type

5.2.1 Filtering by payload type and header

The payload-type-filter ensures that a message is passed only if its payload is of the required type. Let's implement the receiving end of the JMS queues from listing 5.3 (see figure 5.5):

```
<flow name="acceptBrewDefinitions">
    <jms:inbound-endpoint queue="brewsdefinitions"
        connector-ref="Active_MQ" />
    <payload-type-filter expectedType="com.prancingdonkey.domain.Brew" />
    <component
        class="com.prancingdonkey.service.BrewProcessingService" />
</flow>
```

These flows are now guaranteed to only process messages with the matching payload types.¹

It's also possible to filter based on a message's headers. The following flow routes a message based on a priority header (see figure 5.6):

```
<flow name="processOrders">
    <http:inbound-endpoint exchange-pattern="request-response"
        host="${http.host}" port="${http.port}" path="orders" />
    <component
        class="com.prancingdonkey.service.OrderPreProcessingService" />
    <expression-filter
        expression="message.inboundProperties['PD_PRIORITY'] == 'HIGH'" />
        <jms:outbound-endpoint queue="orders.critical"
            connector-ref="Active_MQ" />
</flow>
```

This flow accepts order data (you'll see the exact format of it a little bit later) and sends it to a Java class for processing. If the PD_PRIORITY header is set to HIGH, then the message is routed to a JMS queue for processing critical orders.

Let's take a look at how to filter message payloads with textual or XML content.

¹ The payload-type-filter can also be implemented as an expression-filter that checks the payload's type.

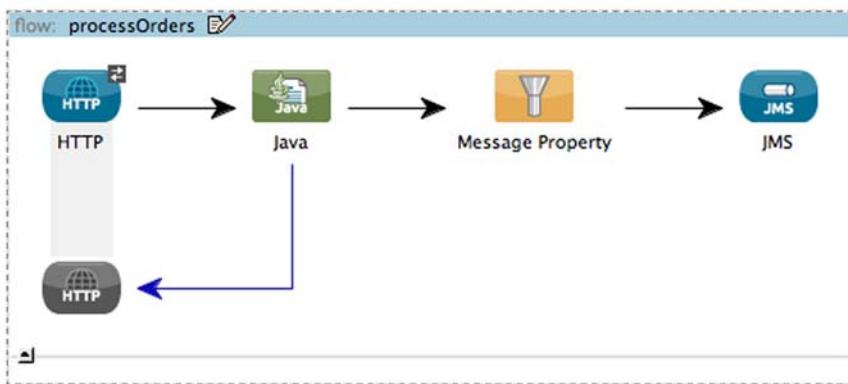


Figure 5.6 Filter messages by priority header

5.2.2 Filtering text and XML

Filtering messages with textual payloads is easy with Mule's wildcard and regular expression filters. The wildcard filter applies a shell-like wildcard to the message's payload. If the payload matches the wildcard, then the message is passed. Let's augment listing 5.2 to log all cooling reports but only dispatch those with SEVERE payloads to JMS (see figure 5.7):

```

<flow name="dispatchCoolingAlerts">
    <imap:inbound-endpoint host="${imap.host}"
        port="${imap.port}"
        user="cooling"
        password="password"
        responseTimeout="10000" />
    <email:email-to-string-transformer/>
    <wildcard-filter pattern="*SEVERE*" />
    <jms:outbound-endpoint
        topic="cooling.alerts"
        connector-ref="Active_MQ"
    />
</flow>
  
```

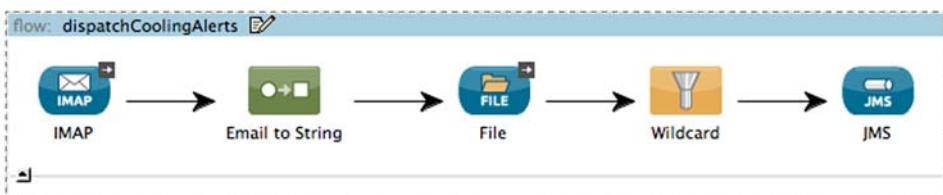


Figure 5.7 Using the wildcard filter to filter text and XML

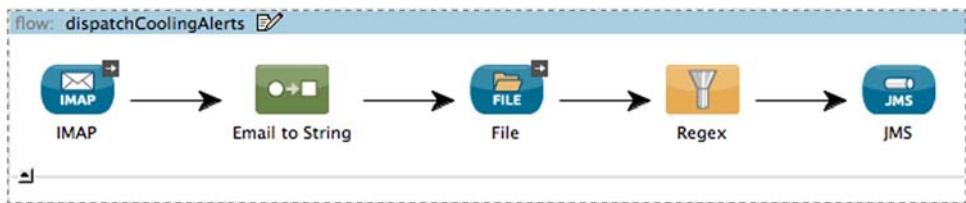


Figure 5.8 Filtering with a regular expression

You can tighten up this flow by filtering with a regular expression filter (see figure 5.8):

```

<flow name="dispatchCoolingAlertsWithRegex">

    <imap:inbound-endpoint
        host="${imap.host}"
        port="${imap.port}"
        user="cooling"
        password="password"/>
    <email:email-to-string-transformer/>
    <file:outbound-endpoint path="./data/cooling/reports"/>
    <regex-filter pattern="^LEVEL: SEVERE$"/>
    <jms:outbound-endpoint topic="cooling.alerts"
        connector-ref="Active_MQ"/>
</flow>

```

The wildcard and regex filters are useful for filtering textual payloads. Using these filters to process structured textual data like XML, however, can be cumbersome. You'll see in the next section how XPath expressions can be used to filter XML. XPath filter evaluations provide a fine-grained way to determine the validity of a message. While this is useful, you often want to ensure an XML document is valid prior to processing it. The schema-validation-filter provides this functionality.

The following example demonstrates an HTTP-to-JMS proxy that rejects messages that don't conform to the given schema (see figure 5.9):

```

<flow name="validateAndDispatchOrders">
    <http:inbound-endpoint exchange-pattern="one-way"
        host="${http.host}" port="${http.port}"
        path="orders"/>
    <mulexml:schema-validation-filter
        schemaLocations="orders.xsd"
        returnResult="true"/>
    <jms:outbound-endpoint topic="orders.submitted"
        connector-ref="jmsConnector"/>
</flow>

```

The filters we've considered so far are powerful, but somewhat simplistic. Let's see how you can use Mule Expression Language to perform powerful filtering without writing any Java code.

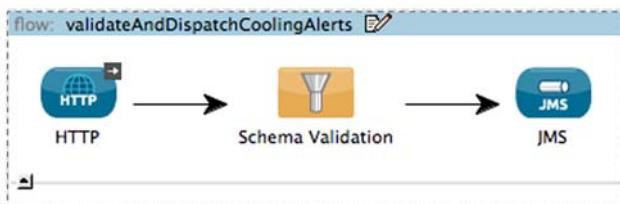


Figure 5.9 Validate and dispatch cooling alerts

5.2.3 Filtering with expressions

It's often the case that a message filtering requirement won't be easily met with the built-in filters you just saw. In many cases, however, Mule's expression filter will give you the same amount of flexibility without writing any Java code.

One of the benefits of working with structured data, such as XML or object graphs, is the availability of structurally aware query tools. In chapter 3, we introduced Java classes that began to model Prancing Donkey's domain. We'll build on this in listings 5.5 and 5.6 by introducing an Order object to model a customer order for cases of Prancing Donkey's brew and show how it can be used with Mule's expression filter.

Listing 5.5 Java class for an Order

```

public class Order implements Serializable {

    private String id;
    private String priority;
    private String customerId;
    private BigDecimal total;
    private BigDecimal shippingCost;
    private List<LineItem> lineItems;
    ...getters / setters omitted
}

```

Listing 5.6 Java class for a LineItem

```

public class LineItem {

    private Brew brew;
    private Integer quantity;
    ...getters / setters omitted
}

```

The Order class contains a collection of LineItem classes. The latter represents an order for a quantity of cases of brew, represented by the Brew object. When serialized to XML, an order might look something like the following listing.

Listing 5.7 XML representation of an Order

```

<order>
    <id>4d3196a9-aecc-4b46-be21-2b7be47a8b19</id>
    <priority>NORMAL</priority>
    <customerId>d69d3d89-9955-45c7-8167-24bf2733b347</customerId>

```

```

<total>100.23</total>
<shippingCost>21.23</shippingCost>
<lineItems>
    <lineItem>
        <brew>11</brew>
        <quantity>2</quantity>
    </lineItem>
    <lineItem>
        <brew>3</brew>
        <quantity>1</quantity>
    </lineItem>
</lineItems>
</order>

```

Now let's implement a flow that subscribes to a JMS topic for order data and performs some processing on messages that are of HIGH priority (see figure 5.10):

```

<flow name="routeHighPriorityOrdersWithXpath">
    <jms:inbound-endpoint topic="orders"
        connector-ref="jmsConnector" />
    <expression-filter
        expression="xpath('/order/priority').text == 'HIGH'" />
    <component
        class="com.prancingdonkey.service.HighPriorityOrderProcessingService" />
</flow>

```

The expression filter takes a Mule expression as its argument. In this example, you use the Mule Expression Language's `xpath` support to evaluate the given expression against the message's XML payload. The `xpath` expression will ensure that messages with high-priority order payloads will get processed by `HighPriorityOrderProcessingService`.

The Mule Expression Language also allows you to navigate arbitrary object graphs using a dot notation. You could use this to perform the same filtering if Prancing Donkey were not serializing their objects to XML prior to publishing them on the topic (see figure 5.11):

```

<flow name="routeHighPriorityOrdersWithProperties">
    <jms:inbound-endpoint topic="orders"
        connector-ref="Active_MQ"/>
    <expression-filter
        expression="payload.priority == 'HIGH'" />
    <component
        class="com.prancingdonkey.service.HighPriorityOrderProcessingService" />
</flow>

```

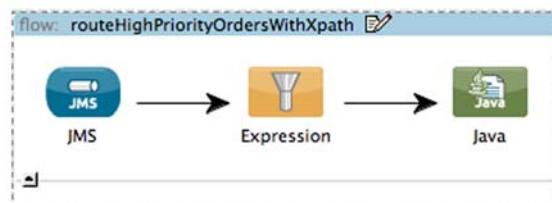


Figure 5.10 Route high-priority orders with XPath

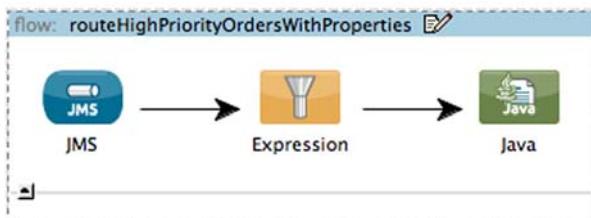


Figure 5.11 Route high-priority orders with properties

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

We'll come back to this domain model and routing expressions later in this chapter. Now we'll take a look at logical filters, which allow you to combine the various filters in composites.

5.2.4 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 modify listing 5.7 to ensure the flow only processes Order objects. Listing 5.8 contains the configuration (see figure 5.12).

Listing 5.8 Using a logical filter

```
<flow name="routeHighPriorityOrdersOfTheCorrectType">
    <jms:inbound-endpoint topic="orders"
        connector-ref="Active_MQ"/>
    <and-filter>
        <payload-type-filter
            expectedType="com.prancingdonkey.domain.Order"/>
        <expression-filter
            expression="payload.priority == 'HIGH'"/>
    </and-filter>
    <component
        class="com.prancingdonkey.service.HighPriorityOrderProcessingService"/>
</flow>
```

As you probably guessed, this filter will consume messages that are of type `com.prancingdonkey.domain.Order` and for which the supplied XPath expression evaluates to true. You're also able to nest logical filters.

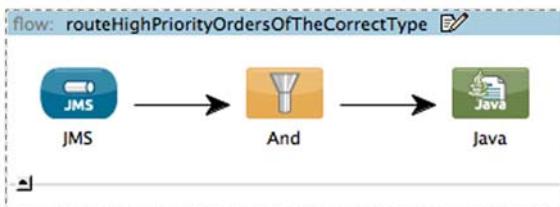


Figure 5.12 Using a logical filter

Listing 5.9 Nesting logical filters

```
<flow name="routeHighPriorityOrdersOfTheCorrectType">

    <jms:inbound-endpoint queue="orders"
                           connector-ref="jmsConnector" />

    <and-filter>
        <or-filter>
            <expression-filter
                expression="payload.priority == 'HIGH'" />
            <expression-filter
                expression="message.inboundProperties['PD_PRIORITY']
                == 'HIGH'" />
        </or-filter>
        <payload-type-filter
            expectedType="com.prancingdonkey.domain.Order" />
    </and-filter>
    <component
        class="com.prancingdonkey.service.HighPriorityOrderProcessingService" />
</flow>
```

In this example, you open the scope of the flow to process messages that have payloads of type `com.prancingdonkey.domain.Order` and have either a priority of HIGH when `Order's getPriority()` method is called or the `PD_PRIORITY` header on the message is set to HIGH.

Let's switch gears and take a look at two slightly more complicated filters.

5.2.5 Ensuring atomic delivery with the `idempotent` filter

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

Mule's `idempotent-message-filter` ensures that a flow only processes a message once. By default, the `idempotent-message-filter` will use the message's ID to determine if it's been delivered. Typically, however, this behavior is overridden by setting the `idExpression` attribute on the filter to something that makes sense in the business domain. Let's extend the example in section 5.2.4 to include an `idempotent-message-filter`. We'll configure the `idExpression` to use the order's ID to determine if the message has been processed already (see figure 5.13).

Listing 5.10 Ensuring idempotent delivery with the `idempotent-message-filter`

```
<flow name="idempotentOrderRouter">
    <jms:inbound-endpoint topic="orders"
                           connector-ref="jmsConnector" />

    <or-filter>
        <expression-filter
```

```

expression="xpath('/order/priority').text == 'HIGH' "/>
<message-property-filter pattern="PD_PRIORITY='HIGH'" 
    caseSensitive="true" scope="inbound"/>
</or-filter>
<idempotent-message-filter
    idExpression="xpath('/order/id').text"/>
<component
class="com.prancingdonkey.service.HighPriorityOrderProcessingService"/>
</flow>

```

① Use XPath to determine ID for idempotency

The `idempotent-message-filter` configured at ① uses the supplied XPath expression to determine the ID of the order. Mule will use this ID to determine if the message has been processed, and pass the message if it hasn't. By default, Mule stores the list of processed messages as text files in a directory local to the filesystem. You can override the location of this directory by setting the `simple-text-file-store` element on the filter:

```

<idempotent-message-filter
    idExpression="xpath('/order/id').text">
    <simple-text-file-store directory="/opt/idempotent"/>
</idempotent-message-filter>

```

Object stores other than files are also supported by the `idempotent-message-filter`. You can configure an in-memory store or a custom-object store. The latter is particularly useful if you're using Mule in a clustered or load-balanced environment. You could implement an object store that stores idempotent IDs in a shared location, like an Apache Cassandra instance, so that the history of processed messages is available to every Mule node in the cluster. You'll see how to implement custom object stores in chapter 12.

USING THE IDEMPOTENT-MESSAGE-FILTER WITH MULE EE The `idempotent-message-filter` is automatically cluster aware if you're running the Enterprise HA version of Mule.

Let's now investigate what happens when messages are blocked by a filter.

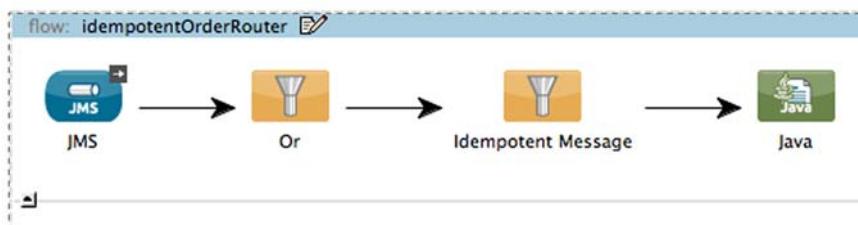


Figure 5.13 Use XPath to determine ID for idempotency

5.2.6 Using the message filter

You may be wondering what happens to messages that are blocked by one of the filters we've described. By default, no action is taken, and the message is dropped. If you want to override this behavior, for instance, to send the message to a dead-letter queue or throw an exception, you need to use a message filter. The message filter is a wrapper around a filter expression that lets you route filtered messages. Let's send blocked messages to a JMS queue, as shown in the following listing.

Listing 5.11 Route to a DLQ

```
<flow name="routeToDLQ">
    <jms:inbound-endpoint topic="orders"
        connector-ref="jmsConnector"/>
    <message-filter onUnaccepted="sendToDLQ">
        <and-filter>
            <payload-type-filter
                expectedType="java.lang.String"/>
            <or-filter>
                <expression-filter
                    expression="xpath('/order/priority').text == 'HIGH'"/>
                <message-property-filter
                    pattern="PD_PRIORITY='HIGH'"
                    caseSensitive="true"
                    scope="inbound"/>
            </or-filter>
        </and-filter>
    </message-filter>
    <component
        class="com.prancingdonkey.service.HighPriorityOrderProcessingService"/>
</flow>

<sub-flow name="sendToDLQ">
    <jms:outbound-endpoint name="DLQ" queue="DLQ"/>
</sub-flow>
```

Wrap and-filter in message-filter and route blocked messages to a DLQ

MESSAGE FILTER AND CATCH-ALL ROUTING The message filter provides functionality similar to catch-all strategies with routers used in Mule 2 services.

Setting the `onAccepted` attribute of the `messageFilter` will route the blocked message to the supplied address. The `messageFilter` takes an additional, mutually exclusive attribute called `throwOnUnaccepted`. Setting this to `true` will cause Mule to throw an exception when the message is blocked. We'll talk more about how to handle exceptions in flows in chapter 9.

At this point, you've seen how to use Mule's choice router to direct a message in the appropriate direction. We also took a look at a number of Mule's filters, which provide facilities to stop a message from being processed any further in a flow. In all the examples so far, however, there's been a one-to-one mapping between a message and its ultimate destination. Let's investigate how you can use Mule to route a message to multiple destinations.

5.3 Routing to multiple recipients

You often need to send a message to more than one endpoint. A monitoring alert, for instance, might need to reach the following destinations:

- The email address of an admin
- A local log file
- A database for archiving

In this section, we'll take a look at Mule's recipient list router, among others. These message processors will allow you to route messages to multiple endpoints. You'll see how you can also synchronously route these messages and aggregate their responses.

5.3.1 Dispatching messages with the all router

The `all` routing message processor sends a message to a group of endpoints. The next listing shows how alerts from Salesforce are routed (see figure 5.14).

Listing 5.12 Multicast data to SMTP and MongoDB endpoints

```
<flow name="processSalesForceAlert" >
    <vm:inbound-endpoint exchange-pattern="one-way"
        path="alerts.salesforce" doc:name="VM" />
    <logger message="#{payload}" level="INFO" /> ← Log alert

    <all >
        <processor-chain>
            <smtp:outbound-endpoint host="localhost"
                port="25"
                responseTimeout="10000" />
        </processor-chain>
        <processor-chain>
            <mongo:json-to-dbobject/>
            <mongo:insert-object
                config-ref="Mongo_DB"
                collection="alerts" />
        </processor-chain>
    </all>
</flow>
```

Annotations for Listing 5.12:

- Log alert**: Points to the `logger` element.
- Use all router to send message to enclosed recipients**: Points to the `<all>` element.
- Send an email based on alert**: Points to the first `<processor-chain>`.
- Write alert to a MongoDB collection**: Points to the second `<processor-chain>`.

The `all` router in this example is sending the alert to the SMTP outbound endpoint to email an administrator, and writing the alert to a MongoDB collection for archival purposes.

THE ALL ROUTER AND MULTICASTING The `all` router is analogous to the multicasting router in Mule 2.

THE ALL ROUTER AND MESSAGE COPIES The `all` router makes a copy of the `MuleMessage` for each recipient.

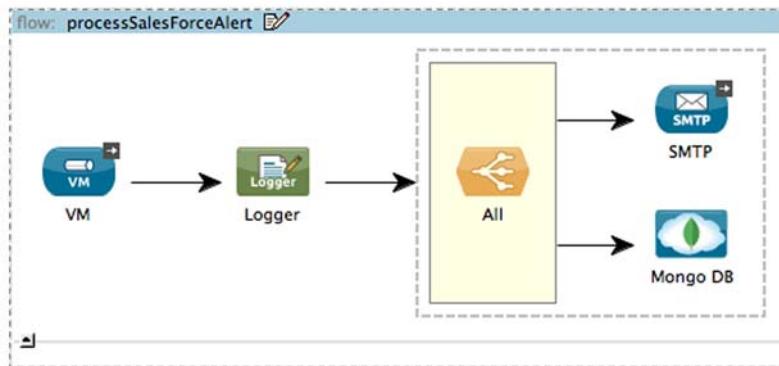


Figure 5.14 Multicast data to SMTP and MongoDB endpoints

5.3.2 Scatter/gather IO with the all router

In the previous example, your use of the all router was fire-and-forget; the goal was to dispatch the alert to the appropriate endpoints and not expect a response. The all router can also be used with the request-response exchange pattern. In this case, the responses from the outbound endpoints can be aggregated together. You'll see more about message aggregation in the next section, but let's look at an example that uses the all router in conjunction with Prancing Donkey's Order domain object to calculate the net price for an order.

Prancing Donkey has two internal web services used to calculate shipping and sales tax for an order. Let's implement a Mule flow that accepts an Order object, routes the object to these two web services, and aggregates the responses to return a net price, shown in the next listing (see figure 5.15).

Listing 5.13 Scatter/gather IO

```

<flow name="aggregateResponses" doc:name="aggregateResponses">
    <vm:inbound-endpoint exchange-pattern="request-response"
        path="order.netprice"
        doc:name="VM"/>
    <all doc:name="All">
        <http:outbound-endpoint
            exchange-pattern="request-response"
            host="api.internal.prancingdonkey.com"
            port="9091" path="shipping" doc:name="HTTP"/>
        <http:outbound-endpoint
            exchange-pattern="request-response"
            host="api.internal.prancingdonkey.com"
            port="9091" path="tax" doc:name="HTTP"/>
    </all>
    <expression-transformer
        expression="payload[0].toBigDecimal()
            + payload[1].toBigDecimal()"
        doc:name="Expression"/>
</flow>
  
```

Synchronously accept request to calculate net price ①

2 Use all router to synchronously request shipping and tax prices from appropriate web services

3 Use expression transformer to add shipping and tax prices together and return result

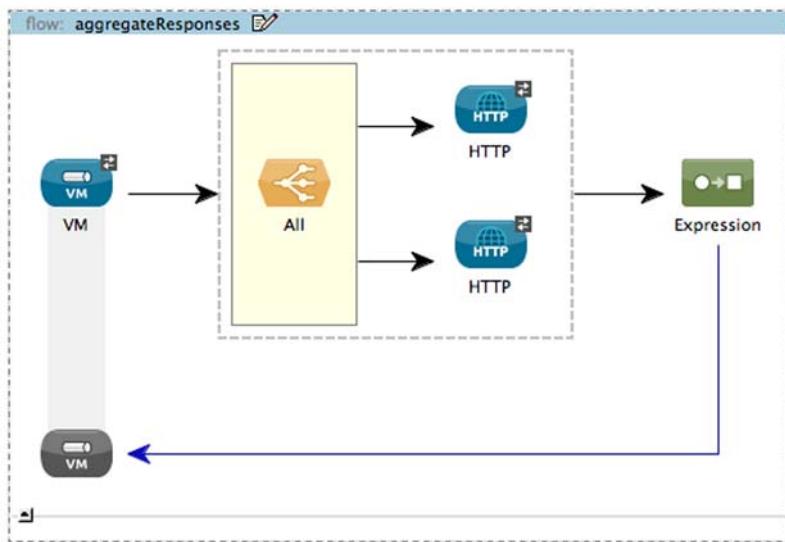


Figure 5.15 Scatter/gather IO

The internal API methods exposed by Prancing Donkey take the XML serialized `Order` objects and return the calculated shipping and tax prices. The VM inbound endpoint ① synchronously accepts the XML representation of the `Order` object. The all router ② then synchronously sends the order to each API. The output of the all router is a Collection containing the response from each endpoint. The response from each API call is added to this collection and is transformed by the expression transformer ③. The summed result is then returned to the caller of the VM inbound endpoint.

Using the all router in this fashion provides a powerful method to aggregate responses from multiple sources. You'll see another way to do this when we discuss message splitting and aggregation in the next section. For now, however, let's consider how you can invoke a sequence of message processors asynchronously using a different thread of execution.

5.3.3 Going async with the `async` processor

The all router is appropriate when you want to synchronously dispatch or aggregate requests to endpoints. What's your option, however, when you want a series of message processors to be processed asynchronously in a different thread? In such a case, the `async` messaging scope is necessary.

To see why this is useful, let's consider a use case for Prancing Donkey. Prancing Donkey's customers are able to place orders via a mobile phone application. On the server side, this process consists of a few steps:

- Validate the request (ensure the credit card is valid, the request is well formed, and so on)
- Perform order preprocessing using a Java class

- Perform order processing using a Java class
- Execute a method on a Java class to perform the shipping operations

The easiest way to implement this would be to have an HTTP inbound endpoint followed by three Java components corresponding to each step. The downside of this approach, however, is that the mobile user's experience is now coupled to pieces of order processing they probably don't care about. It would be ideal if you could accept the HTTP request and return a response to the user after validating the request, allowing the rest of the order processing to occur asynchronously in a different thread (see the following listing).

Listing 5.14 Asynchronously processing an Order

```
<flow name="mobileOrderSubmission">
    <http:inbound-endpoint host="${http.host}"
                           port="${http.port}"
                           path="orders"/>
    <component class=
                "com.prancingdonkey.service.OrderValidationService"/>
    <async>
        <component
            class="com.prancingdonkey.service.OrderPreProcessingService"/>
        <component
            class="com.prancingdonkey.service.OrderProcessingService" />
            <jms:outbound-endpoint queue="orders.completed"/>
    </async>
</flow>
```

Perform order validation and synchronously return response to client

1 The rest of order submission is handled asynchronously

The http:inbound-endpoint will receive the request and invoke OrderValidationService to ensure the request is valid. This will return a response to the client, in this case a mobile phone app, indicating whether or not the order was successfully submitted, along with a tracking ID that the app can use later on to check the status of the order.

The async scope beginning at ① invokes the contained message processors using a different thread. This allows you to quickly return a response to the mobile app, while allowing back-end processing to occur independently.

So far, you've seen how Mule's routing message processors can selectively route messages, stop messages from being routed, route messages to multiple endpoints, and asynchronously handle back-channel responses. One thing we haven't discussed, however, is how to route groups of messages together.

5.4 Routing and processing groups of messages

The messages we've processed so far in this chapter, up until we discussed scatter/gather IO with the all router, were atomic in nature. Processing groups of messages, however, can be as common a use case. You could have a process that's waiting for data from a variety of remote systems and can't proceed until it gets it from all of them. You also might selectively choose to split your data, process it, and then aggregate the

results. This should be familiar to anyone with exposure to the map-reduce programming paradigm.

We'll demonstrate Mule's splitting and aggregation features by parallelizing the processing of Prancing Donkey's orders to multiple Mule nodes over JMS. Prancing Donkey ultimately wants to be able to split `Order` objects up into individual `LineItems`, submit these to JMS for distributed processing, and then aggregate the results to complete the order. Finally, you'll see how the `foreach` message processor allows you to iterate over the payload of a message, potentially routing each element individually.

Let's start off by seeing how the `Orders` will be split up.

5.4.1 **Splitting up messages**

Prancing Donkey passes around order data using XML documents. These documents contain high-level data about the order, including who the customer is and how they're paying, as well as data about each line item. Small orders will only contain a few line items, but larger orders, perhaps for restaurant chains or distributors, might contain hundreds of line items. A certain amount of processing needs to be done for each line item: calculating its shipping cost, updating the inventory system, submitting the line item to an analytics system, and so on. Performing these tasks sequentially isn't feasible for orders with a large number of items.

The following listing demonstrates how Prancing Donkey is splitting the order up into its constituent line items in order to distribute the work required to process each `LineItem` object (see figure 5.16).

Listing 5.15 Splitting line items

```
<flow name="splitLineItems" >
    <jms:inbound-endpoint queue="order.submission"
        connector-ref="jmsConnector" />
    <set-property
        propertyName="ORDER_ID"
        value="#[xpath('/order/id').text]" />
    <splitter
        expression="xpath('//lineItem')"/>
    <set-property
        propertyName="MULE_CORRELATION_ID"
        value="#{flowVars['PD_ORDER_ID']}" />
    <jms:outbound-endpoint queue="lineitem.processing"
        connector-ref="jmsConnector" />
</flow>
```

The diagram shows the XML code from Listing 5.15 with several annotations:

- A red box labeled "Split order into its constituent line items" with an arrow points to the `<splitter` element.
- An annotation "Save order's ID into an invocation message property" with an arrow points to the `<set-property` before the `<splitter`.
- Annotation 1: "Explicitly set MULE_CORRELATION_ID to the ORDER_ID" with an arrow points to the `<set-property` inside the `<splitter>` block.
- Annotation 2: "Route each LineItem to lineitem.processing queue" with an arrow points to the `<jms:outbound-endpoint` inside the `<splitter>` block.

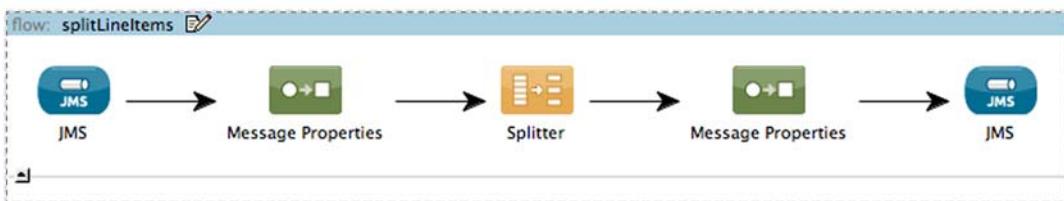


Figure 5.16 Splitting line items

This flow accepts a message containing an Order payload. It will split the Order into its individual LineItem objects, and place each LineItem object in a JMS queue for processing. Because you're sending the LineItems over JMS, it's very likely that a different Mule instance listening to that queue will process the LineItem. Mule uses two special headers, called MULE_CORRELATION_ID and MULE_CORRELATION_GROUP_SIZE, to keep track of these messages when responses come back from the different systems. The fact that the correlation state is maintained in the individual messages means that a different Mule instance than the one that did the splitting can assume aggregation duties for the message group.

Mule will automatically set a MULE_CORRELATION_ID if one isn't explicitly set. In the previous case, Prancing Donkey is explicitly setting the MULE_CORRELATION_ID to be the Order's ID. This will allow them to reference the Order when the LineItems are correlated. The MULE_CORRELATION_GROUP_SIZE is automatically set to the amount of elements in the collection. Each LineItem is then dispatched to the lineitem.processing queue ② with its MULE_CORRELATION_ID header explicitly set ①. An Order with 100 LineItems and an ID of 6ed49087-2765-45f6-a9bf-ade2aea363c9, for example, would result in 100 messages dispatched on the lineitem.processing queue. Each message would have its MULE_CORRELATION_ID set to 6ed49087-2765-45f6-a9bf-ade2aea363c9 and its MULE_CORRELATION_GROUP_SIZE set to 100.

Now let's see how to handle the LineItems once they've been processed.

5.4.2 Aggregating messages

Often a message is split and its constituent parts processed without bearing a result, or a result is produced that's meaningless in aggregate. Other groups of messages, as in the Prancing Donkey use case we're exploring in this section, need to be reassembled. Message aggregators are used for this assembly. Mule aggregators use the MULE_CORRELATION_ID and MULE_CORRELATION_GROUP_SIZE to reassemble a split message.

The following listing demonstrates how Prancing Donkey is using a collection aggregator to reassemble the List of LineItems broken up using the splitter in listing 5.15 (see figure 5.17).

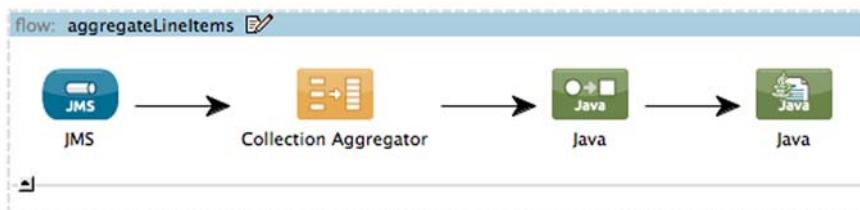


Figure 5.17 Aggregating line items

Listing 5.16 Aggregating line items

```

<flow name="aggregateLineItems">
    <jms:inbound-endpoint queue="lineitem.complete"
        connector-ref="jmsConnector" />
    <collection-aggregator timeout="60000"
        failOnTimeout="true" />
    <custom-transformer
        class="com.prancingdonkey.transformer.LineItemsToOrderTransformer"/>
    <component
        class="com.prancingdonkey.service.OrderProcessingService"/>
</flow>
  
```

Process Order ↗ ① Wait 10 minutes for all LineItems and throw exception if any are missing ↙

② Transform LineItems into an Order ↗

The collection aggregator ① will wait until its timeout is reached for the amount of LineItems equal to the MULE_CORRELATION_GROUP_SIZE for a MULE_CORRELATION_ID. Using the numbers from earlier, if 100 LineItems aren't received by the collection aggregator within 10 minutes, then an exception is thrown. Setting failOnTimeout to false disables this behavior. This can be useful if the entire set of messages isn't critical for processing to continue. This obviously isn't the case for order data, but it could be true for some kinds of analytics or media payloads that can tolerate lost messages.

Now that you have the List of LineItems back, you can call a custom transformer ② to convert this back to an Order object. LineItemsToOrderTransformer uses the message's MULE_CORRELATION_ID, which you explicitly set to the order's ID, to retrieve the order data from a database and populate it with the newly available LineItems collection. The Order is then passed to a Java component for its final processing.

Message aggregation can be used independently of splitting. Manually setting the MULE_CORRELATION_ID and MULE_CORRELATION_GROUP_SIZE will trigger the aggregation behavior without using a splitter. This can be useful to force Mule to process certain kinds of messages as a group, perhaps to enforce certain transaction semantics or batching.

5.4.3 Routing collections

The examples in this section have so far dealt with routing groups of messages. Mule also provides facilities to iterate over the contents of an individual message, typically

its payload, and process each piece individually. This functionality is achieved using the foreach message processor, as demonstrated by the following listing. The flow accepts an XML document as a JMS payload describing an order and its constituent line items. The goal is to use an XPath expression to process each line item separately, sending line items with a HIGH priority to one Java component and all other line items to a different Java component (see figure 5.18).

Listing 5.17 Routing collections

```

<flow name="Chapter05ExamplesFlow1">
    <jms:inbound-endpoint queue="orders.audit"
        connector-ref="jmsConnector"/>
    <foreach collection="xpath('//lineItem')"
        counterVariableName="lineItem">
        <choice>
            <when
expression="xpath('priority/text()').textContent == 'HIGH'">
                <object-to-string-transformer/>
                <component
class="com.prancingdonkey.service.HighPriorityOrderProcessingService"/>
            </when>
            <otherwise>
                <object-to-string-transformer/>
                <component
class="com.prancingdonkey.service.LowPriorityOrderProcessingService"/>
            </otherwise>
        </choice>
    </foreach>
</flow>

```

① Use XPath to split collection; then process each node distinctly

The collection expression defined at ① indicates how to split the message up for processing. In this case, you use an XPath expression. If the payload was an instance of a Java collection, however, then you could simply use the payload variable as the

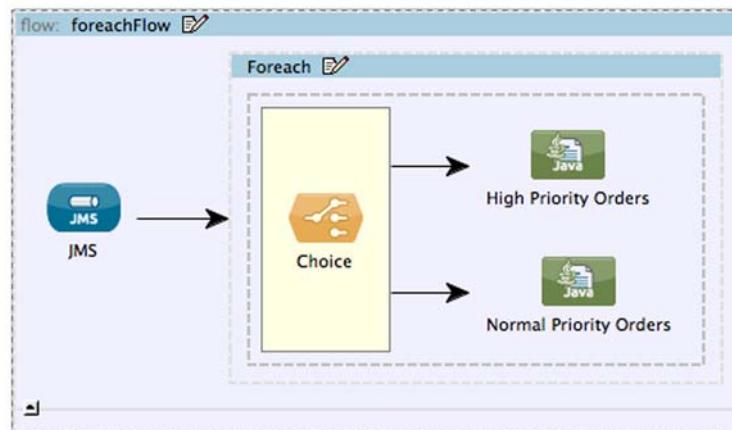


Figure 5.18 Routing the line items

collection for the foreach router. The choice router then makes the determination, on a per-node basis, of how to route the message. Line items with a HIGH priority will be routed to the HighPriorityOrderProcessingService component, while all other line items will get processed by the regular OrderProcessingService.

Now that you're comfortable routing messages in aggregate as well as routing messages as collections, we can turn our attention to the final routing topic of this chapter: routing for reliability.

5.5 **Guaranteed routing**

Certain transports, like JMS, offer guaranteed delivery. Most commonly used transports, particularly HTTP, don't feature such amenities. Decoupling such transports with something like JMS can be a good approach, but message brokers aren't always available and, more importantly, they introduce operational overhead to an environment. Luckily, Mule supplies routers that provide the benefits of decoupling middleware without the overhead of maintaining a messaging infrastructure.

Let's start by looking at the routers that can introduce resiliency for unreliable transports: first-successful and until-successful.

5.5.1 **Resiliency for unreliable transports**

The first-successful and until-successful routers allow you to define how Mule deals with a failure when sending a message to an endpoint. The first-successful router, illustrated next, will attempt to send a message to each endpoint until one succeeds:

```
<first-successful>
    <http:outbound-endpoint
        exchange-pattern="request-response"
        host="localhost" port="9091"
        path="shipping" doc:name="HTTP"/>
    <http:outbound-endpoint
        exchange-pattern="request-response"
        host="localhost" port="9091"
        path="shipping" doc:name="HTTP"/>
    <http:outbound-endpoint
        exchange-pattern="request-response"
        host="localhost" port="9091"
        path="shipping" doc:name="HTTP"/>
</first-successful>
```

The until-successful router, contrasted with the first-successful router, will continue to retry delivery to an endpoint until it succeeds. Here's an example that retries an HTTP endpoint for an hour, waiting five minutes between retry attempts.

Listing 5.18 Reliability routing with the until-successful message processor

```

<spring:bean
    class="org.mule.util.store.SimpleMemoryObjectStore" />

<flow name="storeDocument">
    <vm:inbound-endpoint path="ws.in"/>

    <until-successful objectStore-ref="documentRequestObjectStore"
        maxRetries="12" deadLetterQueue-ref="vm://DLQ"
        secondsBetweenRetries="300">
        <outbound-endpoint address="http://localhost:9091/documents"/>
    </until-successful>
</flow>

```

Annotations for Listing 5.18:

- ① Configure in-memory object store to save requests**: A red callout points to the first line of the XML, which defines a simple memory object store.
- ② Attempt redelivery to HTTP outbound endpoint for an hour with a five-minute delay between attempts**: A red callout points to the configuration of the until-successful router, specifically the maxRetries and secondsBetweenRetries attributes.

The until-successful router is an example of the “store and forward” pattern. As such, an object store is needed to persist the request. For the purposes of this example, we’re using an in-memory object store that’s configured at ①. The until-successful router ② will attempt delivery to the outbound endpoint for an hour, waiting five minutes between each redelivery attempt.²

Now let’s see how to handle the LineItems once they’ve been processed.

5.5.2 Defining failure expressions

In both previous examples, any exception thrown by invoking the outbound endpoint would trigger the resiliency behavior. In the case of the first-successful router, it would cause delivery to be attempted to the next message in the chain. For the until-successful router, it would trigger a retry attempt. By setting the failureExpression attribute on either router, you can control what Mule perceives as an error.

Listing 5.19 demonstrates configuring the until-successful router to only fail on `SocketTimeoutExceptions`. The rationale behind this is that you only want to attempt redelivery if there’s some sort of network problem or outage on the remote system. This goes a step towards avoiding the *poison pill* problem, in which a message is repeatedly delivered that will never succeed, taking up resources on the remote system.

Listing 5.19 Failure expressions and the until-successful router

```

<spring:bean
    class="org.mule.util.store.SimpleMemoryObjectStore" />

<flow name="storeDocument">
    <vm:inbound-endpoint path="ws.in"/>

    <until-successful objectStore-ref="documentRequestObjectStore"
        maxRetries="12" deadLetterQueue-ref="vm://DLQ"
        failureExpression="socketTimeoutException" />

```

² The until-successful router is currently asynchronous only.

```
failureExpression=
    "exception-type:java.net.SocketTimeoutException"
secondsBetweenRetries="300">
    <outbound-endpoint
        address="http://localhost:9091/documents" />
</until-successful>
</flow>
```

Only attempt redelivery if
SocketTimeoutException
is thrown

In this case, you use the exception-type expression evaluator to see if the exception payload is an instance of `SocketTimeoutException`. Assuming that it is, Mule will attempt to send the message to the endpoint in five minutes. If another type of exception is thrown, then the retry attempts will be cancelled, and the message will be sent to the DLQ.

5.6 **Summary**

In this chapter, we explored Mule's routing capabilities. You saw how routing message processors control how data moves back and forth from remote systems. We took a look at Mule's filtering capabilities and how they apply to routing. We examined some of Mule's core filters in depth, and saw examples of each at work. Perhaps most importantly, you may have noticed that we didn't have to implement any custom code to perform the routing.

Some of the examples in this chapter and the previous one demonstrated routing messages to Java service classes. Let's now take a formal look at how to integrate business logic in Mule applications with components.



Working with components and patterns

This chapter covers

- How to use a Mule component
- Simplifying projects using configuration patterns

If you've been around for a little while in the happy field of software development, you've surely been exposed to different flavors of component models. CORBA, EJB, JavaBean, and SCA have all helped to familiarize developers with the notion of a component. You understand that components represent entities that can perform specific operations, with well-defined interfaces and an emphasis 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 have explored these capacities; you've discovered that you can achieve many complex integration scenarios without the need for any particular component. Why bother about components? In this chapter, we'll start by answering that question. Then you'll learn how to reuse your existing components, how to use component annotations, and

how to use scripting languages to write components. We'll also look closely at Prancing Donkey's requirements for a shipping cost calculator using lifecycle methods and the addition of an MD5 signature to messages using scripting components.

As soon as you learn how to use components, you will—by using the knowledge acquired in the previous chapters—be able to connect to external systems, route, transform, enrich, and even apply business logic. With all this power you can start writing full-fledged Mule applications, but beware; Mule applications are usually XML-based and can be verbose if they're not handled with care. To help you to keep your applications as small and maintainable as possible, Mule provides configuration patterns, specialized processors for the more common tasks in Mule: bridging, proxying, validating, and servicing. We'll study them in the second half of this chapter. We'll also cover examples of how each of the patterns can help Prancing Donkey.

Now let's jump into the world of business logic by learning how to use Mule components.

6.1 **Using Mule components**

To incorporate your business logic in your Mule application, you might want to use Mule to integrate the logic by using SOAP or REST; in that case, you probably aren't interested in Mule components. But if instead you plan to host your components alongside your Mule application, Mule components are your best shot to incorporate loosely coupled logic in your application. Mule invites you to write your business logic in a loosely coupled fashion and then wrap it in a component. You'll learn in the next section how to use components to execute business logic, but let's first figure out if you have the need for components.

As you've learned in previous chapters, transformers and routers already provide many message processing capacities, such as splitting or enrichment. Where do components fit in this scheme? There are no hard rules, but the following are a few use cases and guidelines to help you better understand:

- Some message-related operations don't conceptually fit anywhere else. For example, a transformer wouldn't be the best place for executing business logic or logging messages.
- Unlike other Mule moving parts, components don't mandate the implementation of a specific interface. This enables the use of 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 do not. For example, you can pool components to handle heavy concurrent workloads only by means of configuration.
- As a rule of thumb, a transformer shouldn't have state; for the same input it should produce the same output. Therefore, if there is state, a component should be used.

- Exceptions thrown at the 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 it would if you were throwing it from your component.

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. Let's start with learning how to run logic within Mule itself.

6.1.1 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 the many transports of Mule. 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 a matter of configuration. This is great news because the industry was *not* in need of 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 to create components that are independent from Mule's API at compile time. Importantly, this approach will allow you to unit test your component code independently of the Mule container.

This said, there are times when you'll be interested in being coupled with Mule's API. One is when your component needs awareness 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 to access, for instance, message properties.

At this point, you'll preferably use Mule annotations, such as those we'll cover in section 6.1.4, by defining properties or arguments and then annotating them. Alternatively, 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 retrieving the message payload, applying inbound transformers, or looking up other moving parts via the Mule registry.

Another reason to be 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 of section 6.1.5.

Two mechanisms—the annotations and the `Callable` interface—will couple you to the Mule API, but using annotations is considered the more favorable solution given that they’re less intrusive; they don’t force the class to use specific method signatures, and they let the methods be injected with more atomic data instead of the whole context. You’ll see this in an example in section 6.1.4 in which you annotate specific parameters instead of passing the whole Mule context.

Instantiation policy

By now, you’re certainly wondering, how does Mule take care of instantiating your objects before calling the right method on them? Unless you decide to pool these objects, which is discussed in section 6.1.5, there are three main possibilities:

- Let Mule create a unique instance of the object and use it to service all the requests.
- Let Mule create one new object instance per each request it’s servicing. This is done by using the `prototype-object` element or the short syntax in which 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.

Granted, creating custom components mostly amounts to creating standard business logic classes, but there are technical aspects to consider when it comes to configuring these custom components. Indeed, because 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 the upcoming sections:

- How does Mule locate the method to call—a.k.a. 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?

COMPONENT VERSUS COMPONENT If you’ve looked at Mule’s API, you might have stumbled upon the `org.mule.api.component.Component` interface. Is this an interface you need to implement if you create a custom component? No. This interface is mostly for Mule’s use only. It’s implemented internally by Mule classes that wrap your custom components. We conjecture that this interface has been made public because Java doesn’t have a “friend” visibility specifier.

Let’s start with the important notion of entry point resolution.

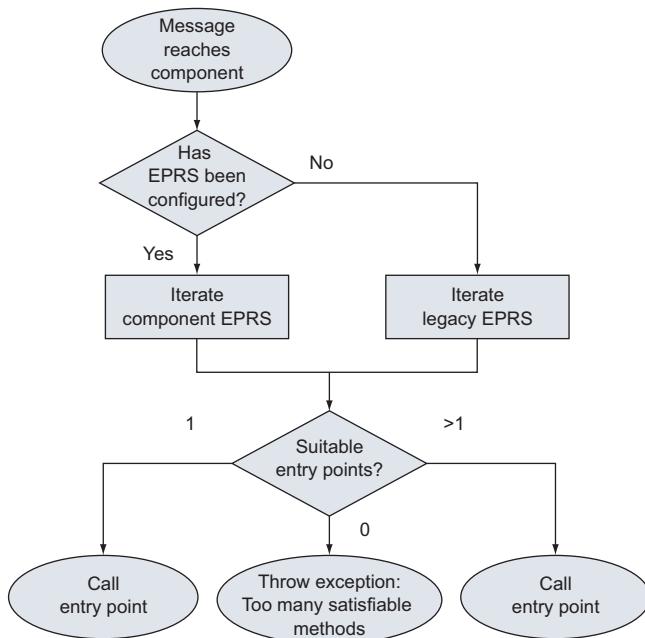


Figure 6.1 The component entry-point resolution strategy (EPRS) used by Mule

6.1.2 Resolving the entry point

As you've discovered, Mule components can be objects of any type. This flexibility raises the 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 named an *entry point*. To find the entry point to use, Mule follows a strategy that's defined by a set of *entry-point resolvers*.

Figure 6.1 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, 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 great variety of entry-point resolvers that you can use to ensure that messages reach the intended method on your components. Table 6.1 gives you a list of the available resolvers.

Table 6.1 Available entry-point resolvers

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.

Table 6.1 Available entry-point resolvers (continued)

Entry-point resolver name	Behavior
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 want to receive an array as an argument, you must use the <code>array-entry-point-resolver</code>). If the message has a null payload, the resolver will look for a method without an argument.
custom-entry-point-resolver	In the rare case in which these resolvers can't satisfy your needs, you can roll out your own implementation. It's preferable to subclass <code>org.mule.model.resolvers.AbstractEntryPointResolver</code> rather than implement <code>org.mule.api.model.EntryPointResolver</code> from scratch.

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.

If you remember the discussion at the beginning of chapter 4, you should remember that we said *it's up to the component to decide to apply it or not*. 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 messages are at risk of dispatch 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.

The default entry-point resolver

As of 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 `transformFirst` disabled

It sure 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 performs all these contortions in order to be compatible with the behavior of Mule 1.

Let's look at a simple example. Listing 6.1 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 6.1 A random integer generator service

```
<flow name="shippingCost">
  <inbound-endpoint address="vm://random-integer-service.in"
    exchange-pattern="request-response" />
  <component>
    <no-arguments-entry-point-resolver>
      <include-entry-point method="nextInt" />
    </no-arguments-entry-point-resolver>
    <singleton-object
      class="java.util.Random" />
  </component>
</flow>
```

Annotations for Listing 6.1:

- A callout points to the line `class="java.util.Random" />` with the text "Targets nextInt() method of the random object".
- A callout points to the line `class="java.util.Random" />` with the text "Instantiates standard Java random number generator".

BEST PRACTICE Adapt Mule to your components, not the other way around. If necessary, use a transformer before the component to adapt the payload.

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.

6.1.3 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 and 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, you'll configure a simple random integer generator using the stock JDK random class. You'll set its seed to a predetermined configured value (don't try this in your own online casino game).

PROPERTY RESOLUTION CHALLENGES There are cases in which you'll have to use Spring instead of Mule for configuring a component. Mule uses property resolvers that 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 (that is, the infamous `[B.. string]`) instead of having the value correctly set. This doesn't happen if you use Spring to configure this component.

Listing 6.2 shows this service with its component configured using a Mule object. Notice how the seed value is configured using a property placeholder. Notice also that,

like Spring, Mule supports the notion of singleton and prototype objects. Because `java.util.Random` is thread-safe, you only need a unique instance of this object to serve all the requests coming to the service. This is why you can use the `singleton-object` configuration element.

Listing 6.2 A fixed-seed random integer generator service

```
<flow name="fixed-seed-random-integer">
  <inbound-endpoint
    address="vm://fixed-seed-random-integer-service.in"
    exchange-pattern="request-response" />
  <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>
</flow>
```

Annotations for Listing 6.2:

- A callout points to the `method="nextInt"` attribute in the `<include-entry-point>` block, with the text: "Targets `nextInt()` method of the random object".
- A callout points to the `value="${seed}"` placeholder in the `<property>` block, with the text: "Instantiates and calls `setSeed()` to configure the random object".

There are cases in which using a single component instance to serve all the service requests isn't desirable. For example, this is the case 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 consists of using the `prototype-object` configuration element instead of the `singleton-object` one. We'll come back to this subject in the next section.

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 8).

Now let's look at the Spring version of this random integer generator service. As listing 6.3 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 6.3 A Spring-configured, fixed-seed, random integer generator service

```

<spring:bean id="Random" class="java.util.Random">
    <spring:constructor-arg value="${seed}" />
</spring:bean>

<flow name="spring-fixed-seed-random-integer">
    <inbound-endpoint
        address="vm://spring-fixed-seed-random-integer-service.in"
        exchange-pattern="request-response" />
    <component>
        <no-arguments-entry-point-resolver>
            <include-entry-point method="nextInt" />
        </no-arguments-entry-point-resolver>
        <spring-object bean="Random" />
    </component>
</flow>

```

The code block shows annotations and configurations for a Spring-configured random integer generator service. Annotations include `@Spring:bean`, `@Spring:constructor-arg`, `@Flow`, `@InboundEndpoint`, `@Component`, `@NoArgumentsEntryPointResolver`, and `@IncludeEntryPoint`. Callout boxes explain these annotations:
 - `@Spring:bean` and `@Spring:constructor-arg` are grouped under 'Instantiates and configures the random number generator'.
 - `@Flow`, `@InboundEndpoint`, and `@Component` are grouped under 'References the Spring-configured random object'.

Depending on your current usage of Spring and the need to share the component objects across services and configurations, you'll use either the Mule or the Spring way to configure your custom service component objects.

Now you know how to properly configure a component. Let's look at how to receive context information using Mule annotations in the next section.

6.1.4 Annotating components

Since version 1.5, Java supports annotations. At the same time, one of the leitmotifs of the third major version of Mule is to reduce the amount of XML and Java necessary to write a Mule application. Therefore, the marriage of Mule 3 and annotations probably shouldn't surprise you.

Annotations will let you perform a fine-grained injection in your components. The classic counterpart, the `Callable` interface, will force you to implement one method that will receive the whole Mule context; you'll see this approach in action in listing 6.9. `Callable` adds some boilerplate to the Java code to extract the required parts. Mule offers a plethora of annotations to do this work in a less-coupled and more fine-grained way; in appendix B, we'll cover most of the available annotations, but given that you're learning about components, let's first deal with how to use the more common annotations for components here:

- `@InboundHeaders` and `@OutboundHeaders` pass to the annotated object *inbound* or *outbound* properties as a single object, a list of values, or a map of values depending on whether the annotated object is a regular argument, a `List`, or a `Map`. It will accept a value to filter the headers that should be passed. That value can be a single header name, a wildcard expression (covered previously in section 4.3.3)—for example, `X-*`—or a comma-separated list of values—for example, `Content-Type, Content-Length`.
- `@Payload` will inject the message payload into the annotated parameter. If the type of the message payload is different than the type of the parameter, Mule

will do its best to transform the message payload to the parameter type, as you learned in section 4.3.6.

PARAMETERS: ANNOTATE ALL OR NONE You should take into consideration that Mule has to figure out what to pass for each of the arguments. That was simple with the Callable interface, but with annotations Mule won't know what to do if there's a non-annotated argument. If you want to use annotated arguments, you should annotate all of them.

Prancing Donkey needs to calculate the taxes that should be applied to the shopping cart of the users depending on the state from which they're making their purchases. This sounds like a great opportunity to put annotated components to work by creating a tax calculator component.

Listing 6.4 A method of a component making use of annotations

```
public BigDecimal calculateTax(
    @Payload BigDecimal cartValue, // Marks parameter to be
    @InboundHeaders("state") String state) // Annotates parameter to be injected with an inbound header called "state"
{
    return calculateTaxForState(state, cartValue);
}
```

Annotations aren't specifically for components; other Mule elements, such as transformers, can benefit from them. We'll cover many other useful annotations that are available in Mule, from the injection of the result of the execution of expressions to attachment injection in appendix B.

Let's now look at advanced configuration options that can allow you to control the workload of your components.

6.1.5 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 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 11 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 important that the total number of component object instances remains under control.
- *The component is thread-unsafe*—It's essential to ensure that only one thread at a time will ever enter a particular instance of the component.

Pool exhaustion

By default, if no component object instance is available to serve an incoming request, Mule will disregard the maximum value you have set on the polling profile and create an extra component instance to take care of the request. You can change this and configure the pooling 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 used by Prancing Donkey to compute the MD5 hash code 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 they've received the expected file and that they can proceed with it (like pushing it to their client's server farm in the cloud).

Because this computation is expensive for large files, they'll use pooled components. Listing 6.5 demonstrates the configuration for this service. The pooling profile element is self-explanatory: one instance of the `com.prancingdonkey.component.Md5FileHasher` component object will be created initially in the pool and a maximum of five active ones will exist at any point in time. They don't expect this service to receive heavy traffic, but should this happen, they allow a maximum of 15 seconds of waiting time in the event that the component pool becomes exhausted.

Listing 6.5 Service limits its resource usage thanks to a pooled component

```
<flow name="pooled-random-integer">
    <inbound-endpoint
        address="vm://pooled-md5-service.in"
        exchange-pattern="request-response" />
    <pooled-component>
        <prototype-object
            class="com.prancingdonkey.component.Md5FileHasher">
            <property key="fileConnector"
                value-ref="NonDeletingFileConnector" />
            <property key="sourceFolder"
                value="${java.io.tmpdir}" />
        </prototype-object>
        <pooling-profile
            initialisationPolicy="INITIALISE_ONE"
            maxActive="5"
            exhaustedAction="WHEN_EXHAUSTED_WAIT"
            maxWait="15000" />
    </pooled-component>
</flow>
```

The listing includes three annotations:

- A red callout pointing to the `<pooled-component>` tag with the text "Declares component as a pooled one".
- A red callout pointing to the `<pooling-profile>` tag with the text "Defines pooling profile".

```
</pooled-component>
</flow>
```

Using connectors inside components

Notice how in listing 6.5 you inject a reference to a Mule file connector instance. Indeed, you use the file connector to read from a particular directory. One could wonder why you're doing that, because it's a trivial task to read a file using plain Java code. In fact, using the Mule connector provides you with all the statistical, exception handling, and transformation features of Mule. In case you're curious, here's the declaration of NonDeletingFileConnector:

```
<file:connector name="NonDeletingFileConnector" autoDelete="false" />
```

Similarly, you can inject global transformers and even global endpoints in your component. Doing so couples your component with the API of Mule.

You can use the pooling component configuration with Spring beans too. For this, use a `spring-object` reference element, as you saw in listing 6.3, and you're done. Bear in mind that you'll have to define the scope of the bean to `factory` so that Mule will be able to populate the component pool with different object instances.

Before we close this section on pooling, here's a little 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 indiscriminately oftentimes amounts to premature optimization.

The last feature we'll cover will allow your components to reach out and touch...other services.

6.1.6 Scripting components

You might find yourself in a situation in which 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 a case like this. Let's assume you want to add a property to messages as they pass through a service. Perhaps you want to do an evaluation of a message's payload and, based on that, set a property header stating what the message's priority is. This property could subsequently be used by selective consumers to determine whether to accept the message or not. The following listing illustrates how this might be accomplished by using a JavaScript component.

Listing 6.6 Using a JavaScript component to enrich a message

```
<flow name="rhino-message-enrichment-service">
  <inbound-endpoint
    address="vm://rhino-message-enrichment-service.in"
    exchange-pattern="request-response" />
  <scripting:component>
```

1 Declare scripting component

```

<scripting:script engine="javascript">
    <scripting:text>
        if (payload.search("STATUS: CRITICAL") != -1) {
            message.setProperty("PRIORITY", 'HIGH');
        }
        result = message
    </scripting:text>
</scripting:script>
</scripting:component>
</flow>

```

Set specified property on message

② Define script and scripting engine

③ Search message payload for specified expression

By using the scripting namespace, you can begin to inline scripts into your Mule configuration. You first configure a scripting component ①, telling Mule that the component logic will be using a JSR 223-compliant scripting engine. Mule will subsequently spin up an appropriate environment for the script to execute it, with common Mule references defined in the script's context, as you'll see in a moment. Because the logic for this component is fairly straightforward, you 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 engine¹ will declare a name, which is set at ②. In this case, you have it set to javascript, indicating your script is written in JavaScript and will be executed using Rhino. The script itself begins at ③, where you're searching the payload variable for the presence of the specified expression. You might be wondering where you defined the payload variable; you haven't. Mule has. Mule will prepopulate the script's context with the 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 6.2.

Table 6.2 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
result	A variable to explicitly set the result of a script

¹ Make sure the engine you intend to use is available on Mule's classpath. If you're using Rhino or Groovy, you shouldn't need to do anything. Using a scripting engine not included in the JDK or Mule distribution will require you to place the appropriate JAR files in \$MULE_HOME/lib/user or in your application server's classpath, depending on how you've deployed Mule.

You use the message variable 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 6.5, you 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. You'll implement a Rhino script that will 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 the payload of a message hasn't changed when processed by future services. The following listing illustrates a Rhino script external to the Mule configuration to accomplish this.

Listing 6.7 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);
}
```

Write log messages at DEBUG level: A vertical line with an arrow pointing to the first line of the script, labeled "Write log messages at DEBUG level".

Check if property has been set on the message: A red callout pointing to the `if (!message.getProperty(PROPERTY_NAME))` condition.

Set MD5SUM property on the message: A red callout pointing to the `message.setProperty(PROPERTY_NAME, getMD5Sum(payload));` line.

Explicitly set result of the component: A red callout pointing to the `result = message;` line.

This script is pretty similar to that of listing 6.6. You have a bit more ceremony in listing 6.8, however, so it seems more natural to store it externally to the Mule configuration. This will also give you the flexibility to change the script as Mule's running, as you'll see in a bit. The Mule configuration to load the externally defined script is illustrated in the next listing.

Listing 6.8 Using an externally defined Rhino script to add an `MD5` property

```
<flow name="external-rhino-message-enrichment-service">
    <inbound-endpoint
        address="vm://external-rhino-message-enrichment-service.in"
        exchange-pattern="request-response" />
    <scripting:component>
        <scripting:script file="md5.component.js" />
    </scripting:component>
</flow>
```

1 Declare location of the externally defined script: A red callout pointing to the `file="md5.component.js"` attribute.

By setting the `file` parameter ①, you 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, because the script ends with a .js extension, Mule will run the script using the Rhino engine.

USING EXPRESSIONS TO CREATE COMPONENTS

Scripting can be simpler than Java but isn't necessarily the simplest method. Because you learned how to implement full-blown components with Java and easily embeddable scripting components, you might have figured that there's still room for an easier kind of component that, even when doing business logic, still doesn't need all the power of Java or the scripting languages.

For that reason, Mule provides an expression-component using the ubiquitous Mule Expression Language that provides the easiest (and not the slowest) way to implement a component. You're probably familiar with MEL; Mule intentionally uses it almost everywhere, so you can use and reuse your knowledge about it to create useful expressions. The expression component is no exception in regard to using MEL.

To understand how the expression-component works, let's visit an expression-component that stamps the payload with the time and date the message is processed:

```
<flow name="process-stamp">
    <inbound-endpoint address="vm://process-stamp.in"
        exchange-pattern="request-response" />
    <expression-component>
        message.payload.processedTime = new Date();
    </expression-component>
</flow>
```

Here you find the use of the special variable `message`. Similar to scripting components for which Mule offered the variables you saw in table 6.2, the expression component has available some top-level objects and functions. In appendix A, there's a comprehensive list of them.

The expression-component is also capable of using files instead of embedded expressions, like you saw in listing 6.7. Whether to use a file to store the expressions or not is up to you; there will be no significant differences in the execution time. Let's see how to configure an expression-component using a file:

```
<flow name="external-process-stamp">
    <inbound-endpoint address="vm://external-process-stamp.in"
        exchange-pattern="request-response" />
    <expression-component file="stamp.txt" />
</flow>
```

The `file` parameter sets the location of the expression file in Mule's CLASSPATH. To be equivalent to the listing using an embedded expression shown previously, the contents of the file should be the same as the body defined in the expression-component in that listing.

BEST PRACTICE Choose your approach to configuring components wisely. As a rule of thumb, you should use a full-blown component for complex matters, scripting components for tasks that don't have dependencies or the need for unit testing, and for simple tasks—expressions.

6.1.7 Component lifecycle

Between the time you bootstrap a Mule component and the time it's up and ready, many things happen, to which the numerous log file entries can attest. Moving parts are created and made ready for production. These moving parts are configured and transitioned through lifecycle phases, which technically amount to calling specific methods in a predefined order. The same happens when you dispose of a Mule component: all moving parts are transitioned to their ultimate destruction through another series of lifecycle method calls.

Your components can benefit from these configuration and lifecycle method calls the same way Mule's do. Table 6.3 gives an overview of the interface or annotated methods that are called and the order in which this happens for the component you create.

Table 6.3 Interfaces and annotations for the configuration of lifecycle methods

Configuration and lifecycle methods	Interface	JSR 250 annotation
initialise	org.mule.api.lifecycle.Initialisable	javax.annotation.PostConstruct
start	org.mule.api.lifecycle.Startable	
stop	org.mule.api.lifecycle.Stoppable	
dispose	org.mule.api.lifecycle.Disposable	javax.annotation.PreDestroy

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 implementing `org.mule.api.lifecycle.Lifecycle`, which extends the four standard ones.

STARTED BUT NOT READY When your components are started, don't assume that the whole Mule instance is up and running. Starting happens 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 you'll see in section 12.3.3.

You can either implement the desired interfaces or use the `PostConstruct` or `PreDestroy` annotations in a method. Sadly, there are no equivalents to the JSR 250 annotations for the `startable` and `stoppable` lifecycle interfaces.

The `initialise` phase is invoked when all the properties of the component have been set, meaning that all injectors on the component have already been called. Once every single moving part of Mule is ready to go, the `start` lifecycle will be invoked.

When shutting down, the `stop` phase will first be called, letting the component free any allocated resources. Right before the component is completely disposed of, the `dispose` phase will be invoked.

The code shown in listing 6.9 demonstrates how Prancing Donkey's client validation service (introduced in the previous section) is configured and initialized. Note that they have implemented the `initialise` method to be idempotent: should Mule call it several times, it will perform the initialization sequence only once.² The `initialise` method will be called because the `ClientValidatorService` component implements the `org.mule.api.lifecycle.Initialisable` interface.

Listing 6.9 Prancing Donkey's shipping cost calculator component

```
public class ShippingCostCalculator
    implements Initialisable, Callable {
    boolean initialized = false;
    EndpointBuilder errorProcessorChannelBuilder;
    OutboundEndpoint errorProcessorChannel;

    public void setErrorProcessorChannel(
        EndpointBuilder errorProcessorChannelBuilder) {
        this.errorProcessorChannelBuilder =
            errorProcessorChannelBuilder;
    }

    public void initialise() throws InitialisationException {
        if (initialized) {
            return;
        }
        try {
            errorProcessorChannel = errorProcessorChannelBuilder
                .buildOutboundEndpoint();
            initialized = true;
        } catch (final EndpointException ee) {
            throw new InitialisationException(ee, this);
        }
    }

    /
    /* Other logic here */
}
```

Builds error channel outbound endpoint

Receives error channel endpoint builder via setter injection

Ensures that component has not yet been initialized

The configuration of the component itself is trivial:

```
<flow name="random-integer">
    <inbound-endpoint address="vm://calculateShippingCost.in"
        exchange-pattern="request-response" />
    <component>
        <singleton-object
            class="com.prancingdonkey.component.ShippingCostCalculator">
            <property key="errorProcessorChannel"
                value-ref="ErrorProcessorChannel" />
        </singleton-object>
    </component>
</flow>
```

² 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.

In this configuration, you inject a reference to a global endpoint named `ErrorProcessorChannel` in the designated setter method. As you've seen in listing 6.9, what you inject is a reference to an endpoint builder, from which you derive an outbound endpoint. This makes sense when you take into account that a global endpoint can be referred to from both inbound and outbound endpoints.

DECOUPLING THE LIFECYCLE The same way entry-point resolvers (see section 6.1.2) allow you to target component methods without implementing a Mule interface, it's also possible to create custom lifecycle adapters³ that will take care of translating Mule lifecycle methods into any methods on your objects. Whether or not creating such an adapter is worth the effort for creating Mule-free components is a moot point, because it's also possible to rely on the non-invasive Spring lifecycle methods. Our own experience is that this is more than enough.

You're now able to use components alongside of the rest of Mule's moving parts: transports, routers, transformers, and so on. With the exception of Mule's fine-tuning, security, and other fine arts that you'll learn soon, you now have Mule's reins in your hands. You'll find soon that there are many tasks for which you're repeating almost identical pieces of configuration or configuring your Mule flows with too much detail. To simplify the processing of the information and the configuration files, Mule provides configuration patterns. Let's explore how to use them.

6.2 Simplifying configuration with configuration patterns

Mule has the capacities of routing, filtering, transforming, and processing with components. Each of those capacities entail a large quantity of fine-grained processors. The configuration file of a Mule application that combines those elements can end up being big.

While writing Mule applications, you might find that sometimes you're doing repetitive work. *Don't repeat yourself* is not only a phrase that deserves to be on a T-shirt, it's also a basic principle of software development. Mule helps you to reduce the noise in your projects by offering the *Mule configuration patterns*. There's no new functionality in these patterns, and you can replicate the behavior of the patterns with a combination of other Mule message processors. They're a simpler and more readable way of expressing common configurations that you will probably need. In this section, we'll review the usage of each of the configuration patterns provided by Mule:

- *Simple service pattern*—A clean way to expose a service in an endpoint
- *Bridge*—Directly connects an inbound endpoint to an outbound endpoint
- *Validator*—Validates input received in the inbound endpoint against a filter, handling ACK/NAK responses and the routing of the valid or invalid messages

³ Namely, custom implementations of `org.mule.api.component.LifecycleAdapterFactory` and `org.mule.api.component.LifecycleAdapter`.

- *HTTP proxy*—Bypasses the HTTP requests received in the HTTP inbound endpoint to a different HTTP outbound endpoint
- *WS proxy*—Links the SOAP request received in the inbound endpoint to a different SOAP outbound endpoint handling the required WSDL transformation

Let's visit them one by one in the following sections. We'll start with the simple service pattern.

CONFIGURATION PATTERNS AND STUDIO Currently there isn't support for the configuration patterns in Mule Studio. You'll need to rely on the XML configuration when using configuration patterns in your Mule applications.

6.2.1 Using the simple service pattern

All the orchestration in the world amounts to nothing without the business logic. Sometimes you might need to expose to the world the business logic that resides in your Mule application to create real value.

The simple service pattern can expose to the world Mule components or services implemented in some of the most common technologies for services:

- *Mule components*—Studied earlier in this chapter; they represent the classic Mule way to implement logic
- *JAX-RS*—The Java API for RESTful web services; thanks to them you'll be able to create REST-based services
- *JAX-WS*—The Java API for XML web services; they give support for SOAP-based services

This configuration pattern uses the word *simple* in its name, and that's risky (even if it's related not to the pattern but to the service). Let's check how appropriate the use of that word is in the next listing by exposing in an endpoint one of the components used in the previous section.

Listing 6.10 Simple service

```
<pattern:simple-service
    name="recicling-service"
    address="vm://recicledCount.in"
    component-class=
        "com.prancingdonkey.recyclingplant.RecycledCountComponent"/>
```

That was simple! You set the name for the service, a request-response address, and the component class that will handle the request, et voilà! A component is facing the world. But you passed a component class and an address instead of a reference to an endpoint or the component.

All the Mule configuration patterns are flexible in terms of what they accept; you'll find this now for the simple service and later on for the rest. For most of the accepted values, it will also accept references. But let's not cross that bridge before we come to

it. The following listing defines a simple service using a global endpoint and a component reference.

Listing 6.11 Simple service using a global endpoint

```
<spring:bean id="recicledCountComponent"
    class="com.prancingdonkey.recyclingplant.RecicledCountComponent" />

<endpoint name="recicledCountServiceEndpoint"
    address="vm://recicledCount.in" />

<mulexml:xml-to-dom-transformer
    name="xmlToDom"
    returnClass="org.w3c.dom.Document" />

<mulexml>xpath-extractor-transformer
    name="countExtractor"
    expression="//count" />

<pattern:simple-service
    name="recicling-service"
    endpoint-ref="recicledCountServiceEndpoint"
    component-ref="recicledCountComponent"
    responseTransformer-refs="xmlToDom countExtractor"/>
```

Here you create an equivalent to the previous example, but using a global endpoint instantiated with Spring and a component reference instead of an address and a component-class. Although you added a slight modification, you set a chain of response transformers to modify the response. This can be useful if the response of the component/service is not exactly the response you want to send back to the requestor.

Given that you can set the component as component-class or component-ref and the component entry point as address or endpoint-ref, it's not difficult to figure out that you could also set them as child elements, as shown in the next listing.

Listing 6.12 Simple service using child elements

```
<mulexml:xml-to-dom-transformer
    name="xmlToDom"
    returnClass="org.w3c.dom.Document" />

<mulexml>xpath-extractor-transformer
    name="countExtractor"
    expression="//count" />

<pattern:simple-service
    name="recicling-service" >
    <inbound-endpoint address="vm://recicledCount.in"
        exchange-pattern="request-response">
        <response>
            <transformer ref="xmlToDom" />
            <transformer ref="countExtractor" />
        </response>
    </inbound-endpoint>
```

```
<component class=
    "com.prancingdonkey.recyclingplant.RecycledCountComponent" />
</pattern:simple-service>
```

Now that you know how to configure a service using a good old Mule component, you're ready to start studying how to expose REST services in the next section.

CONFIGURING A SIMPLE SERVICE PATTERN WITH JAX-RS

JAX-RS is the Java API for RESTful web services. You might remember JAX-RS from section 3.3.2; there you configured a small service to provide a JSON representation of the brew catalog.

Here you'll create exactly the same service, but you'll configure it by the use of a simple service pattern that will comprise the processors configured in section 3.3.2 in one element. The following listing reviews the service you're exposing.

Listing 6.13 Service class to be exposed

```
package com.prancingdonkey.service;

import java.util.List;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

import com.prancingdonkey.model.Brew;

@Path("/brews")
public class BrewServiceImpl implements BrewService {

    @GET
    @Produces("application/json")
    @Path("/list")
    public List<Brew> getBrews() {
        return Brew.findAll();
    }
}
```

Now you'll put the magic of the configuration pattern in practice. In order to do it, you'll set up a simple service similar to the ones you've already seen using Mule components. But this time you'll add a type attribute to identify the service as jax-rs (shown in the next listing).

Listing 6.14 Simple service using JAX-RS

```
<pattern:simple-service name="brewServiceRest"
    address="http://localhost:6099/rest"
    component-class="com.prancingdonkey.service.BrewServiceImpl"
    type="jax-rs" />
```

That's it! You have a working JAX-RS-based web service working. You can check it with curl in the same way you did in section 3.3.2. That should do the trick for those who love the flexibility of REST. For those who prefer the contract-first approach, we'll cover JAX-WS in the next section.

CONFIGURING A SIMPLE SERVICE PATTERN WITH JAX-WS

In a way similar to how JAX-RS provides annotations to create REST services, JAX-WS does so for SOAP services. Remember section 3.3.2, in which you saw how to expose a SOAP service with Mule and its support for Apache CXF?

Now you'll simplify the configuration needed to create a SOAP service by using the simple service pattern. Like you did in section 3.3.2, you'll expose the same service you already exposed in the previous section but using JAX-WS annotations; the next listing shows what the service annotated with JAX-WS looks like.

Listing 6.15 Service class to be exposed

```
package com.prancingdonkey.service;

import java.util.List;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

import com.prancingdonkey.model.Brew;

@Path("/brews")
public class BrewServiceImpl implements BrewService {

    @GET
    @Produces("application/json")
    @Path("/list")
    public List<Brew> getBrews() {
        return Brew.findAll();
    }
}
```

Nothing new under the sun; you already know what a JAX-WS looks like. To see how the simple service pattern helps you to simplify the configuration, you'll have to declare it (see the following listing).

Listing 6.16 Simple service using JAX-WS

```
<pattern:simple-service name="brewWS"
    address="http://localhost:8090/soap"
    component-class="com.prancingdonkey.service.BrewServiceImpl"
    type="jax-ws" />
```

Now we're talking! See how you set the `type` to `jax-ws`; the rest is in essence the same as JAX-RS. With this, you're ready to expose a service to the world in a few lines. You can see this behavior in figure 6.2. Now let's see how to bridge a request to an already existent service using the bridge configuration pattern.

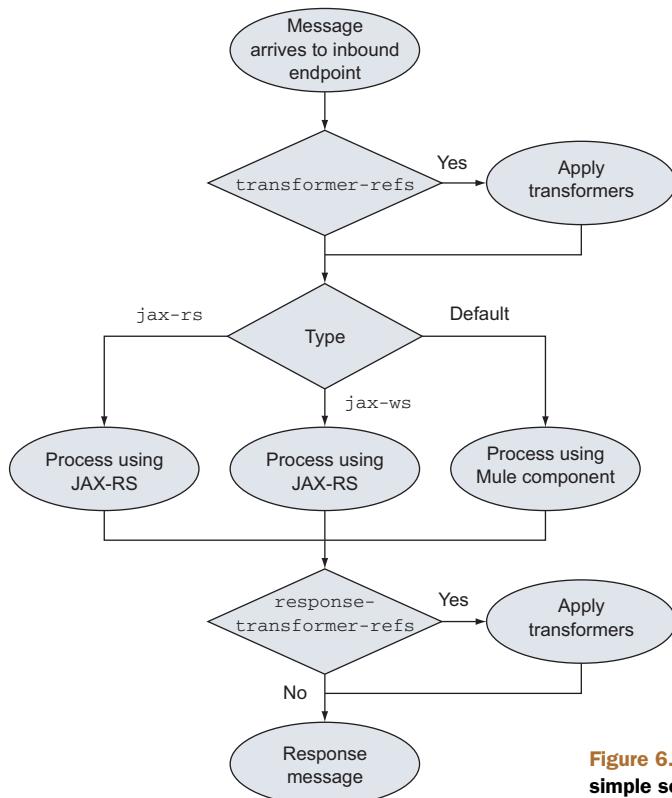


Figure 6.2 Behavior of the simple service pattern

6.2.2 Using the bridge

You've learned how to build services. It's just a matter of time before you need to connect two endpoints to share services; for example, you might need to create an API layer to expose internal services to the world, like the ones you created. Or perhaps you might want to wrap a legacy CSV service for JSON support by adding some transformation and adaptation to the bridge.

To simplify the connection of systems, Mule 3 introduced the bridge pattern. The name can't be more explicit: It's a bridge that connects an inbound endpoint to an outbound endpoint. No more, no less.

The simplest form of bridge that configures the name, inboundAddress, and outboundAddress will set a request-response conduit and will behave synchronously between its endpoints; any response coming from the outbound endpoint will be routed back to the inbound endpoint. The following listing illustrates such a synchronous bridge.

Listing 6.17 Synchronous bridge

```
<pattern:bridge name="a-simple-bridge"
  inboundAddress="http://localhost:8888"
  outboundAddress="vm://my-local-service.in" />
```

Here you expose, using a single element, a synchronous service that's listening to the inboundAddress and bridging the requests to the outboundAddress. The same response obtained from the outboundAddress will be sent back to the requestor at the inboundAddress.

Sometimes, however, you may have no need to send a response back, having effectively an asynchronous behavior. You won't be surprised to discover that this is defined like in any other Mule processor, by setting the exchange-pattern to one-way, shown next.

Listing 6.18 Asynchronous bridge

```
<pattern:bridge name="a-simple-async-bridge"
    exchange-pattern="one-way"
    inboundAddress="http://localhost:8888"
    outboundAddress="vm://my-local-service.in" />
```

The real world usually won't make things as simple as this; you did some *adaptation* by coupling two disparate protocols like HTTP and JMS, and that's something, but you might need more. For instance, you may need to wrap a legacy CSV service for JSON support. Or even more difficult, Prancing Donkey wants to have their legacy order status system exposed to their website using JSON. You need to receive a JSON request, transform it to a legacy payload before the request is bridged to the order status system, and then transform the CSV response to JSON again, so that the website provides and receives JSON, as shown in the next listing.

Listing 6.19 Bridge using transformers

```
<pattern:bridge name="a-simple-bridge"
    inboundAddress="http://localhost:8888"
    outboundAddress="vm://my-local-service.in"
    transformer-refs=
        "jsonToObjectTransformer objectToXmlTransformer"
    responseTransformer-refs=
        "xmlToObjectTransformer objectToJsonTransformer" />
```

You're already familiar with most of the configuration here, with the exception of the transformer-refs attribute that will convert the JSON request to a legacy format and the responseTransformer-refs attribute that will convert CSV to JSON. Both attributes are space-separated references to global transformers like those you studied in section 4.2. You can find the behavior of the exchange-pattern and the transformers and response transformers defined in figure 6.3.

You've learned how to expose services and discovered how to bridge services with a configuration pattern. Now let's explore some defensive techniques to prevent the processing of malformed messages using yet another configuration pattern: the validator.

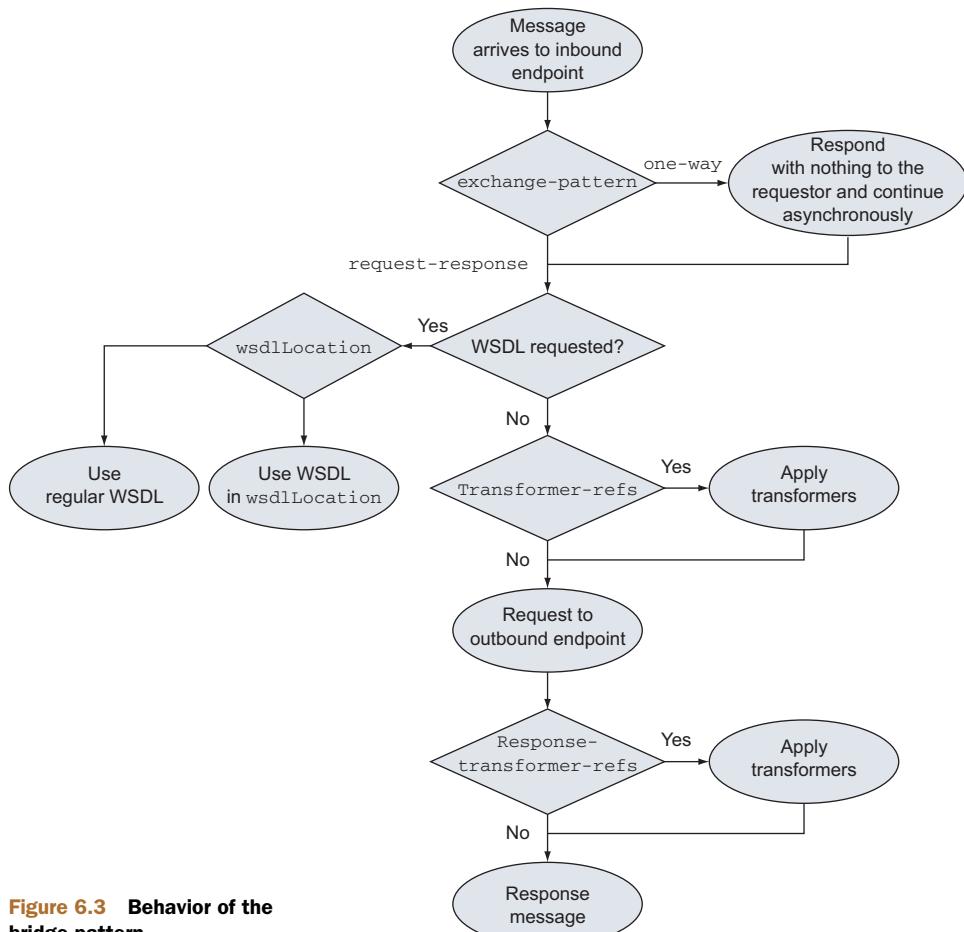


Figure 6.3 Behavior of the bridge pattern

6.2.3 Using the validator

Let's face it: the real world is error prone. Perhaps because of a copy-and-paste error in the documentation, you might find yourself receiving the wrong message in a REST endpoint. Or you could receive JSON instead of your desired XML because of a small misconfiguration in the accept header of your front-end web server.

Although the developer is supposed to be *generous on input, strict on output*, there's a line between generous, with messages that you can handle but that are not totally correct, and the suicidal tendency to accept messages that the application won't understand at all. To reduce the number of lines needed to write a validation, Mule provides the validator pattern.

The validator needs a few Mule expressions to build acknowledgement and rejection messages and a filter to set the conditions for a message to be valid. Let's play with a simple validator that will only validate integer payloads, shown in the following listing.

Listing 6.20 Basic validator

```
<pattern:validator name="integer-validator"
    inboundAddress="vm://service.in"
    ackExpression="#['Message accepted.']")
    nackExpression="#['Message rejected.']")
    outboundAddress="vm://real-service.in">
    <payload-type-filter expectedType="java.lang.Integer" />
</pattern:validator>
```

Here you can easily recognize that you’re instructing Mule to accept messages from an endpoint defined with the `inboundAddress` attribute. Then if the request payload isn’t an integer, the response will be “Message rejected,” and nothing else will happen. But if the request payload is an integer, “Message accepted” will be the response and the message will be sent an endpoint defined with the `outboundAddress` attribute. But it will be sent *asynchronously*.

To send to the outbound endpoint asynchronously means not only that the requestor doesn’t have to wait for the delivery of the message, but also that the requestor won’t be aware of a potential problem in that delivery. This is not necessarily a problem, but you might have different needs. To address the possible scenario in which the requestor has to be informed of an error after the validator, the validator pattern lets you pass the `errorExpression` that will disable its asynchronous behavior and will return the result of the expression to the requestor in case of error; a flow chart with this process is presented in figure 6.4. Let’s put it in action with this listing.

Listing 6.21 Validating with error expression

```
<pattern:validator name="integer-validator"
    inboundAddress="vm://service.in"
    ackExpression="#['Message accepted.']")
    nackExpression="#['Message rejected.']")
    errorExpression="#['Error delivering.']")
    outboundAddress="vm://real-service.in">
    <payload-type-filter expectedType="java.lang.Integer" />
</pattern:validator>
```

The validator pattern shares the aim of the rest of the patterns: to reduce the complexity of the Mule configuration files. Unsurprisingly, the validator pattern will let you express endpoints and the filter as references. In addition, it will let you define the inbound and outbound endpoints as child elements. Let’s put those two styles in practice in listings 6.22 and 6.23.

Listing 6.22 Validator using references

```
<vm:endpoint name="integer-service"
    path="integer-service.in"
    exchange-pattern="request-response" />

<vm:endpoint name="real-integer-service"
    path="real-integer-service.in"
```

```

        exchange-pattern="request-response" />

<payload-type-filter name="integer-filter"
    expectedType="java.lang.Integer" />

<pattern:validator name="integer-service-validator"
    ackExpression="#['Message accepted.']")
    nackExpression="#['Message rejected.']")
    inboundEndpoint-ref="integer-service"
    outboundEndpoint-ref="real-integer-service"
    validationFilter-ref="integer-filter">
</pattern:validator>
```

Listing 6.23 Validator using child elements

```

<pattern:validator name="integer-service-validator"
    ackExpression="#['Message accepted.']")
    nackExpression="#['Message rejected.']")
    outboundAddress="vm://real-integer-service.in">

    <vm:inbound-endpoint path="integer-service.in"
        exchange-pattern="request-response" />

    <payload-type-filter expectedType="java.lang.Integer" />

</pattern:validator>
```

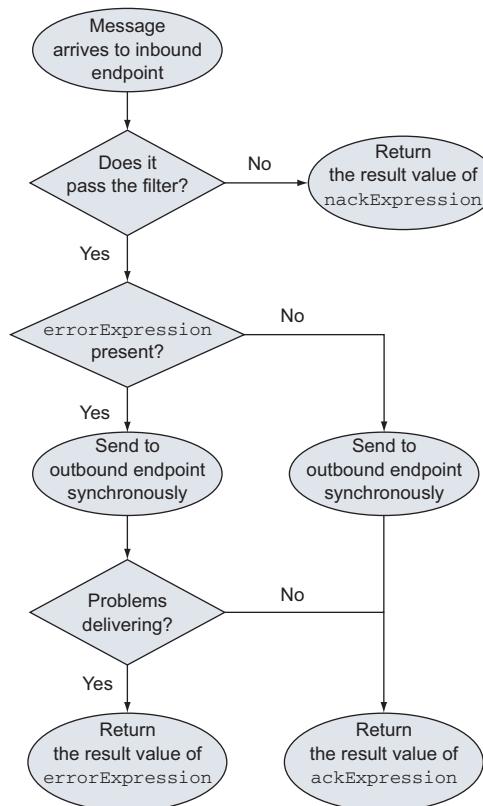


Figure 6.4 Behavior of the validator pattern

You've now acquired enough knowledge about simplifying your validation configuration with the validator pattern. In the next section, we'll take a look at the WS proxy to be able to wrap the services with traversal functionality.

6.2.4 Using the HTTP proxy

Since its creation in 1990, HTTP has been one of the most widely used protocols on the internet. If this were not enough to guarantee it first-class support in Mule, HTTP is also the basis or main transport of dozens of other technologies such as REST and SOAP. You won't be surprised to discover that in the same manner as there's a specialized form of bridge for WS-*, the WS proxy, Mule also has a specialized proxy for HTTP, the http-proxy.

Like the WS proxy, the HTTP proxy can be used, thanks to its middleman nature, in a myriad of scenarios—from the simplest one in which it will proxy the request, to a more sophisticated scenario in which you need the modification of the request itself by using transformers. To see it in action, let's create a simple proxy in the next listing for the Google REST search API.

Listing 6.24 Basic HTTP proxy

```
<pattern:http-proxy name="ajaxSearchProxy"
    inboundAddress="http://localhost:8888"
    outboundAddress=
        "http://ajax.googleapis.com/ajax/services/search/web" />
```

With this simple element you connect any HTTP request to the inboundAddress to the specified outboundAddress. The same response obtained from the outboundAddress will be sent back to the requestor at the inboundAddress.

At this point it's probably needless to say that, like you learned with the rest of the configuration patterns, http-proxy also accepts global endpoints. Let's configure the same proxy created in the last listing, but this time using global endpoints instead of addresses.

Listing 6.25 HTTP proxy using references

```
<pattern:http-proxy name="ajaxSearchProxy"
    inboundEndpoint-ref="ajaxSearch"
    outboundEndpoint-ref="ajaxSearchReal" />
```

Notice the change to the inboundAddress and outboundAddress attributes, with inboundEndpoint-ref and outboundEndpoint-ref making use of global endpoints. Now that you know how to reference global endpoints, you can continue learning how to apply transformers to the proxy.

The request proxied to the outbound endpoint doesn't necessarily have to be an exact copy of the original request; you might need to convert it before passing it, or you may need to convert the response before sending it back to the original requester. You've seen the attributes to enable these transformations in section 6.2.2; if you've

already practiced with it, you'll soon discover that the HTTP proxy has similar configuration and behavior.

Prancing Donkey is an environmentally friendly company. They not only believe in recycling, they are absolutely proud of it. To show that pride to the world, Prancing Donkey wants to show on their website how many beer bottles have been recycled in the last 24 hours. Thankfully, they have a REST service in the software of the recycling plant that, given an XML search document, will return that data also in XML. To pass it to the JavaScript counter of Prancing Donkey, they'll need to convert a JSON request to XML and then vice versa for the response, as shown in this listing.

Listing 6.26 HTTP proxy using transformers and response transformers

```
<pattern:http-proxy name="ajaxSearchProxy"
    inboundAddress="http://localhost:8888"
    outboundAddress=
        "http://127.0.0.1:9999/rest/recycledCount"
    transformer-refs=
        "jsonToObjectTransformer objectToXmlTransformer"
    responseTransformer-refs=
        "xmlToObjectTransformer objectToJsonTransformer" />
```

There's nothing new here; you've already seen the `transformer-refs` and `responseTransformer-refs` attributes in other patterns. They're passed as a space-separated list of transformers. The attribute `transformer-refs` configures the transformers to be applied to the original request before passing them to the outbound endpoint, and the attribute `responseTransformer-refs` configures the transformers for the response of the outbound endpoint before it's proxied back to the original requestor.

The features of the HTTP proxy don't end here. The HTTP proxy also supports the use of a caching strategy to provide cached access to the proxied resource. The next listing shows how to provide a `cachingStrategy` to the HTTP proxy configuration pattern so the requests are effectively cached (illustrated in figure 6.5).

Listing 6.27 HTTP proxy using cache

```
<pattern:http-proxy name="ajaxSearchProxy"
    inboundAddress="http://localhost:8888"
    outboundAddress="http://ajax.googleapis.com/ajax/services/search/web"
    cachingStrategy-ref="cache" />
```

HTTP is one of the more important protocols out there, but it's not the only one. Let's find out how to create a proxy with specific features for the WS-* services.

6.2.5 Using the WS proxy

We already covered WS-* SOAP services in section 3.3.2. They're pretty common services and have been a strong standard for years. Many organizations use them, and you'll need to integrate with them sooner or later. You already know how to expose JAX-WS services using Mule; let's see how to proxy a WS-* service in a simple manner thanks to Mule's WS proxy configuration pattern.

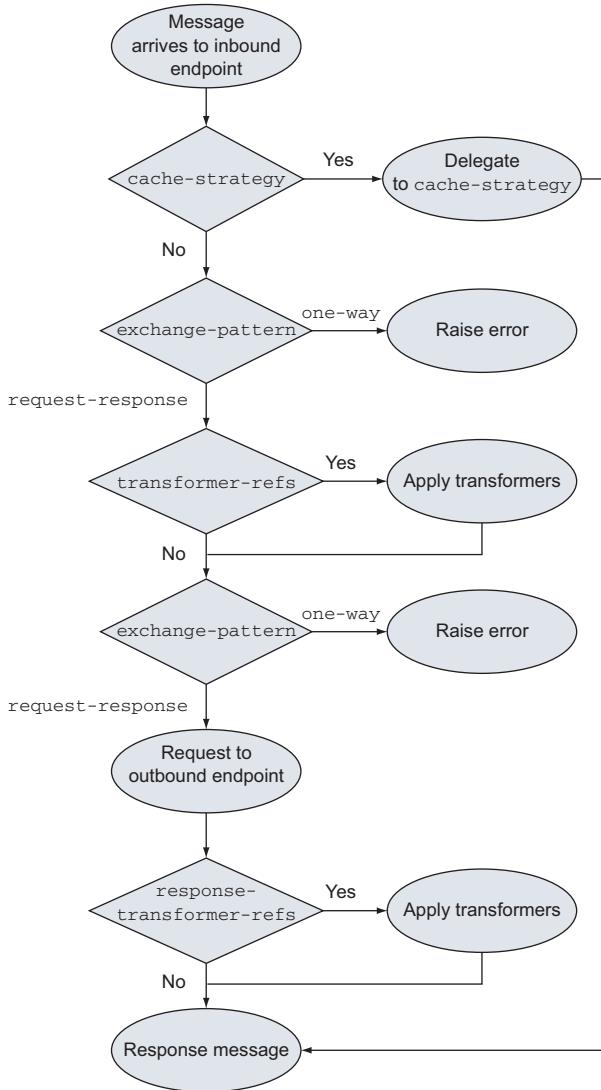


Figure 6.5 Behavior of the HTTP proxy

The WS proxy is a middleman, so it may be used in many scenarios; you might proxy the request or modify the behavior of the request in a transparent (for the requestor) way. Let's put it in practice in the next listing by configuring the simplest form of WS proxy configuration pattern that will proxy the service created in listing 6.16.

Listing 6.28 WS proxy pattern

```
<pattern:web-service-proxy name="weather-forecast-ws-proxy"
  inboundAddress="http://localhost:8080/prancingServices"
  outboundAddress="http://localhost:8090/soap" />
```

There's more magic here than what may appear at first sight. The WS proxy will not only proxy the requests received to the service, but will also provide a WSDL in the address plus /?wsdl that will be the original WSDL with the addresses rewritten to point to the proxy address.

Sometimes you might need to modify the request or the response; with the same attributes that you learned in sections 6.2.2 and 6.2.4, `transformer-refs` and `responseTransformer-refs`, you can modify the request or the response respectively. As you're already familiar with them, let's jump straight into this listing.

Listing 6.29 WS proxy using transformers

```
<pattern:web-service-proxy name="weather-forecast-ws-proxy"
    inboundAddress="http://localhost:8080/prancingServices"
    outboundAddress="http://localhost:8090/soap"
    transformer-refs="add-credentials-transformer"
    responseTransformer-refs="amount-to-words-transformer"
    wsdlFile="localWsdl.xml" />
```

WSDL location

The WS proxy makes an assumption that is *usually* true: The WSDL file is located at the same address as the service plus `wsdl` in the query string `?wsdl`. But the real world has plenty of uncommon practices, so what happens when the WSDL file is *not* in the common location or, even worse, isn't available online at all? You can pass the `wsdlLocation` attribute to indicate an alternative location or `wsdlFile` to use a locally stored file (illustrated in figure 6.6).

Listing 6.30 WS proxy using a different WSDL file

```
<pattern:web-service-proxy name="weather-forecast-ws-proxy"
    inboundAddress="http://localhost:8080/prancingServices"
    outboundAddress="http://localhost:8090/soap"
    wsdlFile="localWsdl.xml" />
```

This is the last of the configuration patterns available in Mule. With it you can create WS proxies with added value like security, auditioning, or transformation. It shares a style and features with the rest of the configuration patterns.

6.2.6 Reusing common configuration elements

The Mule configuration patterns are about real life, and sometimes real life is about multiple messy scenarios with almost but not quite the same requirements. If not handled with care, those many similar use cases can end up leading to a big Mule configuration file that only holds similar lines. To alleviate this problem, the Mule configuration patterns support inheritance.

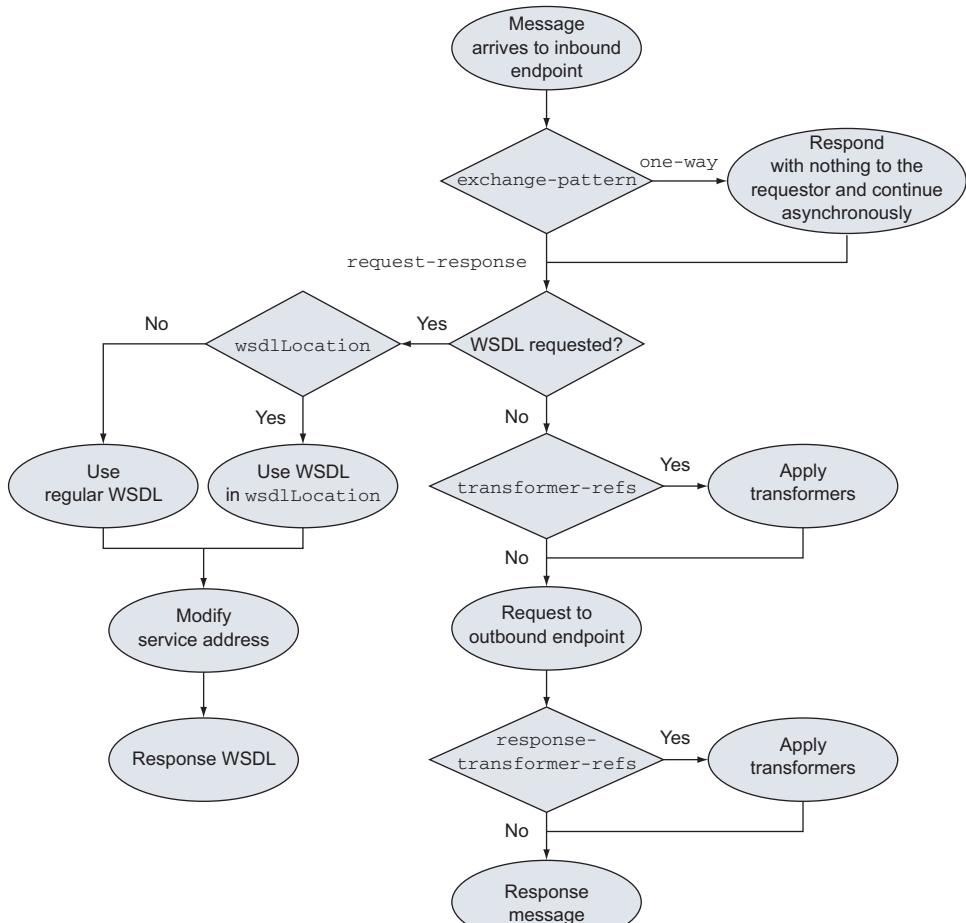


Figure 6.6 Behavior of the WS proxy

What does inheritance mean for the configuration patterns? The concept of inheritance in the Mule configuration files is similar to the programming paradigm, but here it's only applied to reuse of configuration elements. If you have a few patterns that are similar, you could set a parent abstract pattern and then create the patterns with a parent attribute pointing to the abstract one that holds all the shared configuration.

In this way you'd save lines in your configuration files and, at the same time, you'd reduce complexity by configuring the shared parts once and minimizing potential typos in an eventual evolution of the configuration file.

Now let's see how to put inheritance into action.

Prancing Donkey needs two similar validations, one checking that the payload is valid XML and the other checking that the payload is a String. For both, they want to share the same expressions. You can see how to configure this inheritance in the following listing.

Listing 6.31 Validator inheritance

```
<pattern:validator name="abstract-parent-validator"
    abstract="true"
    ackExpression="#['Message accepted.']")
    nackExpression="#['Message rejected.']) />

<pattern:validator name="integer-service-validator"
    parent="abstract-parent-validator"
    inboundAddress="vm://integer-service.in"
    outboundAddress="vm://real-integer-service.in">
    <payload-type-filter expectedType="java.lang.Integer" />
</pattern:validator>

<pattern:validator name="string-service-validator"
    parent="abstract-parent-validator"
    inboundAddress="vm://string-service.in"
    outboundAddress="vm://real-string-service.in">
    <payload-type-filter expectedType="java.lang.String" />
</pattern:validator>
```

Here you can see that you share the expressions defined in the abstract parent of both children configuration patterns by setting the parent attribute. This effectively makes a cleaner configuration than fully configuring both pattern elements. The benefits increase potentially with the number of children.

Keep in mind that inheritance is supported in all the configuration patterns. Using it where necessary will boost the simplicity of your configuration files and will definitely make the eventual maintenance tasks easier.

6.3 Summary

Components are first-class citizens of a Mule configuration. They can be discreet (but efficient) and they can also encompass custom business logic under the form of standard Java objects, JSR 223 scripts, and the Mule Expression Language. You've also learned about the numerous options Mule provides you with when it comes to running your own business logic in custom components.

You've discovered that Mule offers, for the more common tasks, a wealth of configuration patterns that allow you to connect to external systems, route, transform, enrich, and even expose the components to the world.

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, components, and configuration patterns. In part 2, we'll look at more advanced topics such as integration architecture and transaction management. We'll also look at deployment and monitoring strategies to ensure that Mule plays its prime role in your IT landscape in the best possible conditions.

Part 2

Running Mule

I

In part 1, you learned about the fundamentals of Mule. You discovered its philosophy, configuration principles, and major moving parts. You also ran through examples that exercised the main building blocks of Mule services: transports, transformers, routers, components, and patterns. Part 2 will take you further and will guide you through the next steps of running Mule in a production environment.

Mule doesn't come complete with a predetermined approach to architecting solutions. Each project can use whatever methodology best fits its needs. This said, in chapter 7 we'll review what we consider to be the major architectures and patterns through which Mule provides the most bang for your buck.

As an ESB and integration platform, Mule supports several very different deployment strategies. In chapter 8, we'll review these strategies, including how one can successfully run Mule in a highly available fashion.

Problems happen. As tough as it is, Mule can stumble. This usually translates into exceptions being thrown. Chapter 9 will guide you when putting in place a sound error-management strategy. You'll also learn how to recover from connection flakiness and what Mule offers in terms of transaction management, a must for environments in which data integrity is critical.

All enterprise applications are subject to security concerns, and Mule doesn't escape these constraints. Chapter 10 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.

To wrap up the discussion about running Mule in production, we'll focus on Mule's threading model in chapter 11. We'll also discuss how thread pools and processing strategies can be tuned and what techniques you can use to profile and performance-boost your Mule-driven applications.



Integration architecture with Mule

This chapter covers

- Structuring integration applications
- Common architectures used in Mule applications
- Mule application implementation patterns

During the course of this book, we've shown you the building blocks of piecing together integration solutions with Mule. We haven't, however, talked about the architecture of such applications. Ideally your Mule applications should exhibit the properties of any other well-architected software application: they should be easy to refactor and test, modular, and decoupled. They should also be constructed with a view to future needs, making them easy to modify as requirements inevitably change.

In this chapter, you'll see how to architect integration solutions with Mule. We'll start off by presenting three different architectural approaches to structuring Mule applications. We'll then take a look at implementation patterns that can aid in the composition of integration solutions. Seeing how Prancing Donkey approaches and evolves their integration solutions will glue together the concepts we introduce.

Hopefully you'll finish this chapter by getting the big picture of how real-world applications are composed using the techniques in this book.

7.1 Structuring integration applications

Mule can function in multiple ways, from an integration application to the central integration fabric of a Fortune 500 enterprise. The architecture of these applications, whether implicit or explicit, will impact their success. In this section, we'll look at three ways Mule can be used for nontrivial projects.

7.1.1 Guerrilla SOA with hub and spoke

Integration is, by its nature, often ad hoc. All developers wish for unbounded time constraints and an unlimited budget to realize a platonic ideal of architecture, implementation, and delivery. The reality, particularly with integration products trying to realize the cost benefits associated with SaaS solutions, is often the opposite.

A core benefit of integration frameworks such as Mule is that they provide structure around these integration exercises, avoiding the dreaded plate-of-spaghetti, point-to-point solutions. Ideally the integration layer should get out of the way of the developers trying to tie systems together, both when initially getting a solution up and running as well as when paying back the technical debt that has inevitably occurred.

In chapter 1 we mentioned the concept of *Guerrilla SOA*. This is a bottom-up approach to implementing integration architecture that values the pragmatic, iterative delivery of features over the top-down governance model of service-oriented architectures. Guerrilla SOA is best achieved using a hub-and-spoke architecture with Mule. This architecture pattern is best suited to organizations with quickly changing requirements or small-to-medium integration needs. Figure 7.1 illustrates Prancing Donkey's current integration landscape as an example of how hub-and-spoke architectures are implemented with Mule.

This architecture consists of a single Mule deployment containing four Mule applications. Each application hosts services siloed based on business requirements. All the flows implementing accounting use cases, for instance, are placed in the Accounting Services application. This allows Prancing Donkey to use Mule's hot deployment features in the standalone server to manage each application separately. It also decouples unrelated services from each other, allowing the development teams to implement integration solutions in parallel.

As in the other architecture approaches you'll see in this chapter, Mule acts as an intermediary between all interactions with remote services. The front-end applications use the protocols that make the most sense for them to integrate with Mule. The web store, for instance, takes advantage of Prancing Donkey's JMS infrastructure for asynchronous submission. External API clients, not being on Prancing Donkey's network, don't have this luxury.

Although this centralizes all the integration logic in the Mule applications, it also introduces a single point of failure. You'll see in section 7.2 how this can be mitigated by using reliability patterns. The multitude of protocols on the front end also means

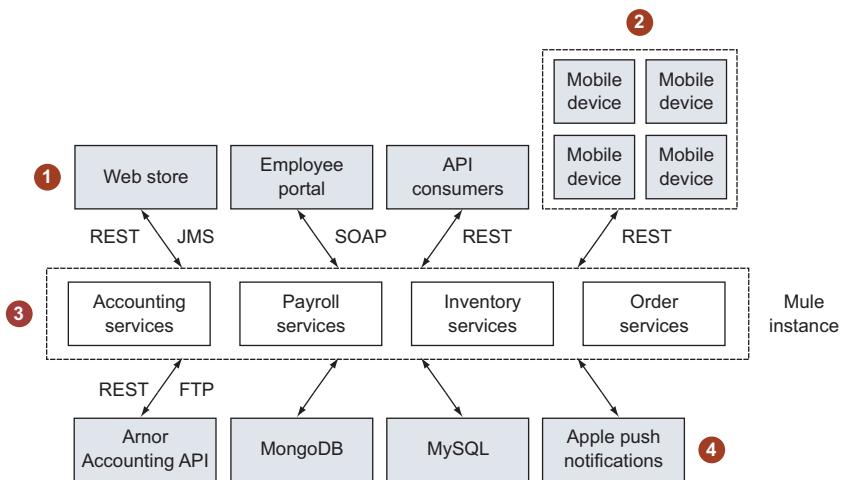


Figure 7.1 Prancing Donkey’s Guerrilla SOA architecture using a hub-and-spoke architecture: ① web application clients to Mule applications; ② mobile device clients to Mule applications; ③ standalone Mule instance hosting four discrete applications; ④ various external integration points decoupled from the front-end client.

that different transformation implementations are used in each Mule application, with little reuse between each application. You’ll see in the next section as well as in section 7.2.1 how the canonical domain model pattern can improve this design.

A nice aspect of hub-and-spoke architectures is that their design is typically iterative. Mule’s flexible deployment architecture, which you’ll see in chapter 8, means you can start with an embedded Mule instance in your application that’s internally decoupling integration points with the VM transport. Eventually this can be refactored into a standalone Mule application in which the services are then decoupled with JMS, a web services protocol, or something like ZeroMQ.

Teams practicing Guerrilla SOA tend to develop and own the integration solutions. This often means there aren’t developers purely focused on Mule, increasing the odds that people won’t jump through hoops to avoid interacting with a potentially slow-moving “integration team.”

Now let’s see how the ESB architecture can potentially improve on some of the deficiencies of hub-and-spoke, particularly for larger organizations.

7.1.2 Mule as the enterprise service bus

Enterprise service bus implementations are a more formal, top-down variant of the hub-and-spoke architecture. An ESB architecture is typically a piece of a larger initiative or a new “green field” development in which the team has the luxury of time and resources to plan out an integration. Usually other applications the ESB will integrate with are either known or being architected up front. In such situations a canonical domain model, which we’ll discuss in section 7.2.1, is often also known up front and is being shared with the applications integrating with the ESB. Figure 7.2 illustrates the ESB architecture of Prancing Donkey’s largest competitor, Mordor Brewing Company.

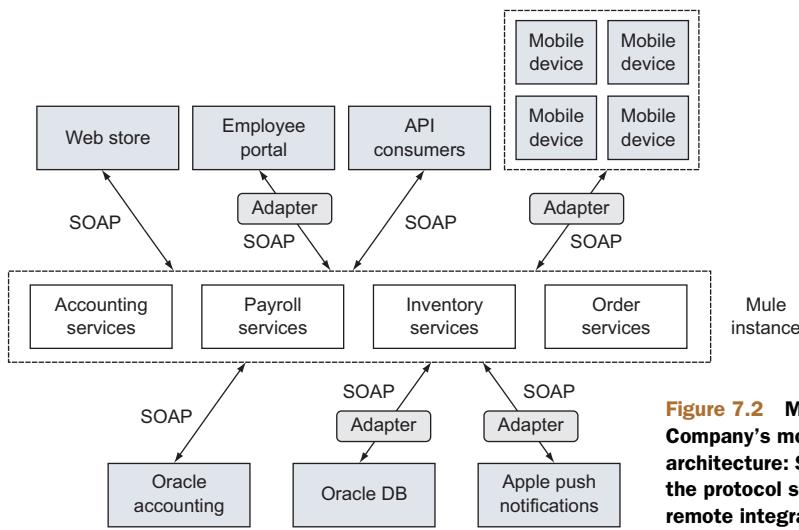


Figure 7.2 **Mordor Brewing Company's monolithic ESB architecture: SOAP is enforced as the protocol standard on all remote integration points.**

At first glance, this architecture seems very similar to hub-and-spoke, and it is. The topologies are, in fact, mostly identical. In Mordor's case, for instance, they're still using Mule's ability to simultaneously host multiple applications to segment their flows by business silo. The crucial difference is that this architecture was defined prior to any implementation, and that shown in figure 7.1 was drawn as a snapshot in time of the system's evolution. The second major difference is that all traffic to and from the ESB will be SOAP over either HTTP or JMS. This limits the integration flows to purely routing, transformation, security, and possibly reliability. This makes SOAP an obvious choice as an implementation protocol, given that there are WS specifications for each of these use cases.

Because SOAP is to be strictly enforced as the communication method to and from Mule, channel adapters must be implemented for the remote integration points that don't natively support SOAP. In this example, Mordor's portal software has no SOAP support. As such, the portal team must implement a mechanism to bridge the portal's internal API to speak to SOAP.

Although monolithic ESB architecture has largely fallen out of favor in recent years, it can still be a good choice for a large organization with a dedicated integration team. SOAP, despite many of its faults, still delivers excellent tooling that's largely unrivaled by competing integration protocols such as REST.

We'll now take a look at how Mule can function purely as a mediation layer.

7.1.3 Mule as a mediation layer

Using Mule purely for its mediation features, particularly routing and security, is emerging as a common architectural pattern. Heavily services-oriented environments introduce a unique set of problems. Multiple versions of services often need to be supported. Implementing cross-cutting concerns between services, such as security, is

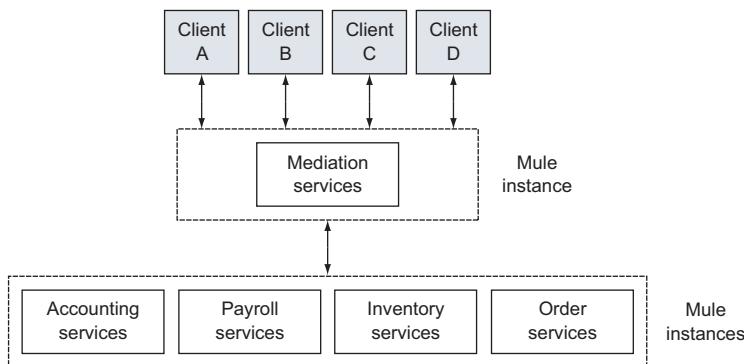


Figure 7.3 Mule as a service mediation layer: a dedicated Mule application to perform mediation

often a challenge. What happens, for instance, if the security infrastructure is augmented or changed in an environment with 100 back-end services? It would be nice to be able to make this change in one place rather than in 100 downstream applications. Mule's routing and security features provide an excellent place to host this functionality.

Prancing Donkey is at the point where they want to centralize authentication and schema validation for their services. Their plan is to introduce a dedicated Mule server hosting a single application that performs mediation, as illustrated in figure 7.3.

The mediation layer will, in this case, be performing schema validation against the XML payload of the messages being passed to it. It'll perform a security check against the messages, ensuring they have an HTTP basic authentication header that validates against Prancing Donkey's LDAP infrastructure. It'll then use a Java component that integrates with an external registry product to determine the final destination of the message. This is demonstrated in the flow shown in the following listing.

Listing 7.1 Using Mule as a service mediation layer

```

<flow name="ServiceMediationFlow">
    <http:inbound-endpoint
        exchange-pattern="request-response"
        host="${http.host}"
        port="${http.port}"/>
    <byte-array-to-string-transformer/>
    <mule-ss:http-security-filter realm="mule-realm"/>
    <mulexml:schema-validation-filter
        schemaLocations="schema/order.xsd"
        returnResult="true"/>
    <component
        class="com.prancingdonkey.service.RegistryLookupService"/>
    <http:outbound-endpoint
        exchange-pattern="request-response"
        host="${http.target.host}" port="${http.target.port}"
        path="#{flowVars['resolvedEndpoint']}"/>
</flow>

```

Accept web requests to proxy

Perform schema validation

Perform security validation

① Perform a registry lookup

② Dynamically route request to appropriate endpoint based on a flow variable populated by registry lookup

In addition to performing schema and security validation of the message, this flow determines how to route the message based on a service registry lookup, performed by the Java component ①. This component populates a flow variable that populates the address of the HTTP outbound-endpoint ②.

You've seen three different ways to structure integration solutions with Mule in this section. This is obviously not an exhaustive list, but instead shows the more common architectures developers typically choose. Other architectural approaches also exist. Mule can be used, for instance, in a grid architecture with multiple nodes performing map-reduce operations on datasets, for instance. Peer-to-peer architectures are also possible.

Now let's take a look at some patterns that simplify the implementation of these architectures.

7.2 **Mule implementation patterns**

In this section, we'll take a look at patterns that simplify the composition of integration applications. These aren't explicit features of Mule but rather are approaches that take advantage of Mule's functionality to ease integration application development. You'll see how adopting a canonical domain model enables you to decouple your business logic from your integration operations, such as routing and transformation. We'll then take a look at how to use asynchronous messaging to build resiliency into your Mule flows.

7.2.1 **Using a canonical data model**

A canonical data model is a useful way to structure the payloads of your Mule messages to simplify the implementation of your Mule applications. To use a canonical data model, you must introduce a common format to which message payloads are transformed prior to any further processing by Mule. Canonical data models are typically Java domain objects, an XML schema, or an agreed-upon JSON format.

Let's consider one of Prancing Donkey's use cases. Prancing Donkey is using Salesforce as their CRM. Mule's Salesforce connector makes the mechanics of this implementation trivial but introduces some challenges when data is retrieved from Salesforce for processing. The objects returned from the Salesforce connector are Java objects generated from the WSDL of Salesforce's SOAP API. These objects tightly couple Prancing Donkey CRM flows in Mule to Salesforce's API. Such a coupling will make it difficult for Prancing Donkey to change CRM providers in the future or augment the CRM functionality with another application.

A canonical data model avoids this coupling. The next listing shows the `Customer` domain object that's part of the canonical model Prancing Donkey introduced.

Listing 7.2 The Customer canonical data model

```

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer implements Serializable {

    String customerId;
    String firstName;
    String lastName;
}

```

Now let's look at the flows responsible for creating and querying contacts in the CRM (illustrated in figure 7.4).

Listing 7.3 Populate and query records in Salesforce

```

<flow name="createContact">
    <vm:inbound-endpoint exchange-pattern="one-way"
        path="crm.contact.create"/>
    <sfdc:create-single config-ref="sfconfig" type="Contact">
        <sfdc:object>
            <sfdc:object key="FirstName">
                #[message.payload.firstName]
            </sfdc:object>
            <sfdc:object key="LastName">
                #[message.payload.lastName]
            </sfdc:object>
        </sfdc:object>
    </sfdc:create-single>
</flow>
<flow name="getContact" >
    <vm:inbound-endpoint exchange-pattern="request-response"
        path="crm.contact.get"/>
    <sfdc:query-single config-ref="sfconfig"
        query=
    "SELECT Name from Contact where Name = '#[message.payload]'">
        <transformer ref="salesforceResultToCustomerTransformer"/>
    </sfdc:query-single>
</flow>

```

The diagram shows two vertical arrows pointing downwards from the flow definitions. The top arrow originates from the 'createContact' flow and points to a callout box containing the text 'Use MEL to create the Salesforce contact object'. The bottom arrow originates from the 'getContact' flow and points to another callout box containing the text 'Custom transformer is used to convert response from the Salesforce connector, a Map, to an instance of Customer'.

You can see that the `createContact` flow uses MEL to populate the appropriate fields in the Salesforce contact object ①. The inverse is done using the custom transformer ②, which converts the result from a Map to a `Customer` instance.

The canonical data model is a great way to further decouple your Mule integrations from the rest of your applications. The example in this section used a POJO data model, but, as we already mentioned, other domain models like JSON and XML are as easily supported. The canonical data model is a powerful technique when combined with Mule's annotation support, allowing you to keep any business logic hosted in Mule independent of Mule's container at runtime. Such an approach makes it easy to unit test business logic code outside of the Mule container and also gives you the ability to move such code into and out of Mule as necessary.

Now let's take a look at how you can use asynchronous messaging to reliably deliver messages with Mule.

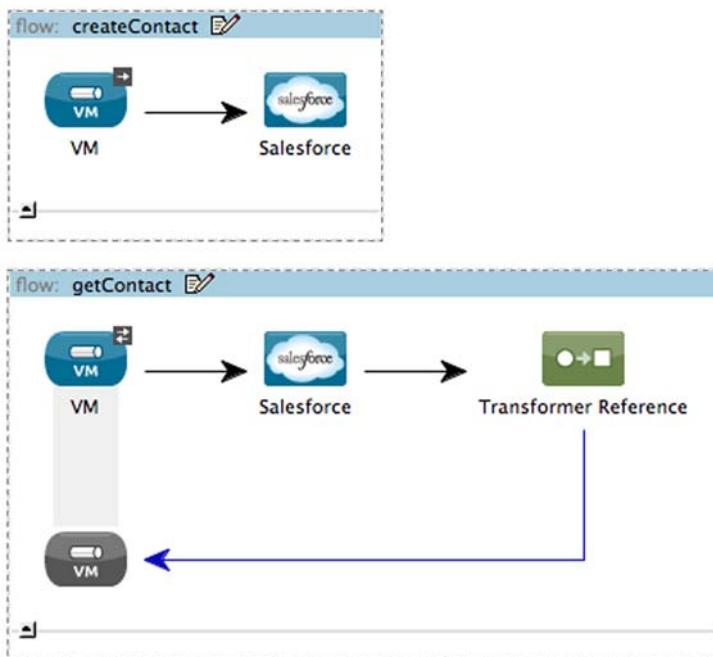


Figure 7.4 Populate and query records in Salesforce

7.2.2 Reliability patterns with asynchronous messaging

Asynchronous messaging providers such as JMS, AMQP, or Mule’s own VM transport provide an opportunity to decompose integration applications into decoupled, reliable segments. As illustrated with Gregor Hohpe’s seminal Starbucks analogy in “Your Coffee Shop Doesn’t Use Two-Phase Commit” (<http://mng.bz/fb70>), such approaches compose an otherwise synchronous transaction into a series of asynchronous steps. This has a variety of benefits, from allowing the client to perform other work while waiting for the “transaction” to finish to layering resiliency into an otherwise unreliable step.

Let’s consider an example. Prancing Donkey is beginning a re-architecture to allow orders to be submitted from their iPhone application. The ordering process on the server side is fairly complicated and involves multiple remote systems that Prancing Donkey has no control over (Salesforce and NetSuite, for instance). One of the goals of the re-architecture is to provide a response as soon as possible to the mobile device from which the order was submitted. This will be accomplished by accepting the order over HTTP, generating an order ID, and transactionally submitting it to a JMS queue for processing. If the JMS transaction that submits the message on the queue succeeds, then a response containing the order ID is returned to the mobile device. This ID can then be used later to track the status of the order. The “front end” part of this flow is shown in the following listing, and illustrated in figure 7.5 (transactions are covered in depth in chapter 9).

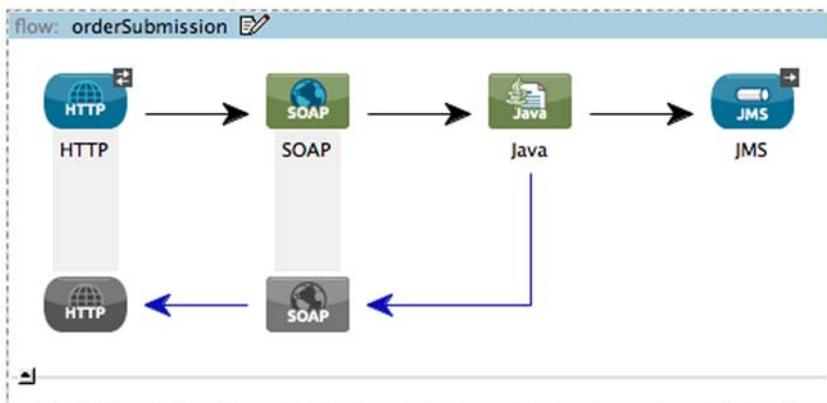


Figure 7.5 Front-end order submission

Listing 7.4 Front-end order submission

```

<flow name="orderSubmission">
    <http:inbound-endpoint exchange-pattern="request-response"
        host="localhost" port="8081"
        path="order"/>
    <component
        serviceClass="com.prancingdonkey.service.OrderSubmissionService"/>
    <component
        class="com.prancingdonkey.service.OrderSubmissionServiceImpl"/>

    <async>
        <jms:outbound-endpoint queue="order.submit">
            <jms:transaction action="ALWAYS_BEGIN"/>
        </jms:outbound-endpoint>
    </async>
</flow>

```

Submit the Order object transactionally to a JMS queue

When the mobile device receives the order ID from the Mule flow, it can be sure that the order has been submitted, provided that Prancing Donkey's JMS infrastructure is robust and the back end of the order submission is implemented properly. This ID will then be used to track the order on another screen of the mobile application's UI. Now let's take a look at the back end of the flow in the next listing (illustrated in figure 7.6).

Listing 7.5 Back-end order processing

```

<flow name="orderProcessing">
    <jms:inbound-endpoint queue="order.submit">
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:inbound-endpoint>
    <all>
        <jms:outbound-endpoint queue="crm.customer.create">

```

Transactionally accepts order from a JMS queue

```

        <jms:transaction action="ALWAYS_JOIN" />
    </jms:outbound-endpoint>
    <jms:outbound-endpoint queue="erp.order.record">
        <jms:transaction action="ALWAYS_JOIN" />
    </jms:outbound-endpoint>
</all>
</flow>

<flow name="orderCompletion">
    <jms:inbound-endpoint queue="order.complete">
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:inbound-endpoint>
    <collection-aggregator timeout="60000"
        failOnTimeout="false"/>
    <jms:outbound-endpoint topic="events.orders.completed">
        <jms:transaction action="ALWAYS_JOIN" />
    </jms:outbound-
    endpoint>
</flow>

```

Joins in previous transaction and submits to erp.order.record queue

Joins in previous transaction and submits to crm.customer.create queue

Waits and aggregates responses from Salesforce and NetSuite

Dispatches order completion events to events.orders.completed JMS topic

These two flows handle the asynchronous routing and aggregation of back-end order processing. The orderProcessing flow accepts a message and submits it to two queues in a single transaction. The orderProcessing flow submits the Order object to a Salesforce flow similar to the one you saw previously. The erp.order.record flow submits the Order object to a flow that uses Mule's NetSuite cloud connector.

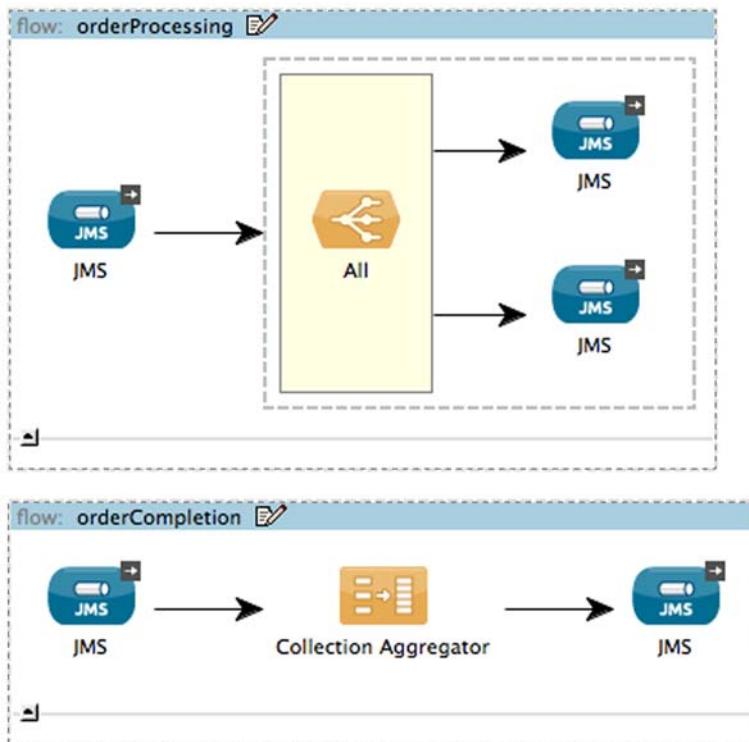


Figure 7.6 Back-end order processing

Prancing Donkey is using ActiveMQ's message redelivery feature to redeliver messages up to a certain point in the event of a transactional rollback. In this case, if there's any failure submitting to any of the queues, then the broker will attempt again after a specified interval. Once this interval is reached, the message is placed on a DLQ. This effectively makes otherwise unreliable services, such as NetSuite or Salesforce, robust.

It's not shown in this example, but after the completion of the NetSuite and Salesforce flows, messages are sent to the `order.complete` queue. Mule's collection aggregator, which you saw in chapter 5, will wait for both responses for an order to arrive before dispatching the Order on `events.order.completed`. Prancing Donkey uses JMS topics to generate events when certain business phenomena occur. You'll see a little bit later how complex event processing can be used to act on this data.

Service hosting with Mule

Web service hosting has typically been the domain of dedicated applications or web servers like JBoss AS or Tomcat. Both platforms provide varying degrees of support for the Java EE stack and multitenancy. For web services, particularly those built on top of JAX-RS and JAX-WS, they were natural choices as host platforms. Although such applications could be hosted on Mule, this was typically a difficult affair—particularly because the standalone server could only host one application at a time, and application deploys required a restart of the server.

Things have changed in Mule 3. As you saw in chapter 3, Mule fully supports the JAX-RS and JAX-WS specifications via Jersey and Apache CXF. The standalone server, as you'll see in chapter 8, is now fully multitenant. Hot deployment of applications is also supported. Finally, Mule's deployment descriptor, a ZIP file, is fully transparent, and those used to working within exploded WAR files will feel completely at home.

Using Mule as a decoupling middleware platform, like you saw in this section, lets you use reliable messaging to make otherwise transient, fatal errors in an unreliable transport recoverable.

7.2.3 **Proxying SOAP requests with CXF**

In chapter 3, you saw how JAX-WS, in conjunction with Apache CXF, can be used to build and consume SOAP web services with Mule. JAX-WS is an annotation-driven mechanism for web services consumption. It requires you to either manually define the Java classes or use an automated tool, such as CXF's `wsdl2java`, to automatically generate JAX-WS classes from a WSDL.

It can be desirable, however, to avoid the overhead of intermediary Java classes and directly work with the XML payloads of SOAP requests. This would enable you to do things such as route a SOAP request based on an XPath evaluation or use XSLT to map one SOAP request to another. This can be significantly easier, and more performant, than incurring the expense of marshaling XML documents back and forth to a Java object graph.

Mule's CXF proxy service and proxy client allow you to work directly with a SOAP message's XML body. Let's see how these work. You'll start off by refactoring Prancing Donkey's brew listing web service that you built in chapter 3 to use an external service invoked over the VM transport to get the listing of brews.

Listing 7.6 Proxying SOAP service requests with the CXF proxy

Transform
XML
document
to a String

```
<flow name="brewSoapService">
    <http:inbound-endpoint address="http://localhost:8090/soap"
        exchange-pattern="request-response"/>
    <cxf:proxy-service wsdlLocation="brew.wsdl" service="BrewService"
        namespace="http://service.prancingdonkey.com/" />
        ↴
        <mulexml:dom-to-xml-transformer/>
        ↴
        <vm:outbound-endpoint path="brew.lookup"
            exchange-pattern="request-response"/>
    </flow>
```

① Strip SOAP
Envelope from
the message

The HTTP inbound endpoint in this example is identical to the one used with the JAX-WS service class in chapter 3. You're not changing the way you accept SOAP requests; in both cases it's over HTTP. Things are different, however, with ①. The cxf:proxy-service here is performing two functions. The wsdlLocation and namespace attributes let you host the WSDL from the service without resorting to a JAX-WS annotated interface. Primarily, however, the cxf:proxy-service is being used to strip the Envelope from the SOAP request. Listings 7.7 and 7.8 show what the SOAP request looks like before and after processing by the cxf:proxy-service.

Listing 7.7 MuleMessage payload before CXF service proxy

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:ser="http://service.prancingdonkey.com/">
    <soapenv:Header/>
    <soapenv:Body>
        <ser:getBrews/>
    </soapenv:Body>
</soapenv:Envelope>
```

Listing 7.8 MuleMessage payload after CXF service proxy

```
<ser:getBrews
    xmlns:ser="http://service.prancingdonkey.com/" />
```

As you can see, the SOAP Envelope has been stripped from the message's payload, allowing you to work directly with the content of the SOAP message. In this case, you're routing the XML document minus the SOAP Envelope to a VM endpoint that's returning the XML document shown in the next listing.

Listing 7.9 Result from vm://brew.lookup

```
<?xml version="1.0" encoding="UTF-8"?>
<ns1:getBrewsResponse xmlns:ns1="http://service.prancingdonkey.com/">
    <ns1:return>
        <ns2:Brew xmlns:ns2="http://model.prancingdonkey.com">
            <ns2:description>Hobbit IPA</ns2:description>
            <ns2:name>Hobbit
                IPA</ns2:name>
        </ns2:Brew>
        <ns2:Brew xmlns:ns2="http://model.prancingdonkey.com">
            <ns2:description>Frodos
                Lager</ns2:description>
            <ns2:name>Frodos Lager</ns2:name>
        </ns2:Brew>
    </ns1:return>
</ns1:getBrewsResponse>
```

Note that the response from the VM invocation is a plain XML document; there's no SOAP Envelope wrapped around it. The CXF proxy service, because the flow is request-response, will wrap the message payload in a SOAP Envelope, allowing you to return a response to the SOAP client. The following listing shows what the final document looks like.

Listing 7.10 Final proxied response

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <ns1:getBrewsResponse xmlns:ns1="http://service.prancingdonkey.com/">
            <ns1:return>
                <ns2:Brew xmlns:ns2="http://model.prancingdonkey.com">
                    <ns2:description>Hobbit IPA</ns2:description>
                    <ns2:name>Hobbit IPA</ns2:name>
                </ns2:Brew>
                <ns2:Brew xmlns:ns2="http://model.prancingdonkey.com">
                    <ns2:description>Frodos Lager</ns2:description>
                    <ns2:name>Frodos Lager</ns2:name>
                </ns2:Brew>
            </ns1:return>
        </ns1:getBrewsResponse>
    </soap:Body>
</soap:Envelope>
```

The CXF module can also be used to proxy SOAP requests. The next listing demonstrates how you can use the CXF proxy client to wrap an arbitrary XML document in a SOAP Envelope.

Listing 7.11 Proxying SOAP client requests with the CXF proxy

```
<flow name="brewClientProxy">
    <vm:inbound-endpoint path="brews"
        exchange-pattern="request-response" />
    <byte-array-to-string-transformer/>
    <cxfrs:proxy-client payload="body"
        enableMuleSoapHeaders="false" />
    <http:outbound-endpoint address="http://localhost:8090/soap"
        exchange-pattern="request-response" />
</flow>
```

① Wrap MuleMessage's payload with a SOAP Envelope

The payload attribute ① lets CXF know if the `MuleMessage` is already a SOAP document. Because it isn't in this case, you'll set the payload to `body`, indicating you want the message's payload wrapped in a SOAP Envelope. The CXF proxy client, after wrapping the message's body in a SOAP Envelope, will send the request to the HTTP outbound endpoint. When the response comes back, the Envelope will be stripped from the message, allowing you to work directly with the XML of the response. Setting `enableMuleSoapHeaders` to `false` ensures that any outbound properties on the `MuleMessage` aren't propagated in the intermediary SOAP request.

Mule's agnosticism to both the payload of message and the architecture of integration applications makes it easy to implement patterns such as the canonical data model, decoupling middleware, and generic SOAP proxies.

7.3 Summary

Defining the architecture for any software application can be a daunting task. This is particularly true of integration applications which, by nature, often involve a multitude of technologies, distributed systems, and long-running processes. In this chapter, you saw how three different approaches to integration application architecture can potentially mitigate the difficulties of such implementations. The choice of an architecture, even if it's an evolving one, can greatly impact the agility of such applications to adapt to ever-changing requirements, technologies, and use cases. Patterns also help here. The canonical data model and the use of decoupling middleware give you additional avenues for agility.

Hopefully the discussion in this chapter will give you confidence in choosing and implementing integration applications with Mule. We'll now take a look at how Mule applications are deployed.

Deploying Mule



This chapter covers

- Deploying standalone Mule applications
- Deploying Mule to a web container
- Embedding Mule into an existing application
- Deploying Mule for high availability

You may have reached this chapter following the natural order of the book: you've learned how to configure Mule and have seen a few samples running. At this point, you might be wondering how to move from a "works on my machine" situation to a "Mule running in production" one.

You may also have directly jumped to this chapter because your main concern is to figure out if and how Mule will fit into your IT landscape. You may have concerns about what you will end up handing off to your production team. They may have operational or skill constraints that you must absolutely comply with. You might also be wondering about the different deployment topologies Mule can support.

In the upcoming sections, you'll come to realize that Mule is incredibly flexible and can adapt to your needs. In this respect, Mule differs from many of its competitors, which often mandate a specific 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 that point, you'll be glad that Mule is such a versatile platform.

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 by directly creating configuration files.

Mule differs from this shrink-wrapped-application approach because of its dual nature. Mule is a lightweight messaging framework and a highly distributable object broker,¹ which means that it supports more deployment strategies and greater flexibility than a traditional ESB does. As an object broker, Mule can be installed as a stand-alone server pretty much like any application server. As a messaging framework, Mule is also available as a set of libraries² 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 constraints. For example, you might need to connect to local-only services in a particular server, or the standard production environment in use in your company might constrain you to a particular web container.

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 one, there's no absolute hindrance that would prevent you from migrating from one deployment strategy to another.

In the following sections, we'll detail the five different deployment strategies that are possible with Mule and present you with 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 it's time to decide which deployment strategy you'll follow.

8.1 Deploying standalone Mule applications

The most simple (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 available on the MuleSoft website (www.mulesoft.org/documentation/display/MULE3INSTALL/). When deployed this way, Mule relies on the Java Service Wrapper from Tanuki Software to control its execution. Figure 8.1 shows a conceptual representation of this deployment model.

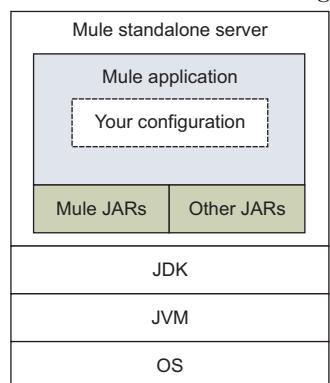


Figure 8.1 Conceptual deployment model of Mule standalone server

¹ Don't confuse messaging framework (Mule, Camel, Spring Integration) with messaging middleware (HornetQ, RabbitMQ).

² More specifically, a set of Maven artifacts.

The wrapper is a comprehensive configuration and control framework for Java applications. By using it, Mule can be deployed as a daemon on Unix-family operating systems or as a service on Microsoft Windows. The wrapper is a production-grade running environment: It can tell the difference between a clean shutdown and a crash of the Java application it controls. If the application has died in an unexpected manner, the wrapper will restart it automatically.

The following is the command line supported by the wrapper:

```
mule [console|start|stop|restart|status|dump] [-app appName]
```

If no particular action is specified, console is assumed; the instance will be bound to the terminal from which it's been launched. If no application name is specified, Mule will start every single application located in the apps directory. The startup script supports several other optional parameters including -debug (to activate the JVM's remote debugging; see section 12.5) and -profile (to enable YourKit profiling; see chapter 11).

Mule as a system service

The mule command line supports the standard System V commands. If your Unix init daemon is compatible with System V, and the odds are definitely in your favor, you can symlink the Mule startup script in /etc/init.d and use it as any other System V service. Follow your Unix distribution manual for more information on how to install services, or refer to the Java Service Wrapper site (<http://wrapper.tanukisoftware.com/doc/english/launch.html>).

If you're using Windows instead, the commands `mule install` and `mule remove` will install and uninstall Mule as a Windows service.

Going through the wrapper

Often you'll need to pass some parameters to your Mule configuration by using Java system properties (for example, to provide values for properties placeholders in your configuration file). Because of the wrapping nature of the wrapper, the system properties you'll try to set with the classic `-Dkey=value` construct will be applied on the wrapper itself. They won't be set on the Mule instance that's wrapped. The solution is to use a specific construct such as this one: `-M-Dkey=value`. For example, the following code will set the `key=value` system property on the Mule instance:

```
mule [-app appName] -M-Dkey=value
```

The Java Service Wrapper, and therefore Mule, has a configuration file in `conf/wrapper.conf` that contains the information necessary to launch a JVM instance and the different wrapper configuration properties. To learn more about the contents of this file, you can refer to the Tanuki Software Wrapper site (<http://wrapper.tanukisoftware.com/doc/english/properties.html>). Using an approach similar to the one you learned before, you can pass configuration properties to the wrapper by using the `-Wkey=value` construct. The key/value pair passed is equivalent to appending a line at the end of the `conf/wrapper.conf` file, following the same rules and semantics:

```
mule [-app appName] -W-Dkey=value
```

Figure 8.2 presents the structure created after the installation of Mule’s complete distribution. The top-level structure presents no particular surprise. For example, bin is where to go to find the control scripts for the different operating systems, as expected. We’ve expanded the lib subdirectory so you can see the structure in which the different types of libraries Mule depends on are stored.

There are three notable directories here. The *user* directory is the first one. This is where you can drop any extra library you’d like Mule to be able to use. Common libraries that could be deployed here are JDBC and JMS drivers. The lib/user directory is also the destination for patches, as its content is loaded before what’s in the *mule* directory. Last but not least we have the *apps* directory, which is where Mule applications should be deployed. We’ll study how to deploy to this directory in section 8.1.2.³

The other outstanding directory is *shared*. When sharing common libraries between applications, it’s not uncommon to find that you need to have two different versions of the same library for two different groups of applications. You can have two different *domains* and place the common libraries in the corresponding shared library directory. Each domain will correspond to a subdirectory of shared named with the domain name. The reasoning behind this behavior is described shortly in the sidebar, “The hierarchical classloader.” You’ll learn how to assign the domain of an application in the next section.

In a production environment, there’s a slight drawback to dropping your own libraries in the user directory. By doing so, you make it harder for your operation team to switch between Mule versions, as they’ll have to do some tweaking of the

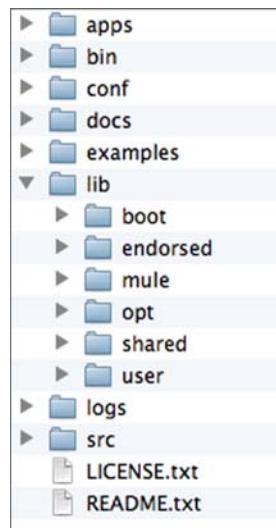


Figure 8.2 Directory structure of Mule standalone server

lib/user and dependency management

If you’re placing common dependencies in lib/user, those dependencies will be automatically present on the deployed application’s classpath. Those libraries therefore should be considered as provided in the dependency management system of your Mule applications. For instance, if you place the hornetq-jms-client library in lib/user and you’re using Maven, you should mark the scope as follows:

```

<dependency>
    <groupId>org.hornetq</groupId>
    <artifactId>hornetq-jms-client</artifactId>
    <version>2.2.21.Final</version>
    <scope>provided</scope>
</dependency>
    
```

³ Mule needs to be restarted to pick up any JAR that you would drop in this directory.

The hierarchical classloader

Mule has a hierarchy of classloaders, similar to what you can find in most application containers, to allow different library domains and each of the deployed Mule applications to have access to different classes and resources.

Classloaders are responsible for locating class containers and libraries, gathering their contents, and loading the contained classes. Classloaders are distributed in a hierarchical tree. When a class is requested to a classloader, it usually will delegate to its parent. This'll happen recursively until the root classloader is reached. A classloader will only try to load a class locally when its corresponding parent can't find the class.

At Mule's startup, a hierarchy of classloaders is created with the distribution in figure 8.3. The Java bootstrap classloader will load the classes present in the JDK. Its child, the Mule system classloader, is responsible for the libraries placed in lib/mule and lib/opt.

Continuing down, the shared domain classloader will load the libraries present in each of the domains used in the applications; in the case shown in the figure, lib/shared/default and lib/shared/custom. Finally the application classloader will load the libraries and plugins bundled with each application.

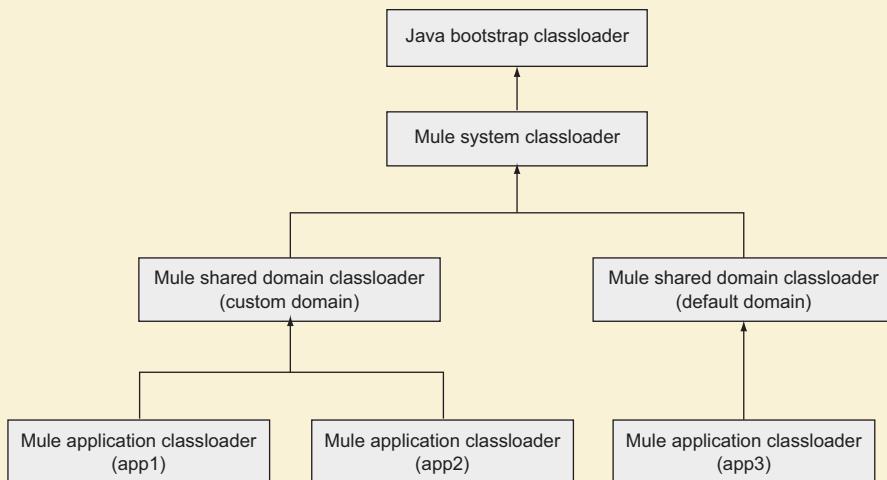


Figure 8.3 The Mule's hierarchy of classloaders

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.

Table 8.1 recaps the pros and cons of the standalone server deployment option.

Table 8.1 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.	Must take care of file and directories organization.
Well suited for ESB-style deployment.	Not suited for a “Mule as a messaging framework” approach.
Direct support for patch installation.	Resources such as JDBC connection pools aren't shared across applications.

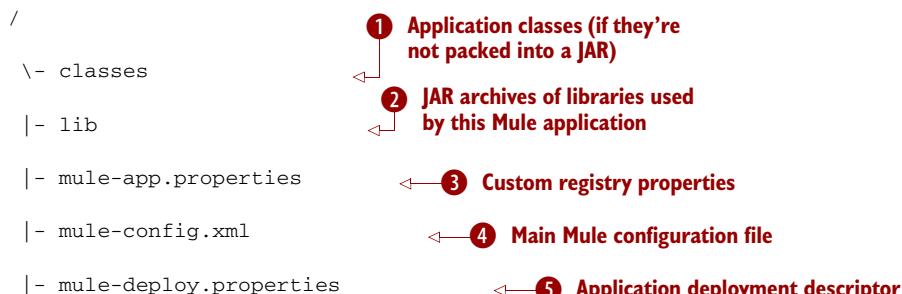
Now you know the structure of a Mule standalone server. Let's now look at how to package a Mule application to be deployed inside one.

8.1.1 Packaging a standalone Mule app

In chapter 1, we went through the creation process of a Mule project using Maven. The last thing we did was export the application as a ZIP file. Now let's focus on the format of that Mule application deployment unit.

A Mule application can be deployed either as a ZIP file or as a regular directory that will contain exactly the same contents a Mule app ZIP file would contain, but exploded. You're probably used to this behavior if you've ever deployed artifacts to containers such as Tomcat or JBoss. Let's visit the structure of a Mule application deployment unit in the following listing.

Listing 8.1 Directory structure of an application that uses the Mule standalone server



Typically the classes directory ① will contain resources of the application that aren't packaged as part of the JAR files, such as secondary Mule config files, logging configuration files, and application properties files. The lib directory ② instead contains the JAR libraries used by the Mule application. The mule-config.xml file ④ is the principal configuration file and entry point of the Mule application; usually from here you'll include other Mule configuration files present in the classes directory.

The mule-app.properties file ③ is a standard properties file used to modify Mule's registry. We won't work with the registry until section 12.1; at the moment you need to understand that the pairs of values present in the properties file will be inserted into the Mule registry.

Finally, the mule-deploy.properties file ⑤, also known as Mule's deployment descriptor, is a standard properties file used to control how a Mule application should be deployed, configuring, for instance, whether hot redeployment should be enabled or which packages should be scanned for annotations. Let's review the more relevant configuration properties it supports:

- domain—Default value is default. Domain is used to load shared libraries in \$MULE_HOME/lib/shared/domain, where domain is the value passed into the configuration property.
- config.resources—Default value is mule-config.xml. There's a comma-separated list of configuration files for this application. It's common to use this configuration property to avoid the use of imports in mule-config.xml.
- redeployment.enabled—Default value is true. When activated, the mule-config.xml file or the files passed into config.resources will be monitored for changes to trigger a redeployment.
- encoding—Default value is UTF-8. It sets the default encoding for this Mule application. This is important for string-related transformers, as you learned in chapter 4, in our discussion of transformers and encoding.
- scan.packages—Empty by default. Before Mule 3.1, the whole classpath was scanned for annotations. If you use annotations that need to be scanned, you can use this comma-separated list of packages.

Now that you understand the format of the Mule application deployment unit, we can jump to the creation of one unit. The building process will depend on how you're building your application. To build a unit using a Maven project created with a Mule Maven archetype, you need to execute the following command in the directory containing the Project Object Model (pom.xml):

```
mvn package
```

This'll trigger the packaging of the application into a ZIP file. You'll find this file in the target directory with the name of the project as the filename and zip as the extension:

```
[INFO] Building zip: /Users/mia/product-registration-application  
/target/productregistrationapplication-1.0-SNAPSHOT.zip  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 15.866s  
[INFO] Finished at: Sat Dec 29 13:51:08 CET 2012  
[INFO] Final Memory: 8M/81M  
[INFO] -----
```

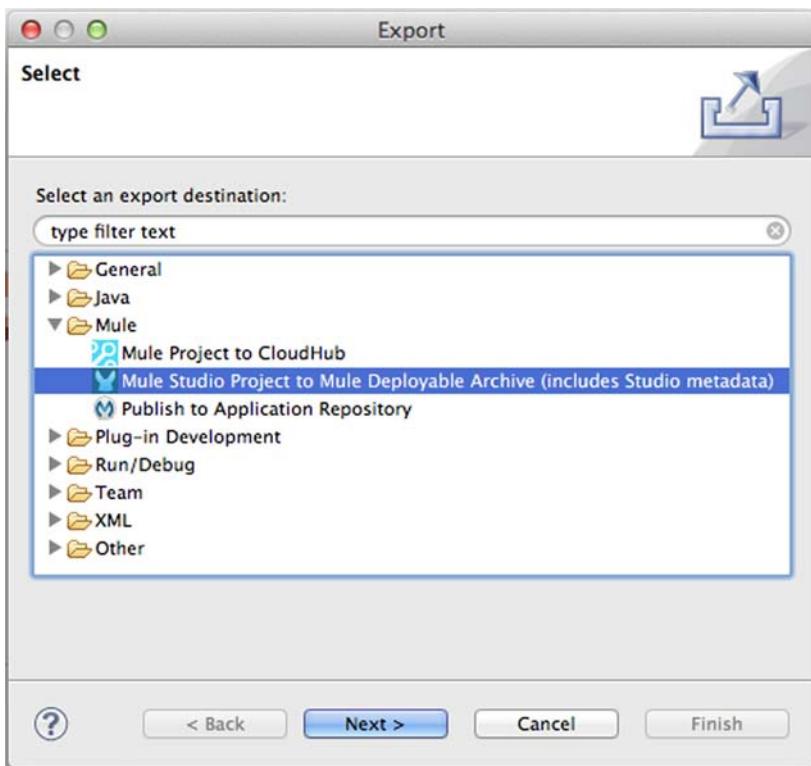


Figure 8.4 Export dialog of Mule Studio

If you've chosen to use the Mule Studio to drive your application build process, to create a Mule application deployment unit for your project you should, from the Mule perspective, right-click the project name in the package explorer pane, and then select Export.... From the export destination dialog, select Mule and then Mule Studio Project to Mule Deployable Archive (Includes Studio Metadata), as shown in figure 8.4.

Click the Next button and you'll find a dialog requesting the export file destination, in which you should type or browse to the desired destination file path, as shown in figure 8.5. Once you click Finish, the file will be created.

You've learned how to create a Mule application deployment unit; now it's time to learn how to deploy your application in a Mule standalone server.

8.1.2 **Deploying applications to Mule**

Mule applications can be deployed by copying either the zipped or exploded deployment unit into the apps directory of the Mule distribution. This mechanism is similar to the deployment method of the most popular application containers that have their own webapps or apps directory.

At Mule's startup, if the application was zipped, it'll be uncompressed into a directory with the same name as the ZIP file minus the extension. The original ZIP file will

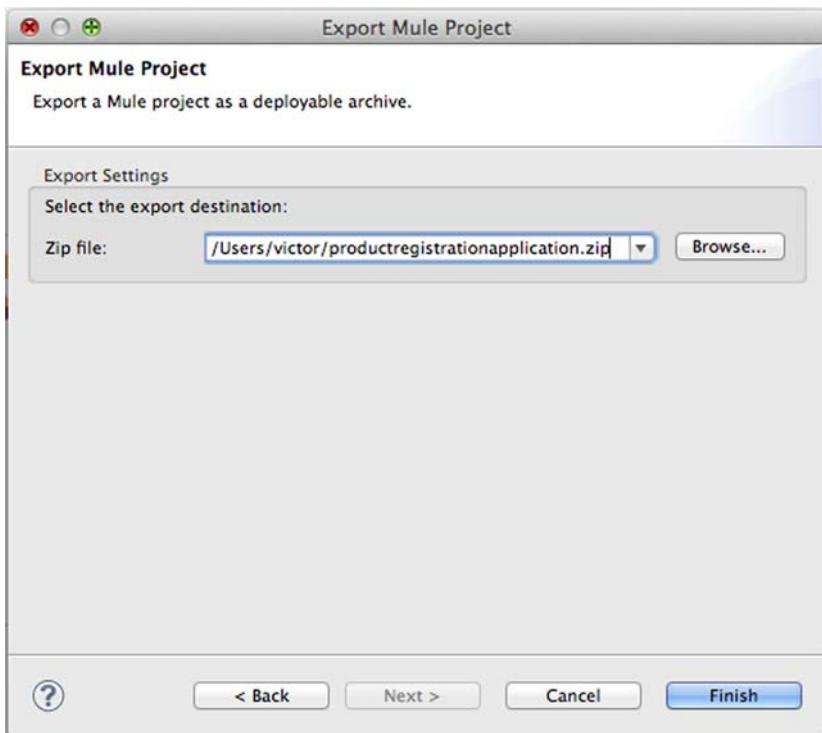


Figure 8.5 Export Mule Project dialog of Mule Studio

also be removed. Again, this is similar to what other containers do; for instance, Tomcat behaves almost identically with its WAR files.

Right before starting the deployment, Mule will create an *anchor* file. This file is monitored by Mule to trigger a clean shutdown when deleted. To undeploy an application, you should remove this anchor file. After a few seconds, Mule will undeploy the application and *completely remove the application directory*. This method will prevent any interference with the hot deployment system and any potential problems with file locking or missing resources.

Deployment and logging

A correct Mule application deployment will follow these steps:

- 1 If the Mule application is a ZIP file, uncompress the application deployment unit into a directory named the same way as the file minus the extension; that is, the productregistrationapplication.zip file will become the productregistrationapplication directory.
- 2 If a Mule application is a ZIP file, delete the ZIP file.
- 3 Create the anchor file, appending -anchor.txt to the directory name.
- 4 Run the application.

(continued)

In the next section, you'll learn how to configure the logs. But at this point it's useful to note that only the fourth point is logged to your application log file, that is, logs/productregistrationapplication.log, whereas all four are logged to the general Mule log file, logs/mule.log. If there's some kind of problem before the application starts, you might need to take a look at the general log file instead of the application-specific one.

Mule applications can be hot deployed. This means that any deployment or redeployment can be made with the server running instead of requiring a container restart. Thanks to the hot deployment feature of Mule, you can do these things:

- Automatically deploy exploded or ZIP-compressed Mule applications placed in the apps directory
- Redeploy applications when an updated ZIP-compressed Mule application is deployed in the apps directory
- Redeploy exploded applications when the mule-config.xml file or the first entry of the list of resources specified in the mule-deploy.properties file as *config.resources* is updated

Therefore, the procedure to hot deploy applications to Mule is the same as normal deployment, but with Mule running. In addition, Mule will monitor the mule-config.xml file and the first config.resources entry in the mule-deploy.properties file for changes, and will redeploy the application if there are any modifications.

Once you're able to deploy your application, you'll need to properly configure the logging so that you can monitor the proper functioning of your application and the Mule standalone server itself. Let's configure the logs in the next section.

8.1.3 Configuring logs

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, however, is by logging them. Mule uses the SLF4J logging facade (<http://www.slf4j.org/>) to allow you to plug and play logging facilities into Mule. By default, Mule ships with the popular log4j logging library, which SLF4J will use without any intervention on your part. You'll see later how you can change this behavior to allow Mule to use other logging implementations such as java.util.logging. Let's start off by looking at how logging works in a freshly installed, standalone Mule instance.

Mule uses log4j as its default logging implementation. Log4j is a robust logging facility that is commonly used in many Java applications. Full documentation for using log4j is available on the project's website at <http://logging.apache.org/log4j/>. Mule provides a default log4j configuration in the \$MULE_HOME/conf/log4j.properties file. The following listing shows the default log4j.properties file.

Listing 8.2 The default log4j.properties file

```

# Default log level
log4j.rootCategory=INFO, console
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

#####
# 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

# Reduce startup noise
log4j.logger.org.springframework.beans.factory=WARN

# Mule classes
log4j.logger.org.mule=INFO

# Your custom classes
log4j.logger.com.mycompany=DEBUG

```

① Specify default log level

② Specify logging format

③ Specify logging level of the org.apache package

④ Specify logging level of org.mule

⑤ Specify logging level for your packages

If you've ever worked with log4j, 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 at ①, which is logging messages of level INFO to the console. The format of the log output is specified at 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 at 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 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 on. For instance, in order to set DEBUG logging for Prancing Donkey's custom classes, you'd change ⑤ to this:

```
log4j.logger.com.prancingdonkey=DEBUG
```

By default, Mule will write one general log file, mule.log, with all the container-level information, and one more file per application named with the application name plus the log extension. These files are 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 MB worth of data to the log files before automatically rotating them. It will 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.

Getting diagnostics from log4j

You occasionally need to gather debugging information from log4j itself. This can be accomplished by setting the `log4j.debug` property. Do this by appending the string `-M-Dlog4j.debug` to the end of the command to launch Mule. Here's what it looks like:

```
mule -M-Dlog4j.debug
```

Now that you've seen how to configure log4j and SLF4J, you've reached the point at which you can completely deploy and review the execution of your Mule applications. A Mule standalone server usually contains more than one Mule application, and it's a matter of time until you need some kind of inter-application communication. Let's take a look at how you can perform this in the next section.

USING A DIFFERENT LOGGER 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. Download the SLF4J implementation your version of Mule is using, and place the appropriate SLF4J bridge in `$MULE_HOME/lib/user`. On your next restart, Mule should be logging using the new implementation.

8.1.4 Inter-application communication with Mule

When deploying multiple Mule applications to a single Mule standalone server, it's a common pitfall to believe that Mule applications are somehow connected because they reside on the same server. The reality is the opposite of the common perception; Mule does its best to keep the applications isolated as much as possible, such as by using a different classloader for each application (as you learned earlier).

Thanks to this behavior, Mule applications hide their internals and are forced to communicate through well-defined transports. Among other benefits, this lets Mule applications be deployed, undeployed, started, or stopped independently without affecting other apps; they can also be used in a cluster or on a single server without making any changes related to interactions with other apps. It also allows for different versions of dependencies without inducing classpath hell (see http://en.wikipedia.org/wiki/Classloader#JAR_hell).

VM endpoints don't communicate between applications

The "VM" part of VM transport mistakenly suggests *virtual machine*. Given that many Mule applications can run in the same Mule standalone server, and therefore in the same Java Virtual Machine, you may think that you can use VM endpoints to communicate with applications in the same server. But you would be wrong.

The VM transport is an in-memory transport, and therefore each application has a different set of VM endpoints that are completely local to that application.

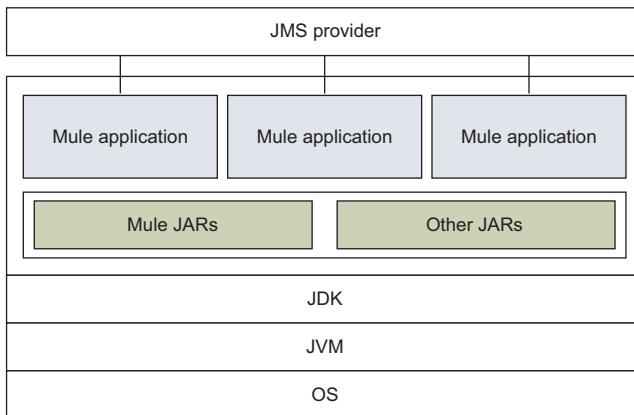


Figure 8.6 Mule applications communicating using JMS

If Mule works hard to make your different apps as isolated as possible, how should you implement inter-application communication? You'll have to treat these communications as if they were communications between applications running on different servers. You could use any Mule transport suitable for communication with external services, as you learned in chapter 3.

Some of the more common approaches to communicating between Mule applications include using an external message broker such as a JMS server, as you can see in figure 8.6, and using protocol formats such as XML, JSON, or the Google protocol buffers. Let's see how you can use these approaches for your purposes.

You may remember we created a shipping cost calculator in section 6.1.7. We exposed it using a VM endpoint. The VM transport isn't suitable to perform inter-application communications, but you want to be able to call this service from other applications. In order to do so, you'll expose it in a JMS queue using XML. You can find the new shipping cost calculator flow in the next listing.

Listing 8.3 Flow exporting a service for inter-application communication

```
<flow name="shipping-cost-service">
    <jms:inbound-endpoint queue="calculateShippingCost"
        exchange-pattern="request-response" />

    <mulexml:xml-to-object-transformer />
    <component>
        <singleton-object
            class="com.prancingdonkey.component.ShippingCostCalculator" />
    </component>

    <mulexml:object-to-xml-transformer />
</flow>
```

① JMS endpoint for external communication

② Unmarshaling of the XML request message payload into an object

③ Marshaling of the response object payload into XML

You've moved from the service in section 6.1.7 that accepts POJOs to the new service in listing 8.3 that works with XML and JMS. The outstanding points in this evolution are therefore the inclusion of XStream marshalers (② and ③) and the use of a JMS

inbound endpoint instead of the VM inbound endpoint ①. The JMS endpoints won't work with object references because the messages are intended to be sent to different virtual machines. Therefore the marshalers allow you to communicate using XML messages instead of Java object references.

At this point, you can use the calculate shipping cost service from a different flow. You know you have to send a message to the JMS queue defined at ①, using the XML format defined by a shared class. In the following listing, let's call the service from a different Mule application to retrieve the shipping cost of a purchase.

Listing 8.4 Flow requesting information in an inter-application communication

```
<mulexml:object-to-xml-transformer />          1 Marshaling of request  

<jms:outbound-endpoint queue="calculateShippingCost"    object payload into XML  

    exchange-pattern="request-response" />          2 Communicate with shipping  

                                                cost calculator application  

<mulexml:xml-to-object-transformer />          3 Unmarshaling of XML response  

                                                message payload into an object
```

You start from a payload with an object that represents a shipping cost calculator service request. In practice, this is an instance of a class that can be marshaled using XStream, as you may recall from chapter 4. This object is marshaled into XML before you send it ①. Then the request is performed using a JMS outbound endpoint ②. This will return an XML response. You'll want to transform this response back into an object to continue your work. You do this transformation at ③, doing the inverse of what you did before, transforming an XML payload into an object.

In this section, you've learned how to use the techniques you already visited in chapters 3 and 4, with a special emphasis on how to use them for inter-application communication. You also learned you shouldn't use local transports like the VM transport for this kind of communication. Now let's see how a Mule application can also be a web application container using Jetty.

8.1.5 Embedding web applications in Mule

We've studied how to deploy Mule applications to a Mule standalone server. Now we'll go deeper: imagine an M.C. Escher lithograph in which you follow a path. After some time, it seems that you're back at the starting point. That's how you may feel knowing that not only can Mule applications be deployed in a web application container, as you'll learn in section 8.2, but they can also act as containers of Java EE web applications through the use of Jetty. You'll start by learning how to configure an embedded Jetty server in Mule, as illustrated in figure 8.7.

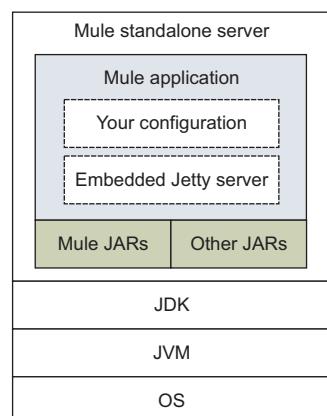


Figure 8.7 Conceptual deployment model of Mule standalone server with Jetty embedded in an app

Mule's Jetty connector offers a lightweight but full-fledged web application container for Mule. Why is this useful? Sometimes you may need to serve small web applications to use your AJAX endpoints, or perhaps you have to serve some static content and you don't want more complexity in your architecture. If you need it for any reason, Mule is ready to act as a web application container.

Mule provides two different namespaces for Jetty, the plain HTTP one and the HTTPS one. Let's start by configuring an HTTP-based embedded Jetty server in the next listing.

Listing 8.5 Configuring an embedded Jetty server

```
<jetty:connector name="jettyConnector">
    <jetty:webapps
        directory="${app.home}/webapps" port="${jettyPort}" />
</jetty:connector>
```



Defines the webapp directory and the port to be used

In the previous example, you configure an embedded Jetty server using the Jetty connector. It'll serve the web applications stored in the path specified in the attribute `directory` using the TCP port passed with the attribute `port`.

This'll expose a web application using HTTP, an unsafe protocol in terms of security. To use the Security Sockets Layer (SSL) over HTTP (and therefore obtain HTTPS), you should use a different namespace; let's do that in the next listing.

Listing 8.6 Configuring an embedded Jetty SSL server

```
<jettyssl:connector name="jettySslConnector">
    <jettyssl:tls-key-store path="keystore.jks"
        keyPassword="password" storePassword="password" />
    <jettyssl:webapps directory="${app.home}/webapps"
        port="${jettySslPort}" />
</jettyssl:connector>
```

You use a different namespace and pass the new element `tls-key-store`. This element adds a key store to Jetty to be able to communicate using HTTPS. The transport security configuration of the Jetty SSL namespace is intentionally similar to the configuration HTTPS. You'll learn how to configure an HTTPS server in chapter 10; all the elements you'll use for SSL on HTTP can be used to configure SSL for Jetty.

Jetty is known for being lightweight, but that doesn't mean it's not powerful. You can configure some of the basic options of Jetty using Mule XML elements. If you need advanced features, you can pass a Jetty configuration file to the connector, as shown next.

Listing 8.7 Configuring an embedded Jetty using a config file

```
<jetty:connector name="jettyConnectorWithConfigFile"
    configFile="${app.home}/jetty-conf.xml" />
```

Given that Mule itself doesn't necessarily have to be a container and that it can be deployed to other containers, let's see now how you can accomplish the opposite of what you just learned: how to embed Mule in web applications to be deployed inside web containers.

8.2 **Deploying Mule to a web container**

For the same reasons stated at the beginning of the previous section, you might be interested in embedding Mule in your web application. This approach used to be popular in the Mule 2.x days, but now it's superseded by the deployment to a standalone server, as we discussed in section 8.1.

Mule provides all that you need to hook it to your favorite servlet container. Why is this desirable? 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 8.8, you need to make sure the necessary libraries are packaged in your WAR file. If you use Maven, this packaging will be done automatically, including for 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 to your web application's lifecycle (which itself is bound to the web container's one). The following demonstrates the entries you would 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 is stopped. Notice how the configuration of the Mule context listener is done by using context-wide initialization parameters.

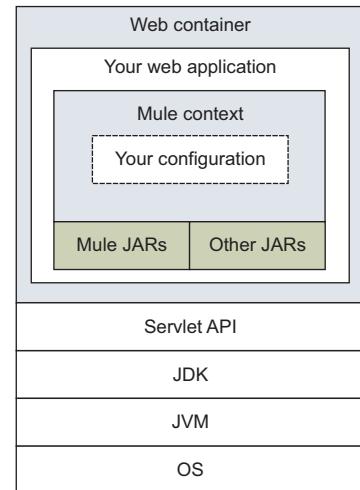


Figure 8.8 Conceptual deployment model of Mule embedded in a web application

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. Because the context listener took care of starting Mule, you have no reference to the context, unlike the case in which you bootstrap Mule yourself. This is why the Mule client is instantiated with a context retrieved from the `servletContext`:

```
muleContext = (MuleContext) getServletContext()
    .getAttribute(MuleProperties.MULE_CONTEXT_PROPERTY);

client = muleContext.getClient();
```

There's another great benefit to 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 rely on either the stock HTTP transport or the Jetty one. But this doesn't need to 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 use these if you use the servlet transport for your inbound endpoint.

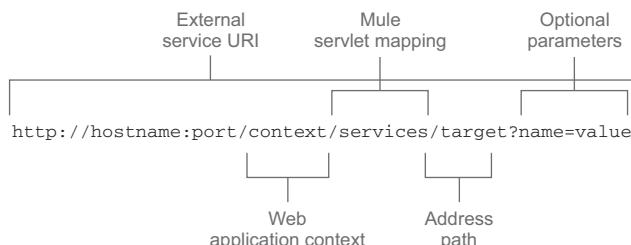
The servlet transport only works in coordination with a servlet configured in your `web.xml`. This servlet takes care of receiving the messages and sending them to the inbound endpoint that can accept them. Here's an example of how to configure this servlet:

```
<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://target">
```

A Mule service using such a configuration will be accessible to the outside world with a URI following this pattern:



The payload of the message that will be received in the service will depend on the HTTP method used by the caller.

SERVING REST INSIDE A WEB CONTAINER Mule ships with another servlet that's more oriented toward REST-style services. Named `org.mule.transport.servlet.MuleRESTReceiverServlet`, this other servlet can be used in lieu of or conjointly with the standard receiver servlet. Refer to the Mule servlet transport documentation for more information on this.

Table 8.2 shows a list of pros and cons for the web application–embedded deployment option of Mule.

Table 8.2 Pros and cons for 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 lifestyle events	No direct support for scripted configurations
Familiar deployment platform for operations	Possible pesky classloading issues on some web containers
Resources like JDBC connection pools can be shared across applications	

Embedding Mule in a web application is a popular and powerful way to deploy Mule. Should you need easier integration with the cloud, particularly for Software-as-a-Service (SaaS) integration, the next deployment strategy that we'll look at is for you.

8.3 *Deploying applications to CloudHub*

In July 2012, MuleSoft rebranded its integration platform as a service from Mule iON to CloudHub. Since the announcement of the first release, CloudHub has gained more and more momentum in the cloud integration industry. CloudHub offers a platform to deploy Mule applications that can use features like real-time message visibility, message replay, root-cause analysis, and multitenancy. For a full description and more details about CloudHub, you can refer to the CloudHub site (www.cloudhub.io).

Although this book is primarily focused on the community edition of Mule, you'll learn how to deploy packaged Mule application deployment units to CloudHub, as it represents the natural cloud platform for Mule. To start the process, sign up in the CloudHub site (www.cloudhub.io) as a developer. Once the registration is completed, you should see a console similar to that in figure 8.9.

At this point, you can add your application. Click Add to get a dialog box, as shown in figure 8.10.

At this point, upload the Mule application deployment unit by clicking on the file selector button next to the Application field. You should select a unique subdomain

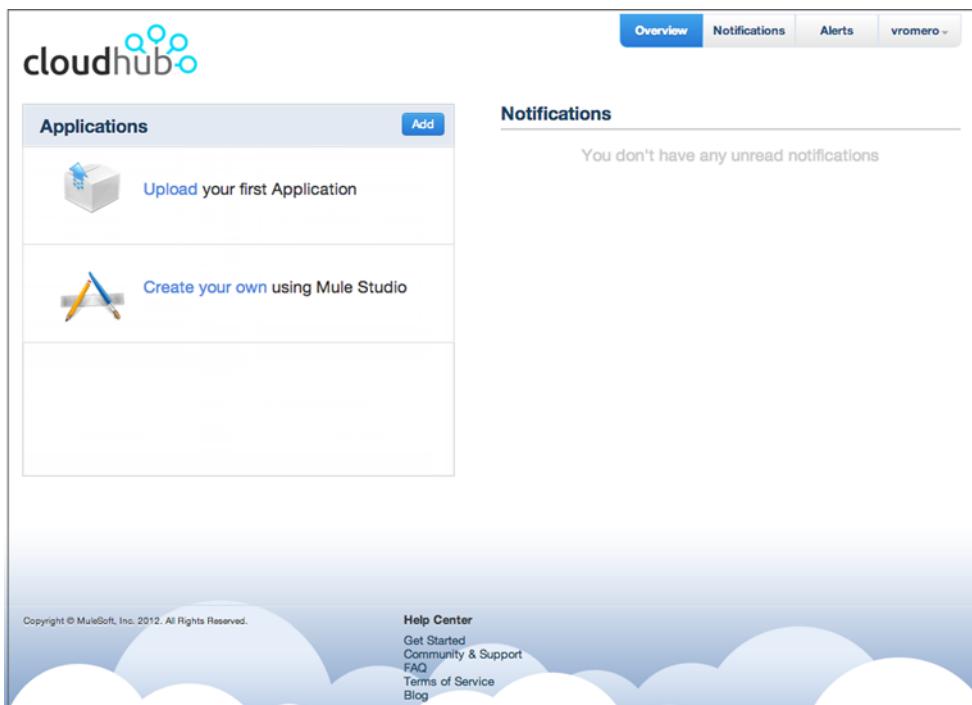


Figure 8.9 CloudHub management console

name for your application. If your application exposes any endpoint, the indicated full domain name will represent the host name you'll use to reach the endpoint. The drop-down next to the Version field will let you choose the Mule version that'll be used to run this Mule application.

The screenshot shows the "New Application" dialog box. It has fields for "Domain" (containing ".cloudhub.io"), "Application" (with a note "No file has been loaded" and a "Upload" button), and "Version" (set to "CloudHub Mule Runtime (Oct 20)"). On the right side, there's a "Related FAQs" section with links to "What are applications?", "How can I build them?", "What are workers?", and "What are environments variables?". At the bottom left is a "Advanced Details" link, and at the bottom right is a large blue "Create" button.

Figure 8.10 CloudHub's New Application dialog

Small adjustments

If you intend to deploy your application in CloudHub, you may need to make some small adjustments to your software.

For instance, the servlet transport is not allowed. Instead you should use the HTTP transport, and when using it the variables \${http.port} and \${https.port} will define the HTTP and HTTPS ports you've assigned to your application. The outside world will always see your application running at yoursdomain.cloudhub.io at port 80 for HTTP and 443 for HTTPS, but internally you should use those variables.

For further information about potential adjustments in your application, visit MuleSoft's CloudHub documentation at <http://mng.bz/yjyG>.

Once you've clicked the Add Application button and the Mule Application deployment unit is uploaded, the application deployment will automatically start, and your browser will be pointed to the log view of the CloudHub console. Once you get a log line showing Successfully deployed [your application name], you'll know your application's been successfully deployed. Table 8.3 shows the pros and cons for CloudHub deployment.

Table 8.3 Pros and cons for CloudHub deployment

Pros	Cons
Extremely easy management	More expensive than the community edition, but probably less expensive than the Enterprise Edition, which has a fee associated
Enterprise Edition features	
Great scalability and availability	
Good for integrating cloud services	Not suitable for applications with special legal requirements

EASILY TEST MULE ENTERPRISE FEATURES IN CLOUDHUB CloudHub features some exciting capabilities only present in the Enterprise Edition of Mule, such as clustering support and Insight, a useful tool to, among other things, analyze the root cause of failures.

CloudHub represents the natural cloud evolution of Mule. It's not only a great platform for deploying Mule applications, but also an excellent playground for testing features. Now let's leave behind all kinds of application containers to look at how you can use Mule in an existing application.

8.4 Embedding Mule into an existing application

If you've built applications that communicate with the outside world, you've 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 had the need to compose or

orchestrate calls to different remote services, things started to get a little 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 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 8.11, Mule can be embedded in a standard Java application. This makes sense if the application’s not destined to be run as a background service; in that case, you’d be better off using the standalone server deployment we talked about in section 8.1.

In the embedded mode, it’s up to you to put on the classpath of your application all the libraries that will be needed by Mule and the underlying transports you’ll need. Because Mule’s built with Maven, you can benefit from its clean and controlled dependency management system by using Maven for your own project.⁴

Bootstrapping Mule from your own code is easy, as illustrated in the following code snippet that 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 = muleContext.getClient();
```

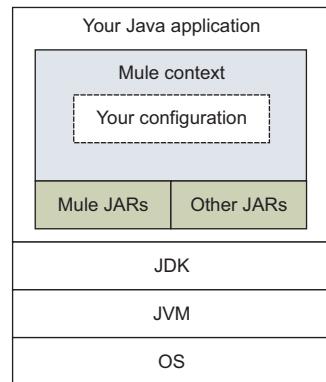


Figure 8.11 Conceptual deployment model of Mule embedded in a standard Java application

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).

Mule über Spring?

The previous example shows how Mule leads the dance relative to Spring; it's Mule that takes the responsibility to load and bootstrap the bean factory from which your beans will be managed. But what if you have an existing Spring application with its own context (or hierarchy thereof) and you want Mule to be running within this environment and have access to its beans? In that case, you'll have to pass your existing Spring context to the Mule configuration builder so it'll use it as its parent. This'll allow Mule to use beans managed in a parent context as its service components:

```
SpringXmlConfigurationBuilder("my-config.xml");
builder.setParentContext(parentContext);
builder.configure(muleContext);
```

Note that the Mule client supports other construction parameters that we'll discuss in section 12.2.

Table 8.4 summarizes the pros and cons for embedding Mule in a standard Java application.

Table 8.4 Pros and cons for embedding Mule in a standard Java application

Pros	Cons
Flexibility to deploy only 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	The high-availability Enterprise Edition feature isn't available in embedded mode

You now know how to make your standalone Java applications use Mule to communicate and integrate with other applications. Let's now look at how you can deploy Mule to achieve high availability.

8.5 Deploying Mule for high availability

Being able to ensure business continuity is one of the main goals of any IT department. Your Mule-driven projects will not 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 you can take Mule instances down at any time without 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 downtime. In this scenario, each of

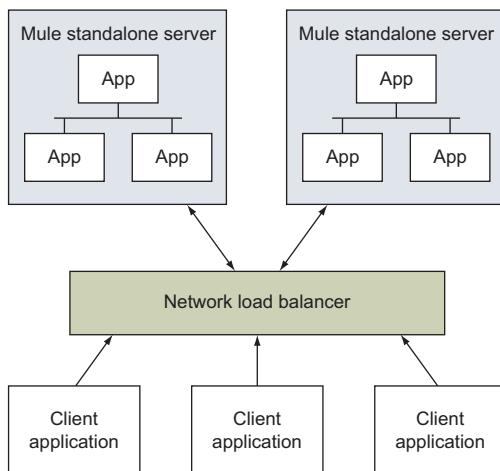


Figure 8.12 A network load balancer provides high availability to Mule instances

the Mule instances behind the indirection layer is taken down and brought 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 (see figure 8.12). Obviously, this is only an option if the protocol used to reach the Mule instances can be load-balanced (for example, HTTP). With a network load balancer in place, one Mule instance can be taken down, for example, for an upgrade, whereas 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 in the pool and have it handle part of the load.

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 the queue. Only one Mule instance will pick up a message that's been published in the queue. If one instance goes down, the other one will take care of picking up messages if your JMS middleware supports the competing consumers pattern (see www.eapatterns.com/CompetingConsumers.html). Moreover, if messages aren't processed fast enough, you can easily throw in an extra Mule instance to pick up part of the load. This implies that you're running a highly available JMS provider that will always be up and available for client applications. The canonical ESB topology, represented in figure 8.13, can therefore be easily evolved into a highly available one.

If your Mule instance doesn't contain any kind of session state, then it doesn't matter where the load balancer will dispatch a particular request, as all your Mule instances are equal as far as incoming requests are concerned. But, on the other hand,

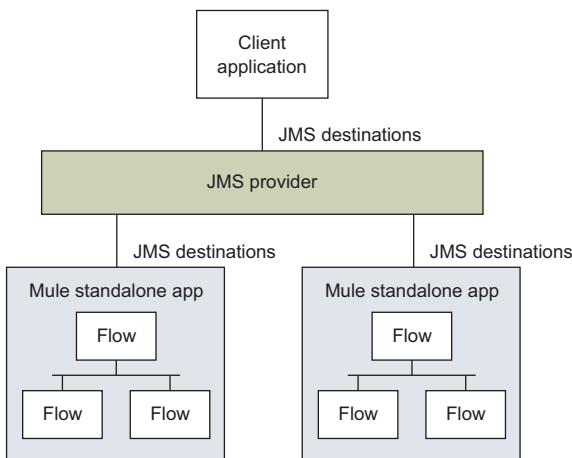


Figure 8.13 The canonical ESB deployment topology of Mule instances relies on a central JMS provider.

if your Mule instance carries any sort of state (for example, idempotency, aggregators, resequencers, or components with their own 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.⁵ This is usually achieved either with a shared database or with clustering software, depending on what needs to be shared and performance constraints.

Note that as of this writing, there's no officially supported clustering mechanism for the Mule community edition; you can work around some of the clustering limitations of the community edition using the object stores, as you'll learn in the next section. The Enterprise Edition, however, has full-fledged support for clustering.

Using the Mule Enterprise Edition, all Mule features become cluster aware in a completely transparent fashion. A cluster of Mule Enterprise Edition servers will create a distributed shared memory, as you can see in figure 8.14, that'll contain all the necessary shared state and coordination systems to cluster a Mule application without a specific cluster design in the

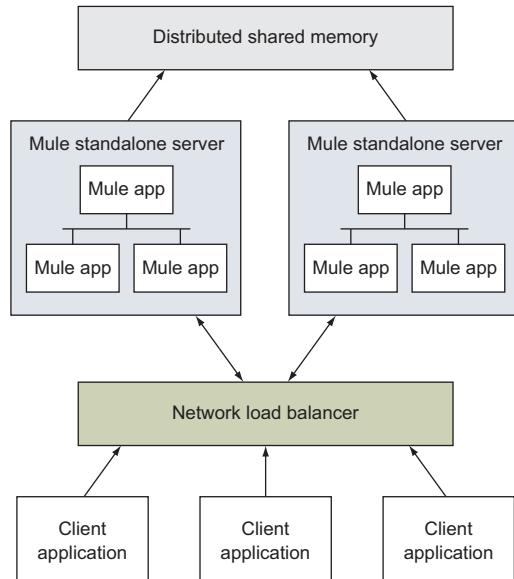


Figure 8.14 Clustered and load-balanced Mule EE instances

⁵ 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 it wouldn't guarantee a graceful failover in case of a crash.

Mule application. To learn more about Mule Enterprise Edition, the key differences between it and the community edition, and how it can help you with easier clusterization, you can visit the Mule Enterprise Edition site (www.mulesoft.com/mule-esb-enterprise).

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 as well.

8.5.1 High availability via 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) ones. In the happy world of Mule, because the vast majority of transports don't support the notion of transactions, 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 are stored on the filesystem when they move between the different services of a Mule instance, as shown in figure 8.15. In case of a 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 transactional consumption of messages.

The main drawback of VM-persisted queues⁶ is that you need to restart a dead instance in order to resume processing. This can conflict with high-availability requirements you may have. When this is the case, the best option is to rely on an external, highly available JMS provider and use dedicated queues for all intra-instance communications. This is illustrated in figure 8.16.

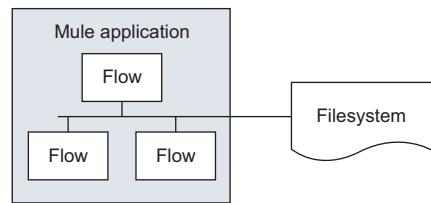


Figure 8.15 Simple filesystem-based persisted VM queues are standard with Mule.

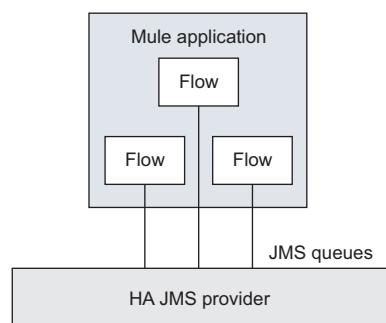


Figure 8.16 An HA JMS provider can host queues for all communications within a Mule application.

⁶ Other than forcing you to move around serializable payloads only.

CLUSTERING INTERNAL STATE USING OBJECT STORES

Using the aforementioned mechanisms, you can reach a certain level of fault tolerance with regard to message delivery. But you must be aware that certain moving parts of Mule have an internal state that's necessary to process messages correctly. This state is persisted with an internal framework that's called the *object store*. The object stores of your application play a key role in the application design, as they contain data that probably should be shared in a clustered application.

This internal state can exist in different moving parts of Mule, such as in the idempotent receiver, which must store the messages it's already received. You may recall using an object store in section 5.2.5, when you were introduced to the idempotent filters. There you learned how to configure an idempotent filter to discard duplicated orders. You can see how it interacts with an object store in figure 8.17.

The community version of Mule offers a set of object stores:

- **in-memory-store**—A nonclustered object store that stores the contents in local memory
- **simple-text-file-store**—Stores String objects by key to a text file; only suitable for storing string key/values
- **custom-object-store**—Used to refer to custom implementations of the object store; see how to develop your own object stores in section 12.3.4
- **managed-store**—Gets an “object store inside an object store” by getting a partition of a `ListableObjectStore` retrieved from the Mule registry
- **spring-object-store**—Represents a reference to a Spring-managed object store

None of those object stores are inherently clustered. You can use JVM-level clusterization in conjunction with the `in-memory-store` or an NFS shared mount point with the `simple-text-file-store`, but that would be complex at minimum. Therefore, we'll turn our attention to one of the contributed modules of Mule, the Redis connector (see www.mulesoft.org/extensions/redis-connector).

Redis (<http://redis.io/>) is an open source, networked, in-memory, key/value data store with optional durability. Its design renders Redis specially suitable to be used as a

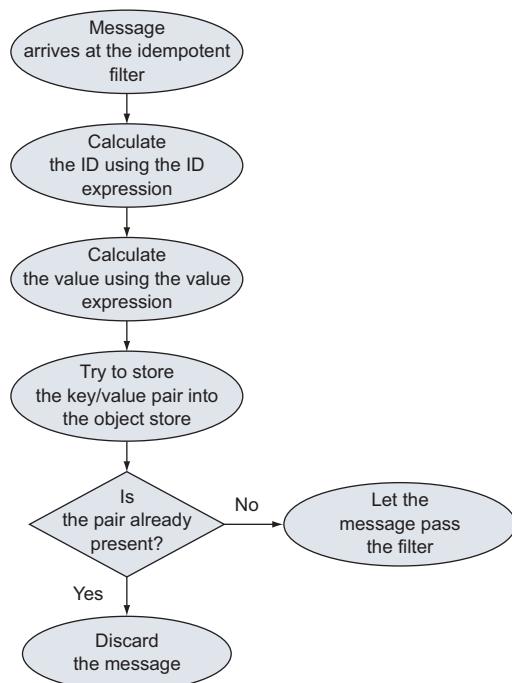


Figure 8.17 Behavior of an idempotent filter

Mule object store, and by using the Redis connector, the usage of Redis as a Mule object store is straightforward.

Given that Redis will represent an external highly available object store, you can include it in the previous design of our application (in figure 8.16) to store the internal state of your Mule moving parts, as you can see in figure 8.18.

The Prancing Donkey commitment to high availability is unavoidable. They've decided to use a highly available configured Redis server to store the internal state of some moving parts of Mule. Start the high-availability implementation by configuring the connectivity with Redis:

```
<redis:config name="localRedis" />
```

This will configure a local nonpassword-protected Redis instance running on the standard port, so it should connect straight to a brand-new Redis installation. The server in production will be placed in a different host and will be strengthened with a password. You'll eventually use the host, port, and password attributes to connect to the server.

Now you're ready to use the Redis connector to store the internal state of your processors. Configure the previously mentioned idempotent filter to use Redis as an object store, as in the next listing.

Listing 8.8 Configuring an idempotent filter to use Redis as an object store

```
<idempotent-message-filter
    idExpression="xpath('/order/id').text">
    <managed-store storeName="localRedis" />
</idempotent-message-filter>
```

① Configure idempotent filter to use an object store

Here you declare an idempotent filter almost identical to the one configured in section 5.2.5. The only exception is found at ①, where you instruct the filter to use the object store with an ID equal to localRedis, which you declared before.

Redis isn't the only option that implements an object store; another available extension is, for instance, the MongoDB connector. The Mule Enterprise Edition supplies a myriad of other options such as JDBC or Spring cache-based object stores. But not every possible solution is covered as a Mule connector or as an Enterprise Edition feature. You'll learn how to implement your own object store in section 12.3.4.

You've seen that shooting for 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.

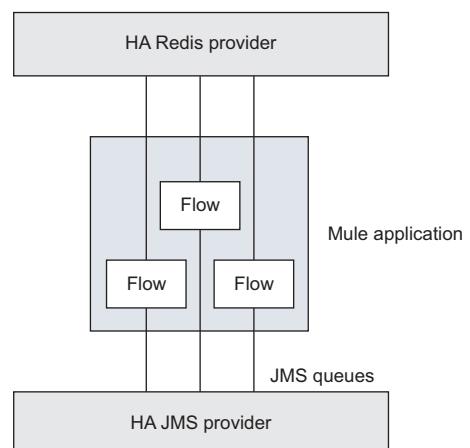


Figure 8.18 An HA JMS provider can host queues for all communications within a Mule instance.

8.6 Summary

We're done with our exploration of Mule deployment topologies. If there's one concept you need to take away from this rather disparate chapter, 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 to look at but, once again, Mule will go where you need it to.

You've also learned how to design your application for high availability. We'll continue this subject, learning how to react to errors and how to use transactions, in the next chapter.



Exception handling and transaction management with Mule

This chapter covers

- Error handling strategies
- Using transaction mechanisms in Mule
- Using transaction demarcation

Dealing with the unexpected is an unfortunate reality when you're 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 events include 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's causing your data to be routed improperly. In any case, it's undesirable for your entire application to fail because of a single unanticipated error.

On the other hand, 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 running total.
- 3 The sum of the prices is calculated and the total cost is communicated to 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, can't total their prices unless they're scanned, 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 total cost of the items, you probably won't hand over 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 above 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.

In the first half of this chapter, we'll examine how Mule implements exception handling. We'll first consider the reconnection strategies to deal with problems related to connections, and then we'll explore exception strategies, in which you'll see how Mule lets you react when errors arise with a message involved.

In the second half, you'll see how to add transactional properties to your Mule services. We'll cover the two major types of transactions supported by the Java and Java EE ecosystems: single- and multiple-resource transactions. We'll start off by examining how single-resource transactions let you operate on a single resource, such as a database or JMS broker, transactionally. You'll then see how you can use transactions across multiple resources. As usual, we'll examine each of these features through the lens of Prancing Donkey. You'll see how Mule's transactional support augments the reliability of the integration projects.

Transaction management and exception handling are related issues. The first defines, among other factors, how a task should be accomplished and retried. Exception handling defines how to react to errors, and this reaction may include retries.

Sometimes both topics can overlap; at other times they can complement each other. In this chapter, you'll learn how to always use them in your favor.

9.1 Dealing with errors

Mule's exception handling recognizes that things in the real world can fail. It lets you plan for and react to errors that would otherwise bring your integration process to a screeching halt. You'll find yourself using Mule's exception-handling ability to identify and troubleshoot failures in your connectors that will be handled with reconnection strategies and in your message processors, in which exception strategies will be used to react accordingly. You'll find that Mule exceptions are grouped into two kinds:

- *System exceptions*—Errors that happened when there was no message involved; for instance, during system startup or when a transport tries to establish a connection
- *Messaging exceptions*—Errors that happened with a message involved; this will likely happen inside a flow, when a message processor fails to accomplish its duty

The default exception strategy for *messaging exceptions* handles errors that occur in a flow (and a message is involved). By default, it will log the exception and Mule will continue execution. You'll learn how to change this behavior in section 9.1.3.

The default exception strategy for *system exceptions* is responsible for handling transport-related exceptions such as SSL errors on an HTTPS endpoint or a connection failure on a JMS endpoint. It will notify the registered listeners and only if the exception is caused by a connection failure will it use a reconnection strategy. You'll learn more about reconnection strategies in the next section. Although reconnection strategies can be configured, the default exception strategy for system exceptions can't be configured.

9.1.1 Using reconnection strategies

It's an unfortunate reality that services, servers, and remote applications are occasionally unavailable. Thankfully, however, these outages tend to be short-lived. Network routing issues, a sysadmin restarting an application, and a server rebooting all represent common scenarios that typically don't take long to recover from. Nevertheless, such failures can have a drastic impact on applications dependent on them. In order to mitigate such failures, Mule provides reconnection strategies to dictate how connectors deal with failed connections.

Let's imagine a scenario in which you're connecting to a legacy system that often goes into maintenance at night or, even worse, fails at random hours. Good news: you can handle this situation with reconnection strategies.

Mule offers both a mechanism to implement custom reconnection strategies, that we'll visit soon, and a couple of ready-to-use reconnection strategies:

- reconnect, with the use of the two attributes (count and frequency), allows configuration of how many times a reconnection should be attempted and the interval there should be between attempts.
- reconnect-forever works like reconnect, but instead of reconnecting a number of times, it will reconnect ad infinitum, waiting the number of milliseconds specified in the frequency attribute between attempts.

Both strategies share an attribute called blocking; if set to true, the Mule application message processing will halt till it's able to reconnect to the external resource. Otherwise, Mule won't wait to have all connectors online to process messages, and in the case of reconnection it will handle it in a separate thread. The choice depends on the scenario you're working on. If you're working with reliable communication like JMS, you probably can afford to have a nonblocking reconnection strategy. Otherwise, you may want to work with a blocking strategy.

Let's find out how to use reconnection strategies by configuring a JMS connector with a reconnect element in XML in the following listing. The same can be accomplished in Mule Studio using the dialog shown in figure 9.1.

Listing 9.1 Using the reconnect strategy

```
<jms:activemq-connector name="jmsConnector">
    <reconnect count="5" frequency="1000"/>
</jms:activemq-connector>
```

You may realize that you set one reconnection strategy for the connector instead of setting one for inbound and another one for outbound connections. If you need separate inbound and outbound behavior, you can get it by using two different connectors, and then applying one to the inbound endpoints and the other to the outbound ones (see the next listing).

Listing 9.2 Using different strategies for inbound/outbound

```
<jms:activemq-connector name="jmsInboundConnector">
    <reconnect-forever frequency="1000" />
</jms:activemq-connector>
<jms:activemq-connector name="jmsOutboundConnector">
    <reconnect count="5" frequency="1000" />
</jms:activemq-connector>
```

The built-in reconnection strategies of Mule are good enough for many cases but not for everything. For instance, one of the associates of Prancing Donkey supplies an excellent barley, but their software services are less than good: their endpoints fail often and so need a restart. Sadly, their maintenance engineers are only available during working hours, so if something fails at night, the chances that a reconnection will succeed are lower.

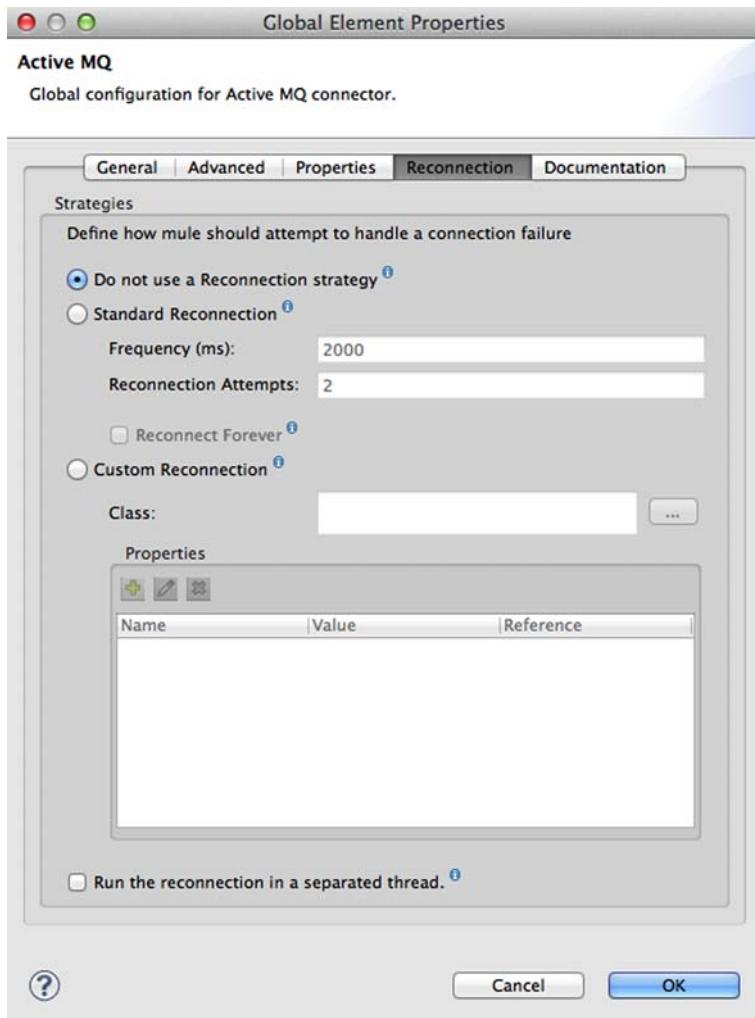


Figure 9.1 Reconnection tab for a connector in Mule Studio

9.1.2 Creating reconnection strategies

Thankfully, Mule offers the ability to writing custom reconnection strategies. The situation with the barley provider looks like an excellent opportunity to put them into practice. Reconnection strategies are defined by implementing the `RetryPolicy` interface. Listing 9.3 demonstrates a simple reconnection strategy that instructs a connector to reconnect to the failed resource, waiting a different amount of time depending on if the reconnection happens during working hours or not.

Listing 9.3 A working hours-aware retry policy

```

public class WorkingHoursAwareRetryPolicy implements RetryPolicy
{
    int companyWorkStartHour;
    int companyWorkEndHour;
    int intervalInWorkingHours;
    int intervalInNonWorkingHours;
    TimeZone timeZone;

    public WorkingHoursAwareRetryPolicy(int companyWorkStartHour,
                                         int companyWorkEndHour, TimeZone timeZone,
                                         int intervalInWorkingHours, int intervalInNonWorkingHours)
    {
        this.companyWorkStartHour = companyWorkStartHour;
        this.companyWorkEndHour = companyWorkEndHour;
        this.intervalInWorkingHours = intervalInWorkingHours;
        this.intervalInNonWorkingHours = intervalInNonWorkingHours;
        this.timeZone = timeZone;
    }

    public PolicyStatus applyPolicy(Throwable cause)
    {
        Calendar cal = Calendar.getInstance();
        cal.setTime(new Date());
        cal.setTimeZone(timeZone);

        int hour = cal.get(Calendar.HOUR_OF_DAY);
        boolean withinCompanyHours =
            (hour >= companyWorkStartHour && hour < companyWorkEndHour);

        try
        {
            Thread.sleep(withinCompanyHours ?
                intervalInWorkingHours : intervalInNonWorkingHours);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return PolicyStatus.policyOk();
    }
}

```

Sleeps the defined amount of time

Throws a runtime exception if interrupted

This retry policy will sleep at ①; if not interrupted, it will return a `PolicyStatus` of OK. This informs the connector to attempt to retry again. If the thread is interrupted, a runtime exception will be thrown ②. In addition to returning a `policyOK` state, `PolicyStatus` can also return an exhausted state. This is useful if you want to limit the amount of retry attempts to a particular resource and is demonstrated in the next listing.

BEWARE OF THE LEGACY NAMES In Mule 2, reconnection strategies were called *retry policies*; this name mistakenly suggests that Mule will retry operations doing message redelivery. In Mule 3, the name was changed, and *retry policies* became officially deprecated. Nevertheless, you'll still find it in the Mule API.

Listing 9.4 An exhaustible retry policy

```
public class WorkingHoursAwareExhaustibleRetryPolicy
    implements RetryPolicy
{
    int companyWorkStartHour;
    int companyWorkEndHour;
    int intervalInWorkingHours;
    TimeZone timeZone;

    public WorkingHoursAwareExhaustibleRetryPolicy(
        int companyWorkStartHour,
        int companyWorkEndHour, TimeZone timeZone,
        int intervalInWorkingHours)
    {
        this.companyWorkStartHour = companyWorkStartHour;
        this.companyWorkEndHour = companyWorkEndHour;
        this.intervalInWorkingHours = intervalInWorkingHours;
        this.timeZone = timeZone;
    }

    public PolicyStatus applyPolicy(Throwable cause)
    {
        Calendar cal = Calendar.getInstance();
        cal.setTime(new Date());
        cal.setTimeZone(timeZone);

        int hour = cal.get(Calendar.HOUR_OF_DAY);
        boolean withinCompanyHours =
            (hour >= companyWorkStartHour && hour < companyWorkEndHour);

        if (!withinCompanyHours) {
            return PolicyStatus.policyExhausted(cause);
        }

        try
        {
            Thread.sleep(intervalInWorkingHours);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    return PolicyStatus.policyOk();
}
```

① Stop the retrying by returning policyExhausted

The `WorkingHoursAwareExhaustibleRetryPolicy` will attempt to connect to the remote resource only during working hours; if reconnection is issued outside of working hours, then a `PolicyStatus` of `exhausted` is returned ①, along with the cause of the retry. This causes the connector to stop retrying to connect to the failed resource.

Before you can use either of the retry policies defined in the previous listing, you must implement a policy template. This is accomplished by extending the `AbstractPolicyTemplate` class. The next listing illustrates a policy template for the `WorkingHoursAwareRetryPolicy`.

Listing 9.5 A policy template

```
public class WorkingHoursAwareRetryPolicyTemplate
    extends AbstractPolicyTemplate
{
    int companyWorkStartHour;
    int companyWorkEndHour;
    int intervalInWorkingHours;
    int intervalInNonWorkingHours;
    TimeZone timeZone;

    public RetryPolicy createRetryInstance()
    {
        return new WorkingHoursAwareRetryPolicy(
            companyWorkStartHour, companyWorkEndHour, timeZone,
            intervalInWorkingHours, intervalInNonWorkingHours);
    }

    public void setCompanyWorkStartHour(int companyWorkStartHour)
    {
        this.companyWorkStartHour = companyWorkStartHour;
    }

    public void setCompanyWorkEndHour(int companyWorkEndHour)
    {
        this.companyWorkEndHour = companyWorkEndHour;
    }

    public void setIntervalInWorkingHours(int intervalInWorkingHours)
    {
        this.intervalInWorkingHours = intervalInWorkingHours;
    }

    public void setIntervalInNonWorkingHours
        (int intervalInNonWorkingHours)
    {
        this.intervalInNonWorkingHours = intervalInNonWorkingHours;
    }

    public void setTimezone(TimeZone timeZone)
    {
        this.timeZone = timeZone;
    }
}
```

You need to implement the `createRetryInstance` method of `AbstractPolicyTemplate` to return an instance of the `RetryPolicy` you wish to use. In this case, you'll return a `WorkingHoursAwareRetryPolicy`, which will cause the connector to use the logic defined in listing 9.3.

THE RIGHT TOOL FOR THE RIGHT JOB Listing 9.4 represents a simple Java implementation of a `java.util.Calendar`-based retry policy. For complex scenarios you may want to use more powerful tools like the Drools Expert rules engine. Sadly, there's no out-of-the-box support to do this in Mule. You should create your own adapter implementing `RetryPolicy` to delegate the logic to Drools.

Finally, to tie everything together, when you define your connector you can pass your recently created policy template using the element reconnect-custom-strategy, as shown in the next listing.

Listing 9.6 Configuring a connector with a custom strategy

```
<jms:activemq-connector name="jmsWithCustomReconnect">
    <reconnect-custom-strategy
        class=
"com.prancingdonkey.reconnect.WorkingHoursAwareRetryPolicyTemplate">
    <spring:property name="companyWorkStartHour" value="9"/>
    <spring:property name="companyWorkEndHour" value="18"/>
    <spring:property name="intervalInWorkingHours" value="5000"/>
    <spring:property name="intervalInNonWorkingHours"
        value="360000000"/>
    <spring:property name="timeZone"
        value="#{T(TimeZone).getTimeZone('America/Los_Angeles')}"/>
</reconnect-custom-strategy>
</jms:activemq-connector>
```

Now you're able to react to system exceptions caused by a connection problem. Let's find out how you can use exception strategies to react when an error is raised with a message involved.

9.1.3 Handling exceptions

Being able to define different reactions for system exceptions caused by connection problems and for messaging exceptions lets you handle each sort of error independently. You learned how you can react to connection problems; now let's find out how exception strategies work for situations where there are exceptions with messages involved.

You have the option of explicitly defining exception strategies in multiple places in your Mule configurations. Exception strategies can be configured on a per-flow basis. This is done by defining an exception strategy at the end of each flow definition. You can additionally define exception strategies globally; in this case, to establish the exception strategy in a flow, you'll have to use a reference. And finally, you can also configure the default exception strategy that Mule will use when there's no strategy configured for a flow with a problem.

Let's start by configuring an exception strategy inside a flow. We still haven't studied the different built-in exception strategies, but don't worry; we'll do so soon. In the meantime, let's use a catch-exception-strategy that will catch and process all exceptions thrown in the flow, shown in the next listing.

Listing 9.7 A locally defined exception strategy

```
<flow name="tax-calculator">
    <vm:inbound-endpoint
        path="tax-calculator-service.in"
        exchange-pattern="request-response" />
```

```

<component>
    <singleton-object
        class="com.prancingdonkey.component.TaxCalculator" />
</component>

<catch-exception-strategy>
    <set-payload value="Error: #[exception.summaryMessage]" />
</catch-exception-strategy>

</flow>

```



① Catch any exception

Here you can find one new element, the `catch-exception-strategy` ①. This element instructs the flow to respond with a String that summarizes the exception message, for any kind of error. This approach is useful for special cases, but requires you to configure different exception strategies for flows that could reuse exception strategies. To avoid this, you can use the global exception strategies.

To configure a global exception strategy, like you already learned with other message processors, two steps are required. You should first configure a global exception strategy and then put a reference in the desired flows. Let's configure a global exception strategy, shown in the following listing.

Listing 9.8 A globally defined exception strategy

```

<catch-exception-strategy name="globalCatchStrategy">
    <set-payload value="Error: #[exception.summaryMessage]" />
</catch-exception-strategy>

<flow name="tax-calculator">

    <vm:inbound-endpoint
        path="tax-calculator-service.in"
        exchange-pattern="request-response" />

    <component>
        <singleton-object
            class="com.prancingdonkey.component.TaxCalculator" />
    </component>

    <exception-strategy ref="globalCatchStrategy" />

```



① Global exception strategy

② Exception strategy reference

You can find two outstanding points here: At ① you introduce a global exception strategy, and then you reuse it at ②. This is extremely useful when you want to reuse an exception strategy in a few flows. But there's still another scenario to cover. What happens when you need to configure an exception strategy for all (or most) of the flows? For these cases, you can use the default exception strategy.

The default exception strategy is configured by defining a `defaultException-Strategy-ref` attribute on the Mule configuration element and pointing it to the desired global exception strategy, which is configured like a regular global endpoint. Defining the default exception strategy on the configuration element will cause all flows in the Mule application with no specific exception strategy to use the defined exception strategy. Let's revisit listing 9.8 and use the global endpoint it configures as the default exception strategy in the next listing.

Listing 9.9 Configuring the default exception strategy

```

<catch-exception-strategy name="globalCatchStrategy">
    <set-payload value="Error: #[exception.summaryMessage]" />
</catch-exception-strategy>

<configuration defaultExceptionStrategy-ref="globalCatchStrategy" /> ← ① Global exception strategy

<flow name="tax-calculator">
    <vm:inbound-endpoint
        path="tax-calculator-service.in"
        exchange-pattern="request-response" />

    <component>
        <singleton-object
            class="com.prancingdonkey.component.TaxCalculator" />
    </component>
</flow> ← ② Default exception strategy configuration

```

The default exception strategy configured at ② will ensure all exceptions thrown in the flow will be handled by the catch-exception-strategy defined at ①.

9.1.4 Using exception strategies

Now that you're more comfortable with the different ways to configure an exception strategy in Mule, it's a good moment to learn about the exception strategies Mule supports. As we mentioned before, the default exception strategy will log exceptions and move on. This is often the right degree of action to take, but sometimes you'll want to take more elaborate measures when an exception occurs. This is particularly 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 or rollback and retry. To suit your needs, Mule provides five exception strategies ready to use:

- *Default exception strategy*—Implicitly present by default in every flow; will log the exception and roll back the transaction.
- *Catch exception strategy*—Customize the reaction for any exception; will commit the transaction and so will consume the message.
- *Rollback exception strategy*—Customize the reaction for any exception; will roll back the transaction, hence won't consume the message.
- *Choice exception strategy*—Using a Mule expression will route to one exception strategy within a set; if no exception strategy is suitable in the set, the exception will be routed to the default exception strategy.
- *Reference exception strategy*—Delegate the responsibility to handle exceptions to a global exception handler.

Those strategies can be found in Mule Studio in the form you can see in figure 9.2. You're already familiar with the default and reference exception strategies. Let's learn a little bit more about the other three strategies with a couple of examples.

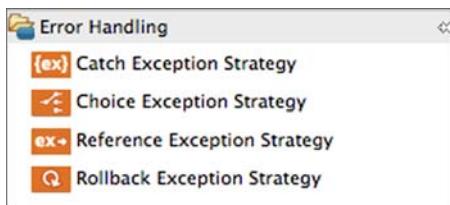


Figure 9.2 Exception strategies as shown in Mule Studio's toolbox

You recall Prancing Donkey's shipping cost calculator from section 6.1.7. When we introduced it for the first time, we approached the error handling naïvely. This time we'll add some more reality bites with more complex scenarios.

Since Prancing Donkey deployed the brand new shipping cost calculator, many different possible errors have been found. The first reaction of the integration architects was to return a message explaining something went wrong. They used a catch exception strategy to be able to catch the error and react to it without inducing a retry, as you can see in listing 9.10 and in figure 9.3.

Listing 9.10 Flow with a catch exception strategy

```
<flow name="ShippingCostCalculatorFlow">
    <vm:inbound-endpoint path="calculateShippingCost.in"
        exchange-pattern="request-response" />
    <component>
        <singleton-object
            class="com.prancingdonkey.component.ShippingCostCalculator" />
    </component>
    <catch-exception-strategy>
        <set-payload value="Error: #[exception.summaryMessage]" />
        <logger message=
            "ShippingCost error : #[exception.summaryMessage]"
            level="ERROR" />
    </catch-exception-strategy>
</flow>
```

① Catches any exceptions
② Logs exception summary

This approach worked as expected. The catch-exception-strategy at ① grabbed all the errors and created a specific response for them. But thanks to the logger at ②, Prancing Donkey's integration architects found that sometimes the errors were caused by an `IllegalStateException`. They decided to retry three times and then give up, returning another informative message to the requestor.

Fortunately, Mule provides the rollback exception strategy that fits perfectly in this situation. It will use a `maxRedeliveryAttempts` attribute in combination with an `on-redelivery-attempts-exceeded` child element to accomplish the desired behavior.

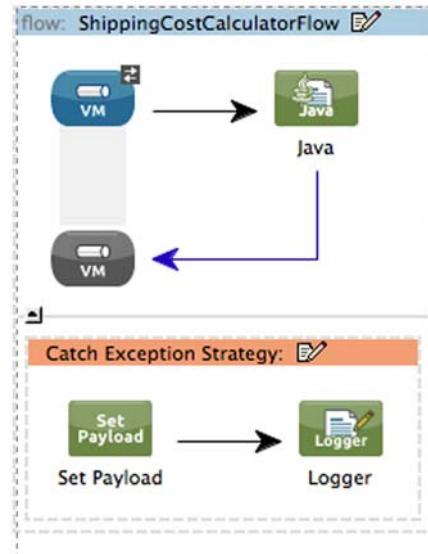


Figure 9.3 Mule Studio equivalent for listing 9.10

The problem with this is that it can only handle the `IllegalStateException` problem, whereas Prancing Donkey still needs to process the rest of the exceptions as they did in listing 9.10. Here the last of the exception strategies, the choice exception strategy, comes in useful.

Useful when expressions

The when attribute of the exception strategies should be a Mule expression. This will let you create complex routing without writing extra code:

- `exception.causedBy(com.prancingdonkey.MyPrancingException)` evaluates if the exception was caused by an instanceof the provided exception type.
- `exception.causedExactlyBy(com.prancingdonkey.MyPrancingException)` evaluates if the exception was caused by the type provided, and only by that type provided.
- `exception.causeMatches('com.prancingdonkey.*') && !exception.causedBy(com.prancingdonkey.MyPrancingException)` evaluates if the cause exception type name matches the '`com.prancingdonkey.*`' regex and is not caused by `com.prancingdonkey.MyPrancingException`.

The choice exception strategy is the router of the exception strategies: it can contain other exception strategies, routing between them by leveraging a when attribute in the child strategies. Thanks to the choice exception strategy, you can take advantage of any combination of catch and rollback exception strategies. The only limitation is that a choice exception strategy can't contain another choice exception strategy. Let's use it in listing 9.11 and in figure 9.4.

Listing 9.11 Flow with a choice exception strategy

```
<flow name="ShippingCostCalculatorFlow"
  processingStrategy="synchronous">
  <vm:inbound-endpoint path="calculateShippingCost.in"
    exchange-pattern="request-response">
    <vm:transaction action="ALWAYS_BEGIN"/>
  </vm:inbound-endpoint>

  <component>
    <singleton-object
      class="com.prancingdonkey.component.ShippingCostCalculator" />
  </component>
  <choice-exception-strategy>
    <rollback-exception-strategy
      when="exception.causedBy(java.lang.IllegalStateException)"
      maxRedeliveryAttempts="3">
      <logger message="Retrying shipping cost calc." level="WARN" />
      <on-redelivery-attempts-exceeded>
        <logger message="Too many retries shipping cost calc."
          level="WARN" />
    </rollback-exception-strategy>
  </choice-exception-strategy>
</flow>
```



**Choice exception strategy
nesting to strategies**

Set
redelivery
limit to
three



```

<set-payload value="Error: #[exception.summaryMessage]" />
</on-redelivery-attempts-exceeded>
</rollback-exception-strategy>

<catch-exception-strategy>
    <logger message=
        "ShippingCost error : #[exception.summaryMessage]"
        level="ERROR" />
    <set-payload value="Error: #[exception.summaryMessage]" />
</catch-exception-strategy>

</choice-exception-strategy>

</flow>

```

The choice exception strategy ① will route between the child strategies it has. When the exception is caused by an `IllegalStateException`, the `rollback-exception-strategy` will be invoked, whereas if it's of any other type, the `catch-exception-strategy` will be used. The `maxRedeliveryAttempts` attribute ② instructs the `rollback-exception-strategy` to restart the transaction up to three times; after that, the `on-redelivery-attempts-exceeded` block will be executed.

DIFFERENT REDELIVERY ATTEMPTS Some reliable transports, like JMS, have their own redelivery mechanism that can include a max redelivery limit. As this works at the transport level, this mechanism could be faster than the rollback exception strategy's redelivery policy. If you want to rely on your transport's redelivery policy, don't set the `maxRedeliveryAttempts` attribute of the `rollback-exception-strategy`.

Now we'll see how to use the built-in exception strategies. This will enable you to apply different behaviors for further processing and responses when dealing with exceptions.

9.2 Using transactions with Mule

At the beginning of this chapter, we considered a real-life transaction example. Similar scenarios are present in software applications. Updating related data in a database, for instance, usually requires that all or none of the tables be 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 data that has been updated to the point of failure. This makes the database update *atomic*; even though a sequence of disconnected events is taking place (the updating of different tables), they're treated as a single operation that's either completely successful or completely rolled back.

Having atomicity allows you 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 that other transactions aren't affected. This is closely related to consistency. If another process is querying a table

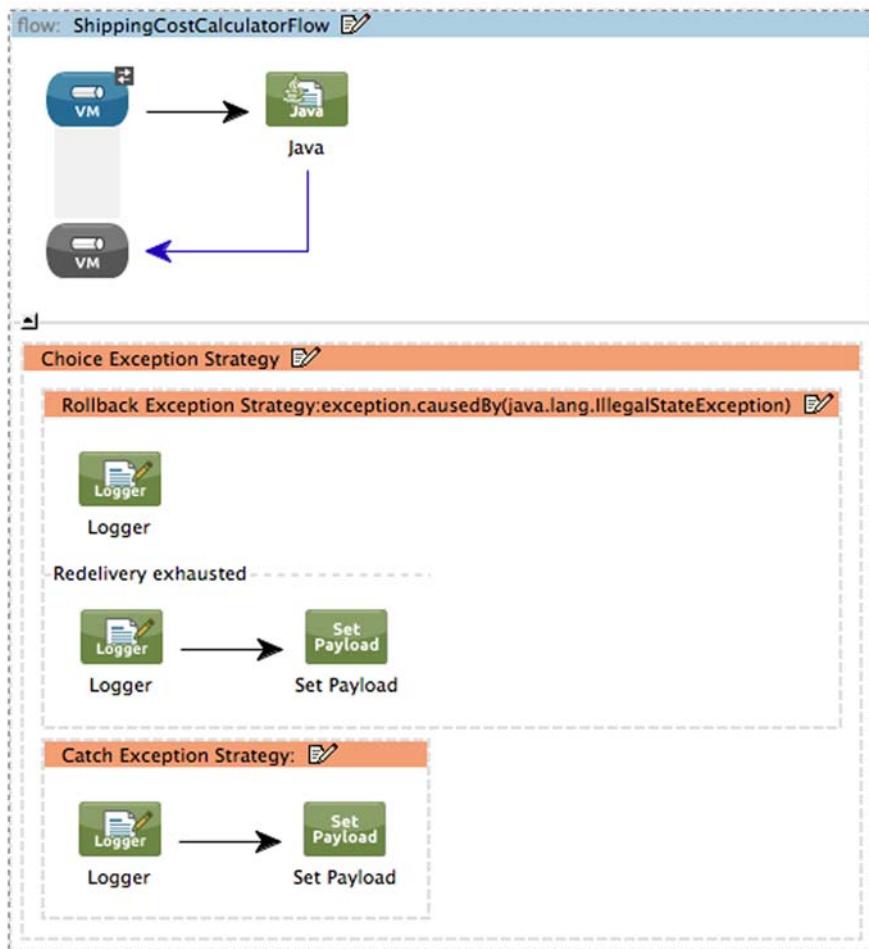


Figure 9.4 Mule Studio equivalent for listing 9.11

while the database update we discussed previously is occurring, and then the update is subsequently rolled back, the other process might read data that is 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 you 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—together form the acronym *ACID*. This term is commonly used to discuss the transactions we'll examine 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. You'll see in this chapter how to add transactional properties to your Mule services. We'll cover the two major types of transactions supported by the Java and Java EE ecosystems: single- and multiple-resource transactions. We'll start off by examining how single-resource transactions let you operate on a single resource, such as a database or JMS broker, transactionally. You'll then see how you can use transactions across multiple resources. Finally, we'll look at how you 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 Prancing Donkey. You'll see how Mule's transactional support augments the reliability of the integration projects.

9.2.1 **Single-resource transaction**

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 static recipient list 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 to use these same techniques to consume and send JMS messages in transactions.

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 the update doesn't happen. 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.

Prancing Donkey makes extensive use of a relational databases for its day-to-day operations. Suppose that recently Prancing Donkey's operations team has deployed a performance monitoring application to run against their client's web applications. This application periodically runs a series of tests against a client's website 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 Prancing Donkey'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. The following listing illustrates how Prancing Donkey is accomplishing this.

MYSQL AND TRANSACTIONS For transactions with MySQL to work properly, they must be supported by an underlying engine that supports transactions, such as InnoDB.

Listing 9.12 Using a JDBC outbound endpoint transactionally

```

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

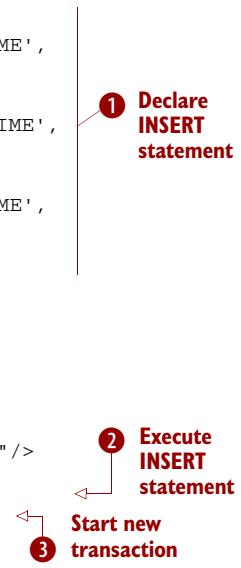
<jms:activemq-connector name="jmsConnector" specification="1.1" />

<jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
    <jdbc:query key="statsInsert"
        value="INSERT INTO PERF_METRICS VALUES
            (0, #[message.payload.CLIENT_ID], 'AVG_RESPONSE_TIME',
            #[message.payload.AVG_RESPONSE_TIME],
            #[message.payload.TIMESTAMP]),
            (0, #[message.payload.CLIENT_ID], 'MED_RESPONSE_TIME',
            #[message.payload.MED_RESPONSE_TIME],
            #[message.payload.TIMESTAMP]),
            (0, #[message.payload.CLIENT_ID], 'MAX_RESPONSE_TIME',
            #[message.payload.MAX_RESPONSE_TIME],
            #[message.payload.TIMESTAMP]) "/>
</jdbc:connector>

<flow name="URLAlertingFlow">
    <jms:inbound-endpoint topic="monitoring.performance"
        exchange-pattern="request-response" />

    <component class=
        "com.prancingdonkey.component.URLMetricsComponent" />

    <jdbc:outbound-endpoint queryKey="statsInsert">
        <jdbc:transaction action="ALWAYS_BEGIN" />
    </jdbc:outbound-endpoint>
</flow>
```



The insert statement is declared at ①. It inserts data into the `PERF_METRICS` table using data from the map populated by the `URLMetricsComponent`. The `URLMetricsComponent` builds this map from the JMS message received on the monitoring `.performance` topic. The insert statement is then executed ②.

The action parameter 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 9.1 lists the valid action values for a Mule transaction.

Table 9.1 Available options for configuring a transaction action

Action value	Description
NONE	Never participate in a transaction, and commit any previously existing transaction.
NOT_SUPPORTED	Will execute within an existing transaction context if it's present but without joining it, because there's no transactional resource to join.
ALWAYS_BEGIN	Always start a new transaction, committing any previously existing transaction.
BEGIN_OR_JOIN	If there is an existing transaction, join that transaction. If not, start a new transaction.
ALWAYS_JOIN	Always expect and join an existing transaction. Throw an exception if no previous transaction exists.
JOIN_IF_POSSIBLE	Join an existing transaction if one exists. If no transaction exists, run nontransactionally.

Because the JDBC outbound endpoint in listing 9.12 isn't participating in any other transaction, you naturally use `ALWAYS_BEGIN` to start a new transaction for the insert.

Now that you've seen how transactions work with the JDBC transport, let's take a look at how you can send and receive JMS messages transactionally.

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. A rollback in this case means the message is destroyed, causing the JMS provider to attempt redelivery of the message. A transaction on a JMS outbound endpoint indicates that the message was sent successfully. If there was a failure, the message is rolled back and redelivery is attempted. As you'll see shortly, JMS transactions can be used in conjunction with the all router, allowing multiple messages to be dispatched in the same transaction. First, let's see how to accept JMS messages transactionally.

Given that Prancing Donkey is using this service in a production manner for important data (such as messages containing order or provisioning information), instead of passing the message to the console for display, they're instead forwarding

the message to another JMS queue for further processing. In this case, Prancing Donkey wants to be sure the messages aren't lost, either by a failure occurring at the inbound endpoint or a failure occurring on an outbound endpoint. Listing 9.13 shows a modified configuration that adds transactional semantics to the inbound and outbound endpoints, ensuring that at each step the message processing is successful, and if not, is rolled back. Figure 9.5 represents an JMS endpoint using transactions in Mule Studio.

Listing 9.13 Accepting and sending JMS messages transactionally

```
<jms:activemq-connector name="jmsConnector" specification="1.1" />

<flow name="URLAlertingFlow">
    <jms:inbound-endpoint queue="messages"
        exchange-pattern="request-response">
        <jms:transaction action="ALWAYS_BEGIN" />
    </jms:inbound-endpoint>

    <regex-filter pattern="^STATUS: (OK|SUCCESS)$" />

    <component class=
        "com.prancingdonkey.component.BusinessComponent" />

    <jms:outbound-endpoint queue="processed.messages">
        <jms:transaction action="ALWAYS_BEGIN" />
    </jms:outbound-endpoint>
</flow>
```

The diagram illustrates the flow of a JMS transaction across three main stages:

- 1 Accept JMS messages in new transaction**: This stage is indicated by a red circle with the number 1 and a red arrow pointing to the `<jms:transaction action="ALWAYS_BEGIN" />` block within the inbound endpoint configuration.
- 2 Enrich messages not caught by forwarding-router's regex-filter**: This stage is indicated by a red circle with the number 2 and a red arrow pointing to the component configuration block.
- 3 Send JMS messages in new transaction**: This stage is indicated by a red circle with the number 3 and a red arrow pointing to the `<jms:transaction action="ALWAYS_BEGIN" />` block within the outbound endpoint configuration.

Messages are received off the messages queue ①. If a message has a payload that matches the regular expression defined by the regex-filter, it's passed to the component ②. The JMS transaction is defined at ①, where a new transaction is created for every message received. The transaction will span receiving the message on the inbound endpoint and its processing by the Spring object. When the message reaches the JMS outbound endpoint, the `ALWAYS_BEGIN` action of the JMS transaction defined at ③ will cause the previous transaction started at ① to commit and a new transaction to begin.

Failures in the component are handled differently. Unless overridden by the methods discussed previously in this chapter, 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. You'll see later in this chapter how you can override this behavior and trigger a commit based on the type of exception thrown by a component.

Once the message is passed to outbound endpoint, it's sent to the processed-messages queue. In listing 9.13, the transaction configured at ③ will ensure this message is sent in a new transaction. If the message can't be sent, the transaction won't begin and the message will be lost. To solve this problem, let's see how you can have the outbound endpoint join in the transaction created at ①. Then a failure to start the transaction on the outbound endpoint will trigger a rollback up to the beginning of the transaction created at ①. Listing 9.14 illustrates how to do this.

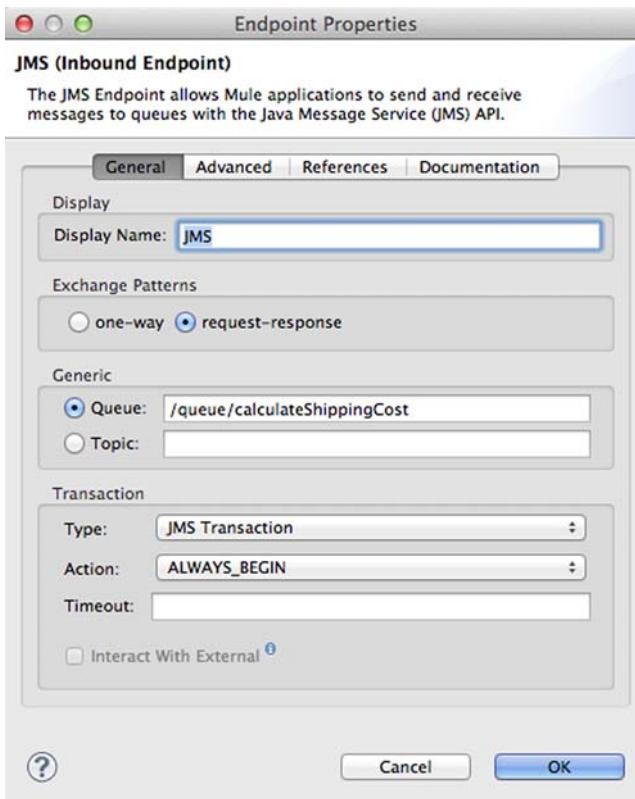


Figure 9.5 JMS endpoint using transactions

Listing 9.14 Using ALWAYS_JOIN to join in an existing transaction

```
<jms:activemq-connector name="jmsConnector" specification="1.1" />

<flow name="URLAlertingFlow">
    <jms:inbound-endpoint queue="messages"
        exchange-pattern="request-response">
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:inbound-endpoint>

    <regex-filter pattern="^STATUS: (OK|SUCCESS)$" />

    <component class=
        "com.prancingdonkey.component.BusinessComponent"/>

    <jms:outbound-endpoint queue="processed.messages">
        <jms:transaction action="ALWAYS_JOIN"/>
    </jms:outbound-endpoint>
</flow>
```

- 1 Accept JMS messages in new transaction
- 2 Join in existing transaction

By changing the transaction action to `ALWAYS_JOIN` ②, you ensure that the sending of the JMS message will always join a preexisting transaction. In the event the message

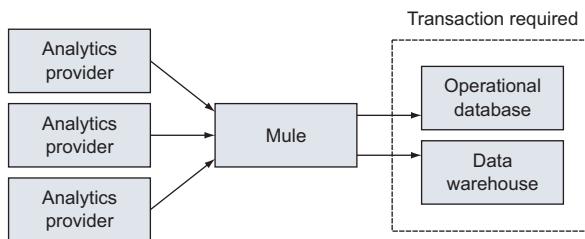


Figure 9.6 Prancing Donkey's approach for decoupled data management

can't be sent, the transaction will roll back to where it began, at ①, and the message will be pushed back to the JMS provider for subsequent redelivery.

It's also possible to send multiple messages transactionally. Using an all router with JMS outbound endpoints along with a JMS transaction will cause all 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 Prancing Donkey.

Prancing Donkey currently collects various real-time analytical data about its customer's web applications. Web application response times, network metrics, and billing data are all collected from Prancing Donkey's data centers and published to JMS queues for processing by Prancing Donkey's Mule instances. This data must ultimately be processed and fed into Prancing Donkey's operational database as well as its data warehouse. In order to decouple the operational database and the data warehousing, your team has decided to republish the data in an appropriate format for each destination. The resultant data is then placed on a dedicated JMS queue where it's consumed and saved. Because 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 9.6 illustrates this.

The following listing shows the corresponding Mule configuration.

Listing 9.15 Using the multicasting router transactionally

```

<jms:activemq-connector name="jmsConnector" specification="1.1" />
<flow name="transactedMulticastingRouterService">
    <composite-source>
        <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>
    </composite-source>
    <component class=
        "com.prancingdonkey.component.AnalyticsService"/>
    <all>
  
```

Process analytical data

2

1 Accept JMS messages from analytics providers

3 Define the multicasting router

```

<jms:outbound-endpoint queue="operational-database">
    <jms:transaction action="ALWAYS_BEGIN" timeout="1000" />
</jms:outbound-endpoint>
<jms:outbound-endpoint queue="data-warehouse">
    <jms:transaction action="ALWAYS_BEGIN" timeout="1000" />
</jms:outbound-endpoint>
</all>
</flow>

```

Send analytical data transactionally.

①

②

③

④

The JMS endpoints defined at ① will accept messages from each analytics provider transactionally. This data will be processed by the analyticsService Spring object configured at ②. If there's a failure on the JMS endpoints, the transaction will be rolled back. Exceptions thrown by analyticsService will be logged but will have no impact on the transaction. The processed message will then be passed to the all router defined at ③. The JMS transactions defined at ① and ④ will ensure that the message is sent to both JMS queues successfully. If there's a failure on either queue, then the transactions won't begin and the message will be lost. The timeout value defined at ④ specifies how many milliseconds to wait before rolling back the transaction.

MESSAGE REDELIVERY IN JMS You might be wondering how often and for how long message redelivery is attempted. The JMSTimeToLive header property defines how long a message exists until it's destroyed by the JMS provider. The redelivery fields are dependent on your JMS provider. These can be important properties to reference when tuning and troubleshooting JMS and Mule performance.

In order to ensure messages aren't lost by such a failure, you can make this entire message flow transactional, from JMS inbound endpoints to JMS outbound endpoints. The next listing shows how to do this.

Listing 9.16 Making an entire message flow transactional

```

<jms:activemq-connector name="jmsConnector" specification="1.1" />

<flow name="transactedMulticastingRouterService">
    <composite-source>
        <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>
    </composite-source>
    <component class=
        "com.prancingdonkey.component.AnalyticsService"/>
    <all>
        <jms:outbound-endpoint queue="operational-database">
            <jms:transaction action="ALWAYS_JOIN" timeout="1000" />

```

Join
previously
existing
transaction

```
</jms:outbound-endpoint>
<jms:outbound-endpoint queue="data-warehouse">
    <jms:transaction action="ALWAYS_JOIN" timeout="1000" />
</jms:outbound-endpoint>
</all>
</flow>
```

① Join previously existing transaction

You change the transaction action from `ALWAYS_BEGIN` to `ALWAYS_JOIN` at ①. Now a failure on either queue in the all 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.

Transactions with the VM transport

The VM transport can also be used transactionally. For instance, to begin a new VM transaction instead of a JMS transaction in listing 9.16, you would use the `vm:transaction` element, as follows:

```
<all>
    <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>
</all>
```

A transacted VM endpoint shows more reliability than a nontransacted one, but still can't be considered a completely reliable transport as message loss may occur if the Mule instance goes down. But this isn't true if you're using a clustered Mule (only available with Mule Enterprise Edition; see appendix C) where the messages are backed by other nodes in the cluster. You could also improve reliability by using a reliability pattern, as discussed in the sidebar "Implementing a reliability pattern" in section 9.2.2.

Transactions, by their nature, are synchronous operations. As such, Mule endpoints participating in transactions will operate synchronously. In the previous listing, the JMS inbound endpoint that receives a JMS message will block until the transaction is either committed or rolled back on the outbound endpoint. Keep this in mind when structuring transactions in your services.

In this section, you 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 a JMS provider. Let's investigate Mule's support for that now.

Decoupling middleware

In 2007, Michael Nygard's book *Release It!* (www.pragprog.com/titles/mnee/release-it) presented the Decoupling Middleware pattern to decouple remote service invocation, in both space and time, using a messaging broker. This allows you to use the features of the messaging broker, such as durability and delayed redelivery, to improve the resiliency of communication with a remote service.

Let's consider some of the benefits of this indirection:

- The service can be taken down/brought up without fear of losing messages; they'll queue on the broker and will be delivered once the service is brought back up.
- The service can be scaled horizontally by adding additional instances (competing consumption off the queue).
- Messages can be reordered based on priority and sent to the remote service.
- Requests can be resubmitted in the event of a failure.

The last point is particularly important. It's common to encounter transient errors when integrating with remote services, particularly web-based ones. These errors are usually recoverable after a certain amount of time.

Mule does have support for message-oriented middleware protocols like JMS or AMQP, and it also has support for transactions, as you learned in the first half of this chapter. This makes Mule a perfect citizen for decoupled systems, although there could be a certain level of feature collision between Mule and your decoupling middleware, depending on the middleware you're using.

In section 9.1.3, you learned how to use the `rollback-exception-strategy`. It triggers Mule's capacity to redeliver failed messages. You also learned how to use the `maxRedeliveryAttempts` attribute and the `on-redelivery-attempts-exceeded` child element of the same exception strategy that incidentally represents another one of Michael's patterns: the "circuit-breaker" (http://en.wikipedia.org/wiki/Circuit_breaker_design_pattern).

Your decoupling middleware, usually present in the form of a JMS broker, is probably able to redeliver and circuit break when necessary. You can use redelivery in conjunction with transactions to periodically re-attempt the failed request. HornetQ supports this by configuring address-settings on the queue. The following address-setting for `exampleQueue` specifies 12 redelivery attempts with a five-minute interval between each request:

```
<address-setting  
    match="jms.queue.exampleQueue">  
        <max-delivery-attempts>12</max-delivery-attempts>  
        <redelivery-delay>300000</redelivery-delay>  
    </address-setting>
```

9.2.2 Transactions against multiple resources

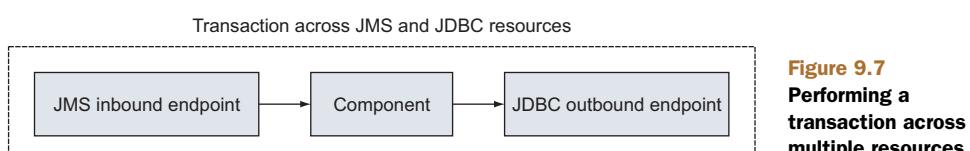
Performing transactions on a single resource is appropriate if the operations to be conducted transactionally all use the same connector. You saw examples of this in the last section, in which we showed how you can use the JDBC and JMS transports transactionally. But what if the operations you wish to 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 9.7 illustrates this scenario.

The XA standard is a distributed transaction protocol designed to meet this need. For resources that support XA transactions, like many JDBC drivers and JMS providers, this is possible through the use of the Java Transaction API. An XA transaction uses the *two-phase commit* (2PC) protocol to ensure that all participants in the transaction commit or roll back. During the first phase of the 2PC, the transaction manager polls each participant in the transaction and makes sure each is ready to commit the transaction. In the second phase of 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.

1.5 PHASE TRANSACTIONS WITH MULE EE The Mule Enterprise Edition provides another multiple resource transaction mechanism using a 1.5 phase commit. This is faster than a two-phase commit, but is less reliable and also supports fewer transports. For more information on this system, refer to the Mule documentation: <http://mng.bz/i1X8>.

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 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 also



introduce scalability issues when locking occurs in the 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.

You'll see in this section how Mule uses the Java Transaction API (JTA) 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.

SPANNING MULTIPLE RESOURCES WITH JBossTS

It used to be the case that running JTA transactions required the use of a Java EE application server, like JBoss AS or WebLogic. Thankfully, there are now standalone JTA implementations that don't require this amount of overhead. One such implementation, which is supported out of the box by Mule, is JBossTS. Let's see how you can use Mule's JBossTS support to improve Prancing Donkey's data warehousing service.

You saw in listing 9.15 how Prancing Donkey 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 the queues. This approach was taken to decouple message reception from message publication. The JMS inbound endpoints would be able to transactionally receive and process JMS messages independently of messages being successfully consumed by the queues defined in the outbound endpoints.

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 or not 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 sure that the billing data is saved to the operational database and the data warehouse successfully. To support this, Prancing Donkey has added another service that's dedicated to receiving billing data. Figure 9.8 illustrates the new service, which forgoes the JMS outbound endpoints and will write to the database and data warehouse directly.

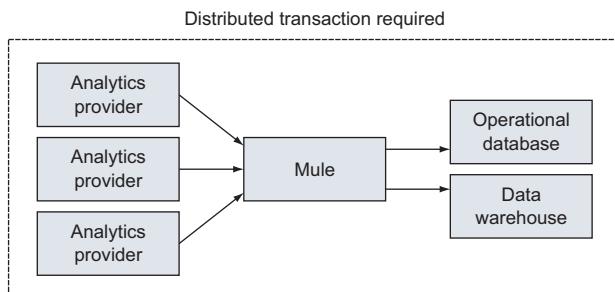


Figure 9.8 Wrapping billing data reception into a single transaction

Assuming Prancing Donkey's JMS provider and its JDBC drivers for the database and data warehouse support XA transactions, you can use Mule's JBossTS support to wrap this entire operation in a single transaction. The next listing illustrates how Prancing Donkey has accomplished this.

Listing 9.17 Sending outbound messages to list of endpoints using an XA transaction

```

Define ActiveMQ XA connector for the data source ①
  > <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,#[message.payload.STAT])"/>
  </jdbc:connector>

  <jdbc:connector name="warehouseDb"
    dataSource-ref="warehouseDataSource">
    <jdbc:query key="warehouseBillingInsert"
      value=
        "INSERT INTO billing_stats VALUES (0,#[message.payload.STAT])"/>
  </jdbc:connector>

  <jbossts:transaction-manager/>
<flow name="transactedMulticastingRouterService" >

  <jms:inbound-endpoint queue="billingData">
    <xa-transaction action="ALWAYS_BEGIN" />
  </jms:inbound-endpoint>

  <component class="com.prancingdonkey.component.BillingService" />

  <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>
</flow>

```

You start off by defining an ActiveMQ connector that supports XA transactions ①. Because the database and data warehouse require separate data sources, you need to configure two JDBC connectors to reference each data source. These are configured at ② and ③. The referenced data sources for both of these connectors, configured in Spring, should support XA transactions. You specify that you're using JBossTS to manage the XA transactions ④. When a JMS message is received on the inbound endpoint, a new XA transaction is started by the `ALWAYS_BEGIN` value given to the `xa-transaction` ⑤. The message is processed by the component and sent to the two following outbound endpoints.

② Define JDBC connector for operational data source

③ Define JDBC connector for warehouse data source

④ Use JBossTS to manage XA transactions

⑤ Begin new XA transaction

⑥ Expect and join existing transaction

ALWAYS_BEGIN AND PREVIOUS TRANSACTIONS If there's a current transaction running when using `ALWAYS_BEGIN` in an XA context, that transaction will be suspended and a new transaction started. Once this new transaction has completed, the previous transaction will be resumed.

The `xa-transaction` configuration ⑥ has an action of `ALWAYS_JOIN`. This means it's expecting 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 take a look at how Prancing Donkey could modify this configuration to run in one of their application servers.

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 Java EE 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 9.2 lists the supported application servers along with their associated configuration elements.

Table 9.2 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 9.17 assumed we were running Mule standalone and as such were leveraging JBossTS outside of any JBoss AS context. Let's now assume that the Mule configuration in listing 9.17 is running as a WAR application inside Resin. To use Resin's JTA implementation, you'd replace the `jboss-transaction-manager` element with Resin's, as illustrated in the next listing.

Listing 9.18 Using an application server's transaction manager for XA

```

<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,#[message.payload.STAT])" />
</jdbc:connector>

<jdbc:connector name="warehouseDb"
    dataSource-ref="warehouseDataSource">
    <jdbc:query key="warehouseBillingInsert"
        value=
            "INSERT INTO billing_stats VALUES (0,#[message.payload.STAT])" />
</jdbc:connector>

<resin-transaction-manager/>           ←— Use Resin's JTA support

<flow name="transactedMulticastingRouterService" >

    <jms:inbound-endpoint queue="billingData">
        <xa-transaction action="ALWAYS_BEGIN" />
    </jms:inbound-endpoint>

    <component class="com.prancingdonkey.component.BillingService" />

    <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>

</flow>

```

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" />
```

Implementing a reliability pattern

By using Mule's support for single and multiple transactions in combination with transactional middleware like JMS or JDBC, you can guarantee reliable processing of messages. The messages might be processed correctly, retried, or even fail (informing the requester), but they will never be lost.

(continued)

But reliable middlewares isn't always the case. Often you'll find yourself in situations in which messages are received through an unreliable transport such as HTTP. In this scenario, when you still want to push the reliability to the limit, even if you have to use an unreliable transport, you'll usually apply a reliability pattern.

You can implement a reliability pattern by coupling with a reliable transport in two phases: the reliable acquisition phase and the application logic phase (see figure 9.9).

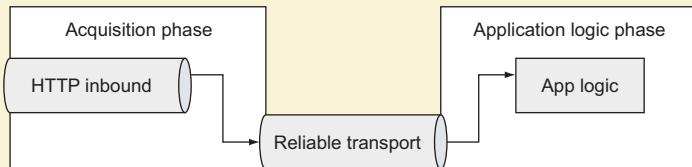


Figure 9.9
Reliability pattern
for HTTP

Following this design, the reception of the message is placed in the reliable acquisition phase, and it will take the message and try to queue it in a reliable transport. The success or the failure of this process will be reported back to the requestor so the whole operation can be retried in case of error. If the message is accepted with success, the requestor will know for sure that the message won't be lost.

The reliable transport can vary—the most common scenario will include a JMS broker. Nevertheless, if you have a Mule HA cluster (not available in the Community Edition), a VM transport can well be a favorable solution. In the following listing, you can find an implementation of this pattern using JMS.

Listing 9.19 Implementing a JMS reliability pattern

```

<flow name="reliable-acquisition-phase"
processingStrategy="synchronous">
    <http:inbound-endpoint address="http://127.0.0.1:8081/inbound"
        exchange-pattern="request-response" />
    <object-to-string-transformer />
    <jms:outbound-endpoint queue="application-logic-queue"
        exchange-pattern="one-way"/>
</flow>

<flow name="application-logic-phase">
    <jms:inbound-endpoint queue="application-logic-queue">
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:inbound-endpoint>
    <logger message="business-logic is performed here." />
</flow>

```

Message acquisition in request-response

Send message using a reliable transport

Receive message from the reliable transport

Application logic placeholder

9.2.3 Transaction demarcation

Transactions can be created on inbound endpoints or outbound endpoints. When used on inbound endpoints, the started transaction can be leveraged by the subsequent endpoints. For instance, you can read a database on a JDBC inbound endpoint and on the same transaction perform two different inserts. If one of the inserts fails, the whole transaction will be rolled back. You can see this behavior in figure 9.10.



Figure 9.10 Using transactions created on inbound endpoints

You can use transactions on outbound endpoints to join, commit, or refuse an already created transaction. You can't, however, create a transaction in an outbound endpoint that's intended to be used again later in Mule. If you create a new transaction in an outbound endpoint, the effect may vary on the different transports, but the typical result will be that the previous transaction will be committed and the requested action will be performed by itself transactionally, as you can see in figure 9.11.

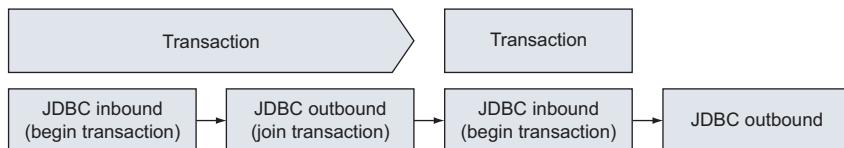


Figure 9.11 Creating transactions on outbound endpoints

What happens when you want to call transactionally different endpoints within a transaction that's not created in an inbound endpoint? To accomplish this, you'll have to do transaction demarcation. It will let you scope a group of elements that should be executed inside a transaction, as shown in figure 9.12.

Transaction demarcation is established in Mule flows by nesting the Mule logic you want to be transactional inside one of the available transaction demarcators shown in table 9.3.

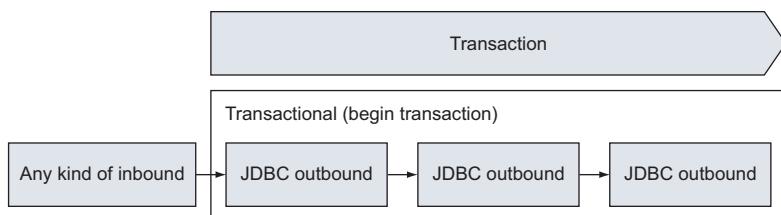


Figure 9.12 Using transaction demarcation

Table 9.3 Transaction demarcation elements

Demarcation type	Description
transactional	Also known as single-resource transaction demarcation. Establishes a transaction for the first resource type found.
ee:xa-transactional	Uses a distributed transaction for all of the resource types found. <i>Only available in the Mule Enterprise Edition.</i>
ee:multi-transactional	Starts multiple transactions for each of the resource types found. <i>Only available in the Mule Enterprise Edition.</i>

The usage of the transactional demarcators couldn't be simpler. You have to wrap the elements you want to use transactionally within the element:

```
<transactional>
    <jms:outbound-endpoint queue="out1"/>
    <jms:outbound-endpoint queue="out2"/>
</transactional>
```

Here you set a single resource transaction using the transactional element ①. Mule will find the first resource, in this case a JMS endpoint ②, and will set a transaction for it.

This raises a question: If this is a single-resource transaction, can you mix and match resources? You can mix resources, but you should explicitly specify that the transactions shouldn't be used in the nonprimary resources. If you do need to have all of them in the same transaction, you'll have to use the enterprise transaction demarcators specified in table 9.3. Let's add a VM endpoint to the previous single-resource transaction example:

```
<transactional>
    <jms:outbound-endpoint queue="out1"/>
    <vm:outbound-endpoint path="alternate">
        <vm:transaction action="NOT_SUPPORTED"/>
    </vm:outbound-endpoint>
    <jms:outbound-endpoint queue="out2"/>
</transactional>
```

You specify NOT_SUPPORTED for the VM outbound endpoint ①, and therefore you leave it outside the transaction.

By default, transaction demarcators will always begin a transaction. You could also instruct the element to join the transaction if it already exists. This is handy when using subflows in combination with demarcators:

```
<flow name="transactionalEntryPoint">
    <jms:inbound-endpoint queue="orders">
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:inbound-endpoint>
```

```
<flow-ref name="transactionalFlow"/>
</flow>

<flow name="nonTransactionalEntry">
    <vm:inbound-endpoint path="nonTransactional.in"/>
    <flow-ref name="transactionalFlow"/>
</flow>

<subflow name="transactionalFlow">
    <transactional action="BEGIN_OR_JOIN">
        <jms:outbound-endpoint queue="billingOrders"/>
        <jms:outbound-endpoint queue="productionOrders"/>
    </transactional>
</subflow>
```

Finally, we should note that transactional demarcators also support the use of exception handlers in the same fashion you learned at the beginning of this chapter. For instance, let's say that you want to log a message when the transaction goes wrong. You can do that by adding a rollback-exception-strategy to the transactional element and including a logger in it:

```
<subflow name="transactionalFlowWithLogger">
    <transactional action="BEGIN_OR_JOIN">
        <jms:outbound-endpoint queue="billingOrders"/>
        <jms:outbound-endpoint queue="productionOrders"/>
        <rollback-exception-strategy>
            <logger message="Problem in the transaction! " />
        </rollback-exception-strategy>
    </transactional>
</subflow>
```

9.3 Summary

In this chapter, we investigated Mule's error-handling capabilities. You saw how to use reconnection strategies to define how errors are handled when they're related to reconnection problems. You then saw how to use exception strategies to manage what happens after an exception occurs with a message involved. We then turned our attention to Mule's transaction support.

Transactions play a critical role when grouping otherwise distinct operations together atomically. They can be indispensable in an integration scenario, in which the nature of such operations is often distinct. You 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, with 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. This, too, is easily configurable by modifying an exception strategy's XML configuration.

In the next chapter, we'll discuss another aspect crucial to Mule: security.

10

Securing Mule

This chapter covers

- Securing Mule with Spring Security
- Securing HTTP using SSL
- Using SOAP's WS-Security
- Encrypting messages with Mule

Security is a challenge in application development and deployment—a challenge that’s exacerbated by application integration. Single sign-on (SSO) technologies like OpenAM, CAS, and LDAP minimize these burdens, but it’s an unlikely scenario 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 you saw in part 1 to handle 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. You’ll see how Prancing Donkey makes use of 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.

10.1 Spring Security 3.0 and Mule

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`, Mule's default security manager, provides support for basic security functionality, such as password and secret key-based encryption. Mule provides additional `SecurityManager` implementations that support more sophisticated security implementations such as Spring Security, JAAS, and PGP.

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 take a look at how to use Spring Security's security managers and security providers to secure your Mule services. We'll examine authenticating users against in-memory and LDAP user databases. Spring Security, formerly known as Acegi Security, is the officially supported security product of the Spring Portfolio. Using Spring Security with Mule follows the pattern we described previously. After defining your Spring Security configuration, which you'll see how to do shortly, you define Mule Spring Security providers and managers that can be used on your endpoints.

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 10.1 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 a database. You may want to try LDAP authentication first and, if that fails, try authentication against a database.

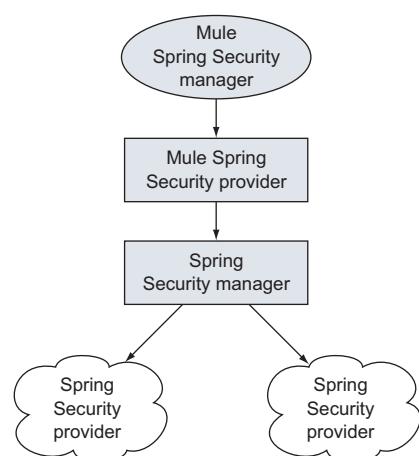


Figure 10.1 Delegation between Mule Spring Security and Spring Security

Let's take a look at two ways in which you 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.

10.1.1 User security with an in-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 10.1 demonstrates configuring the user-service, along with a Spring Security manager and provider.

Listing 10.1 Defining an in-memory user service for endpoint authentication

```

<spring:beans>
    <ss:authentication-manager>           ← Begin Spring bean configuration
        alias="authenticationManager"
        <ss: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 name="victor"
                    password="password"
                    authorities="ROLE_ADMIN" />
            </ss:user-service>
        </ss:authentication-provider>
    </ss:authentication-manager>
</spring:beans>

<mule-ss:security-manager>
    <mule-ss:delegate-security-provider
        name="memory-dao"
        delegate-ref="authenticationManager" />
</mule-ss:security-manager>

```

The Mule Spring Security manager is defined at ⑤. The security provider is then configured ⑥ to delegate to the Spring Security manager defined at ①. The manager will implicitly use the Spring Security authentication-provider defined at ②, which is configured with an in-memory user-service ③. The definitions for the user-service start at ④. The format used to define a user is shown in figure 10.2.

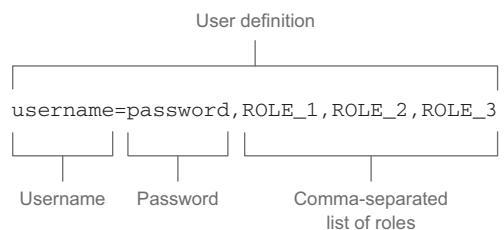


Figure 10.2 File format for using the Spring user service

Although 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.

10.1.2 User security with LDAP

LDAP has emerged as a common directory implementation in many organizations, as is 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 10.2 defines a security manager and provider that authenticate off of Prancing Donkey's OpenLDAP server.

Listing 10.2 Defining an LDAP directory for endpoint authentication

```
<spring:beans>
    <ss:ldap-server
        root="dc=prancingdonkey,dc=com"
        url="ldap://ldap.prancingdonkey.com:389" />
    <ss:authentication-manager
        alias="authenticationManager">
        <ss:ldap-authentication-provider
            user-dn-pattern="uid={0},ou=people"
            group-search-base="ou=groups" />
    </ss:authentication-manager>
</spring:beans>

<mule-ss:security-manager>
    <mule-ss:delegate-security-provider
        name="memory-dao"
        delegate-ref="authenticationManager" />
</mule-ss:security-manager>
```

The diagram illustrates the structure of the XML configuration with four numbered callouts:

- ① Define LDAP server**: Points to the `<ss:ldap-server` element.
- ② Define Mule Security manager**: Points to the `<mule-ss:security-manager>` element.
- ③ Define LDAP authentication provider**: Points to the `<ss:ldap-authentication-provider` element within the `<ss:authentication-manager>`.
- ④ Define Spring Security provider**: Points to the `<mule-ss:delegate-security-provider` element within the `<mule-ss:security-manager>`.

Prancing Donkey's LDAP server is defined at ①. Define the URL of the server along with the root distinguished name (DN) you want to query from. The Spring Security LDAP authentication provider is then defined at ③. Define two dn-patterns, 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, which is 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, a user 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 at ② and finally referenced by the Mule Spring Security manager ④. Endpoints referencing this security-manager will now use Prancing Donkey'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 (see <http://projects.spring.io/spring-security/>) for more info. Now that you're comfortable with configuring Spring Security for both in-memory and LDAP authentication scenarios, let's add authentication to the HTTP transport using security filters.

10.1.3 Securing endpoints with security filters

Security filters allow you to control access, authorization, and encryption on your endpoints. In this section, we'll cover how to filter HTTP inbound endpoints using Spring Security and the http-security-filter.

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. In the following listing, you'll see how to configure an http-security-filter using Spring Security and an in-memory user-service.

Listing 10.3 Defining a basic HTTP security filter on an HTTP inbound endpoint

```
<spring:beans>
    <ss:authentication-manager alias="authenticationManager">
        <ss:authentication-provider>
            <ss:user-service id="userService">
                <ss:user name="victor" password="password"
                    authorities="ROLE_ADMIN" />
            </ss:user-service>
        </ss:authentication-provider>
    </ss:authentication-manager>
</spring:beans>

<mule-ss:security-manager>
    <mule-ss:delegate-security-provider name="memory-dao"
        delegate-ref="authenticationManager" />
</mule-ss:security-manager>

<flow name="authenticatedHttp">
    <inbound-endpoint address="http://localhost:8081/secure">
        <mule-ss:http-security-filter realm="mule"/>
    </inbound-endpoint>
    <logger message="User logged in" level="INFO" />
</flow>
```

The code listing is annotated with three red callouts:

- Annotation ①: Points to the `<ss:authentication-manager` element with the text "Define user service beans".
- Annotation ②: Points to the `<mule-ss:security-manager` element with the text "Define Mule Security manager".
- Annotation ③: Points to the `<mule-ss:http-security-filter` element with the text "Define security filter on HTTP endpoint".

The in-memory user-service is defined by the beans ①, which are delegated to by the Mule security-manager defined at ②. Now you'll see where these come into play. The http-security-filter 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 an LDAP directory, by changing the delegate reference inside the security-manager. No changes need to be made to the security-filter.

We've talked about security filters, Spring's authentication mechanisms, and how to combine them. They're mainly oriented toward credential checking. There are other forms of security; the transport-level security is particularly useful in many cases. Let's find out how to implement transport-level security using SSL.

BEST PRACTICE Change the delegate reference in a security-manager to swap out authentication mechanisms.

10.2 Securing HTTP using SSL

In section 3.3.1, we used the HTTP transport to send and receive data. We passed clear data through the network, and it worked. Sadly, the high security standards of Prancing Donkey won't allow sending or receiving sensitive data without protection. HTTPS comes in handy here.

The Hypertext Transfer Protocol Secure (HTTPS) is a protocol that combines HTTP with SSL/TLS, giving to the widespread HTTP the security capabilities it lacks. HTTPS differs from HTTP not only in the URL scheme and port, but also by adding a whole new layer of security by encryption. Unfortunately, this layer also brings more complexity. If you want to use HTTPS, you have to deal with certificates: either having one to serve or adding external certificates to your key ring to make requests.

Let's start with HTTPS by looking at how to set up a server.

10.2.1 Setting up an HTTPS server

To serve content, the HTTPS connector requires a *keystore*. A keystore contains the private key and any certificates necessary to complete a chain of trust and establish the trustworthiness of the primary certificate.

Before you start using the HTTPS connector, you should create a self-signed certificate in a keystore or import a certificate into a keystore from a certificate authority. Thankfully, Java comes with the keytool command that lets you create keystores. If you have the Java bin directory in your PATH, you're ready to use it; otherwise, look for it in your Java installation directory:

```
keytool -genkey -alias prancingdonkey -keyalg RSA  
-keystore keystore.jks -keysize 2048
```

After responding to a few questions, this command will generate a file called keystore.jks that's ready to be used with the HTTPS connector. It will contain a self-signed certificate. Keep in mind the passwords you passed to keytool; they'll be necessary later.

If you plan to use a certificate from a certificate authority, the process is a bit longer, as you'll have to create a certificate signing request (CSR) that might differ between the different certificate authorities. Please refer to the Java keytool instructions of your certificate authority.

Once you have your keystore ready, you need to tell the HTTPS connector how to use it; you'll do so by passing a `tls-key-store` to the HTTPS connector:

```
<flow name="expenseReportSecureCallback">

    <https:inbound-endpoint host="127.0.0.1"
        port="8443" path="expenseReportCallback" method="POST"
        exchange-pattern="one-way" connector-ref="httpsServerConnector" />

    <file:outbound-endpoint path="/tmp"
        outputPattern="#{java.util.UUID.randomUUID().toString()}.xml" />

</flow>
```

With the connector already configured, you need to set up an inbound HTTPS endpoint in a way similar to how you did it for HTTP. The only outstanding part is that you'll point the `connector-ref` attribute to the connector you configured.

Listing 10.4 Using an HTTPS server in a flow

```
<https:connector name="httpsServerConnector">
    <https:tls-key-store path="keystore.jks"           ← Define keystore
        keyPassword="password" storePassword="password" />
</https:connector>
```

With a few steps, you set up an HTTPS service. Now it's time to learn how to create an HTTPS client at the other end of the pipe.

10.2.2 Setting up an HTTPS client

Prancing Donkey's commitment to security isn't only related to the received data; when sending data over the internet, the same security standards should be applied. You may recall that when we studied the HTTP connector in section 3.3.1, we created a `postExpenseReports` flow that uploaded XLS expense reports to Arnor Accounting's web service. Expense reports are sensitive information, so it was decided to switch from HTTP to HTTPS.

When connecting to an HTTPS server, you need to validate the server's certificate against a trusted root certificate or against your keystore. If the certificate can be validated against the trusted root certificates, the configuration needed in Mule to connect to that server is straightforward. Let's find an endpoint by using GET against Google:

```
<flow name="getGoogle">
    <vm:inbound-endpoint address="vm://getGoogle.in"
        exchange-pattern="request-response" />

    <https:outbound-endpoint host="www.google.com"
        port="443" path="/" />
```

```

method="GET" />

<object-to-string-transformer />
</flow>
```

As you can see, when you connect to servers with certificates signed by a trusted root certificate, the HTTPS client looks similar to the HTTP equivalent. But what happens when you have to use self-signed certificates?

Java trusted certificates

Java is shipped with a few trusted root certificates. If you want to list the trusted certificates in your Java installation, use the following command:

```
keytool -keystore "$JAVA_HOME\jre\lib\security\cacerts"
    -storepass changeit -list
```

When connecting to an HTTPS server with a self-signed certificate, use a keystore with the certificates you trust. Prancing Donkey trusts Arnor Accounting, so what they need is a trust store with Arnor Accounting's public key. In order to create it, fetch the certificate from the server using openssl:

```
openssl s_client -showcerts -connect
    api.accounting.com:443 >arnor.pem </dev/null
```

Sadly, openssl will export not only the certificates but also some extra information that keytool won't be happy to receive, so you'll need to edit the file arnor.pem with your favorite editor to remove anything above BEGIN CERTIFICATE and below END CERTIFICATE, to effectively have something similar to this Google certificate:

```
-----BEGIN CERTIFICATE-----
MIIDIZCCAoygAwIBAgIEM... UzEXMBUGA1UEChMOVmVyaVNpZ24...
bG1jIFByaW1hcngQ2Vyd... MDAwWhcNMTQwNTExMjM1OTU5WjB...
d3R1IENvbnN1bHRpbmcgK... QTCBnzANBgkqhkiG9w0BAQEFAAO...
PKzMyGT7Y/wySweUvW+Au... 5/0ItY0y3pg25gqtAHvEZEo7hHU...
3nWhLHp039XKHIdYYBkCA... A1UdDwQEAwIBBjARBglghkgBvhv...
BgNVBAMTEFByaXzhGVMY... L2Nybc52ZXJpc2lnbi5jb20vcGN...
AQUFBzABhhZodHRwOi8vb... BwMBBgggrBgEFBQcDAgYJYIZIAyB...
BQUAA4GBAFWsY+reod3Sk... q3J5Lwa/q4FwxKjt61M07e8eU9k...
bcV0oveifHtgPHfNDS5IAN8BL7abN+AqKjbc1YXWrOU/VG+WHgWv
-----END CERTIFICATE-----
```

Once the file's ready, you can create a trust store using the following command:

```
keytool -importcert -alias arnor -keystore
    truststore.jks -file arnor.pem
```

Now you're ready to configure the HTTPS connector. The result will be similar to the connector you configured for the server in the previous section, with the exception that the `tls-key-store` will now become `tls-client`:

```
<https:connector name="httpsClientConnector">
    <https:tls-client path="clientkeystore.jks"
```

```
    storePassword="password" />
</https:connector>
```

With the HTTPS connector configured, you can finally convert the insecure expenseReportCallback created in section 3.3.1 into a secure service:

```
<flow name="securePostExpenseReports">
  <file:inbound-endpoint path=".//data/expenses/in"
    pollingFrequency="60000">
    <file:filename-regex-filter pattern=".xls"
      caseSensitive="false"/>
  </file:inbound-endpoint>

  <https:outbound-endpoint host="api.accounting.com"
    port="443" path="expenseReports"
    method="POST" exchange-pattern="one-way"
    connector-ref="httpsClientConnector"/>
</flow>
```

Now you know how to secure the HTTP transport using SSL. Next, let's add a security layer to SOAP.

10.3 Securing SOAP with Mule

If you recall from sections 3.3.2 and 6.2.1, you already know how to configure WS-* services using Mule. You already exposed the Brew Service thanks to the Mule support of JAX-WS through the use of Apache CXF.

In sections 3.3.2 and 6.2.1, you created naïve services that had no concerns at all about security. It's time to build a security wall around them to let pass whom you want. To secure WS-* services, Mule allows you to use WS-Security (see <http://en.wikipedia.org/wiki/WS-Security>), an extension to SOAP that defines standard security headers for SOAP, by using the Apache WSS4J library (see <http://ws.apache.org/wss4j/>).

WSS4J has support for validating security credentials. In order to add authentication to the service, you'll add a validator in combination with an in-memory security-manager you'll reuse from listing 10.1. Let's see them in action in this listing.

Listing 10.5 Setting authentication in a SOAP service

```
<spring:beans>
  <ss:authentication-manager
    alias="authenticationManager">
    <ss: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 name="victor"
          password="password"
          authorities="ROLE_ADMIN" />
      </ss:user-service>
```

```

</ss:authentication-provider>
</ss:authentication-manager>
</spring:beans>

<mule-ss:security-manager>
    <mule-ss:delegate-security-provider name="memory-dao"
        delegate-ref="authenticationManager" />
</mule-ss:security-manager>

<cxft:ws-security name="inboundSecurityConfig">      ← ① Define encryption strategy
    <cxft:mule-security-manager />
    <cxft:ws-config>
        <cxft:property key="action" value="UsernameToken" /> ← ② Encrypt payload
    </cxft:ws-config>                                     of outbound message
</cxft:ws-security>

<flow name="securedBrewSoapService">
    <http:inbound-endpoint address="http://localhost:8090/soap"
        exchange-pattern="request-response">

        <cxft:jaxws-service
            serviceClass="com.prancingdonkey.service.BrewService">
            <cxft:ws-security ref="inboundSecurityConfig"/> ← ③ Decrypt payload
        </cxft:jaxws-service>                               of inbound message
    </http:inbound-endpoint>

    <component class="com.prancingdonkey.service.BrewServiceImpl" />
</flow>

```

In general, this might look similar to the service you saw in section 3.3.2; the differences start at ①, where you configure the cxf security, establishing at ② that you want to check the username before letting pass the request to the component. There's another difference compared with section 3.3.2; there's now a WS-Security element inside the jaxws-service at ③. With it, you're configuring the service to use the security defined at ①.

SOAP security is a wide topic; there are other use cases such as validation using certificates and signing. The WS-Security page of the CXF site (<http://cxf.apache.org/docs/ws-security.html>) is probably a good starting point for finding more information. Now let's leave behind authentication and start learning how to use encryption.

10.4 Message encryption with Mule

At Prancing Donkey, 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, in addition to the authentication you added before, you decide to force clients to encrypt the data being sent to the inbound endpoint using a key shared with the remote office.

In the next couple of sections, you'll learn how to encrypt and decrypt messages, effectively making the Prancing Donkey website a safer place. Let's start applying the simplest form of encryption, using a shared password.

10.4.1 Using password-based payload encryption

Prancing Donkey's security requirements can be implemented by Mule's security support. You'll encrypt 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. Let's now take a look at how to perform end-to-end payload encryption of a message, with both encryption and decryption. Listing 10.6 illustrates a service that accepts a JMS message off a queue, encrypts it, and sends it to another JMS queue, which decrypts the message and forwards it on.

Listing 10.6 Encrypting the payload of messages using password-based encryption

```
<security-manager>
    <password-encryption-strategy name="PBE"
        password="password" />
</security-manager>

<flow name="password-based-encryption">
    <jms:inbound-endpoint queue="messages.in" />
    <encrypt-transformer strategy-ref="PBE" />
    <object-to-string-transformer />
    <jms:outbound-endpoint queue="services.decryption" />
</flow>

<flow name="password-based-decryption">
    <jms:inbound-endpoint queue="services.decryption" />
    <decrypt-transformer strategy-ref="PBE" />
    <object-to-string-transformer />
    <jms:outbound-endpoint queue="messages.out" />
</flow>
```

① Define security profile
② Set security profile for the encryption
③ Set security profile for the decryption

You configure the password-encryption-strategy at ① as you have previously. The encrypt-transformer is then used at ② to encrypt the outbound message payload before it's placed on the JMS queue. When the message is received ③, it's decrypted by the decrypt-transformer. Both the encrypt-transformer and decrypt-transformer refer to the encryption strategy by its name, PBE.

Transports supporting SSL or TLS

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
- IMAPS
- POP3S
- XMPPS
- SMTPS

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.

10.4.2 Decrypting message payloads with PGP

You've seen 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 for 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, that user can encrypt the message using the recipient's public key. The recipient is then able to decrypt the message using their own private key. The sender is also able to "sign" a message using their 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 (GnuPG) and OpenPGP are popular client-side alternatives. Using public key encryption securely, however, 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 the hand-delivery of public keys to individuals on CD-ROM to ensure authenticity. For a larger organization, 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 the signatures of and decrypt messages. Let's take a look at using a PGP security-filter to only accept PGP messages that you can decrypt and on which you can perform successful signature verification. The following listing illustrates how to do this.

Listing 10.7 Decrypting PGP-encrypted JMS payloads using a PGP security-filter

```
<spring:bean id="pgpKeyManager"                                ① Define pgpKeyManager
    class="org.mule.module.pgp.PGPKeyRingImpl"
    init-method="initialise">
    <spring:property name="publicKeyRingFileName"
        value="public.key.gpg" />                                ← Public key ring
    <spring:property name="secretKeyRingFileName"
        value="secret.key.gpg" />                                ← Private key ring
```

```

<spring:property name="secretAliasId"
    value="9071504784255173009" />
<spring:property name="secretPassphrase"
    value="mule" />
</spring:bean>

<spring:bean id="credentialAccessor"
    class="org.mule.security.MuleHeaderCredentialsAccessor" />

Configure PGP security-manager → <pgp:security-manager>
    <pgp:security-provider name="pgpSecurityProvider"
        keyManager-ref="pgpKeyManager" />
    <pgp:keybased-encryption-strategy
        name="keyBasedEncryptionStrategy"
        keyManager-ref="pgpKeyManager"
        credentialsAccessor-ref="credentialAccessor" />
</pgp:security-manager>

<flow name="password-based-decryption">
    <jms:inbound-endpoint queue="messages.in" />

    <decrypt-transformer name="pgpDecrypt"
        strategy-ref="keyBasedEncryptionStrategy" />
    <object-to-string-transformer />
    <jms:outbound-endpoint queue="messages.out" />
</flow>

```

Define credentialAccessor ②

Pass valid messages through to outbound endpoint ③

Start off by defining a key-manager to handle the details required for PGP ①. Specify the location of your key rings, the key alias, and the secret password. The credential accessor is defined at ②. This instructs Mule how to infer the user who encrypted or signed the payload from the message. In this case, use the `MuleHeaderCredentialsAccessor`, which will have Mule use the `MULE_USER` message property for identification of who encrypted or signed the message. The decryption and signature verification occur at 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.¹

PGP AND THE SUN JVM 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 7 Unlimited Strength Jurisdiction Policy Files, for instance, are located at <http://mng.bz/sz1g>. Installation instructions are included in the archive.

¹ You'll need a PGP implementation to generate a key pair and encrypt messages. Good options include Cryptix (www.cryptix.org/) for programmatic manipulation and GNU Privacy Guard (www.gnupg.org/) for manipulation via the command line.

In this section, you saw how to use public key encryption to securely decrypt and verify the payloads of Mule messages.

10.5 **Summary**

In this chapter, you saw how Mule simplifies security for integration projects. This simplification is hopefully evident from our discussion of security managers and security providers. You learned how to use a security manager such as Spring Security to independently provide security services for endpoint filters. You saw how to use endpoint filters to provide basic authentication for HTTP endpoints. We showed how to create HTTPS servers and clients and how to secure SOAP using WS-Security. Finally, we demonstrated how to encrypt payloads using PGP.

Now that you're comfortable with security in Mule, we'll talk about two other cross-cutting concerns in many integration scenarios: transactions and monitoring.

11

Tuning Mule

This chapter covers

- Identifying performance bottlenecks
- Staged event-driven architectures
- Configuring processing strategies

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. Indeed, Mule, like any middleware application, is constrained by the limits of memory size, CPU performance, storage, and network throughput. Tuning Mule is about finding the sweet spot in which 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 in 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.

Let's start by looking deeper into the architecture of Mule than ever before in order to learn how threading works behind the scenes.

11.1 Staged event-driven architecture

An understanding of how Mule processes messages is critical in order to tune Mule applications effectively. Mule flows will use one or more pools of threads to process every message. This architecture, in which an application's event-processing pipeline is decomposed into processing stages, is one of the core concepts behind a *staged event-driven architecture*, or *SEDA* (see figure 11.1). SEDA, as described in Matt Welsh's paper "SEDA: An Architecture for Highly Concurrent Server Applications,"¹ is summarized as follows:

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.

Staged event-driven architectures use event queues to split up an application's processing stages. Each processing stage uses a thread pool to consume messages of a work queue, eliminating the need to spawn a thread for every request the application needs to process. Because a single thread is no longer handling every request, it becomes more difficult to exhaust the application's ability to spawn threads. It also ensures that other stages of the processing pipeline aren't starved of the ability to spawn threads.



Figure 11.1 A typical staged event-driven architecture (SEDA) implementation

¹ See SEDA, Matt Welsh, at www.eecs.harvard.edu/~mdw/proj/seda.

In this chapter, you'll see how processing strategies and exchange patterns dictate how much a given flow can take advantage of Mule's SEDA underpinnings. Assuming your application is tuned properly, you'll be able to take full advantage of two important benefits of this architecture. The first, which is made clear in Matt's paper, is the graceful degradation of your application under load—both expected and contrived (that is, through a denial-of-service attack). The second, and perhaps more important, benefit is that your application will make better use of the available cores on the system it's running on. The techniques introduced in this chapter will aid you in taking advantage of the latter to ensure you're making the most use of the cores available to your application.

11.1.1 Roll your own SEDA

Most of the tuning you'll see in this chapter occurs at the flow level of granularity. You'll see how Mule's various tuning knobs allow you to define how a flow receives, processes, and dispatches messages. It's advisable, however, to consider the concepts of SEDA and how they might apply to your application as a whole.

Mule's VM transport, discussed in chapter 3, is an excellent mechanism to help break your flows into smaller, composite flows that are decoupled with in-memory queueing. This allows you to realize the benefits of SEDA at the macro level, breaking your application into smaller flows that can be individually tuned and are decoupled with VM queueing.

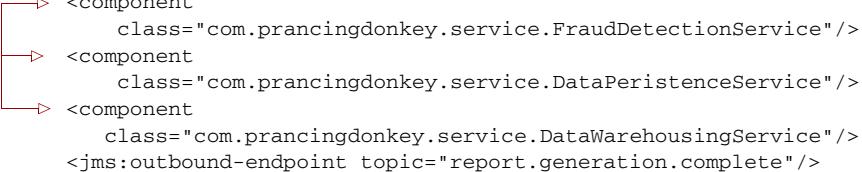
RELIABILITY PATTERNS AND SEDA In chapter 7, you saw how the VM transport can be used to implement reliability patterns. The technique discussed here is an extension of that approach.

The following example illustrates a simple flow that Prancing Donkey is using to process reporting data. The processing stages include three Java components.

Listing 11.1 Report processing in a single flow

```
<flow name="processExpenseReports">
    <jms:inbound-endpoint queue="report.processing"
        exchange-pattern="one-way" />
    <component
        class="com.prancingdonkey.service.FraudDetectionService"/>
    <component
        class="com.prancingdonkey.service.DataPeristenceService"/>
    <component
        class="com.prancingdonkey.service.DataWarehousingService"/>
    <jms:outbound-endpoint topic="report.generation.complete"/>
</flow>
```

Use three
Java
components
to process
reports



Each component is doing a bunch of nontrivial processing for each message received by the flow, whose payload is a large XML report. In order to maximize throughput, refactor the flow using the VM transport. You'll break this flow out into three different flows that each use a VM queue to decouple each processing stage.

Listing 11.2 Report processing in three flows

```
<flow name="processExpenseReportsStage1">           ← First processing stage
    <jms:inbound-endpoint queue="report.processing"
                           exchange-pattern="one-way"/>
    <component
        class="com.prancingdonkey.service.FraudDetectionService"/>
    <vm:outbound-endpoint path="report.generation.stage.2"/>
</flow>

<flow name="processExpenseReportsStage2">           ← Second processing stage
    <vm:inbound-endpoint path="report.generation.stage.2"/>
    <component
        class="com.prancingdonkey.service.DataPeristenceService"/>
    <vm:outbound-endpoint path="report.generation.stage.3"/>
</flow>

<flow name="processExpenseReportsStage3">           ← Third processing stage
    <vm:inbound-endpoint path="report.generation.stage.3"/>
    <component
        class="com.prancingdonkey.service.DataWarehousingService"/>
    <jms:outbound-endpoint topic="report.generation.complete"/>
</flow>
```

Now that the flow has been decomposed into three discrete flows, you have the ability to tune each individually. You also buy yourself some resiliency if any of these flows becomes a bottleneck. Processing is still able to happen in the other two flows. You could also introduce transactions between the flows, facilitating the reliability patterns discussed in chapter 7.

CONCURRENCY, SEDA, AND MULE APPLICATIONS Make sure any component code you're using with Mule is thread-safe. Mule applications, particularly fully asynchronous ones, almost guarantee that multiple threads of execution will be running through your components under load. If you know your code isn't thread-safe, consider using a pool of component objects rather than a singleton.

Now let's take a look at how you can tune the thread pools used in Mule's SEDA implementation.

11.2 Understanding thread pools and processing strategies

Mule is designed with various thread pools at its heart. Each thread pool handles a specific task, such as receiving or dispatching messages or invoking message processors in a flow. 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 and hence an impact on performance.

Processing strategies dictate how a message is passed off between these pools. The processing strategy is determined by the exchange-pattern of the endpoints in a flow, whether or not the flow is transactional or the processingStrategy attribute on the flow is configured.

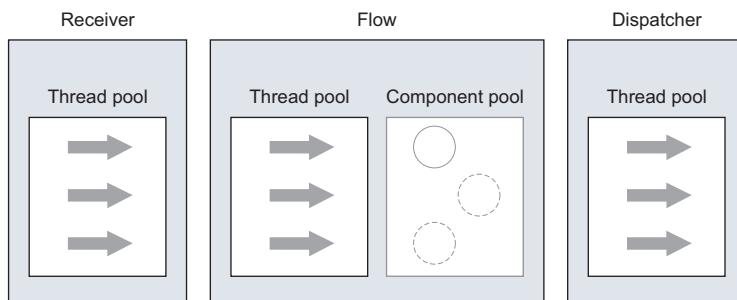


Figure 11.2 Mule relies on three thread pools to handle message events before, inside of, and after each flow.

WARNING Don't mix up your thread pools! The upcoming figures and discussions will help you tell them apart.

Let's start by digging into the different thread pools that exist in Mule. Figure 11.2 represents the three thread pools that can be involved when a flow handles a message event. It also represents the component pool that may or may not have been configured for any components in the flow (see the discussion in section 6.1.5).

UNDERSTANDING THE THREAD POOL FIGURES In the coming figures, pay attention to the thread pool box. If it's full (three arrows represented), that means that no thread from this pool is under use. If a thread is used, its arrow is filled in (black) and moved out of the thread pool box into the stage in which 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, whereas the flow thread pool is specific to the flow. This is illustrated in figure 11.3. A corollary of this fact is that if you want to segregate receiver and dispatcher pools for certain flows, you have to configure several connectors.

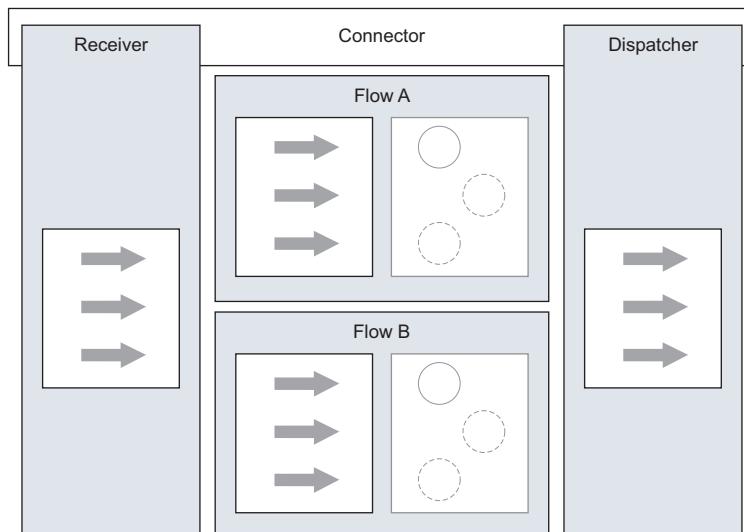


Figure 11.3 Two flows using the same connector share the same receiver and dispatcher thread pools.

If you “zoom” in on a thread pool, you discover that there’s a buffer alongside the threads themselves, as represented in figure 11.4. 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. You’ll see more options in section 11.2.3.

You now have a better understanding of the overall design of the Mule threading model. Next, we’ll look at how the synchronicity of a message event impacts the way thread pools are used.

11.2.1 Processing strategies and synchronicity

At each stage of message event processing, Mule decides if it needs to borrow a thread from the corresponding pool or not. A thread is borrowed from the receiver or the dispatcher pool only if this stage is set to be asynchronous. By extension, the flow thread pool is used only if the message source’s exchange-pattern is one-way. There are several factors that determine if the receiver or the dispatcher will be synchronous or not:

- *Configured exchange pattern*—If you’ve set an inbound or outbound endpoint’s exchange pattern to be request-response, the receiver or dispatcher will be request-response, respectively.
- *Incoming event*—If the received message event is synchronous, a receiver can act synchronously even if its inbound endpoint is configured to be one-way.
- *Outbound message processor*—The final message processor can enforce the dispatching stage to be synchronous.
- *Transaction*—If the flow is transactional, its receiver and dispatcher will be synchronous no matter how the endpoints are configured.

If you look at figure 11.5, you’ll notice that when both the receiver and the dispatcher have one-way exchange patterns, a thread is borrowed from each pool at each stage of the message processing. This configuration fully uses the SEDA design and therefore must be preferred for flows that handle heavy traffic or traffic subject to peaks. This configuration can’t be used if a client is expecting a response from the flow. Table 11.1 summarizes the pros and cons of this configuration model.

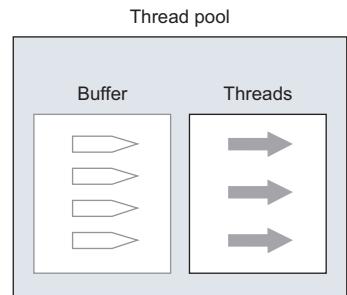


Figure 11.4 Each thread pool has a dedicated buffer that can accumulate pending requests if no thread is available to process the message.

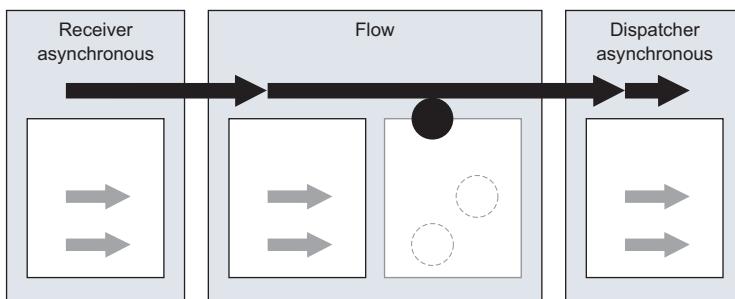


Figure 11.5 In fully asynchronous mode, a thread is borrowed from each pool at each stage of the message processing.

Table 11.1 Pros and cons of fully asynchronous mode

Pros	Cons
Uses SEDA model with three fully decoupled stages	Flow can't return a response to the caller
Highly performant	Complicates integration with clients that need a response

When a flow needs to return a response to the caller, its message source is configured to be request-response. It's still possible to use the dispatcher thread pool by configuring the exchange-pattern on the outbound endpoint to be one-way. The threading model of this configuration is shown in figure 11.6. Notice how the receiver thread is used for calling the message processors in the flow; this makes sense when you consider that the last processor in the flow will return the result to the caller on that thread. Table 11.2 presents the pros and cons of this approach.

Table 11.2 Pros and cons of the synchronous-asynchronous mode

Pros	Cons
Returns the component response to the caller Dispatching decoupled from receiving and servicing	Receiver thread pool can still be overwhelmed by incoming requests

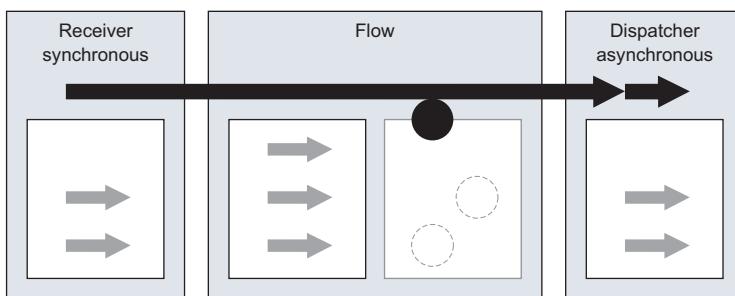


Figure 11.6 If only the receiver is synchronous, one thread from its pool will be used up to the component method invocation stage.

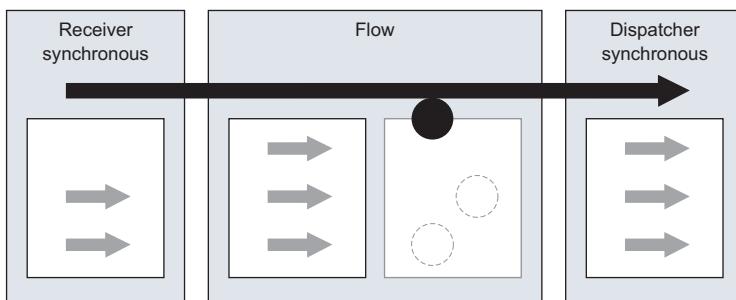


Figure 11.7 In fully synchronous mode, the receiver thread is used throughout all the stages of the message processing.

As discussed in the introduction, a transactional flow will automatically have its receiver and dispatcher running synchronously. Transactions, as you saw in chapter 9, guarantee that message processing in a flow happens in the correct order. To ensure this, transaction processing needs to occur in the same thread. Figure 11.7 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 11.3.

Table 11.3 Pros and cons of the fully synchronous mode

Pros	Cons
Supports transaction propagation	All load handled by the connector's receiver threads
Simple to implement	Doesn't use SEDA queueing

As noted earlier, using an outbound endpoint with a request-response exchange will cause the dispatcher to act synchronously. In that case, an asynchronously called flow can end up dispatching synchronously, using a thread from the flow thread pool for that end. This is illustrated in figure 11.8. This can also occur if the flow uses the event context (see section 12.1.1) to perform a synchronous dispatch programmatically. Look at table 11.4 for a summary of the pros and cons of this threading model.

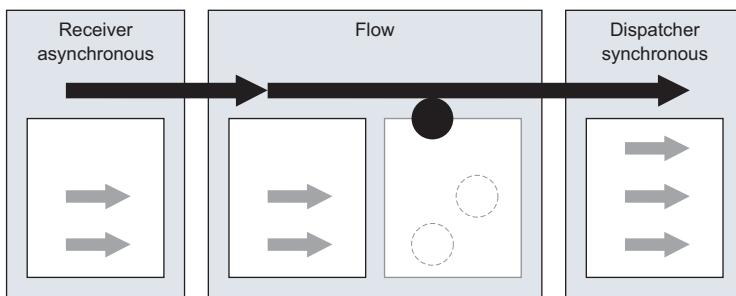


Figure 11.8 If only the dispatcher is synchronous, the receiver thread is used up to the flow in which a thread is taken from the pool and used throughout the dispatcher.

Table 11.4 Pros and cons of the asynchronous-synchronous mode

Pros	Cons
Allows synchronous routing in the dispatcher (as with a chaining router)	Can't return the component response to the caller
Receiving decoupled from dispatching and servicing	Flow thread pool size constrains the outbound dispatching load capacity

BEST PRACTICE Use the request-response message exchange pattern on endpoints sparingly and only when it's justified.

As you've learned, the default usage of the thread pools that can be involved in the processing of a message depends on the message exchange patterns being used and whether or not the flow is transactional. There will be times, however, when you'll want to override this behavior, for instance, to force an otherwise asynchronous flow to be synchronous. You'll see how to accomplish that in section 11.3. We'll now look at how transports can also directly influence the thread pools' usage.

11.2.2 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 11.9. It's possible for a message event to be fully processed in a flow without using any thread from any of its existing pools. Why? 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 flow by piggybacking the same thread across flows. This allows, for example, a message to be transactionally processed across several flows by the same thread.

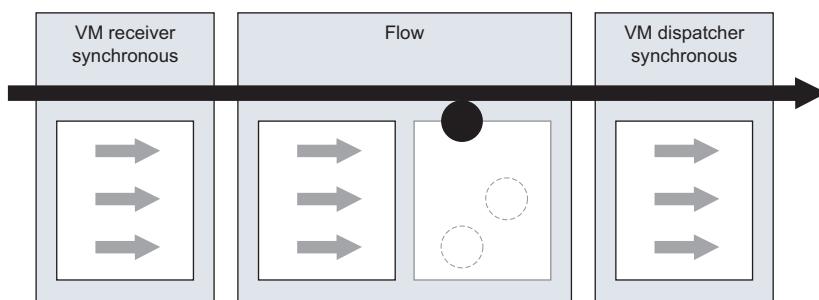


Figure 11.9 A fully synchronous flow using the VM transport can piggyback an incoming thread for event processing.

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 11.10. Because 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 11.11. With the JMS transport, for instance, 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 11.12. For example, the TCP and HTTP transports can handle this situation gracefully by stacking incoming requests in their specific backlogs. 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.

As you can see, transports can influence the way thread pools are used. The fact 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.

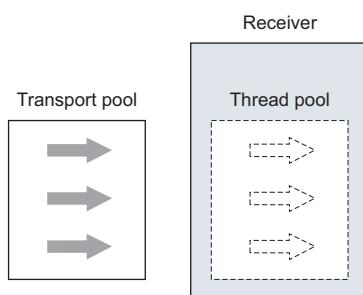


Figure 11.10 Some transports have their own receiver thread pool handled outside of Mule's infrastructure.

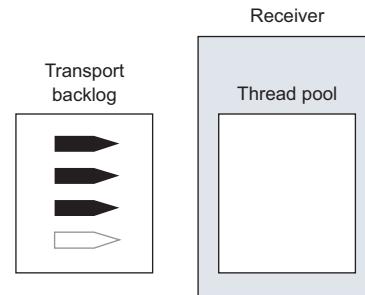


Figure 11.12 Some transports support the notion of a request backlog used when the receiver thread pool is exhausted.

² The pool of JMS dispatcher threads is still fully controlled by Mule.

Let's now look at the different options Mule gives you for configuring these thread pools.

11.2.3 Tuning thread pools

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 flow-level). See figure 11.13.

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 flow) 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 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 flow should never reject any request? You'd then override this Mule-wide default with a flow-level thread pool configuration, as shown in the following excerpt:

```
<flow name="CriticalSection">
    ...
    <threading-profile
        maxBufferSize="100" maxThreadsActive="20"
        maxThreadsIdle="10" threadTTL="60000"
        poolExhaustedAction="RUN" />
</flow>
```

With this setting, the flow 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,

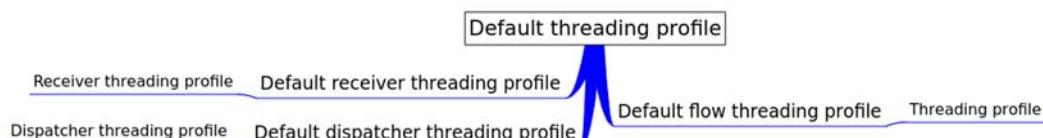


Figure 11.13 Thread pools are configured via a hierarchy of profiles whose scope goes from the most generic to the most specific profile.

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 you have to duplicate all the values defined in the global threading configuration, as there's no way to inherit individual setting values.

Besides `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 timeout occurs, and `DISCARD` and `DISCARD_OLDEST`, which silently drop the incoming event or the oldest one in the buffer, respectively.

You're now probably wondering this: how do I size all these different thread pools? MuleSoft 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 timeout 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 be better off most of the time leaving the default values in place. 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 to do so.

An example of JMS transport tuning

Prancing Donkey has to deal with peaks of activity coming from batch processes happening in the existing systems of one of its bottling suppliers. To abide by their SLA, they have to ensure that they can process a batch of 10,000 messages within 30 minutes. Because these messages are sent over JMS and they use Mule's standard receiver⁴ to consume them transactionally, they 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. They need only to configure the number of JMS concurrent consumers. A load test allowed them to measure that they can sustain an overall message process time of four seconds. A simple computation, similar to the ones explained in Mule's thread pool sizing methodology, gives a minimum number of concurrent consumers of $10000 \times 4 / (30 \times 60) = 22.22$. Margin being one of the secrets of engineering, they've opted for 25 concurrent consumers and, since then, have not broken their 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 12.2.2), use a specific connector for this TCP endpoint so it'll have dedicated thread

³ See <http://mng.bz/zpr2> for more information.

⁴ By which we mean `org.mule.transport.jms.MultiConsumerJmsMessageReceiver`, not one of the polling message receivers that are also available.

pools. That way, in case the TCP transport gets overwhelmed with messages, you'll still be in a 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 flow thread pool size. It's easy to define a global default thread pool size and forget to size a component pool accordingly.
- *Pooled components decrease throughput.* Pools of components are almost always less performant than singletons. Under load there will be contention of threads for objects in the pool. There's also overhead in growing the pool. Use singleton components unless you have a good reason not to.
- *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's taken out of a pool, your main goal is to have this thread back in the pool as fast as possible. Let's see how you can tune processing strategies to accomplish this.

11.2.4 Tuning processing strategies

Processing strategies, like thread pools, are also configurable. We discussed in section 11.2.1 how a processing strategy for a given flow is largely determined by the exchange patterns of the endpoints and whether or not a transaction's in place. It's possible, however, to explicitly set what the processing strategy for a given flow should be. This is done by setting the `processingStrategy` attribute on a flow, as illustrated next:

```
<flow processingStrategy="synchronous">
    <vm:inbound-endpoint path="in" exchange-pattern="one-way"/>
    <vm:outbound-endpoint path="out" exchange-pattern="one-way"/>
</flow>
```

By explicitly setting the `processingStrategy` for the above flow to `synchronous`, you tell Mule to bypass the default queued asynchronous behavior and use a single thread per every request.

Something we've glossed over until now is the flow thread pool. By now, hopefully it's clear that a pool of threads exists in a receiver thread pool to consume messages and, optionally, a dispatcher thread pool to send messages. For message processors in between the message source and the outbound endpoint, Mule will borrow a thread from the flow thread pool to execute all message processors in that flow for a given message. This means all flow processing besides message reception and message dispatch occurs in a single thread.

Often, this behavior isn't desirable. You saw an example at the beginning of this chapter of how the VM transport can be used to decompose message processing in a flow. That approach gives you the maximum amount of flexibility in that you have full control over tuning the receiving flow's thread pools and processing strategies. Sometimes, however, you want to parallelize work across cores in your flow, but you don't necessarily want to decouple everything with VM queues. In such situations, you can use the `thread-per-processor-processing-strategy` and the `queued-thread-per-processor-processing-strategy`.

The `thread-per-processor-processing-strategy` is the simpler of the two and ensures that, after the message source, each message processor in the flow processes the message in a separate thread. The `queued-thread-per-processor-processing-strategy` behaves identically, except that an internal SEDA queue is placed between each message processor, allowing you to introduce a buffer to hold messages when the flow thread pool is exhausted.

The following listing provides an alternative approach to tune listing 11.1 from the beginning of this chapter. Instead of using the VM transport, you use the `queued-thread-per-processor-processing-strategy` to ensure that the processing handled by each component of the flow occurs in its own thread.

Listing 11.3 Report processing in a single flow

```
<queued-thread-per-processor-processing-strategy  
    name="queuedPerProcessorStrategy"  
    maxThreads="16" minThreads="4" threadTTL="5000"  
    poolExhaustedAction="WAIT" threadWaitTimeout="5000"  
    maxBufferSize="10000"/>  
  
<flow name="processExpenseReports"  
processingStrategy="queuedPerProcessorStrategy">  
    <jms:inbound-endpoint queue="report.processing"  
                           exchange-pattern="one-way"/>  
    <component  
        class="com.prancingdonkey.service.FraudDetectionService"/>  
    <component  
        class="com.prancingdonkey.service.DataPersistenceService"/>  
    <component  
        class="com.prancingdonkey.service.DataWarehousingService"/>  
    <jms:outbound-endpoint topic="report.generation.complete"/>  
  
</flow>
```

Define queued-thread-per-processor-processing-strategy

←
Declare flow's processingStrategy

Listing 11.3 is a little bit different from the previous declaration of a flow's processing strategy. In this case, you explicitly define the `queued-thread-per-processor-processing-strategy`, set some tuning parameters for it, and reference it from the flow. Similar namespace elements exist for the other processing strategies. At the time of writing, they include `queued-thread-per-processor-processing-strategy`, `thread-per-processor-processing-strategy`, `asynchronous-processing-strategy`, and `queued-asynchronous-processing-strategy`.

Declaring a global processing strategy using one of these elements provides two benefits. First, you can configure the processing strategy once and reference it from multiple flows. Second, you have the ability to tune the processing strategy using the attributes enumerated in table 11.5.

Table 11.5 Configuring a processing strategy

Name	Description
maxBufferSize	The maximum number of messages to queue when the thread pool is at capacity
maxThreads	The maximum number of threads that can be used
minThreads	The minimum number of threads to keep ready when the flow is idle
poolExhaustedAction	What to do when the thread pool runs out
threadTTL	How long to keep an inactive thread in the pool prior to evicting it
threadWaitTimeout	How long to wait for a thread when the <code>poolExhaustedAction</code> is <code>WAIT</code>

CUSTOM PROCESSING STRATEGIES You can write your own processing strategies by implementing `org.mule.api.processor.ProcessingStrategy` and defining a `custom-processing-strategy` element.

Processing strategies, combined with the receiver and dispatcher thread pool tuning you saw in the previous section, give you almost complete control over how Mule allocates threads for processing performance. This level of control often isn't necessary; Mule's out-of-the-box defaults for both receiver and dispatcher pools, as well as its processing strategy assumptions based on endpoints, are typically good enough for most scenarios. For high-throughput use cases, or use cases with special considerations such as slow consumers or performance-sensitive service components, some level of thread pool or processing strategy tuning is unavoidable.

In this section, you've seen approaches to tuning Mule and structuring Mule flows to achieve throughput. But knowing about these techniques isn't worth anything unless you know where to apply them. Let's take a look at how you can identify performance bottlenecks in Mule applications.

11.3 Identifying performance bottlenecks

Although threading profiles and processing strategies define the overall capacity of your Mule instance in terms 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 increases under load, you can end up

in a position in which your thread pools will be exhausted no matter how many threads or how big a buffer you use.

Therefore, fine-grained performance matters. On the other hand, we all know Donald Knuth's words of caution:⁵

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

Should performance optimization be avoided entirely? No. Premature optimization is the issue. This encourages following 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 that subject (hopefully) covered. In this section, we'll first talk about gathering metrics to pinpoint performance pain. Then we'll give some advice about what you can do 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.

11.3.1 Profiler-based investigation

The most convenient way to locate places in an application that are good candidates for optimization (a.k.a. *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 transparent when it comes to profiling. To make things even easier, you can download a free Profiler Pack from www.mulesoft.org that contains the libraries required to profile a Mule instance with YourKit (<http://yourkit.com>). You still need to own a valid license from YourKit in order to analyze the results of a profiling session.

The way you activate the profiler depends on the way you deploy and start Mule (see section 8.1). With a standalone deployment, adding the `-profile` parameter to the startup command does the trick. If you deploy Mule in a Java EE container or bootstrap it from an IDE, you'll have to refer to YourKit's integration guidelines to have the profiler active.

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. You can use one of the test tools we'll discuss in section 12.4 or create your own ad hoc activity generator if you have specific or trivial needs. Figure 11.14 shows YourKit's memory dashboard while profiling the LoanBroker sample application that ships with Mule. This demonstration application comes with a client that can

⁵ From "Structured Programming with go to Statements," ACM Journal Computing Surveys. <http://dl.acm.org/citation.cfm?id=356640>.

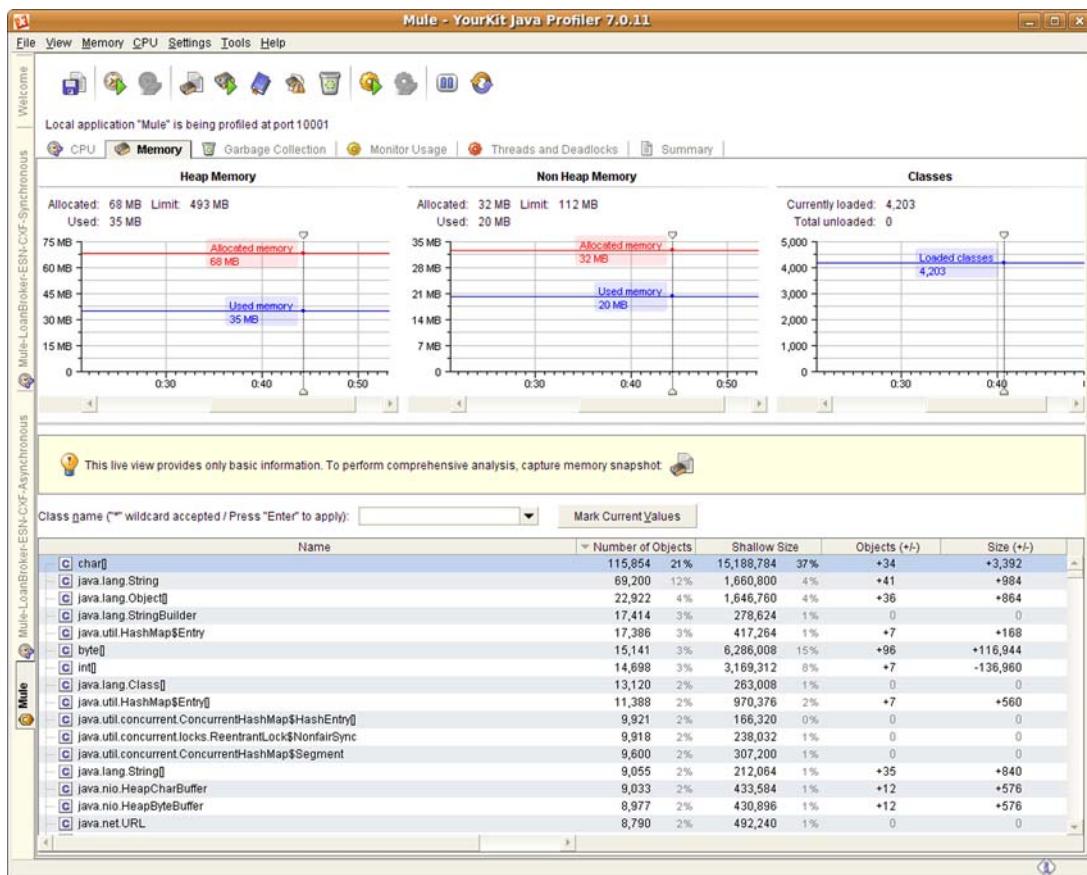


Figure 11.14 Profiling Mule’s memory with YourKit while messages are being processed

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*—Lead to memory leaks
- *Monitor locks*—Excess can create contentions between threads
- *Deadlocks*—Can take down a complete application
- *Excessive objects or threads creation*—Can lead to unexpected memory exhaustion

Once you’ve identified areas of improvement, it’ll become possible to take corrective action. As far as profiling is concerned, a good strategy is to perform differential and incremental profiling sessions. This consists of 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 make one change at a time and reprofile after each change; otherwise, you'll have a hard time telling what the cause was for an improvement or a degradation.

The LoanBroker sample application that 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 11.15, confirm the impact of the configuration change: the asynchronous version depends more on inbound routing and filtering, whereas the synchronous one uses the response routers for quotes aggregation.

In-application profiling

Using a profiler is great to identify bottlenecks during development or load testing. Having on-demand and historical performance data is equally critical. A great tool for instrumenting Java applications for live profiling data is Yammer's Metrics: <http://metrics.codahale.com/>. This framework contains profiling tools such as gauges, meters, timers, and histograms to monitor various pieces of data from your applications.

Mule Management Console, available with a Mule EE subscription, also provides mechanisms to display historic performance data from your Mule applications.

YOURKIT COMPATIBILITY The Profiler Pack and the Profiler 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 are shipped with your version of YourKit in lieu of the ones distributed by MuleSoft.

Using a profiler is the best way to identify performance-challenged pieces of code. But whether you use a profiler or not, there are general guidelines that you can apply to your Mule projects. We'll go through them now.

11.3.2 Performance guidelines

Whether you've used a profiler or not, this section offers a few pieces of advice about coding and configuring your Mule applications for better performance. You'll see that some of this advice is generic, whereas some is 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

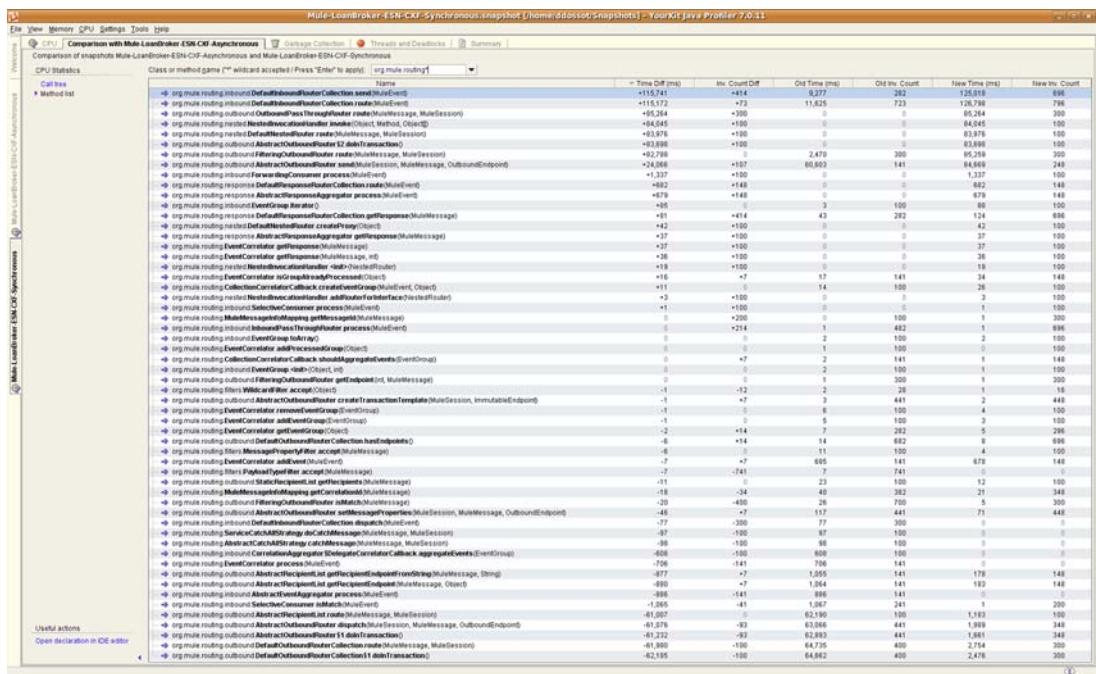


Figure 11.15 Comparing two profiling session snapshots helps determine the impact of configuration changes.

as being different from, say, a standard Java EE one because far 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.
 - Avoid generating useless garbage (favor singleton-scoped beans over prototypes).

REDUCE BUSYWORLD

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 12.3.3), 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 will ignore it. Doing so will also save you the time spent in the component interceptor stack and the entry point resolvers.

BEST PRACTICE Investigate high-performance message processors such as SXC or Smooks if you intend to perform processing or transformation on messages with huge payloads.

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, and these strategies mostly depend on the transport you use. Here's an incomplete list of possible options to consider:

- 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 file content needs to be read.
- Some transports can receive incoming request 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 11.2.1), 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 use an existing transformer to deal with the stream serialization (see section 4.3.1).
- The Claim Check pattern (documented here: <http://eaipatterns.com/StoreInLibrary.html>) can be used to store a reference to the message payload in an external source. This allows it to be loaded on demand rather than being passed around in the message. An example of this would be to use a database row ID as the payload of the message and use this to selectively load the data, or parts of the data, into Mule when it's necessary for processing.

TUNE TRANSPORTS

As we discussed in section 11.2.2, transports have their own characteristics that can influence threading and, consequently, performance. The general advice about this is that you should know the transports you use and how they behave. You should look for timeout 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 deployment over the default HTTP transport.

TUNE MULE'S JVM

Finally, because Mule is 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 and tuning its garbage 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;⁶ 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 advice you can follow to remedy these issues.

11.4 Summary

In this chapter, we investigated the notions of thread pools and performance optimization. We 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 gave you a handful of general tips for better use of your system's available resources (memory and CPU).

Part 1 of this book covered the basics of Mule: working with flows, using connectors and components, and routing and transforming data. That laid the groundwork for part 2, designed to give you confidence in deploying, securing, and tuning Mule applications, and handling errors. In part 3 we'll turn our attention to Mule's API, which lets you go "under the hood" of Mule applications and fully harness its flexibility.

⁶ Your Mileage May Vary.

Part 3

Traveling further with Mule

Although all the fundamentals of Mule have been covered in parts 1 and 2, we're not 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.

Chapter 12 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. We'll also discuss your different options when it comes to thoroughly testing your integration application.

MuleSoft has made it very easy to extend Mule by providing a toolkit (DevKit) that streamlines the creation of custom connectors for Mule. In chapter 13, you'll learn how to create a feature-rich connector using this toolkit.

We'll close this book with a review of the techniques that can be used to run long-lasting business processes and scheduled tasks on Mule. Chapter 14 will also cover two popular approaches for enhancing the business capacities of Mule: complex event processing and business rules execution.

Developing with Mule

This chapter covers

- The Mule context in depth
- How to interact with Mule from code
- Mule's internal API
- Testing tools and practices
- Debugging tools and practices

The previous 11 chapters of this book should've convinced you that there's a lot that can be achieved with Mule with minimal coding, or at least without writing any Mule-specific code. For example, you'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 the disadvantages of 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 2500 classes in Mule, this chapter can by no means be an exhaustive tour of all of them; instead, it 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’ll look at it as a gateway to Mule’s live moving parts. You’ll see how Prancing Donkey makes good use of the API to implement behaviors in their applications that couldn’t be implemented by configuration alone. Indeed, as an expert Mule user, Prancing Donkey uses most of the internal API-related features we’ll cover in this chapter, from message interception to notification listening. Learning how they implement such advanced features will give you a better and deeper insight into how things work inside of Mule.

As your familiarity with the Mule API grows, you’ll start producing more complex applications, increasing your need to apply to your Mule projects all the tools and practices you’re using in your standard software development projects. This is why we’ll also focus, in the second part of this chapter, on development-related activities. You’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. We’ll conclude by taking a look at some debugging techniques you can use while developing your applications.

This chapter is one of the most in-depth of the book; it will take time for its content to sink in, and you’ll surely come back to it several times to fully grok it. Because it covers everything development with Mule, it may feel like a mixed bag of concepts, and to some extent that’s what it is. But as you progress in your understanding of Mule internals and capacities, you’ll progressively realize that all the concepts covered here are tightly intermingled. With this said, let’s start by discovering the Mule context, a useful part of the API that you’ll happily start mining soon!

12.1 **Understanding the Mule context**

In previous chapters of this book, you’ve read about the Mule context. You’ve certainly understood that it’s an in-memory handle to a running Mule application. As figure 12.1 shows, each Mule application that runs is assigned a Mule context.

The Mule context that’s used by a Mule application is created when it’s bootstrapped. Mule supports multiple bootstrapping options (as discussed in chapter 8), which include these:

- The application deployer of the Mule standalone server
- The Mule servlet context listener of a WAR-packaged Mule application
- Custom Java bootstrapping code
- A functional test case (see section 12.4)
- The application launcher of Mule studio

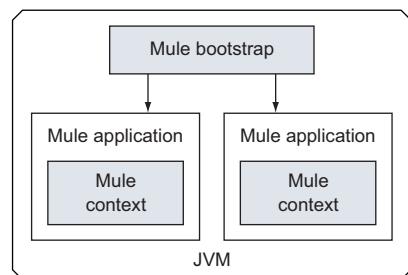


Figure 12.1 Each Mule application has a specific Mule context.

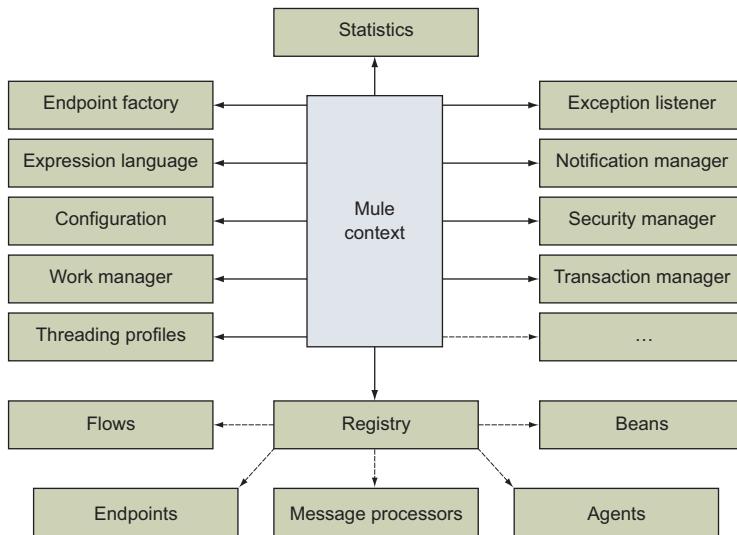


Figure 12.2 The Mule context gives access to a Mule application's internals.

ONCE THERE WAS A SINGLETON In the versions of Mule before 3, the Mule context was a singleton. With the introduction of the Mule application model (see section 8.1) in Mule 3, the Mule context couldn't be a singleton anymore in order to support multiple applications in the same JVM.

Figure 12.2 is a partial dependency graph that represents some of the Mule moving parts you can reach from the Mule context. As you can see, once you get ahold of the Mule context, it's party time: all of Mule's internals become accessible! This figure is far from a complete representation of the reality; if you look at the Javadoc of the `org.mule.api.MuleContext` interface and follow some of the classes it links, like the registry, you'll find far more objects than are 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.

But before we dig deeper into the Mule context, we need to answer this crucial question: how can one get access to this treasure trove?

12.1.1 Accessing the Mule context

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 was demonstrated in section 8.4.
- *Get it from the client*—If you use the client to bootstrap a Mule instance, you can ask it for the context it has created behind the scenes. We'll use this approach in listing 12.12.
- *Get it from the event context*—Components implementing `org.mule.api.lifecycle.Callable` receive an event context that, among other things, gives direct access to the Mule context (see section 6.1.1).

- *Receive it by injection*—If you write a custom class that implements org.mule.api.MuleContextAware and is instantiated by Spring or Mule, you’ll receive a reference to the current Mule context before it gets initialized (see section 12.3.1). Alternatively, you can also have the context injected by annotating an org.mule.api.MuleContext field with either @Inject (standard JSR 330 annotation) or @Lookup (Mule annotation).
- *Get it from the servlet context*—When running Mule deployed as a web application (see section 8.2), the Mule context is accessible in a servlet context attribute named mule.context.
- *Use it as a script variable*—Mule binds the context in the evaluation context of all the scripts it evaluates under the name muleContext.

If at all possible for your particular use case, prefer receiving the Mule context by injection.

WHAT ABOUT MEL? The Mule context isn’t directly accessible in MEL, but much of the configuration information it contains and the registry are bound as context objects. Check out appendix A to find out more about these objects.

Let’s now review some of the actions you can perform once you have a reference to the context.

12.1.2 Using the Mule context

In this section, we’ll review four actions that the Mule context allows you to perform:

- Controlling a Mule instance
- Getting the active configuration of a Mule application
- Accessing the application’s statistics
- Looking up objects in the registry

Let’s start with controlling a Mule instance.

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();
```

SUSPENDED ANIMATION 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().

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 object of type `org.mule.api.config.MuleConfiguration`. Though the vast majority of all configuration values you'll handle will come from XML or properties files and will be directly injected into your objects, some values are computed by Mule itself and made available via the configuration object.

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 the next listing.

Listing 12.1 Using the Mule context to access the system configuration

```
String serverId = muleContext.getConfiguration().getId();  
  
ObjectName listenerObjectName = ObjectName  
    .getInstance(JmxSupport.DEFAULT_JMX_DOMAIN_PREFIX + ".  
        + serverId + ":"  
        + JmxServerNotificationAgent.LISTENER_JMX_OBJECT_NAME);
```

ACCESSING STATISTICS

Mule keeps detailed statistics for all its moving parts. In an instance with numerous flows, these statistics are a convenient means to keep track of the activity of a Mule application. Prancing Donkey uses a custom component to regularly dump the full statistics of an application into XML files. The logic inside this component, which is shown in the next listing, is simple thanks to the supporting classes offered by Mule.

Listing 12.2 Using the Mule context to access the global statistics

```
final AllStatistics allStatistics = muleContext.getStatistics();  
allStatistics.logSummary(new XMLPrinter(xmlStatisticsWriter));
```

LOOKING UP THE REGISTRY

One of the richest objects the context gives you access to is the registry, where, as its name suggests, all of 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 that delegates to a read-only Spring bean factory. This means that you can access all Mule moving parts and your own Spring beans through the registry.

In terms 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 and hence can't be statically configured. This is the case for the XML statistics component at Prancing Donkey. As we said before, this component can output the whole of the statistics of an application, but it can also do it for only

one flow, if its name has been sent to the component. Because of the dynamic nature of this behavior, you can use the registry to look up the flow by name and then render its statistics, as shown next.

Listing 12.3 The registry gives access to all Mule's moving parts.

```
final FlowConstruct flow = muleContext.getRegistry()
    .lookupFlowConstruct(flowName);

FlowConstructStatistics flowStatistics = flow.getStatistics();
new XMLPrinter(xmlStatisticsWriter)
    .print(Collections.singleton(flowStatistics));
```

There's another situation in which you would prefer lookup over injection: when no configuration element can be referenced (or injected). Let's consider the `jmx-default-config` configuration element that automatically configures a bunch of agents. This element doesn't give you a handle for each agent it creates, so if you need to reach one of these agents, as shown in the following listing, you'd need to perform a registry lookup.

Listing 12.4 JMX agents can be looked up in the registry.

```
JmxAgent jmxAgent = (JmxAgent) muleContext.getRegistry()
    .lookupAgent("jmx-agent");

MBeanServer mBeanServer = jmxAgent.getMBeanServer();
```

As suggested by figure 12.2, the Mule context gives access to the innards of the whole application. The few examples we've just gone through should have 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.

Registry reloaded

It's possible to store your own objects in the registry with the `registerObject` method and to retrieve them with `lookupObject`. You can also register objects from the XML configuration using the `global-property` element.

Though thread-safe, the registry shouldn't be mistaken for a general-purpose object cache. If that's what your application needs, you'd do better to use a dedicated one (like Ehcache or Hazelcast) and have it injected into 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 of Mule's moving parts configured in the XML file.

With the Mule context in hand, you can do many things, such as interacting with a Mule application using the Mule client. But what is the Mule client? Let's learn more about it.

12.2 Connecting to Mule

As you'll see in the coming sections, the main purpose of the Mule client 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, and that instance can be used to directly tap into the transports infrastructure of Mule.

The Mule client provides a rich messaging abstraction and complete isolation from the transports' particularities. The client supports the three main types of message interactions that are supported by Mule (refer to section 2.2 for the complete list of interactions):

- *Send message*—A synchronous operation that waits for a response to the message it sent (relies on the message dispatcher infrastructure). This corresponds to the request-response message exchange pattern (MEP).
- *Dispatch message*—An asynchronous operation that expects no response for the message it dispatched (also relies on the message dispatcher infrastructure). This corresponds to the one-way MEP.
- *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 interactions are offered under numerous variations that allow you, for example, to directly target a component instead of an inbound endpoint, receive a future¹ message, or disregard the response of a synchronous call.

TRANSACTIONAL CLIENT—It's possible, but not trivial, to involve several operations of the Mule client in a single transaction, provided that the transports used are transactional. This is seldom necessary and won't be detailed here. You can study `org.mule.test.integration.client.MuleClient-TransactionTestCase` for more information.

Depending on the way you gain access to the Mule client, you will work with either of these:

- An instance of the `org.mule.api.client.MuleClient` interface
- An instance of the `org.mule.module.client.MuleClient` class, which doesn't implement the `org.mule.api.client.MuleClient` interface, but offers the same API

¹ As in `java.util.concurrent.Future`, i.e., the pending result of a computation that can be waited on or polled regularly for a result.

Let's start by discovering how the client can be used to interact with a local Mule instance.

12.2.1 Reaching a local Mule application

The most common usage of the Mule client is to interact with an already running local Mule application, that is, an application that runs within the same JVM. This is illustrated in figure 12.3.

In-memory calls from the client to a local Mule application are common in integration testing (see the discussion in section 12.2). The majority of the examples that accompany this book use the Mule client, for that matter. Listing 12.5 shows an excerpt of a method that Prancing Donkey uses to run integration tests on a flow that attaches PDF files to a Mule message (we talked about this in section 2.3.3).

Listing 12.5 The client facilitates testing by granting access to a Mule application.

```
MuleClient muleClient = muleContext.getClient();

MuleMessage result = muleClient.send("vm://add-attachments.in",
    TEST_PAYLOAD,
    Collections.<String, Object>singletonMap(
        "pdfFilePath",
        usageReportPdfFilePath));

assertThat(
    result.getInboundAttachmentNames().contains(TEST_USAGE_REPORT_PDF),
    is(true));
```

Because this test runs in a class in which the Mule context is available, you retrieve an instance of the Mule client directly from the context. In this particular case, you use an instance of `org.mule.api.client.MuleClient` that's provided by the Mule context (it's an instance of `org.mule.api.client.LocalMuleClient`, which extends `org.mule.api.client.MuleClient` with a few methods).

You may have noticed that in listing 12.5 you send a message to a VM transport URI. You may wonder if you'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, we'll look at the test code used to exercise the flow shown in listing 2.11. Notice in listing 12.6 how you send over HTTP a message to the URI that the flow you want to test listens to.

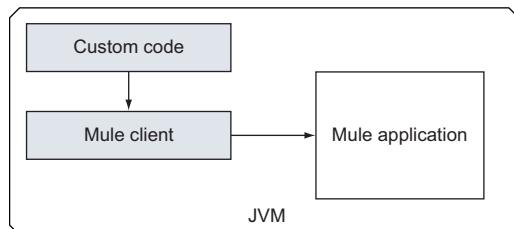


Figure 12.3 The client is a convenient way to reach a Mule application within the same JVM.

Listing 12.6 Flows can be reached over any protocol via the Mule client.

```
MuleMessage result = muleClient.send("http://localhost:8080/json/data",
    TEST_PAYLOAD, null);
assertThat((String) result.getInboundProperty("Content-Type"),
    is("application/json"));
```

You've discovered how the Mule client facilitates connectivity with a local application running in the same JVM. What if the Mule application 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 as well, as you'll now see.

12.2.2 Reaching a remote Mule application

The Mule client can connect to a remote application and use the same richness of interaction it does with a local one. For this to work, a specific remote dispatcher agent must be configured in the Mule application that the client wants to connect to, as illustrated in figure 12.4. Because the dispatcher agent is 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 for the representation of the data that's sent over the wire in both directions (the client request to Mule and its response).

Listing 12.7 shows the configuration of a Mule application 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 Mule-Messages (payload and properties) to and from byte arrays. The default wire format relies on standard Java serialization and hence can only carry serializable payloads.

Listing 12.7 A remote dispatcher agent using the default wire format over TCP

```
<client:remote-dispatcher-agent>
    <client:remote-endpoint
        address="tcp://localhost:5555"
        exchange-pattern="request-response" />
</client:remote-dispatcher-agent>
```

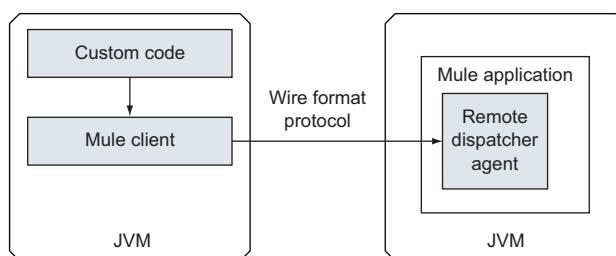


Figure 12.4 Using the client to send messages to a remote Mule application via a dedicated dispatcher agent

ONLY AT HOME Expose the remote dispatcher agent over TCP only in a secure and trusted environment, as it allows you to completely control your Mule instance without enforcing any security.

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 application. The client you've seen so far isn't able to do this directly, but must be used to spawn a specific client for the remote dispatcher agent (shown in the next listing).

Listing 12.8 The client is the factory for creating remote dispatchers.

```
MuleClient muleClient = new MuleClient(true);           ↪ Creating client running
RemoteDispatcher remoteDispatcher =                   empty Mule application
                                                     ①
muleClient.getRemoteDispatcher("tcp://localhost:5555");
FutureMessageResult asyncResponse =
remoteDispatcher.sendAsyncRemote(
"clientServiceChannel",
TEST_CLIENT_ID,
null);
```

**Sending message to
endpoint named
clientServiceChannel**

**Using client
to build
remote
dispatcher
for TCP URI**

②

In this case, you work with an instance of the `org.mule.module.client.MuleClient` class that's instantiated directly at ①. The client for the remote dispatcher agent, which is an instance of `org.mule.module.client.RemoteDispatcher`, is created at ②. This object is thread-safe.

In this example, you perform an asynchronous send to an endpoint named `clientServiceChannel` with a client ID as the message payload and no properties. Note that the `sendAsyncRemote` method does the same thing as `sendAsync` of the local Mule client. Other methods have slightly different names; for example, the methods that send directly to a component are suffixed with `ToRemoteComponent` instead of `Direct`.

Notice at ① how you create a Mule client without providing it a context (nor a configuration file to load) and asking it, with the `true` parameter, to start the empty context it will create. Because you're only interested in the remote Mule you connect to, there's no reason for the Mule client to try to interact with a local application.

Once you're done using the remote dispatcher, you can get rid of it by disposing of the Mule client that created it:

```
muleClient.dispose();
```

Note that disposing of a Mule client terminates all the remote dispatchers it could have created.

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.

All your clients are belong to us

The remote dispatcher is a convenient way to reach a remote Mule application 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 flows hosted on Mule. For example, you can still use an HTTP API to form a request to a flow hosted on Mule and exposed over HTTP. Or you can still use a JMS client to send to a destination consumed by a Mule flow.

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 need for the messaging abstraction the Mule client proposes.

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 is able to serialize any Java object to an XML representation. Listing 12.9 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 12.9 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="8080"
    path="_remoting"
    exchange-pattern="request-response" />
```

You surely observed in listing 12.9 how you create a global HTTP endpoint and reference it from the remote dispatcher agent. This is by no means a requirement; you could declare the HTTP endpoint on the `remote-endpoint` element, as in listing 12.7. The idea was to demonstrate that the remote dispatcher agent uses the standard transport infrastructure of the Mule instance that hosts it.

Now look at listing 12.10, which demonstrates how you create a client for this HTTP remote dispatcher agent. Except the URI that's changed, there isn't much difference from the previous code; where is 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 12.10 Creating a remote dispatcher to connect to a distant Mule over HTTP

```
MuleClient muleClient = new MuleClient(true);
RemoteDispatcher remoteDispatcher =
    muleClient.getRemoteDispatcher("http://localhost:8080/_remoting");
```

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`.

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. As 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 flows hosted by the remote Mule instance, as detailed in section 10.1.3. If you follow that path, you'll need to pass a username and a password to the remote dispatcher when it's created:

```
RemoteDispatcher remoteDispatcher =
    muleClient.getRemoteDispatcher(
        "http://localhost:8181/_remoting",
        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 application 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.

12.2.3 Reaching out with transports

In the opening discussion of this section, we said that the client is able to bootstrap a Mule application if no context is passed to it. This capability can be used to exploit Mule's transports by bootstrapping an empty instance and using the messaging abstraction directly offered by the client. This approach, illustrated in figure 12.5, truly promotes Mule as an integration framework on which an application can lean to connect to remote flows without having to deal with the particulars of the protocols involved.

Listing 12.11 shows the code that makes this happen. The client is started without providing it with a Mule context or configuration, so it bootstraps an empty application. This empty application can be used from the client to reach any URI whose related transport library is available in the classpath. It will also use any module that's

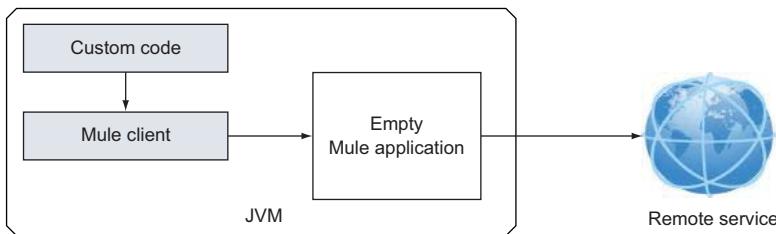


Figure 12.5 The client can bootstrap a minimal Mule instance to use its transport layer.

available in the classpath. In the case of this example, the HTTP transport and the WSDL connector of the CXF module are used in order to call a web service that performs currency conversions.

Listing 12.11 Bootstrapping an empty Mule application with the client

```

MuleClient muleClient = new MuleClient(true);           ↪ Creates empty Mule application
MuleMessage result = muleClient.send(                  ↪ and client connected to it
    "wsdl-cxf:http://www.webservicex.net/CurrencyConvertor.asmx?WSDL"
    +"&method=ConversionRate",
    new String[] {"CAD", "USD"},
    null,
    15000);
Double cadToUsd = (Double)result.getPayload();
muleClient.dispose();                                ↪ Disposes of client, releasing
                                                       ↪ all underlying resources
  
```

Performs synchronous send using HTTP transport and CXF module

Retrieves result of web service call

Such a simple setup is possible because neither the HTTP transport nor the CXF module needed specific configurations for them to work. Should the transport you plan to use need to be configured, you'll have to create a minimal Mule configuration and load it from the client. For example, this would be required if you use the JMS transport, as shown in listing 12.12. The raw-jms-muleclient-config.xml configuration file² still doesn't contain any flow.

Listing 12.12 Using Mule's transports and messaging layers via an empty application

```

MuleClient muleClient = new MuleClient("raw-jms-muleclient-config.xml");           ↪ Creates configured Mule application
muleClient.getMuleContext().start();                                                 ↪ and client connected to it
  
```

① Starts Mule instance

② Performs synchronous read using JMS transport

```

MuleMessage response = muleClient.request(
    "jms://" + queueName + "?connector=amqConnector", 1000);
  
```

² Available in the book's source code.

```
muleClient.getMuleContext().dispose();
muleClient.dispose();
```

③ Disposes of configured
Mule instance, then client

Because there's no constructor of the Mule client that loads a configuration and starts it, you have to do it explicitly at ①. As we showed before, you define the connector name you want to use in the request URI ②. This would allow you to configure several connectors in the same Mule configuration and select the one you want to use in the client code. The same way you start the instance explicitly, you have to dispose of it ③ before terminating the client.

Almost a full Mule

Mule started with the approach shown in listing 12.12 is functional but doesn't offer its full features. For example, handling classes that have Mule annotations won't be supported with this approach. As an example, if you'd like to have support for Mule annotations, you'd need to start it this way:

```
List<ConfigurationBuilder> configurationBuilders =
    Arrays.<ConfigurationBuilder>asList(
        new AnnotationsConfigurationBuilder(),
        new SpringXmlConfigurationBuilder("raw-jms-muleclient-config.xml"));

MuleContextBuilder muleContextBuilder = new DefaultMuleContextBuilder();
MuleContextFactory muleContextFactory = new DefaultMuleContextFactory();
MuleContext muleContext =
    muleContextFactory.createMuleContext(
        configurationBuilders,
        muleContextBuilder);

muleContext.start();

MuleClient muleClient = muleContext.getClient();
```

As you'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 has become an important tool in your Mule toolbox, as it's a convenient gateway to a Mule application, whether this application is running locally or in a remote JVM.

Let's continue our exploration of Mule internals, and see how advanced operations can be achieved by getting closer to the metal.

12.3 Using the Mule API

As an integration framework, Mule exposes a rich internal API that can be coded to in order to implement custom behaviors. Though throughout the book we advocated

against that approach, there are times when tying into Mule is necessary. In this section, we'll focus on the following features for which coding to Mule's API is a must:

- *Lifecycle events*—The API allows you to use lifecycle events that occur from startup time, when moving parts get instantiated, configured, and put into service, to shut down 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.
- *Data persistence*—Some Mule components need to persist data in the course of their normal activity; Mule defines an internal API that allows plugging in any persistence mechanism to store this data.

We won't look into creating custom message sources and processors (introduced in sections 2.2.1 and 2.2.2, respectively) by hand because it's a much better approach to generate such Mule extensions by using the DevKit, which we'll discuss in chapter 13. That said, as you'll see in this section, creating interceptors will expose you to the manual creation of an `InterceptingMessageProcessor`, which is both a message source and processor!

HOW MUCH COUPLING IS ENOUGH? Deciding to couple your application to the Mule API shouldn't be taken lightly, as is true with any framework. As a general rule, code to Mule's API only when it's necessary and not as a default way of building Mule applications. Use Mule-unaware POJOs as much as possible, and configure Mule to use them as-is instead of modifying them to fit in Mule.

Let's start by looking at the lifecycles of the main moving parts that constitute a Mule configuration and how your custom implementations can become aware of their own lifecycles.

12.3.1 Being lifecycle aware

Between the time you bootstrap a Mule application and the time it's up and ready, many things happen (as testified to by the numerous log file entries). Moving parts are created and made ready for service. 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 application; all moving parts are transitioned to their ultimate destruction through another series of lifecycle method calls.

You'll remember from our discussion in section 6.1.7 that your own custom components can benefit from these configuration and lifecycle method calls the same way Mule's do. This applies not only to components, but to all sorts of custom moving parts you can create and configure in Mule. Table 12.1 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 (`initialise`,³ `start`, `stop`, `dispose`), this table also shows the custom properties setters, the Mule-specific setters (`context` and `flow construct` injection), and the Spring-specific lifecycle methods.

Table 12.1 Configuration and lifecycle methods are honored differently depending on the object type.

Configuration and lifecycle methods	Prototype component (for each instance)	Singleton component	Spring Bean component	Any Spring Bean	Transformer	Filter	Custom message processor
Custom properties setters	✓	✓	✓	✓	✓	✓	✓
Mule context setter	✓	✓	✓	✓	✓	✓	✓
Spring init method	n/a	n/a	✓	✓	n/a	n/a	
Mule flow construct setter	✓	✓	✓	n/a			✓
<code>initialise</code>	✓	✓			✓	✓	✓
<code>start</code>	✓	✓	✓	✓	✓		✓
<code>stop</code>	✓	✓	✓	✓	✓		✓
<code>dispose</code>	✓	✓					✓
Spring destroy method	n/a	n/a	✓	✓	n/a	n/a	

REPEATED CALLS As of this writing, Mule happens to invoke `setMuleContext` twice in some circumstances, such as when initializing transformers. Therefore, don't place any logic except standard variable assignments in this setter if you want to avoid running it more than once.

How does Mule decide if it should call a particular lifecycle method or setter on a custom object? It 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
```

³ British English courtesy of Ross Mason, Mule's father!

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 implementing `org.mule.api.lifecycle.Lifecycle`, which extends the four standard ones.

And here are the interfaces for the Mule-specific setters:

```
org.mule.api.context.MuleContextAware  
org.mule.api.construct.FlowConstructAware
```

We mentioned implementing `MuleContextAware` in section 12.1.1 as a convenient way to get ahold of the Mule context. Flow construct awareness allows a component to access the flow element it's hosted by and all its internals: state, statistics, message source, and message processors.

STARTED BUT NOT READY When your components are started, don't assume that the whole Mule instance is up and running. Starting happens long 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 you'll see in section 12.3.3.

You're now able to make custom logic execute when lifecycle events occur in Mule. Let's discuss how to attach cross-cutting logic to components.

EXTENDS OVER IMPLEMENTS Always check if an abstract implementation is provided for a Mule interface; extending provided abstract classes will future-proof your application, as any interface change will be accounted for in the provided abstract class.

12.3.2 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; see <http://mng.bz/w8o5>).

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 `org.mule.api.MuleEvent` destined 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 the event to the component or, to be more accurate, to the next interceptor in the stack, as interceptors are always members of a stack (the last member of the stack is the component itself).

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 interceptor stack by storing the time before passing to the next member, and comparing it with the time when the execution flow comes back to it.

TO INTERCEPT OR NOT? Harness Mule's interceptors for implementing messaging-level, cross-cutting concerns, if you need to run before/after actions around the rest of the flow or if you want the ability to conditionally stop a flow mid-course.

Mule also comes with an abstract interceptor implementation named `org.mule.api.interceptor.EnvelopeInterceptor` whose behavior is limited to adding behavior before and after the invocation of the next member of the stack, but without the ability 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 logs a message before forwarding to the next one, and another message after the execution flow comes back to it.

Springing into AOP

If you're a savvy Spring user, you might be interested in using your Spring AOP skills with Mule. Using Spring AOP can indeed 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 Mule and Spring sides. For example, a Spring autogenerated AOP proxy 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 particularly true for plain POJO components that don't implement `org.mule.api.lifecycle.Callable`. Therefore, it's entirely possible that advising⁴ components can lead you to rework part of your configuration.

Listing 12.13 shows the interceptor stack that Prancing Donkey uses in front of components that are costly to call and whose results can be cached.⁵ The stack first defines the standard `timer-interceptor` that's used to record how efficiently the `custom-interceptor` is configured after it. The cache is provided by Ehcache (<http://ehcache.sourceforge.net>), itself configured and injected into the interceptor by Spring.

⁴ From AOP's "advice."

⁵ Note that, as of this writing, an independent caching module exists for Mule but isn't part of the standard distribution. Prancing Donkey will switch from their custom caching interceptor to a standard one when Mule provides it out of the box.

Listing 12.13 Prancing Donkey's caching interceptor stack defines two interceptors.

```
<interceptor-stack name="PayloadCacheInterceptors">
    <timer-interceptor />
    <custom-interceptor
        class="com.prancingdonkey.interceptor.PayloadCacheInterceptor">
            <spring:property name="cache" ref="ehCache" />
        </custom-interceptor>
    </interceptor-stack>
```

Note that an `org.mule.api.interceptor.Interceptor` is an `org.mule.api.processor.InterceptingMessageProcessor`, which itself is both an `org.mule.api.processor.MessageProcessor` and an `org.mule.api.source.MessageSource` (concepts that were introduced in section 2.2). Being a message source is necessary for the interceptor to be able to stop further processing; it can do that by not calling the next interceptor in the chain, for which it acts as a message source. Other than that, the `PayloadCacheInterceptor` implementation itself is fairly trivial,⁶ as shown in the following listing. The implementation is concise because all of the heavy lifting is done by Ehcache.

Listing 12.14 The cache interceptor uses Ehcache to store and replay payloads.

```
public class PayloadCacheInterceptor implements Interceptor
{
    private MessageProcessor next;
    private Ehcache cache;

    public void setListener(MessageProcessor listener)
    {
        next = listener;
    }

    public void setCache(final Ehcache cache)
    {
        this.cache = cache;
    }

    public MuleEvent process(MuleEvent event) throws MuleException
    {
        MuleMessage currentMessage = event.getMessage();
        Object key = currentMessage.getPayload();
        Element cachedElement = cache.get(key);

        if (cachedElement != null)
        {
            DefaultMuleMessage cachedMessage =
                new DefaultMuleMessage(cachedElement.getObjectValue(),
                                      currentMessage,
                                      event.getMuleContext());

            return new DefaultMuleEvent(cachedMessage, event);
        }
    }
}
```

The code is annotated with several callouts:

- A red callout on the left side points to the `key` variable in the `process` method with the text "Uses incoming message payload as cache key".
- A red callout on the right side points to the `cache` field with the text "Defines class as Interceptor implementation".
- A red callout on the right side points to the `setCache` method with the text "Setter to receive cache by injection".
- A red callout on the right side points to the `setListener` method with the text "Setter to receive next interceptor by injection".
- A red callout on the right side points to the `process` method with the text "Cache hit— returns a new message built with cached payload and incoming message properties".

⁶ Multithread-minded readers will notice that no effort is made to ensure that only a single thread ever invokes the next interceptor; this is acceptable in Prancing Donkey's usage scenarios, but may need to be enforced in other ones.

```

        }
        // we don't synchronize so several threads can compete to fill
        // the cache for the same key: this is rare enough to be
        // acceptable
        MuleEvent result = next.process(event); //
        cache.put(new Element(key, result.getMessage().getPayload())); ←
        return result;
    }
}

```

Cache miss—invokes next interceptor and returns its result after caching its payload

Notice at ① that in case of a cache hit, you do return the cached payload, but you keep the incoming message as the previous one when instantiating the response message. This is because you want to preserve all the preexisting properties and other extra message context alongside the new payload.

ABSTRACT PARENT, LESS WORK To write a little less code, you could have extended `org.mule.processor.AbstractInterceptingMessageProcessor` (while still implementing `org.mule.api.interceptor.Interceptor`). Doing so would have provided you with a ready-made message source implementation, leaving the implementation of `process()` to be done, while offering a convenient `processNext()` method to call in order to forward the event to the next interceptor.

Prancing Donkey's file-hasher component is a perfect candidate for this interceptor stack. Indeed, it receives a message whose payload is a filename and returns a message whose payload is the MD5 hash of the file's content. Because the considered files don't change in content, you can then use the filename as the key and the MD5 hash as the value in a standard cache. Adding this interceptor stack to your existing file-hashing service (discussed in section 6.1.5) is a no-brainer, as shown in this configuration excerpt:

```

<pooled-component>
    <interceptor-stack ref="PayloadCacheInterceptors" />
    <prototype-object
        class="com.prancingdonkey.component.Md5FileHasher">

```

The following console transcripts clearly show the efficiency of the payload cache interceptor, as reported by the timer-interceptor:

```

12-Sep-2012 5:34:14 PM org.mule.interceptor.TimerInterceptor intercept
INFO: Md5FileHasher took 21ms to process event [25e11b6f-a488-11dd-a659-
-2f0fd1014e82]
12-Sep-2012 5:34:24 PM org.mule.interceptor.TimerInterceptor intercept
INFO: Md5FileHasher took 1ms to process event [25e5884c-a488-11dd-a659-
-2f0fd1014e82]
12-Sep-2012 5:34:34 PM org.mule.interceptor.TimerInterceptor intercept
INFO: Md5FileHasher took 0ms to process event [25e5fd85-a488-11dd-a659-
-2f0fd1014e82]

```

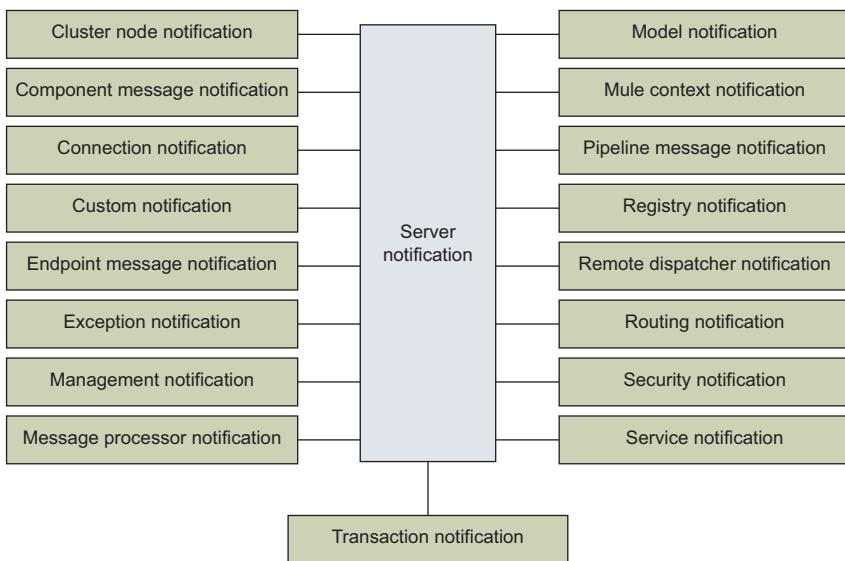


Figure 12.6 The family of Mule server notifications is rich and diverse.

Thanks to the interceptor framework, you’re now able to share transversal logic across components and flows 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 into the notification framework that we’ll detail now.

12.3.3 Listening to notifications

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.

There’s much you can do by directly tapping into the notification framework. For this you need to write custom classes that can receive notification events. Figure 12.6 shows an overview of the notification types, which are all descendants of `ServerNotification`. Each notification type also defines actions that further specify the context in which they happen.

For example, let’s create a listener that can be notified of routing problems that occur when messages don’t get correlated on time in an aggregator router. When this happens, a routing notification is broadcast. The next listing shows part of the implementation of this correlation timeout notification listener.

Listing 12.15 A custom class can be created to receive specific notifications.

```

public final class CorrelationTimeOutListener implements
    RoutingNotificationListener<RoutingNotification> {
    ...
    Defines class as  
RoutingNotificationListener  
implementation
}
  
```

```

public void onNotification(final RoutingNotification notification) {
    if (notification.getAction()
        != RoutingNotification.CORRELATION_TIMEOUT) {
        return;
    }

    final MuleMessage uncorrelatedMessage =
        (MuleMessage) notification.getSource();
}

```

① Disregards any notification event that's not CORRELATION_TIMEOUT

Extracts expected content from notification source

Notice at ① how you programmatically narrow the scope of the notification received to the correlation timeout action only. You do this because routing notifications encompass more types of events than correlation timeouts. Because of the way the notification listener API is designed, notice 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, you 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 the next listing.

Listing 12.16 A custom object can be registered with Mule to receive notifications.

```

<spring:bean name="correlationTimeOutListener"
    class="com.muleinaction.listener.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 at ② and the registration of the listener at ③. The instantiation and initialization of this listener shown at 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 in an example that combines several of the concepts you've learned in this chapter.⁷ Some of Prancing

⁷ Context injection, interceptor, and notification listener.

Donkey's services aren't able to process any event until the Mule instance they run into is fully initialized (because they dispatch messages to other flows that may not themselves be ready). To prevent an incoming event from reaching the component of these sensitive flows, you create an interceptor that rejects any traffic until the whole Mule instance is fully initialized. The next listing shows the full code of this interceptor.

Listing 12.17 Interceptor that rejects events until the context is initialized

```
public final class BrokerNotReadyInterceptor
    extends AbstractInterceptingMessageProcessor
    implements Interceptor,
               MuleContextNotificationListener<MuleContextNotification>,
               Initialisable
{
    private volatile boolean brokerReady = false;

    public void initialise() throws InitialisationException
    {
        try
        {
            muleContext.registerListener(this);
        }
        catch (final NotificationException ne)
        {
            throw new RuntimeException(ne);
        }
    }

    public void onNotification(MuleContextNotification notification)
    {
        int action = notification.getAction();

        if (action == MuleContextNotification.CONTEXT_STARTED)
        {
            brokerReady = true;           ← Sets broker readiness status to
                                         true when context is started
        }
        else if (action == MuleContextNotification.CONTEXT_STOPPED)
        {
            brokerReady = false;         ← Sets broker readiness status to
                                         false when context is stopped
        }
    }

    public MuleEvent process(MuleEvent event) throws MuleException
    {
        if (!brokerReady)
        {
            throw new IllegalStateException(
                "Invocation of service "
                + event.getFlowConstruct().getName()
                + " impossible at this time!");
        }

        return next.process(event);
    }
}
```

The code is annotated with several red callout boxes and numbers:

- Annotation ①: A red callout box pointing to the registration of the listener: `muleContext.registerListener(this);`. The text inside the box is: "Programmatically registers listener with Mule's notifications".
- Annotation 1: A red callout box pointing to the assignment of `brokerReady` to `true`: `brokerReady = true;`. The text inside the box is: "Sets broker readiness status to true when context is started".
- Annotation 2: A red callout box pointing to the assignment of `brokerReady` to `false`: `brokerReady = false;`. The text inside the box is: "Sets broker readiness status to false when context is stopped".
- Annotation 3: A red callout box pointing to the `IllegalStateException` thrown in the `process` method: `throw new IllegalStateException("Invocation of service " + event.getFlowConstruct().getName() + " impossible at this time!");`. The text inside the box is: "Rejects any event until broker is ready".

Did you notice how you apply our previous suggestion and extend `AbstractInterceptingMessageProcessor` to have less code to write in the interceptor?

Because a custom interceptor configuration element doesn't expose an ID that you could use to register it with Mule's notifications infrastructure, you have to perform this registration programmatically ①.

This interceptor is configured using the `custom-interceptor` element, as you saw in the previous section. Again, notice how there's no ID attribute that could allow you to register it as a notification listener by configuration:

```
<custom-processor  
    class="com.prancingdonkey.interceptor.BrokerNotReadyInterceptor" />
```

Therefore, the notification configuration related to this interceptor consists only of the activation of the context family of events:

```
<notifications>  
    <notification event="CONTEXT" />  
</notifications>
```

More notification goodness!

Mule's notification framework offers advanced features:

- 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 12.6), listening to them, and broadcasting them programmatically

Mule's extensive notification framework can help you graft advanced event-driven behavior to your applications. But not everything that flows in Mule is as ephemeral as events; some information needs to be persisted. Let's now discover how Mule manages its own persistence needs.

12.3.4 Configuring Mule data persistence

Have you ever wondered where the messages in a persistent VM queue are stored? Or where Mule keeps track of unique IDs in an idempotent message filter? Indeed, Mule has to manage state for many of the moving parts that you can use in a configuration. This state is managed independent of whatever transports or modules you're using in your configuration. It's persisted with an internal framework called the *object store*. It's important to know about it so that you can understand where Mule's state data is going and how to configure and customize its persistence.

Take a look at the configuration shown in listing 12.18, which consists of a bridge that receives messages on a VM channel, filters out the ones it has already processed, and attempts dispatching to a proxied service until successful.

Listing 12.18 A flow that relies heavily on Mule's internal object store

```
<spring:bean name="untilSuccessfulStore"
    class="org.mule.util.store.PartitionedInMemoryObjectStore" />

<vm:connector name="vmConnector">
    <vm:queue-profile>
        <default-persistent-queue-store />
    </vm:queue-profile>
</vm:connector>

<flow name="idempotent-bridge">
    <vm:inbound-endpoint path="ib.in" />
    <idempotent-message-filter>
        <simple-text-file-store name="ib.idem" />
    </idempotent-message-filter>
    <until-successful objectStore-ref="untilSuccessfulStore" >
        <outbound-endpoint ref="proxiedService" />
    </until-successful>
</flow>
```

The idempotent-bridge flow shown in this example relies on Mule's internal persistence to store the state of three of its moving parts:

- The persistent inbound VM queue uses the default-persistent-queue-store to store the messages pending delivery.
- The idempotent filter uses a simple-text-file-store named ib.idem to store the unique identifier of each message it processed.
- The until-successful routing message processor uses a Spring-configured object store to accumulate messages pending outbound dispatch.

At first glance, it seems that these three configuration mechanisms are quite different, but they all rely on the same internal object store API. Figure 12.7 shows the hierarchy of interfaces that compose Mule's internal object store API. There are many classes that implement these interfaces, and numerous XML configuration elements that allow for their configuration; this figure only lists (in notes) the ones that are in use in listing 12.18.

By default, the file-based persistent object stores save data under the Mule work directory,⁸ in a structure of subdirectories that's created on the fly. For the previous example, you'd find these subdirectories: /queuestore/ib.in and /objectstore/ib.idem.dat, respectively used for the data of the persistent queue and the idempotent filter.

⁸ Typically a directory named .mule located at the root of the execution directory.

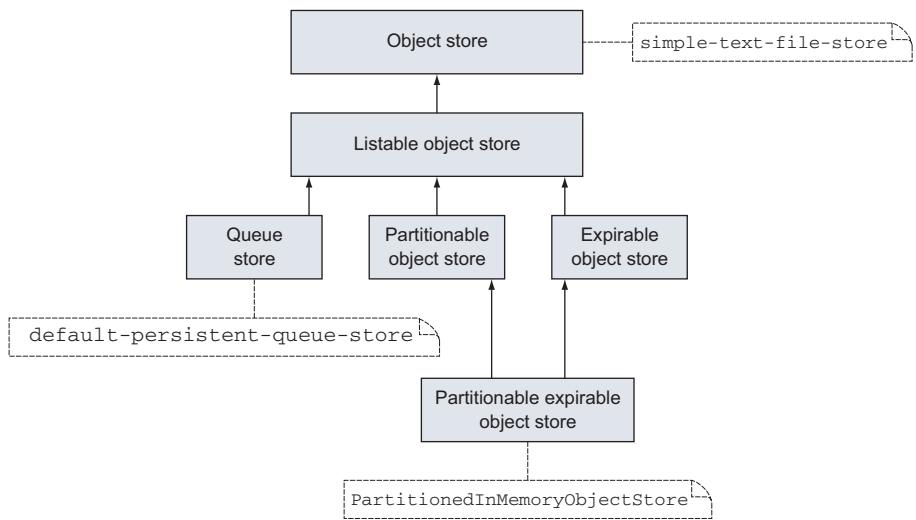


Figure 12.7 The interfaces of the object store API and some elements or classes using them

DATABASE PERSISTENCE If you want to use a database to persist Mule data, look no further than the `org.mule.transport.jdbc.store.JdbcObjectStore`, an object store that, behind the scenes, uses the standard JDBC transport of Mule. It works by using custom queries that you craft to provide the basic CRUD operations needed by the object store.

Prancing Donkey has developed a great deal of experience with Ehcache, so they've decided to create a custom object store implementation that uses Ehcache as its backend storage. They'll implement it as an `org.mule.api.store.ListableObjectStore` (which will cover the needs of `until-successful` and `idempotent-message-filter`) and will use the `org.mule.util.store.QueueStoreAdapter` to adapt it as an `org.mule.api.store.QueueStore` so it can be used for VM queue persistence. They'll also extend `org.mule.util.store.AbstractObjectStore` to have a big part of the implementation done for them already.

Listing 12.19 shows the implementation in all its gory glory! There's not much to say about it except that it delegates all operations to the Ehcache instance that's been provided by injection. There's some general impedance mismatch between the object store API and standard caches; for example, the object store API expects exceptions to be thrown in many circumstances in which a typical cache wouldn't report any problem (like when overriding a key). The mismatch is visible in the `doRemove()` method, in which you jump through hoops to return the last value of a key you delete from the store.⁹

⁹ This implementation is far from perfect; the value may have changed between the retrieve and remove operations.

Listing 12.19 An Ehcache-backed listable object store

```
public class EhCacheObjectStore<T extends Serializable>
    extends AbstractObjectStore<T>
    implements ListableObjectStore<T>
{
    private Ehcache cache;

    public void setCache(final Ehcache cache)
    {
        this.cache = cache;
    }

    public boolean isPersistent()
    {
        return cache.getCacheConfiguration().isDiskPersistent();
    }

    public void open() throws ObjectStoreException
    {
        // NOOP
    }

    public void close() throws ObjectStoreException
    {
        // NOOP
    }

    @SuppressWarnings("unchecked")
    public List<Serializable> allKeys()
        throws ObjectStoreException
    {
        return cache.getKeys();
    }

    @Override
    protected boolean doContains(Serializable key)
        throws ObjectStoreException
    {
        return cache.isKeyInCache(key);
    }

    @Override
    protected void doStore(Serializable key, T value)
        throws ObjectStoreException
    {
        cache.put(new Element(key, value));
    }

    @SuppressWarnings("unchecked")
    @Override
    protected T doRetrieve(Serializable key)
        throws ObjectStoreException
    {
        Element element = cache.get(key);
        return element == null ? null : (T) element.getValue();
    }
}
```

```

@Override
protected T doRemove(Serializable key)
                     throws ObjectStoreException
{
    T removedValue = doRetrieve(key);
    if (removedValue == null) {
        throw new ObjectDoesNotExistException();
    }
    boolean removed = cache.remove(key);
    if (!removed) {
        throw new ObjectDoesNotExistException();
    }
    return removedValue;
}
}

```

With this implementation in hand, you can now reconfigure the previous example in such a way that different Ehcache caches are used for each of the configuration elements that need persistence. Listing 12.20 shows this new configuration. The configuration of each individual Ehcache cache (`vmEhCache`, `idempotentEhCache`, and `untilSuccessfulEhCache`) isn't shown; Spring or Ehcache's own configuration mechanisms could be used for that. Remember how we mentioned that you would use a `QueueStoreAdapter` to fit an Ehcache object store into a configuration element that takes a queue store? You can see it done in the configuration of the Spring bean named `vmQueueStore`.

Listing 12.20 Using custom Ehcache-backed persistence

```

<spring:beans>
    <spring:bean name="vmStore"
                 class="com.prancingdonkey.objectstore.EhCacheObjectStore"
                 p:cache-ref="vmEhCache" />

    <spring:bean name="vmQueueStore"
                 class="org.mule.util.store.QueueStoreAdapter"
                 c:store-ref="vmStore" />

    <spring:bean name="idempotentStore"
                 class="com.prancingdonkey.objectstore.EhCacheObjectStore"
                 p:cache-ref="idempotentEhCache" />

    <spring:bean name="untilSuccessfulStore"
                 class="com.prancingdonkey.objectstore.EhCacheObjectStore"
                 p:cache-ref="untilSuccessfulEhCache" />
</spring:beans>

<vm:connector name="vmConnector">
    <vm:queue-profile>
        <queue-store ref="vmQueueStore" />
    </vm:queue-profile>
</vm:connector>

<flow name="idempotent-bridge">
    <vm:inbound-endpoint path="ib.in" />
    <idempotent-message-filter>

```

```
<spring-object-store ref="idempotentStore" />
</idempotent-message-filter>
<until-successful objectStore-ref="untilSuccessfulStore">
    <outbound-endpoint ref="proxiedService" />
</until-successful>
</flow>
```

Mule's internal persistence mechanism no longer has any secrets for you. Indeed, you should now be able to configure Mule's persistence to fit your needs and constraints.

CLUSTERING FOR EE The Enterprise Edition of Mule provides a clustered object store that can be used to perform distributed deployments of Mule applications and have their state transparently shared across a cluster.

In this section, we've reviewed how using Mule's internal API can allow you to use life-cycle events, intercept messages, and listen to notifications in order to implement advanced behavior in your applications. You've also learned how to configure and customize Mule's data persistence.

Now you're at the tipping point of your learning journey. You've learned a lot about how to build applications with Mule, from the simplest to the most complex ones. It's now time to address the crucial point of testing: how can you test all this great stuff that you're building with Mule? Read on to find out!

12.4 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 stub out component behavior. We'll wrap up by taking a look at JMeter, an open source load-testing tool, to load-test Mule endpoints.

12.4.1 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 do some manual verification that it worked. During development this can be an effective technique to explore potential configurations. 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 flow used by Prancing Donkey to asynchronously transform XML messages from one format to another. The configuration in listing 12.21 shows this flow, which accepts a JMS message off a queue, processes it with an XSLT transformer, and then sends it to another JMS queue.

Listing 12.21 A flow used to asynchronously transform XML messages

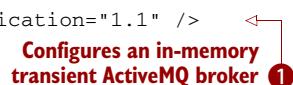
```
<flow name="jmsXmlV1V2Bridge">
    <jms:inbound-endpoint queue="messages.v1" />
    <mule-xml:xslt-transformer xsl-file="v1toV2.xsl" />
    <jms:outbound-endpoint queue="messages.v2" />
</flow>
```



You want to be able to test this flow independently of any JMS provider. This means that you'll have to configure the Mule application differently for functional tests and deployment. This is achieved by externalizing the connectors' configuration in a separate file and having two versions of this file: one for tests and one for deployment. The flows will be in another file, which will remain the same for all environments. When running functional tests, you'll then need to start a JMS provider before the tests start, and stop it after they're done. For this you'll use an in-memory, nonpersistent JMS provider, as shown in the next listing.

Listing 12.22 A connector's configuration file used during functional tests

```
<jms:activemq-connector name="amqConnector" specification="1.1" />
```



This configuration is fairly straightforward. One thing to note is the use of the ActiveMQ connector ①. By default, this creates an in-memory ActiveMQ broker that lives for the duration of the functional test.

IN-MEMORY INFRASTRUCTURE PROVIDERS Use in-memory Java providers to test endpoints that typically require enterprise infrastructure like JMS, JDBC, SMTP, or POP3/IMAP. Popular options include ActiveMQ's in-memory transient (nonpersistent) broker, Apache Derby or HSQL in-memory databases, and GreenMail in-memory mail server.

Let's write a functional test for this configuration. The goal is to start Mule, send a JMS message to the `messages.v1` queue, and then consume a message off the `messages.v2` queue and confirm that the transformed payload is present. Use the `FunctionalTestCase` along with the `MuleClient` to get this done, as illustrated in the following listing.

Listing 12.23 A functional test case that exercises a flow with JMS endpoints

```
public class TransformingBridgeTestCase extends FunctionalTestCase
{
    @Override
    protected String getConfigResources()
    {
        return
            "functional-test-connectors.xml,jms-transforming-bridge.xml";
    }
}
```



```

    @Test
    public void testJmsXmlV1V2Bridge() throws Exception
    {
        MuleClient muleClient = muleContext.getClient();

        muleClient.dispatch("jms://messages.v1",
                            MESSAGE_XML_V1,
                            null);

        MuleMessage response =
            muleClient.request("jms://messages.v2",
                               1000 * getTestTimeoutSecs());
        assertEquals(response.getPayloadAsString(),
                     is(MESSAGE_XML_V2));
    }

```

The code is annotated with numbered callouts:

- ① Get MuleClient instance from context**: Points to the line `MuleClient muleClient = muleContext.getClient();`
- ② Request message from destination queue**: Points to the line `muleClient.request("jms://messages.v2", 1000 * getTestTimeoutSecs());`
- ③ Dispatch message to input queue**: Points to the line `muleClient.dispatch("jms://messages.v1", MESSAGE_XML_V1, null);`
- ④ Test that payload has been transformed**: Points to the line `assertEquals(response.getPayloadAsString(), is(MESSAGE_XML_V2));`

You start off by specifying the locations of the Mule configuration files. Load side-by-side functional-test-connectors.xml, which contains the in-memory JMS connector, and jms-transforming-bridge.xml, which contains the flow to be tested. Typically the test configuration files would be located in `src/test/resources` and the deployable files in `src/main/app` or `src/main/resources`, depending on the deployment you target (see chapter 8).

The test method is annotated with a standard JUnit 4 `@Test` annotation. In it, you get the `MuleClient` ① from the `muleContext` present in the `org.mule.tck.junit4.FunctionalTestCase` superclass.

You're sending a JMS message to a one-way endpoint, so you don't expect a synchronous response (hence the use of the `dispatch` method of `MuleClient` to send a test payload to the `messages.v1` queue). The message will now be sent to the in-memory ActiveMQ broker configured in listing 12.22. At ②, you request a message off of the JMS `messages.v2` queue with a timeout equal to the maximum possible timeout for a functional test.¹⁰ Use the maximum possible timeout because there's no way to know beforehand what could be a reasonable wait time; using a too-small timeout can lead to a test that is intermittently failing, something that can be tricky to figure out. In practice, this timeout will never be reached, and the rest of the test will be executed as soon as a message arrives in the `messages.v2` queue.

Assuming everything went well, the transformed message with transformed payload should be waiting for you patiently on the `messages.v2` queue. If you don't receive a message off the queue in that time frame, the test will fail. Assuming a message has been received, you perform your test assertion ③ and ensure the payload has been transformed appropriately.

TRANSFORMER TESTING The `org.mule.transformer.AbstractTransformerTestCase` can simplify (and speed up) transformer testing.

¹⁰ The default value is 60 seconds. This is configurable via the `mule.test.timeoutSecs` Java system property.

Listing 12.23 used Mule to dispatch a test message over JMS. This is convenient, but has the drawback that Mule is used on both sides of the test; it's good practice to write tests that resemble real use cases. This means that if Mule isn't involved in the client side of the real use case, it's better not to use it as a client in tests. There are numerous Java libraries that can help you with that. For example, the Apache HttpClient (<http://hc.apache.org/httpcomponents-client-ga>) would be perfect for testing HTTP endpoints. What about JMS? Let's rewrite the preceding test without Mule as client and find out!

Listing 12.24 Using pure JMS to test a flow with inbound and outbound JMS endpoints

```

@Test
public void testJmsXmlV1V2Bridge() throws Exception
{
    JmsConnector jmsConnector =
        (JmsConnector) muleContext
            .getRegistry()
            .lookupConnector("amqConnector");

    Session session =
        jmsConnector
            .getConnection()
            .createSession(false, Session.AUTO_ACKNOWLEDGE);

    Queue v1Queue = session.createQueue("messages.v1");
    MessageProducer producer = session.createProducer(v1Queue);
    TextMessage v1Message = session.createTextMessage(MESSAGE_XML_V1);
    producer.send(v1Message);

    Queue v2Queue = session.createQueue("messages.v2");
    MessageConsumer consumer = session.createConsumer(v2Queue);
    BytesMessage v2Message =
        (BytesMessage) consumer.receive(1000 * getTestTimeoutSecs());

    byte[] bytes = new byte[(int) v2Message.getBodyLength()];
    v2Message.readBytes(bytes);
    assertEquals(new String(bytes), MESSAGE_XML_V2);
    session.close();
}

```

Annotations for Listing 12.24:

- Consume resulting message from outbound queue**: A callout points to the line `v2Message = (BytesMessage) consumer.receive(1000 * getTestTimeoutSecs());`. A red arrow points from the text to the line.
- Get reference to JMS connector**: A callout points to the line `JmsConnector jmsConnector = (JmsConnector) muleContext.getRegistry().lookupConnector("amqConnector");`. A red arrow points from the text to the line.
- Produce test message on inbound queue**: A callout points to the line `producer.send(v1Message);`. A red arrow points from the text to the line.
- Extract bytes payload from resulting message**: A callout points to the line `assertEquals(new String(bytes), MESSAGE_XML_V2);`. A red arrow points from the text to the line.

Listing 12.24 shows the test rewritten using standard JMS client code to interact with the JMS provider to which Mule is connected when the test is run. As you can see, the code is more involved than when Mule is used to produce and consume the source and result messages, which says a lot about the heavy lifting Mule is doing behind the scenes! In practice, you'd most likely share this JMS code in a utilities class or use a ready-made client, like Spring's `JmsTemplate`.

The `FunctionalTestCase` is capable of great things but is complete when it's working with its sidekick: the `FunctionalTestComponent`. Let's learn more about it.

12.4.2 Behavior stubbing

Another important item in your Mule testing toolbox is a component that manifests itself as the `<test:component />` configuration element and the `FunctionalTestComponent` class. This component allows you to

- Log details about the message it processes
- Set the current message payload to a configured String value or the content of a file
- Append a string to the payload of the message it processes
- Slow down a flow execution by making the current thread sleep for a while
- Break the flow execution by throwing a configured exception
- Capture messages that flow through it
- Invoke a custom callback when it receives a message

You can use the functional test component in any flow you like. Because it's preferable to use the same flows in test and deployments, you likely won't use this component in flows that end up deployed in production. Where this component truly shines is for stubbing out the behavior of external systems that your flows interact with. Let's see how.

TEST IN PRODUCTION If you end up shipping flows that contain the test component in production configuration, prevent its execution by wrapping it with a choice router or a filter router. Then use a message property to flag test-grade messages that will be the only ones allowed to hit the test component.

Let's functionally test an integration flow that you've seen before; listing 12.25 comes straight from section 2.2.3. As you can see, this flow interacts with a remote web service over HTTP. How can you test this without calling this remote service?

Listing 12.25 A flow that interacts with a remote service

```
<flow name="acmeApiBridge">
    <vm:inbound-endpoint path="invokeAcmeAmi" />
    <jdbc:outbound-endpoint queryKey="storeData" />
    <http:outbound-endpoint address="http://acme.com/api" />
</flow>
```

You start by externalizing the endpoint definition in its own configuration file, turning it into a global endpoint, and referring to it from the flow. That's what you see in the next listing.

Listing 12.26 A flow that interacts with a remote service via a global endpoint

```
<flow name="acmeApiBridge">
    <vm:inbound-endpoint path="invokeAcmeAmi" />
    <jdbc:outbound-endpoint queryKey="storeData" />
    <http:outbound-endpoint ref="acmeApiEndpoint" />
</flow>
```

Notice how the HTTP outbound endpoint now references a global endpoint named acmeApiEndpoint. With this in place, you can now create a test configuration file that contains a global HTTP endpoint named acmeApiEndpoint but that doesn't interact at all with a remote service. You'll provide a stub of the remote service in the same test configuration file, using a standard flow to implement it. This flow will use the functional test component to simulate the remote service behavior (including potential failures) and, more generally, instrument the service stub. The following listing shows this test configuration file.

Listing 12.27 A test configuration file with in-memory transports and service stubs

```
<spring:beans>
    <spring-jdbc:embedded-database id="jdbcDataSource" type="HSQL">
        <spring-jdbc:script location="classpath:test-ddl.sql" />
    </spring-jdbc:embedded-database>
</spring:beans>

<http:endpoint name="acmeApiEndpoint"
    address="http://localhost:${port}/api"
    exchange-pattern="request-response" />

<flow name="acmeApiStub">
    <inbound-endpoint ref="acmeApiEndpoint" />
    <object-to-string-transformer />
    <test:component>
        <test:return-data>{"result": "success"}</test:return-data>
    </test:component>
</flow>
```

Notice that, following the advice in the previous section, you use an in-memory database (HSQL, a.k.a. HyperSQL) to support the testing of this flow, which also interacts with a JDBC endpoint. You don't need to provide a global endpoint here because the connection factory is the natural seam you use to replace the database with an in-memory one during testing.

You're most likely wondering where the \${port} value comes from. Hold tight, we'll look into this in a second. The way you use this test configuration file is similar to what was discussed in the previous section; you'll load it side by side with the configuration containing the deployable flow. This means you'll load functional-test-stubs.xml and acme-api-bridge.xml in your functional test case, shown in the following listing.

Listing 12.28 A functional test that uses a stubbed-out HTTP service

```
public class AcmeApiBridgeTestCase extends FunctionalTestCase
{
    @Rule
    public DynamicPort port = new DynamicPort("port"); ① Receive dynamic port by injection

    @Override
    protected String getConfigResources() ② Specify locations of configuration files
    {
        return
    }
}
```

```

        "functional-test-stubs.xml,acme-api-bridge.xml";
    }

@Test
public void testSuccessfulJdbcAndHttpDispatches() throws Exception
{
    MuleClient muleClient = muleContext.getClient();
    muleClient.dispatch("vm://invokeAcmeAmi",
                        ACME_TEST_MESSAGE,
                        null);
    Thread.sleep(5000L);
    MuleMessage dbResponse =
        muleClient.request("jdbc://retrieveData",
                           1000 * getTestTimeoutSecs());
    List<Map<String, String>> resultSet =
        (List<Map<String, String>>) dbResponse.getPayload();
    assertThat(resultSet.get(0).get("DATA"),
               is(ACME_TEST_MESSAGE));
    FunctionalTestComponent ftc =
        getFunctionalTestComponent("acmeApiStub");
    assertThat(ftc.getReceivedMessagesCount(),
               is(1));
    String lastReceivedMessage =
        (String) ftc.getLastReceivedMessage();
    assertThat(lastReceivedMessage,
               is(ACME_TEST_MESSAGE));
}

```

Get MuleClient instance from context

Dispatch message to input VM queue

Naïve attempt to wait for dispatches to occur

Assert that data has been persisted correctly

Get functional test component in stubbing flow

Ensure one message has been received

Retrieve last message received

Assert message has been dispatched correctly

Notice how you receive a port by injection ①? That's how the \${port} value gets generated and how you could use it in your test, if needed. Mule will look for an available port (in the 5000–6000 range) and will assign it to your test. Should you want to interact with the open port from your code, you could call `port.getNumber()` to find out what value has been assigned. You can have as many dynamic ports as you need and use whatever field names you want, as long as each of them is annotated with `@Rule`. This is extremely convenient for running functional tests on a continuous integration server without messing with other tests or with the host server itself.

DYNAMIC PORTS RULE Use dynamic ports in functional test cases to make them play well on developers' workstations and continuous integration servers.

Why the call to `Thread.sleep()` ③? The reason is that without it, your assertions would run before anything has been dispatched by Mule. Why is that? Look again at the flow you're testing; its message source is a VM inbound endpoint, which is one-way by definition. This means that after dispatching the test message to the VM endpoint ②, the execution will immediately proceed to the next step, exposing you to the likely

scenario in which you would run assertions while Mule is, behind the scenes, still routing the message.

Using `Thread.sleep()` to wait for the messages to have been dispatched is naive because there's no way to figure out a sleep time that will work well in all circumstances. Make it too short, and it won't wait long enough on a busy continuous integration server; make it too long, and running the tests on a developer's workstation would be excruciating.

Let's fix that now. You'll use a latch to hold the test execution until the dispatched message hits the functional test component in the remote service stub. For this, you'll instrument the functional test component with an event callback that will release the latch. The code in next listing shows the test method now relieved of its naive `Thread.sleep()`.

Listing 12.29 A latch suspends the test execution until a message is received

```

@Test
public void testSuccessfulJdbcAndHttpDispatches() throws Exception
{
    final Latch latch = new Latch();           ← Create latch used to
                                              hold test execution

    FunctionalTestComponent ftc =
        getFunctionalTestComponent("acmeApiStub");

    ftc.setEventCallback(new EventCallback()          ← Instrument functional
    {                                              test component
        public void eventReceived(
            MuleEventContext context,
            Object component)
            throws Exception
        {
            latch.countDown();                     ← Release latch when a
        }                                         message is received
    });
}

MuleClient muleClient = muleContext.getClient();

muleClient.dispatch("vm://invokeAcmeAmi",
    ACME_TEST_MESSAGE,
    null);                                     ← Block test
                                              execution until
                                              latch is released

latch.await(getTestTimeoutSecs(), TimeUnit.SECONDS);

// DB and HTTP assertions unchanged...

```

AWAIT, DON'T SLEEP Avoid using `Thread.sleep()` in functional tests to wait for dispatch operations in Mule. Instead, use event listeners and concurrency primitives to control the test execution.

You could decide to test your flow to ensure it behaves correctly if the remote service returns an error. For this, you could configure the functional test component differently in another stubbing configuration file, as shown in the next listing.

Listing 12.30 Using the test component to simulate exceptions

```
<test:component throwException="true"
    exceptionToThrow="java.lang.RuntimeException" />
```

You could also test your flow's behavior in case of slowness from the remote service by using again a specific feature of the functional test component, shown in the next listing.

Listing 12.31 Using the test component to simulate slowness

```
<test:component waitTime="15000">
    <test:return-data>{"result": "success"}</test:return-data>
</test:component>
```

By now you should be comfortable writing functional tests with Mule and mocking component behavior using the `test` component. You'll now see how you can perform load tests on a running Mule instance.

12.4.3 Load testing

Extensive functional tests coupled with automated integration testing should provide you with a fair bit of reassurance that your Mule projects will run correctly when deployed. The real world, however, might have different plans. Perhaps Prancing Donkey's asynchronous message transformation service shown in listing 12.21 will have to deal with unexpected peak loads during Black Friday.

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 minute. Suddenly, however, you notice the service is crawling, and the load on your lowly spare blade server is through the roof. You check with Prancing Donkey's JMS administrator and realize that thousands of messages are being sent to the `messages.v1` queue every minute. Deciding it's time to gain deployment flexibility, you deploy Mule to the cloud, automatically provisioning Mule instances to consume messages when the message volume becomes too high. To do this, however, you need to determine how many messages the transformation service deployed on 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, you'll see how to use JMeter with JMS as you load test the service you performed integration testing on previously.

To start off, you'll need to download JMeter from http://jmeter.apache.org/download_jmeter.cgi. The example in this section will test JMS, so you need to provide your JMS provider's client libraries to JMeter before you start it up. This can be done by copying the appropriate JAR files to the `lib/` subdirectory in the root of the JMeter distribution.¹¹ Once this is done, you can start JMeter from the `bin/` subdirectory,

¹¹ Prancing Donkey uses ActiveMQ, so we dropped `activemq-all-5.6.0.jar` in the `lib/` directory.

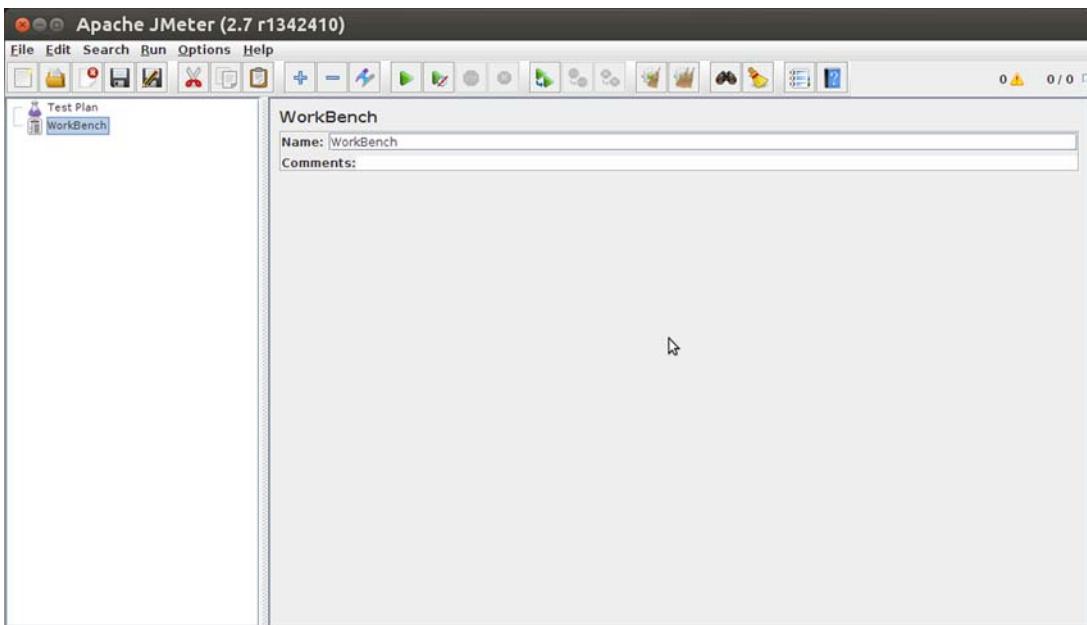


Figure 12.8 Starting JMeter for the first time

either by 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 12.8.

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 you'll use to control the amount and concurrency of the JMS messages you'll send and receive. After talking to Prancing Donkey's JMS administrator a bit more, you manage to nail down the load problems starting to occur with five concurrent producers sending about 20 messages each and a similar number of consumers. 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 12.9.

Set the name and (if you want any) comments. You want your tests to continue if there are any errors along the way. This could allow you to detect and debug failures in the JMS infrastructure itself, for example. Selecting Stop Thread here will cause the 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, you want all five threads to start at the same time, so you set the value to 0. The loop count indicates the number of messages you want each thread to send. In this case, you want each user to send 20 messages, and then stop. Clicking Forever here will cause each user to send messages indefinitely.



Figure 12.9 Specifying a thread group to simulate 5 concurrent JMS producers sending 20 messages

You now need to add a JMS *sampler* to the load test. In JMeter, a sampler is responsible for generating the traffic used for the test. Set up a JMS sampler by right-clicking on JMS Test, selecting Add, selecting Sampler, and then selecting Publisher. The Publisher sampler indicates that you want to generate data for a JMS queue. This should lead to a screen that allows you to configure the details of your JMS broker and specify the `messages.v1` queue as the destination for the test messages. Figure 12.10 illustrates the configuration for Prancing Donkey's staging ActiveMQ broker.

Similarly, you need to configure a JMS Subscriber sampler to consume the transformed messages that have been sent to the `messages.v2` queue. Figure 12.11 shows

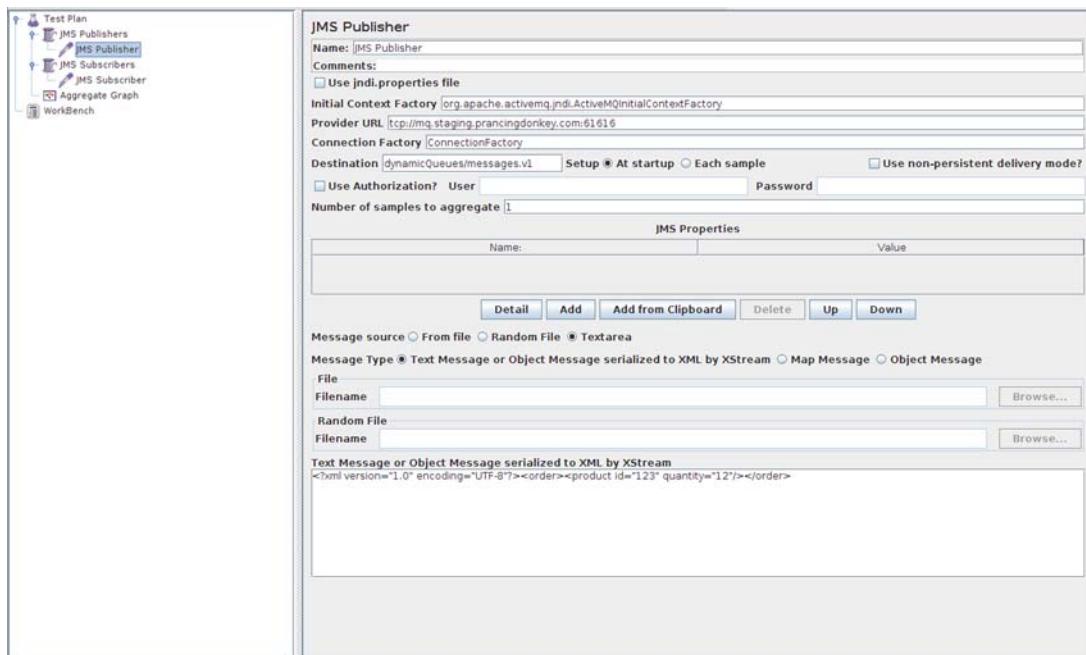


Figure 12.10 Configuring a JMS Publisher sampler with JMeter

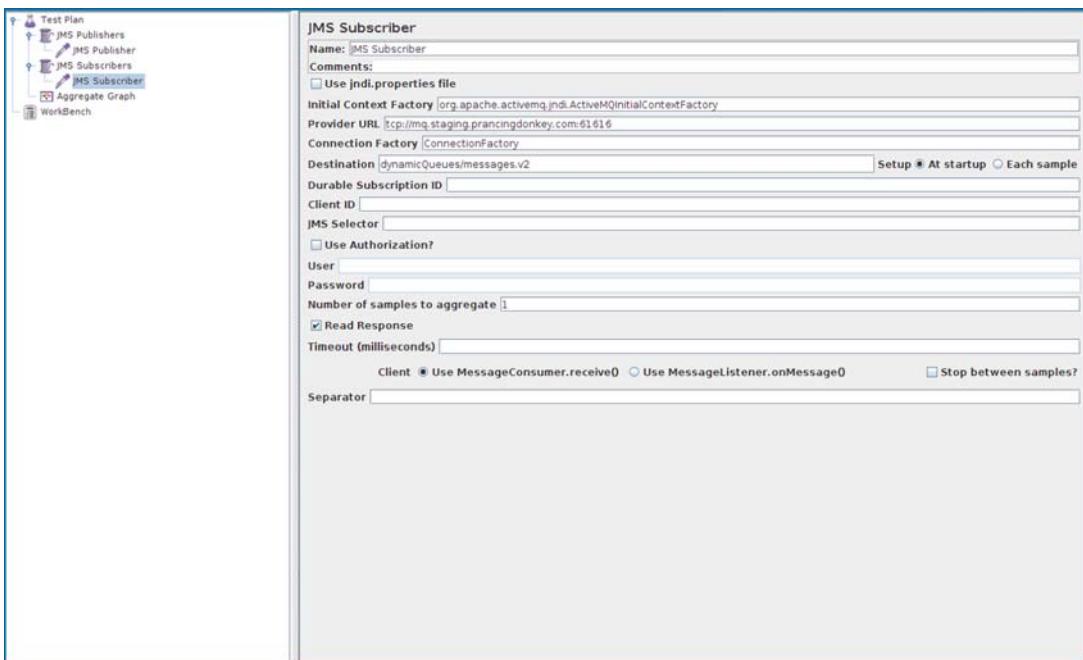


Figure 12.11 Configuring a JMS Subscriber sampler with JMeter

this configuration screen.¹² You don't add any particular assertion to check the content of the transformed messages because you already test this behavior with function tests, and you don't expect to receive garbled messages even under load (it's all or nothing with JMS). If you had a doubt about potential corruption under load, you would need to add an XML assertion node to your JMeter test plan.

With this in place, you're almost ready to start your load test. You could start the test now, but you'll want to collect some data as you run the tests. This is accomplished by adding listeners to the thread group. Right-click on the thread group, select Add, select Listener, and select a listener you'd like to use to interpret the test results. For this case, let's add the Aggregate Graph listener. You can now run the test and view the results. Click on the Run menu and select Start. You should see a green light on the upper right-hand side of the screen as the test runs.¹³

If you click on any of the listeners as the test is running, you should see the results being populated. This should resemble figure 12.12.

The listener results will be appended as subsequent tests are run. This will allow you to trend things like throughput as you increase the load of your tests. You can now use this test to load test the message transformer on the Mule instance. By examining

¹² ActiveMQ binds dynamic queues in JNDI under the dynamicQueues/ root.

¹³ If your test fails to run, or if any failures occur during it, you can view them in the jmeter.log file located in the bin subdirectory you used to launch JMeter.

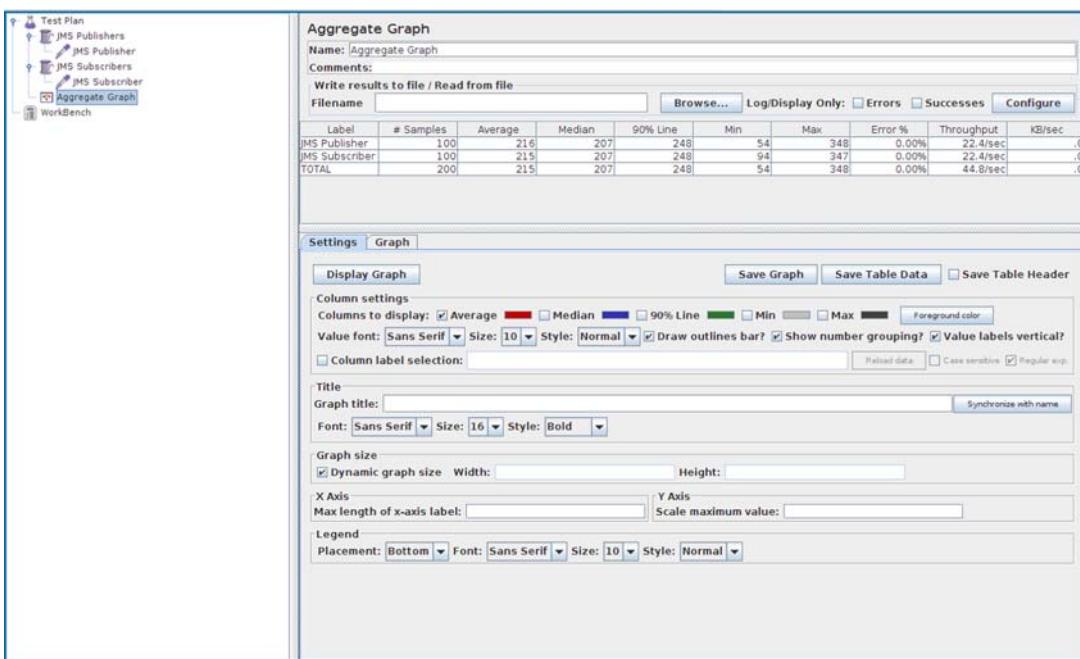


Figure 12.12 Examining results of a JMeter listener

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 begins to drop off. You can then use the techniques you learned about in chapter 11 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 the capabilities of JMeter. As mentioned in the introduction, JMeter can test transports other than JMS, like HTTP or even plain TCP. You can even implement your 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 (<http://jmeter.apache.org/usermanual/>).

By now you should be comfortable testing your Mule projects. You saw how Mule's Test Compatibility Kit facilitates functional testing of your components, transformers, and Mule configurations. You then saw how JMeter, an open source load-testing framework, can be used to load test your Mule endpoints.

As you'll see by developing your Mule applications, the need to grasp what's happening inside Mule at runtime will become more present; this is when debugging will become necessary. Let's now look at this final aspect of developing with Mule.

12.5 Debugging with Mule

Sometimes things don't happen as you expected; a flow doesn't behave the way you initially designed it to. How do you find out why? Or you may need to know all the specific message properties a transport has created on the messages it gives to your flows; because they're not fully documented online, where can you look to find them? If you're asking yourself these questions, it's time for you to learn a few debugging techniques with Mule.

ENTERPRISE DEBUGGING The Enterprise Edition of Mule comes complete with an advanced console, the Mule Management Console (MMC), which provides, among other things, a set of tools that can be used for debugging (like the Flow Analyzer). Mule EE being beyond the scope of this book, we'll focus exclusively on what you can do with the Community Edition of Mule and open source tools.

The Mule Studio visual debugger (see the note "Visual flow debugger") is a convenient tool for inspecting messages that flow through a Mule application running in this IDE. The techniques described in this section are applicable even if the Mule application runs on the standalone broker. They also allow you to get closer to the Mule code if the issue you're investigating can't be resolved by the surface-level inspection provided by the visual debugger.

VISUAL FLOW DEBUGGER Since version 3.4, Mule Studio offers a visual debugger that allows step debugging of the execution of flows, with each step being each message processor in the flow. This debugger, shown in figure 12.13, supports conditional breakpoints and allows a deep inspection of the in-flight message. As of this writing, it's not possible to visually debug a flow started from a functional test.

The first technique we'll look at is as unsophisticated as the good old print statements you used to add to trace your program's execution (don't deny it, we've all done it!).

12.5.1 Logging messages

Logging messages is a simple and effective manner to gain good insight into what's happening in a Mule application. To circle back to one of the questions raised at the beginning of this section, listing 12.32 shows how you would log all the message meta information (properties, payload type, exception payload) of messages received by the HTTP endpoint in your flow.

Listing 12.32 A flow that logs all message meta information

```
<flow name="BeersService">
    <http:inbound-endpoint address="http://localhost:${port}/api/beers"
        exchange-pattern="request-response" />
    <logger level="INFO" />
    <!-- SNIP -->
</flow>
```

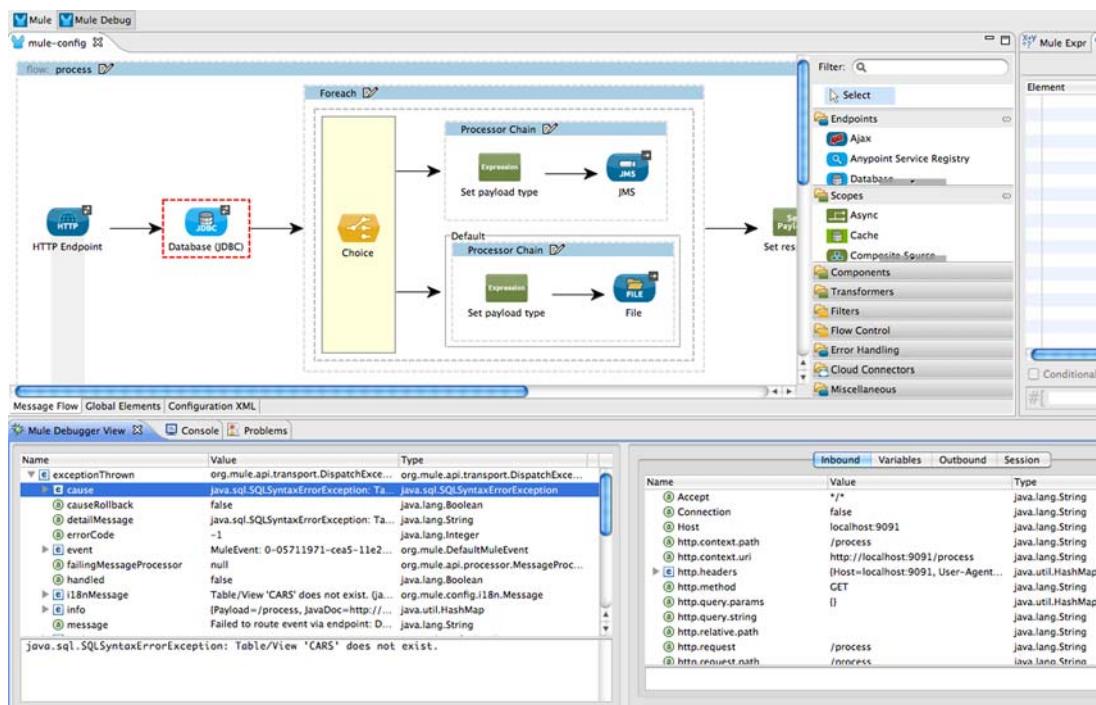


Figure 12.13 The Mule Studio visual debugger

By default, the logger message processor outputs all the message meta information at the configured log level (refer to section 8.1.3 for more information about logging configuration). When an HTTP GET request hits the flow in the previous listing, the log output will look like the following:

```

12:42:25,124 INFO connector.http.mule.default.receiver.02
[org.mule.api.processor.LoggerMessageProcessor]
org.mule.DefaultMuleMessage
{
    id=45dac564-1aee-11e2-84ca-4d33fec1ed8d
    payload=java.lang.String
    correlationId=<not set>
    correlationGroup=-1
    correlationSeq=-1
    encoding=UTF-8
    exceptionPayload=<not set>

    Message properties:
    INVOCATION scoped properties:
    INBOUND scoped properties:
        Connection=false
        Host=localhost:5281
        Keep-Alive=false
        MULE_ORIGINATING_ENDPOINT=endpoint.http.localhost.5281.api.beers
        MULE_REMOTE_CLIENT_ADDRESS=/127.0.0.1:45782
  
```

```
User-Agent=Jakarta Commons-HttpClient/3.1
http.context.path=/api/beers
http.context.uri=http://localhost:5281/api/beers
http.headers={Host=localhost:5281,
              User-Agent=Jakarta Commons-HttpClient/3.1,
              Keep-Alive=false,
              Connection=false}
http.method=GET
http.query.params={}
http.query.string=
http.relative.path=
http.request=/api/beers
http.request.path=/api/beers
http.version=HTTP/1.1
OUTBOUND scoped properties:
  MULE_ENCODING=UTF-8
SESSION scoped properties:
}
```

Notice how the properties scopes are clearly represented. With this information in hand, you can start doing smart things based on the message properties, like routing messages that have a specific `http.context.path` inbound message property to a specific message processor with a choice router. As you can also see, the full payload is not logged. This is a smart default behavior; some payloads can be big or confidential, so logging them could be an issue. Moreover, if the payload is an instance of a `java.io.InputStream` (as is the case for HTTP POST requests), logging the payload would require deserializing the stream, preventing downstream message processors from working in a streaming manner.

TARGET CATEGORY By default, the logger message processor logs under the `org.mule.api.processor.LoggerMessageProcessor` category. You can change that to any value you want by using the `category` attribute of the `logger` XML element.

But what if you need to log the payload? The best way to do so is to use MEL to log the payload rendered as a String. Mule will use its autotransformation capacity (discussed in section 4.3.6) to find the best transformer(s) for rendering the payload as a String. If the payload is streaming, it will be replaced by a serialized form (for example, `byte[]`). The following configuration snippet shows how to configure the logger message processor to log the current message payload. And yes, if you want to log both the payload and the meta information, you'll have to use two loggers:

```
<logger message="#[message.payloadAs(java.lang.String)]"
       level="INFO" />
```

Logging message payload and meta information will help you to gain insight into what's happening within your flows. Sometimes, a finer-grained approach to troubleshooting is needed; this is when step debugging comes in handy.

12.5.2 Step debugging a Mule application

Step debugging in Mule is easy to set up but can be hard to use, and this is because of the long chains of class extensions on which most moving parts rely. Said differently, if you place a breakpoint in a generic abstract class used by many moving parts of Mule, you'll end up breaking the execution constantly without clearly knowing where you stand. Another complication with step debugging in Mule comes from its heavily multithreaded nature; the execution may at some point in time escape your step debugging session because it's been handed off to a different thread.

Without further ado, let's start step debugging a Mule application. This is initiated differently depending on the way you're running Mule:

- *Inside your IDE*—In this case, start in debug mode using your IDE-specific command. This applies to Mule Studio as well.
- *Deployed in a web container*—For this situation, refer to the “Remote debugging” section of your web container's user guide.
- *Using the standalone distribution*—Enable remote debugging by adding the `-debug` parameter to the start command.

Stack traces—longer, deeper, chattier

By default, Mule shortens stack traces logged when exceptions occur. If you want to go for the full monty, start Mule with the `mule.verbose.exceptions` system property set to `true`. With Mule standalone, this is done by adding a `-M` prefixed parameter to the start command:

```
./bin/mule -M-Dmule.verbose.exceptions=true
```

For this discussion, we'll consider remote step debugging Mule standalone with Eclipse IDE. When started with the `-debug` parameter, Mule standalone listens for remote debugging connections on port 5005. Figure 12.14 shows how to connect Eclipse to Mule by creating a remote debug configuration.

In order to step debug in Mule's code, you need to have the Mule source code available to Eclipse. If your project is built with Maven, the Mule source code is most likely attached to your Eclipse project already. Otherwise, you can find it in the `/src/mule-x.y.z-src.zip` archive provided with the Mule standalone distribution.

We've previously warned you about placing breakpoints in counterproductive locations, so what are good places for breakpoints? The following list should give you an idea:

- *Terminal Mule moving parts*—Classes that are at the bottom of class hierarchies, such as, transformers
- *POJO components*—Your own classes that are used as flow components
- *Custom message processors*—Your own custom implementations, including specific interceptors

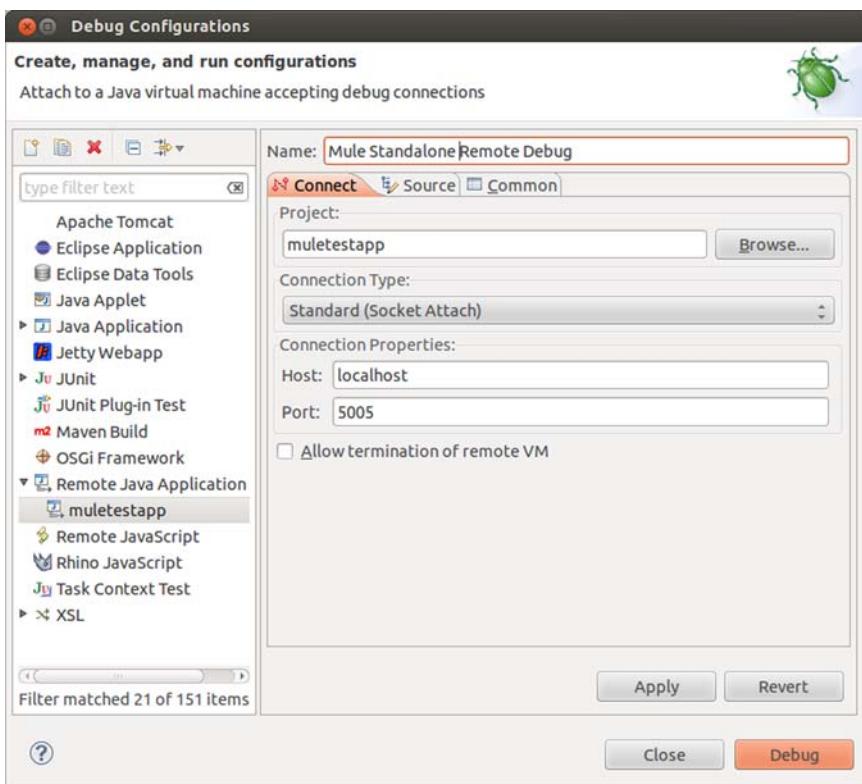


Figure 12.14 Connecting Eclipse to Mule standalone for remote debugging

One powerful debugging tool to consider is conditional breakpoints. Consider the case where you have multiple first-successful routing message processors (discussed in section 5.5.1) used in several flows, and you want to suspend the execution only when one of them gets used in a particular flow. This is when you'll need a conditional breakpoint. Figure 12.15 shows you how to set a breakpoint in the source code of `org.mule.routing.FirstSuccessful` that will be triggered only when it's used in a flow named `orderDeliveryFlow`.

GET OUT OF MY SCRIPT! Scripted components and transformers are convenient, but be aware that you won't be able to put breakpoints in the scripts, and this is true for whatever scripting language you use (MEL, Groovy, or any other ones supported by Mule). This warning should serve as a reminder that scripts must be kept short; if your scripts turn into classes in disguise, rewrite them as Java classes and unit test them properly. The added bonus will be the ability to step debug them!

Conditional breakpoints are the way to go when you want to suspend execution in Mule classes that are used over and over. With the debugging tools we've covered in

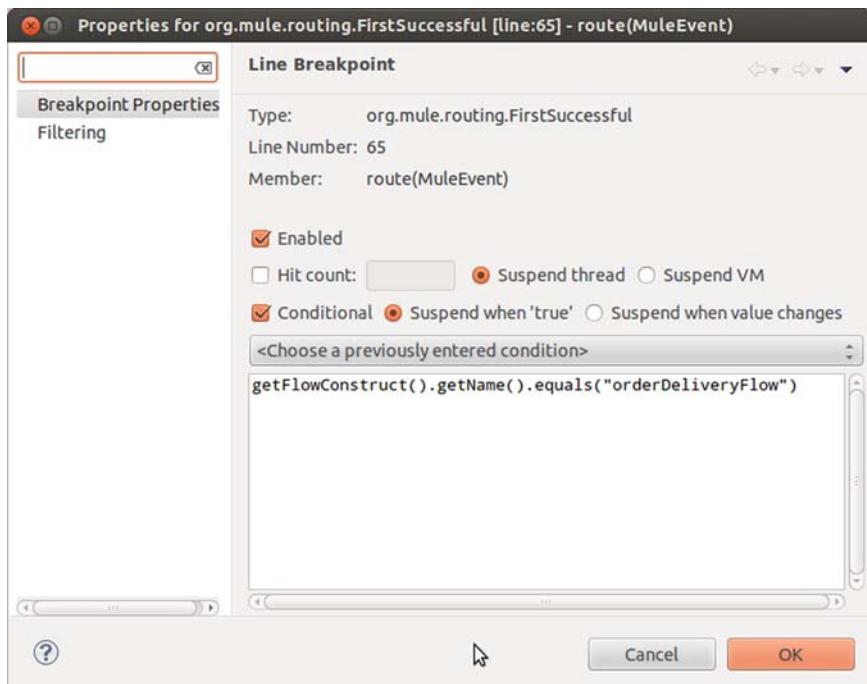


Figure 12.15 Setting a conditional breakpoint in a generic Mule class

this section in hand, you should be able to gain a deep insight into what's happening when Mule runs your applications.

12.6 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 in which the drawbacks of such a coupling are overcome by its benefits.

You saw in this chapter how it's possible to use this API to perform actions of all kinds, including exploring the inner parts its context gives access to, communicating with a running Mule instance, intercepting messages, and listening to internal events. You also learned how to customize Mule's own data persistence. Finally, you discovered how to test Mule applications, from both functional and performance standpoints, and how to debug them.

We haven't covered using the Mule API for creating new connectors, and this is for a reason; Mule offers a specific code generation tool, DevKit, that allows for the creation of new connectors by writing annotated POJOs! This is what we'll cover in the next chapter.

13

Writing custom cloud connectors and processors

This chapter covers

- Building a cloud connector
- Rendering a cloud connector configurable
- Understanding connection management for a cloud connector
- Creating Mule's moving parts within a cloud connector
- Integrating a cloud connector with Studio

You can imagine Mule living on an idyllic Tuscan hill, a place full of tradition that you feel you completely understand. But sooner or later, Mule will need to cross tundra and jungles. The internet state of the art is continuously evolving, leaving behind well-known technologies and stepping into the wild jungle.

Following this evolution, you've seen cloud computing become one of the most relevant technological trends in recent years. New cloud services emerge on a daily basis, providing all kinds of functionalities: social network updates, stock trading, payments, voice calls and SMS, document-oriented databases, and so on. At the

same time, your company may be using several internal protocols that Mule may not support out of the box.

Mule already has a good set of connectors for the most popular cloud services; we discussed them in chapter 3. But it doesn't matter how many Mule provides or how fast the Mule community publishes new ones in the MuleForge. The tsunami of IT evolution will always be faster than the efforts to keep the Mule repository of connectors updated. Instead of how many transports or modules Mule provides, the question probably should be how adaptable to change is Mule?

As Leon C. Megginson said when paraphrasing Charles Darwin, "It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."¹

Not wishing to disappoint the British naturalist, Mule not only tries to be strong and intelligent; it also tries hard to be easily adaptable. Mule 3 has been delivered with the Anypoint Connector DevKit, an annotation-processing tool created to boost developer productivity by automatically generating the boilerplate code and the XML schemas needed to create Mule modules. With the exception of a few annotations, modules created using the DevKit are POJOs.

In this chapter, we'll briefly discuss how DevKit simplifies Mule module development and how to write new Mule modules using the cloud connector DevKit. At the end, we'll create a cloud connector to use the BreweryDB cloud service inside Mule.

13.1 Simplifying Mule development with the DevKit

Mule stands on a solid base: the Spring Framework and its extensible XML authoring. They give Mule a powerful but sometimes complex extension mechanism. All core Mule modules use this mechanism, which usually requires creating the following:

- An XML schema that will most often extend abstract types of Mule
- A namespace handler for the specific namespace of your module
- A bean definition parser for the namespace handler that will parse the schema elements
- Customized connectors or message processors

Some of these tasks are not trivial and require a deep knowledge of the Mule and Spring internals. They also require you to extend Mule or Spring classes, making your code more tightly coupled.

In Java 5, a pluggable annotation processor for the compiler was introduced for the first time. Thanks to this, annotations can be processed at compile time by plugins that can generate new code or resources, modify the annotated classes, do additional checks, and so on. DevKit (www.mulesoft.org/documentation/display/current/Mule+DevKit) is one of those annotation processing tools that autogenerates the Mule- and Spring-specific code required to write Mule modules, effectively

¹ Leon C. Megginson, "Lessons from Europe for American Business," *Southwestern Social Science Quarterly* (1963) 44(1): 3-13, at p. 4.

Traditional Mule extensions

The traditional extension mechanism for Mule isn't deprecated by DevKit. All the moving parts included in the Mule core are done this way. But using DevKit is the preferred way to write extensions.

Even if there's a small subset of situations where it would make sense to write an extension, such as for transaction support and automatic reconnection of message sources, this will lead the extension to a Mule Studio dead end, as it isn't supported.

reducing your module to simple POJOs. You can find this process graphically described in figure 13.1.

You can see that the annotation parser will read the POJO, looking for classes marked with the DevKit set of annotations. Then it'll generate the Spring extension modules that will provide your Mule applications with the different elements you created. The only code you should write here is your annotated POJO. Mule will continue to apply the best practice of trying to keep your code as decoupled as possible.

The DevKit annotation parser, like other standard Java annotation processors, is usually executed at build time (more specifically in the generate-sources phase). Depending on how you build your projects, you'll have to choose between these options:

- The standard Java command `apt` if you're using Java 1.5 or 1.6, or the `javac` command if you're using Java 1.7 or any newer version. You'll have to run it manually or include it in your build scripts.
- The annotation parser support in your favorite IDE.

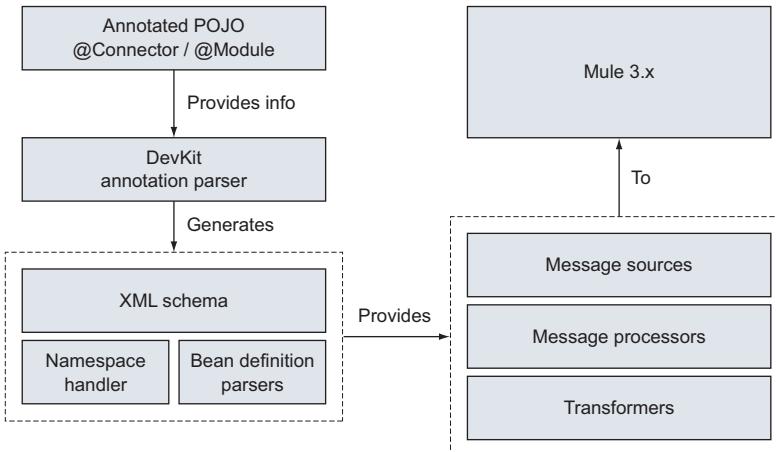


Figure 13.1 A diagram of the relations of DevKit's annotation parsers

- A Maven plugin that automates the process. This is by far the most recommended choice. If you bootstrap your projects using the Maven archetype, as you'll learn in the next section, all the Maven configuration is done automatically. At the moment, this option requires Java 1.6.

At the end of the process of annotation parsing, you'll get all Spring extension-specific parts (XML schema, namespace handler, bean definition parsers) automatically generated and ready for use. This will effectively create message processors and sources. They will have the form of your own namespace, such as `google-tasks`, with custom processors or sources like `google-tasks:get-task-list` and `google-task:delete-task-from-list`.

To better understand what a cloud connector can provide to your Mule application, let's take a look at some of the processors of the Mule Git Connector (<https://github.com/mulesoft/git-connector>), a connector for interacting with Git² repositories that has been built using this technique:

```
<git:clone
    uri="git@github.com:mulesoft/s3-connector.git"/>           ↪ Clone repository
                                                               into new directory
<git:create-branch
    name="myexperiment">                                         ↪ Create local branch
<git:push
    remote="origin"/>                                           ↪ Update remote refs along
                                                               with associated objects
```

Now you understand how the DevKit's annotation parser helps to simplify the development of a cloud connector. Next, let's see how to write a cloud connector.

13.2 Introduction to authoring cloud connectors

Mule cloud connectors are POJOs. What makes these objects special are the annotations that will mark some parts of your code (classes, methods, and members) to be treated in a special way by the annotation processor.

Let's get in touch with a couple of annotations that could be applied to your class:

- `@Module`—Used to provide transformers, filters, routing, custom business logic, and sessionless endpoints. If you're writing a JSR 303 validator, you should use this annotation.
- `@Connector`—Adds out-of-the box connection and session management to the features of `@Module`. `@Connector` would be the right annotation to create an Infinispan transport.

Although `@Module` annotates a Java class to export its functionality as a Mule module, activating the annotation parser for further annotations inside the class, `@Connector` also forces the class to implement some methods that should maintain a permanent connection. In listings 13.1 and 13.2, you can see examples of how to use them.

² Git is a free, open source, distributed version control system.

Listing 13.1 Class annotated to create a Mule cloud connector

```
@Connector
  (name="myconn",
   friendlyName="MyConnector",
   schemaVersion="1.0-SNAPSHOT")
public class MyConnConnector
```

 Marks class as Mule
cloud connector

Listing 13.2 Class annotated to create a Mule module

```
@Module(name = "myMod",
         friendlyName = "MyModule")
public class MyModule
```

 Marks class as Mule
module

Both annotations will let the annotation parser know that the class should be inspected for more annotations. Both annotations have in common some useful attributes:

- **name**—Defines the name the module or connector will have. In the previous examples, the connector namespace will look like `<myConn:processor attribute="value">` and the module like `<myMod:processor attribute="value">`.
- **description**—Used in places where extended information is required, such as documentation.
- **friendlyName**—Sets the name that Mule and Mule Studio will use for labels when a human-friendly name should be shown.
- **minMuleVersion**—Establishes the minimum Mule version required for this module or connector to work.
- **configElementName**—You'll find out how later how to set the configuration parameters of a connector or module; this attribute will set the element name that'll be used to generate the configuration that is, by default, `config`.
- **namespace**—Defines the URI for the namespace of the connector or module; by default it's `http://www.mulesoft.org/schema/mule/${name}`, in which `${name}` represents the value passed in the name attribute.
- **schemaLocation**—Defines the location of the schema file that will describe the namespace. By default, it'll be `http://www.mulesoft.org/schema/mule/${name}/schemaVersion/mule-${name}.xsd`.

To write an extension for Mule without custom message sources, processors, or transformers would probably make no sense. Now that you know how to annotate your class for the basic definition of a module or connector, you'll learn how to annotate your methods to handle connections, to create transformers or custom message processors, or to specify how the extension should be configured:

- Connector-related annotations (`@Connect`, `@Disconnect`, `@ValidateConnection`, `@ConnectionIdentifier`, and `@InvalidateConnectionOn`) are used to generate the connection management using a few POJO methods.
- The `@Processor` annotation can mark a method to be used to generate a general-purpose message processor.

- Argument-passing annotations (@Payload, @InboundHeaders, @Invocation-Headers, and @OutboundHeaders) can mark arguments for receiving the payload, headers, or flow variables.
- REST endpoint annotations (@RestCall and @RestHeaderParam) are used for easier creation of message processors that will perform REST calls.
- The @Transformer annotation identifies a method that will become a Mule transformer.
- The @Configuration annotation sets the arguments that will configure the connection or module to be created.
- Message source annotations (@Source) create a message source that will pass to an inbound endpoint received or generated event.

You've learned how to take the first step toward creating your Mule extension. Now let's see how to make the extension configurable.

13.2.1 Rendering extensions configurable

Probably one of the main reasons for writing a Mule extension is reusability. For better reusability, any value susceptible to being changed in the different uses the extension could have should be configurable.

As with every other mechanism in the Mule cloud connector DevKit, configuration is driven through annotations. Any class member annotated with @Configurable will be considered parameterizable and will be available in the config element. For instance, let's add two configurable values to this connector:

```
@Configurable  
private String parameterOne;  
  
@Configurable  
private String parameterTwo;
```

For these parameters, once the constructor is built, the config element would look like this:

```
<myconn:config  
    parameterOne="valueOne"  
    parameterTwo="valueTwo" />
```

When the connector is configured with this element, the members parameterOne and parameterTwo of your POJO will be populated with the values valueOne and valueTwo. It's worth remembering that if you prefer for any reason to change the name of the config element instead of using the member name, you can pass the attribute configElementName to the Connector annotation.

We'll now take a look at how DevKit helps you simplify connection management.

13.2.2 Managing connections

If you're writing a Mule extension for a connection-oriented protocol, you'll probably be writing a Mule connector. If this is the case, you may need some annotated methods to manage the connections.

Connection handling is a complicated task when you have to deal with pools of connections in a SEDA architecture. Thankfully, Mule will simplify the task by providing your Mule connector with a few annotations that will reduce the required work to the bare minimum. Let's start with the two most relevant annotations:

- `@Connect`—Can be applied to only one publicvoid method that throws `org.mule.api.ConnectionException`. Will be invoked when the connection manager needs to open a new connection.
- `@Disconnect`—Applied to only one publicvoid method with no parameters. Will be called to dispose of a connection.

Let's put them in context in an example connector.

Listing 13.3 Using `@Connect` and `@Disconnect`

```
@Configurable
boolean celsius;

MyExternalApi myExternalApi = new MyExternalApi();

String sessionId;
@Connect
public void connect
    (@ConnectionKey String username,
     String password)
    throws ConnectionException {
    sessionId = myExternalApi.connect(username, password);
}

@Disconnect
public void disconnect() {
    if(this.sessionId != null ) {
        myExternalApi.logout(sessionId);
    }
}
```

The code snippet shows a Java class with two methods: `connect` and `disconnect`. The `connect` method is annotated with `@Configurable` and `@Connect`. The `disconnect` method is annotated with `@Disconnect`. Red callout boxes with numbers 1 and 2 point to the annotations: 1 points to `@Connect` and 2 points to `@Disconnect`. Another red callout box labeled "Set celsius as configurable" points to the `@Configurable` annotation. A red callout box labeled "Annotate argument as connection key" points to the `@ConnectionKey` annotation on the `username` parameter of the `connect` method.

The `connect` method, annotated with `@Connect` at ①, will be in charge of the creation of a connection when required. At the same time, you mark the opposite operation for disconnection at ②. You can find in the `connect` method the arguments `username` and `password`. The `connect` arguments are a special case. They can be passed as if they were `@Configurable` annotated members (as in listing 13.4) and as if they were arguments of the different processors present in the connector (as in listing 13.5).

Listing 13.4 Using connection arguments in the config element

```
<weather:config celsius="true" />

<flow name="configInMp">
    <weather:get-weather
        username="organization1" password="testPassword"
        countryName="Spain" cityName="Malaga" />

    <weather:get-weather
        username="organization2" password="testPassword2"
        countryName="Canada" cityName="Vancouver" />

    <weather:get-weather
        username="organization1" password="testPassword"
        countryName="USA" cityName="New York" />
</flow>
```

The code shows a flow named "configInMp". It contains three sequential "weather:get-weather" steps. Each step has its own configuration block with "username" and "password" attributes. The first step uses "organization1" and "testPassword". The second step uses "organization2" and "testPassword2". The third step also uses "organization1" and "testPassword". To the right of the code, there are three numbered callouts: ① "Open connection with key organization1", ② "Open connection using the key organization2", and ③ "Reuse organizational connection". Callout ① points to the first "weather:get-weather" step. Callout ② points to the second. Callout ③ points to the third, indicating that since the connection for "organization1" is already open from the previous step, it is reused.

Listing 13.5 Using connection arguments as processor attributes

```
<weather:config celsius="true"
    username="xxx" password="yyy" />

<flow name="normalConfig">
    <weather:get-weather
        countryName="Canada" cityName="Vancouver" />
    <weather:get-weather
        countryName="Spain" cityName="Malaga" />
    <weather:get-weather
        countryName="USA" cityName="New York" />
</flow>
```

You may have realized that in listing 13.3 there's an outstanding annotation in the construction arguments: `@ConnectionKey`. When designing a Mule cloud connector, you should keep in mind that the POJO you're writing is used by DevKit; it doesn't represent a connector by itself. Instead, the DevKit connection manager will maintain a pool of instances of this class. To know when one of these instances can be reused, DevKit has to be able to identify the elements in the pool, and this is where the `@ConnectionKey` becomes useful. It'll let DevKit identify each member of the pool, reusing when possible. For instance, in listing 13.4, you start requesting the weather of Malaga at ①. At this point, an instance is created with the pool key `organization1`. The same will happen with `organization2` and `Vancouver` at ②. But in the third call at ③, the `organization1` pool member is already present, and therefore the original connection will be reused.

CONFIGURING RECONNECTION STRATEGIES FOR MULE EXTENSIONS The reconnection strategies we studied in section 9.1.1 can be included in the connector's config element. Unfortunately, as of this writing, the message sources we'll create in section 13.2.7 won't be able to benefit from the reconnection strategies.

This pool can be configured for performance tuning. In each of the DevKit-created cloud connectors, you can find the element connection-pooling-profile with four attributes:

- maxActive—Sets the maximum number of connections allowed at the same time. When set to -1, there won't be a limit.
- maxIdle—Establishes the maximum number of idle connections there can be at the same time. If set to -1, there won't be a limit.
- maxWait—When exhaustedAction (see next) is set to WHEN_EXHAUSTED_WAIT, sets the number of milliseconds to wait for a connection.
- exhaustedAction—Defines how the connector will react when the maximum number of active connections is reached. The possible values are these:
 - WHEN_EXHAUSTED_FAIL—Instructs Mule to throw a NoSuchElementException when the limit for maxActive connections is reached.
 - WHEN_EXHAUSTED_WAIT—Establishes that Mule will wait for a connection for the time set in maxWait; if in that time an available connection isn't found, the same behavior as WHEN_EXHAUSTED_FAIL should be expected. A negative value will set an infinite wait for an available connection.
 - WHEN_EXHAUSTED_GROW—This configuration will create a new connection each time the limit of connections is reached, ignoring the maxActive value. This is the default behavior.

Connection validation (assertion that a connection is alive) and invalidation (establishes the need for reconnection) are handled using the annotations @ValidateConnection and @InvalidateConnectionOn. Although the first one is applied to a method that will return a Boolean if the connection is valid, the second one is applied to methods that are already annotated with @Processor or @Source and if a matching exception is thrown, the connection will be invalidated. Put them in practice in a small weather connector:

```
@Processor
@InvalidateConnectionOn(exception=WeatherException.class)
public Weather getWeather (String countryName, String cityName) {
    return myExternalApi.getWeather
        (sessionId, celsius, countryName, cityName);
}

@ValidateConnection
public boolean isConnected() {
    return myExternalApi.isConnected();
}
```

You learned the mechanisms of connection management to create Mule extensions for connection-oriented protocols. Now let's see how to create processors that can use those connections to invoke logic.

MIXING CONTENT IN EXTENSIONS In the next sections, we'll cover the creation of processors, REST endpoints, transformers, and message sources one by one. This doesn't mean that a connector or module can't have any combination of these. Feel free to mix and match at will.

13.2.3 Creating message processors

The most common use scenario when designing Mule extensions is to create custom message processors. As with the rest of the DevKit, the creation of processors makes use of annotations for a loose coupling between the module's code and Mule.

The methods of your extensions can be designed to be processors by marking them with the annotation `@Processor`. For instance, let's mark a method without parameters to be a processor:

```
@Processor
public void myMethod() {
    // business logic
}
```

This will let you call the business logic in `myMethod` from Mule as a message processor of your extension:

```
<myConn:myMethod />
```

Calling methods without arguments can be useful only up to a certain point. DevKit is capable of discovering the arguments of a method and automatically configuring the processor to be able to accept them as attributes. It also provides two useful annotations that can be used when an argument is optional: `@Optional`, which will mark the argument as optional, and `@Default`, which, if set, will set the default value:

```
@Processor
public void myMethodWithDefaultArg(
    String requiredParam,
    @Optional @Default("1") int optionalParam) {
    // business logic
}
```

Note that the return type of the method is `void`. That means that the message won't alter the payload of the message. If you want to alter the message, you need to set a return type, and the result of the method will become the value of the payload:

```
@Processor
public String myMethodSetPayload() {
    return "I'm a String that will become payload";
}
```

Processor attributes can also be marked to pick up variables, properties, or the payload itself. To pass variables, you should use `@InvocationHeaders`, session properties, and `@SessionHeaders`, and to pass inbound and outbound properties, `@InboundHeaders` and `@OutboundHeaders`, respectively.

@InvocationHeaders, @SessionHeaders, and @InboundHeaders have in common the value attribute. It's used to define what properties will be injected. The acceptable values are as follows:

- A *single named property*—@InvocationHeaders("myHeader") String faz will find and inject the property to the argument and will throw an exception if it's not found. Optionally, all named properties can be followed by the sign ?, which will instruct DevKit not to throw the exception if it can't find the header.
- A *comma-separated list of header names*—@InboundHeaders("myHeader, myOtherHeader?") Map<?, ?> myHeaders.
- A *wildcard expression*—@SessionHeaders("MULE_*") Map<?, ?> myHeaders will inject any property whose name matches the wildcard expression.
- A *comma-separated list of wildcard expressions*—@SessionHeaders("MULE_*, "HTTP_*").

To obtain the payload as an argument of the method, the annotation @Payload should be used. Let's put together a method with the payload and some headers as arguments:

```
@Processor
public String myMethodSetPayload() {
    @Payload String payload,
    @InboundHeaders("MULE_*") Map<?, ?> muleHeaders,
    @InvocationHeaders("header1, optionalHeader2?")
    Map<?, ?> otherHeaders) {

    return "I'm a String that will become payload";
}
```

Now that you're comfortable managing connections and creating message processors, let's look at a special use case: the intercepting message processors.

13.2.4 Creating intercepting message processors

Recall the intercepting message processors from section 12.3.2. You learned in that section when they're useful and how they can decide whether the flow should continue or not. In addition, you implemented an interceptor that stores and replays payloads for similar incoming messages.

For DevKit, the intercepting message processors are a special type of message processor that should have a parameter of type org.mule.api.callback.SourceCallback and the attribute intercepting set to true in the Processor annotation.

Listing 13.6 The cache interceptor using DevKit

```
① @Processor(intercepting=true)
public Object process(SourceCallback afterChain,
    @Payload Object payload) throws Exception {
    Object key = payload;
    Element cachedElement = cache.get(key);
    if (cachedElement != null) {
```

Receives
callback as
a method
argument

Defines method as
an interceptor
implementation

```

        return cachedElement.getObjectValue();
    }

    // we don't synchronize so several threads can compete to fill
    // the cache for the same key: this is rare enough to be
    // acceptable
    Object result = afterChain.process();
    cache.put(new Element(key, result));
    return result;
}

```



Cache miss: invoke next interceptor and return its result

② **after caching its payload**

In the method signature at ①, you can find the return value of the message that will represent the output payload and a `SourceCallback` argument, indispensable for an intercepting processor. At ②, you call the next element in the chain to get the result payload. For more information on intercepting message processors, refer to section 12.3.2.

We've finished looking at the creation of plain message processors. Now let's take a look at the creation of REST consumers, using a specialized form of message processor.

13.2.5 Creating simple REST consumers

You've learned how to create custom message processors with DevKit. The possibilities for the processors are endless, but when writing Mule extensions, one of the more common scenarios will be to write connectors for REST services.

In order to simplify this common case, Mule's DevKit provides a set of annotations that will ease the creation of REST-based connectors. These annotations will be applied to abstract methods whose code will be generated by DevKit, leaving to the developer the task of defining where and how the processor should perform the request, and leaving the dirty work of writing the boilerplate code to the annotation parser.

WS-* AND SPECIALIZED ANNOTATIONS Given the complex nature of WS-* in comparison with REST, this kind of annotation isn't available for WS-*. To write a Mule extension using WS-*, you should rely on the standard `@Processor` annotation.

The main annotation behind this functionality is `@RestCall`. With it, you can define parameterizable calls to REST services like this:

```

@Processor @RestCall(uri =
    "https://www.googleapis.com/language/translate/v2" +
    "?key={apiKey}&source={sourceLang}&target={destLang}" +
    "&q={text}",
    method = HttpMethod.GET,
    contentType ="application/json")
public abstract Object translate(
    @RestUriParam("apiKey") String apiKey,
    @RestUriParam("sourceLang") String sourceLang,
    @RestUriParam("destLang") String destLang,
    @RestUriParam("text") String text)
throws IOException;

```

You apply `@RestCall` to an abstract method to indicate to the DevKit that it has to generate code to realize it. Note that the `uri` attribute of the `@RestCall` has some parts parameterized: `key`, `sourceLang`, `destLang`, and `text`. These parts will be replaced with the values of the arguments marked with the corresponding `@RestUriParam` annotation in the method arguments. At the same time, you request DevKit to return a Java data structure from the `@RestCall` with the attribute `contentType = "application/json"`. This frees your Mule app from the need to use a json-to-object-transformer right after the message processor.

Sometimes, you'll find that there are arguments that are shared by many processors. This is the case of the `key` parameter in the previous example that represents the Google API key. If you have more than one message processor, it's beneficial to configure it once for all the processors. How can you do that?

OAUTH SUPPORT Mule's DevKit features support for OAuth and OAuth2; visit the DevKit documentation site at www.mulesoft.org/documentation/display/current/Authentication+and+Connection+Management.

The answer is related to what you learned in section 13.2.1 about the `@Configurable` annotation. You can mark a member annotated with `@Configurable` with a `@RestUriParam` annotation. With this, you'll be able to use the configured value in the REST URLs:

```
@RestUriParam("apiKey")
@Configuration
private String apiKey;
```

Other possibilities could be considered, like `@RestHeaderParam` that'll define a parameter that will be passed as a header of the requests, and `@RestPostParam` that'll be used to replace parts of the payload with the marked values.

CONFIGURING PARAMETERS FOR ALL ENDPOINTS When configuring parameters for every single endpoint of your module, you can annotate class-level members with `@RestHeaderParam` or `@RestQueryParam`.

One you have a REST processor like this working flawlessly, you have to be prepared for the opposite. You should be able to react to errors. To accomplish this, the `@RestCall` annotation lets you pass the argument `exceptions` that accepts an array of `@RestFailOn`. This annotation, in turn, will accept an `expression` argument with a MEL expression. If the result of this expression is true, an exception will be thrown. Let's clarify this dense explanation with an example:

```
@Processor
@RestCall(uri =
    "https://www.googleapis.com/language/translate/v2" +
    "?key={apiKey}&source={sourceLang}&target={destLang}" +
    "&q={text}", method = HttpMethod.GET,
    exceptions=
        @RestExceptionOn(
            exception = TranslateException.class,
            expression = "#[header:http.status != 200]"))
```

This processor will throw a `RuntimeException` if the resulting status of the call is not 200. If you don't want to throw a runtime exception but another expression, you can use the `exception` attribute:

```
@Processor
@RestCall(uri =
    "https://www.googleapis.com/language/translate/v2" +
    "?key={apiKey}&source={sourceLang}&target={destLang}" +
    "&q={text}", method = HttpMethod.GET,
    exceptions=
    @RestExceptionOn(
        exception = TranslateException.class,
        expression = "#[header:http.status != 200]"))
```

These annotations will let you handle most of the possible scenarios when writing REST extensions for Mule. But if you need to program a special behavior for an endpoint, Mule offers yet another annotation to mark the `HttpClient` the module will use. This way, you avoid creating multiple clients and have the opportunity to perform your own calls or to configure the `HttpClient` to fulfill special needs:

```
@RestHttpClient
HttpClient client = new HttpClient();
```

Now let's see how to use the annotation that will let you create transformers.

13.2.6 Creating transformers

We studied transformers in chapter 4. You know that they change the format or representation of a message's payload. The cloud connector DevKit allows you to write transformers that will be available to you as message processors and that will be also registered in Mule with the rest of the transformers; therefore, they can be used automatically by Mule, such as in autotransformers.

To create a transformer, you need a static public method in your Mule extension that doesn't return `Object`. These constraints are the result of the internal nature of Java and Mule itself. Once you have the method, you need to annotate it with the `@Transformer` annotation. The arguments can be annotated with the same annotations for injection we studied in section 13.2.3 for message processors.

Let's annotate a method to convert from array to `List`:

```
@Transformer(sourceTypes = { Object[].class })
public static List transformArrayToList(@Payload Object[] payload) {
    return Arrays.asList(payload);
}
```

In this example, you require Mule to pass as a payload an object with a specific type. If you want to support multiple input types—for instance, if you want two specific types of fruit—you can use the `sourceTypes` argument of the `@Transformer` annotation:

```
@Transformer(sourceTypes = { Apple.class, Banana.class })
public static FruitBowl transformFruitToBowl(@Payload Fruit payload) {
    FruitBowl bowl = new FruitBowl();
```

```

        bowl.addFruit(payload);
        return bowl;
    }
}

```

If instead of enforcing source types what you want is to receive any kind of object, you could set the type of the payload argument to `Object` and not use the `sourceTypes` attribute of the transformer.

This is the last of the sections covering how to manipulate messages. Now let's *create* messages by annotating methods to create message sources.

13.2.7 Creating message sources

You've learned so far how to create custom processors, simple REST consumers, and transformers. There's one point they have in common: they all perform tasks with existing messages. Message sources will instead *generate* messages. This is easy to understand once you realize that you used message sources all through the book in the form of inbound endpoints.

To create a message source, you'll need a method that accepts at minimum a `SourceCallback` attribute. Annotate this method with `@Source`. Once you've annotated it, in the code of your method you need to call the `SourceCallback` whenever you want to generate a message. For example, when working with Unix sockets, the callback would be invoked when a message is received. Or when working with a custom cron implementation, the message would be generated when the timer triggers an event.

`SourceCallback` provides one overloaded method in three forms:

- `Object process()`—Will send an empty message to Mule
- `Object process(Object payload)`—Will send a message with the passed payload to Mule
- `Object process(Object payload, Map headers)`—Will send a message with the passed payload and inbound properties to Mule

Now that you know how to use `SourceCallback`, let's create a message source that will generate messages every *interval* seconds:

```

@Source
public void subscribeTopic(
    int interval,
    final SourceCallback callback) {

    TimerTask task = new TimerTask() {
        public void run() {
            try {
                callback.process();
            } catch (Exception e) {
                // Process error
            }
        }
    };
    Timer timer = new Timer();

```



```

        timer.schedule(task, interval);
    }
}

```

This will let you effectively create inbound endpoints. This is the last of the element creation mechanisms we'll cover before discussing how to integrate your extensions with Mule Studio.

13.2.8 Integrating Mule extensions with Mule Studio

If you want to publicly distribute your Mule extension, or if you want to have a fancy-looking extension, you'll need to use the different visual interface customization annotations of the DevKit package: `org.mule.api.annotations.display`. With them, you'll be able to customize the user experience in Mule Studio.

The available annotations work at two different levels: the extension level and the field/parameter level. Let's first cover the extension-level annotations:

- `@Category`—Establishes the palette in which your Mule extension will appear
- `@Icons`—Sets the icon set that will be used for the connectors, endpoints, and transformers of your extension

The `@Category` annotation accepts two attributes: `name` and `description`. They should be values within a specific set of values; otherwise, they'll only change the Java-doc of the extension, but not the palette in Mule Studio. The acceptable values are shown in table 13.1.

Table 13.1 Valid values for the `@Category` annotation

name	description
Endpoints	<code>org.mule.tooling.category.endpoints</code>
Scopes	<code>org.mule.tooling.category.scopes</code>
Components	<code>org.mule.tooling.category.core</code>
Transformers	<code>org.mule.tooling.category.transformers</code>
Filters	<code>org.mule.tooling.category.filters</code>
Flow control	<code>org.mule.tooling.category.flowControl</code>
Error handling	<code>org.mule.tooling.ui.modules.core.exceptions</code>
Cloud connectors	<code>org.mule.tooling.category.cloudconnector</code>
Miscellaneous	<code>org.mule.tooling.ui.modules.core.miscellaneous</code>
Security	<code>org.mule.tooling.category.security</code>

The `@Icons` annotation will accept large (48 x 32) and small (24 x 16) attributes for connector, transformer, and endpoint. If this annotation isn't present, DevKit will try to find the icons in the icons directory at the root of your project, concatenating the connector or module name with the element type and the size; for example,

brewerydb-connector-24x16.png or brewerydb-transformer-24x16.png. All icons should use the PNG file format.

At the configuration field or parameter level, you'll be able to annotate with the following:

- @Password—Will mark the field or parameter as a password that should have special UI handling, such as showing asterisks instead of the real values.
- @Path—Any field or parameter marked with this annotation will have a studio file/directory chooser dialog.
- @Text—Set a field or parameter as large text. This will effectively set the input as a multiline input box in Mule Studio.
- @Summary—Add information about the field or parameter to be shown in Mule Studio as summary documentation.

It's also worth noting that DevKit requires a correct license when building extensions. This license will be shown, and should be agreed to, when the extension is installed in Mule Studio. The steps that DevKit will perform to find the correct license for the project are the following:

- 1 Try to use a LICENSE.md file in the same directory as the pom.xml.
- 2 Check for a LICENSE.txt beside the pom.xml.
- 3 If the pom.xml file contains a Licenses section, try to download the licenses.

At this point, you know the more useful mechanisms for creating your Mule extensions using DevKit. Now let's jump into the action, creating your own connector in the next section.

13.3 Creating a REST connector

Prancing Donkey, even though it's a top-tech company in all management matters, still brews its exquisite beer with the original recipe and traditional process that has been used for generations. Over the years, this attachment to authenticity has created a wealth of truly loyal customers. But the managers at Prancing Donkey want to keep an eye out for the preferences of the new generation, and more specifically for the taste of selective customers of regional breweries.

The software architects of Prancing Donkey have decided, after evaluating different options, that the most favorable approach would be to use the information available in BreweryDB (www.brewerydb.com/), an online and API-accessible database that contains a vast amount of data about beer and breweries.

MAKE IT PRETTY! Your Mule extensions can be customized to use custom icons and friendly names, or to change the category in the palette of Mule Studio. You can find more information about this at the DevKit documentation site (<http://mng.bz/cXHG>).

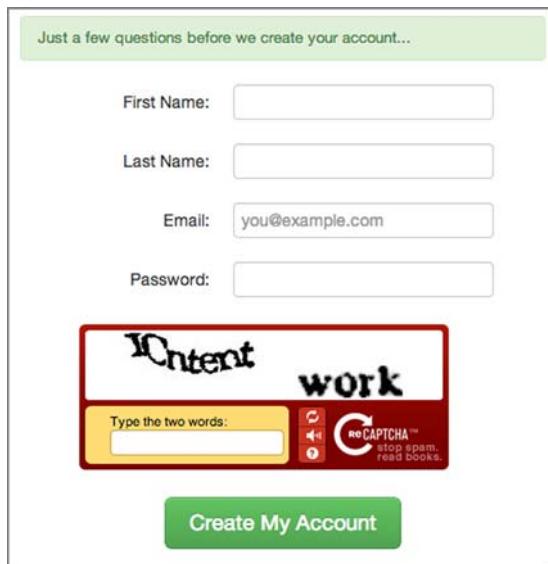


Figure 13.2 Signing up at **BreweryDB**

The Prancing Donkey software architects have decided to create a Mule extension to use BreweryDB in their research. This task could potentially be completed by writing an XML schema, a namespace handler, a bean definition, and some customized message processors. Following this process can take up to a few days of work, depending on how experienced the assigned developer is with the powerful but complex Spring Framework extension mechanism.

Thankfully, they were familiar with the Mule DevKit before starting to work on this project; by using it, they can complete this task in much less time. Let's start from the beginning by signing up at BreweryDB via www.brewerydb.com/auth/signup, as in figure 13.2.

Once you're signed up as a user, you need to register your application with BreweryDB. Register it by visiting www.brewerydb.com/developers/apps and clicking Register a New App to get a dialog like the one shown in figure 13.3.

At the end of the process, you'll have the API key you need to access BreweryDB using Mule. At this point, you should start to work on your Mule extension. Let's create the skeleton with DevKit's Maven Archetype. This will provide you with a working project that'll be able to parse the DevKit annotation in your class and generate the subsequent boilerplate to have a full-fledged Mule extension:

```
mvn archetype:generate \
-DarchetypeGroupId=org.mule.tools.devkit \
-DarchetypeArtifactId=mule-devkit-archetype-cloud-connector \
-DarchetypeVersion=3.4.0 \
-DarchetypeRepository=http://repository.mulesoft.org/releases/
```

The screenshot shows a registration form titled "Register A New App". It includes fields for "App Name", "Description of App" (with a large text area), "Website", "App Icon" (with a "Choose File" button and a "No file chosen" message), and two buttons at the bottom: "Submit For Approval" (red) and "Close".

The "Description of App" field is currently empty.

Figure 13.3 Registering an application at BreweryDB

Maven will ask a few questions at this point:

- `artifactId`—Use `mule-module-brewerydb`.
- `version`—The default value of `1.0-SNAPSHOT` is fine, so press Enter.
- `muleConnectorName`—Use `Brewerydb`.

DON'T USE PATHS WITH SPACES DevKit won't be able to work properly on directories that contain spaces in their full path, such as `C:\Users\Muleteer\My Documents` or `/home/muleteer/My Downloads/mule-modules`. It's highly recommended to use a path without spaces in all Mule development and deployments.

The process should end with a `Build Success` message that will let you know that the skeleton is ready to work. What you've obtained is a Maven project preconfigured with some plugins to read classes in the Java directory in order to find `@Connector` or `@Module` annotations and generate the corresponding extensions for them.

You can find a sample skeleton of your connector in `src/main/java` under the package `org.mule.modules` with the class name `BrewerydbConnector` and an example test case in the corresponding package under `src/test/java`.

Now let's take a look at the API documentation of BreweryDB (www.brewerydb.com/developers/docs). You should find the common configuration of the endpoints to establish them as @Configurable elements in your connector. It seems that the only common configuration element is the ApiKey, so create a @Configurable element in your connector for it:

```
@Configurable
@RestQueryParam("apiKey")
private String apiKey;
```

1 Set property as a
configurable element

At ①, you use the annotation we discussed in section 13.2.1 to set this property as a configurable element; at the same time, you establish that this property should be passed as a query parameter in your REST requests, as you did in section 13.2.5.

At this point, you're ready to create your first endpoint. To meet the requirements of Prancing Donkey, you need an endpoint that tells you which beers are in the market. Visiting again the API documentation of BreweryDB, you can find the Get Beers endpoint (www.brewerydb.com/developers/docs-endpoint/beer_index#1), which will give you the information you need. As the endpoint is a simple REST call, call the required method with the annotation covered in section 13.2.5:

```
@Processor
@RestCall(uri = "http://api.brewerydb.com/v2/beers",
    method = HttpMethod.GET,
    contentType ="application/json",
    exceptions={@RestExceptionOn(
        expression=
            "#[message.inboundProperties['http.status'] != 200]")}
public abstract String getBeers(
    @Optional @RestQueryParam("p") int page,
    @Optional @RestQueryParam("ids") String ids,
    @Optional @RestQueryParam("name") String beerName,
    @Optional @RestQueryParam("abv") String abv,
    @Optional @RestQueryParam("ibu") String ibu,
    @Optional @RestQueryParam("apiKey") String glasswareId,
    @Optional @RestQueryParam("glasswareId") String srmlId,
    @Optional @RestQueryParam("srmlId") String availableId,
    @Optional @RestQueryParam("availableId") String styleId,
    @Optional @RestQueryParam("styleId") String isOrganic,
    @Optional @RestQueryParam("hasLabels") String hasLabels,
    @Optional @RestQueryParam("year") int year,
    @Optional @RestQueryParam("apiKey") int since,
    @Optional @RestQueryParam("status") String status,
    @Optional @RestQueryParam("order") String order,
    @Optional @RestQueryParam("sort") String sort,
    @Optional @RestQueryParam("withBreweries") String withBreweries
) throws IOException;
```

Following the BreweryDB documentation, the endpoint is marked with some optional parameters to refine the search, all of them passed as query parameters in a GET call.

At this point, you should document your connector. DevKit will try to enforce best practices at build time by checking that the Javadoc is preset for each of the members,

methods, and arguments. If you want to disable this feature, you can pass `-Ddevkit.javadoc.check.skip=true` to the Maven command line. For instance, your get-beers processor Javadoc will look like this:

```
/**  
 * Gets a listing of all beers. Results will be paginated with 50  
 * results per page. One of the following attributes must be set:  
 * name, abv, ibu, srmId, availabilityId, styleId.  
 * {@sample.xml ../../../../../doc/brewerydb-connector.xml.sample  
 * brewerydb:get-beers}  
 *  
 * @param page Page Number.  
 * @param ids ID's of the beers to return, comma separated.  
 * @param beerName Name of a beer.  
 * @param abv ABV for a beer  
 * @param ibu IBUs for a beer.  
 * @param glasswareId ID for glassware.  
 * @param srmId ID for SRM.  
 * @param availableId ID for availability.  
 * @param styleId ID for style.  
 * @param isOrganic Certified organic or not (Y/N).  
 * @param hasLabels Has a label (Y/N).  
 * @param year Year vintage of the beer.  
 * @param since What has been updated since that date.  
 * @param status Status of the brewery.  
 * @param order How the results should be ordered.  
 * @param sort How the results should be sorted.  
 * @param withBreweries Include brewery information.  
 * @return JSON structure with the beers in the db.  
 * @throws IOException on error connecting to the api.  
 */
```

Now you're ready to build your connector. A simple `mvn clean install` should be enough to generate either a JAR file for standard Java distribution or a ZIP to distribute as a Mule Studio plugin. In order to generate an update site directory to install your plugin in Mule Studio, pass the environment property `devkit.studio.package.skip` to DevKit as false through Maven:

```
mvn -Ddevkit.studio.package.skip=false clean install package
```

Let's put into practice what we covered in section 13.1 to import your recently created connector in Mule Studio. In Mule Studio, open the Install dialog (see figure 13.4) by using the Help > Install New Software... menu entry. Once there, create a new site by clicking Add and browsing the filesystem, looking in the target folder of your project for an update site. Once completed, you can select your recently created install site in the drop-down menu, select the brewerydb cloud connector, and install.

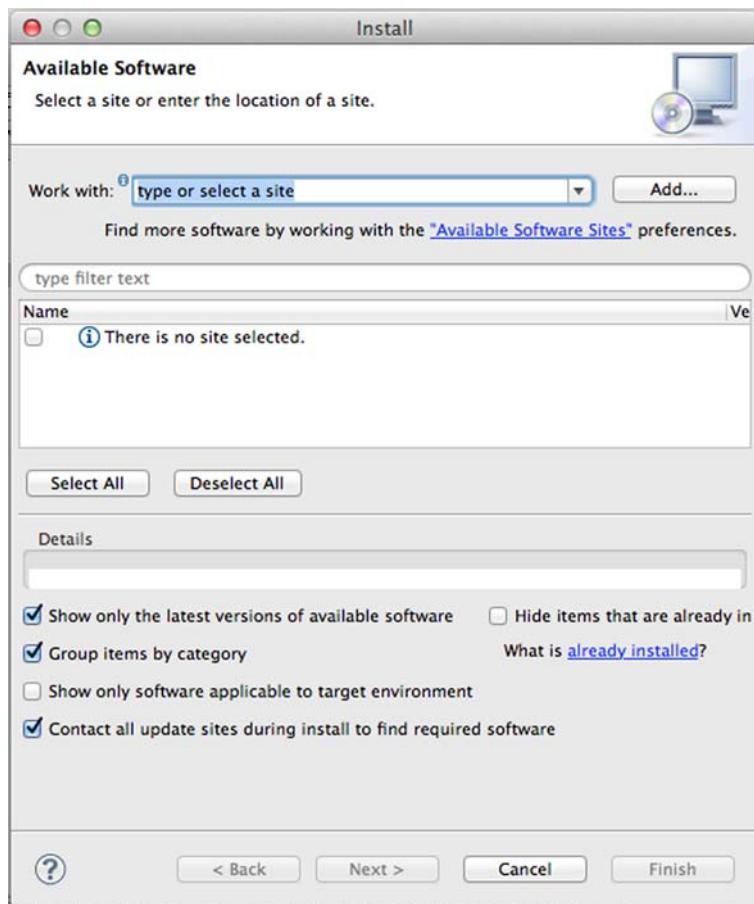


Figure 13.4 Mule Studio's Install dialog



Figure 13.5 Mule Studio's cloud palette

At this point, you can create a new project to use the brewerydb connector. First, create a new connector using the Mule Studio wizard. Then create a flow using your connector, which you'll find in the Cloud Connectors category in the palette, as shown in figures 13.5 and 13.6; also, you can use it in plain XML, like this:

```
<brewerydb:config
    name="Brewerydb" apiKey="YOURAPIKEY" />

<flow name="getBeersFlow">
    <http:inbound-endpoint
        exchange-pattern="request-response"
        host="localhost" port="8081" />
    <brewerydb:get-beers config-ref="Brewerydb" />
</flow>
```

Then you can run the project, access with a browser the address you set up in the inbound endpoint, and see your brand-new connector in action.

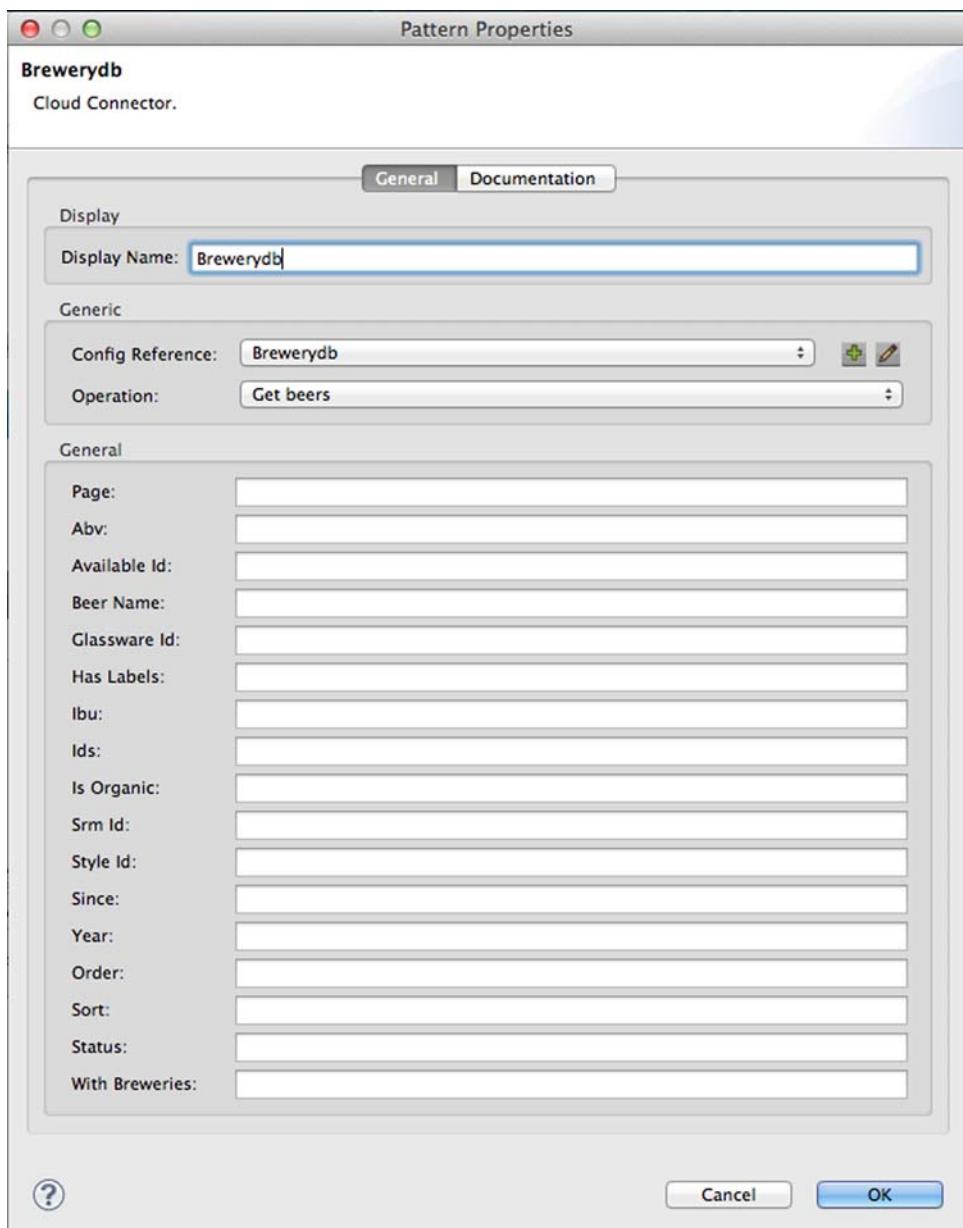


Figure 13.6 The **Brewerydb** connector in Mule Studio

13.4 **Summary**

You saw in this chapter how Mule simplifies the creation of extensions; the different annotations available to create connectors, endpoints, processors, and transformers; and how they're processed by the DevKit annotation parser to create a Mule extension. You learned how to create a REST-based extension for Mule for BreweryDB and how to install it in Mule Studio.

Extensibility is a key feature for any kind of framework, and in Mule it plays a first-class role. DevKit simplifies the creation of Mule extensions so much that the considerable amount of knowledge required to build an extension in the traditional way is reduced to only using a small set of annotations and a Maven plugin. Let's now see how Mule simplifies other common integration challenges—business orchestration, event processing, rules evaluation, and scheduling.

Augmenting Mule with orthogonal technologies

This chapter covers

- Using business process management systems with Mule
- Complex event processing in Mule applications
- Business rules evaluation in Mule applications
- Scheduling with Mule

Mule, as you've seen through the course of this book, offers a wealth of features that simplify the architecture, implementation, and deployment of integration applications. We'll now take a look at some technologies that complement these applications, often simplifying their implementation or offering features that Mule doesn't offer directly. We'll touch on their high-level features and where those commonly intersect with Mule, providing a foundation for your own experiments. Finally, these technologies are often used in conjunction with Mule, so some awareness of them might simplify a future implementation or give you context when you're ramping up on a Mule project that's using them.

14.1 Augmenting Mule flows with business process management

Business process management (BPM) systems or workflow engines, like JBoss's jBPM or Activiti, provide orchestration functionality that complements Mule flows. While most of your integration orchestration requirements can be handled with a Mule flow, there are a few situations in which a dedicated workflow engine might be necessary. One situation is where a workflow requires human intervention. For instance, a manager might need to manually approve an insurance claim. Another situation is when fine-grained management and monitoring of a workflow is required. You might have an automated stock trading system, for instance, that gives the operator the ability to pause or abort processes in a certain state. While both of these use cases can be accomplished with Mule's management facilities, dedicated workflow engines typically provide better abstractions—like prebuilt GUIs and dedicated APIs—to simplify such implementations.

Mule provides a generic interface to interact with BPMs. There's support out of the box for jBPM, along with a community-contributed module for Activiti. Integrating support for other BPMs is typically a trivial exercise.¹ The jBPM module, which we'll look at in this section, provides functionality to start processes from a Mule flow as well as to generate Mule messages from within a jBPM process.

Let's see how Prancing Donkey has begun to introduce jBPM into their infrastructure to augment Mule flows. Prancing Donkey is using Mule flows to orchestrate some of their back-end, order-processing integration with external systems. Part of this submission involves invoking operations with Salesforce and NetSuite's APIs. Prancing Donkey's business operations team wants insight into this process. They'd like to see the current orders in-flight, as well as have the ability to abort or pause orders. While the current order fulfillment only consists of two steps, Prancing Donkey's developers and business operations managers foresee this becoming more complicated. As such, they've made the decision to take advantage of jBPM and refactor order fulfillment into a business process. In the next listing, let's take a look at the business process they've defined.

Listing 14.1 jBPM process definition for order processing

```
<process name="submitOrder"
    xmlns="http://jbpm.org/4.3/jpd1">
    <start name="start">
        <transition to="fork"/>
    </start>
    <fork name="fork">
        <transition to="sendOrderToSalesForce"/>
        <transition to="sendOrderToNetSuite"/>
```

← Name of the
process is
submitOrder

¹ Users wishing to implement support for another BPM engine need to implement the `org.mule.module.bpm.BPMS` interface.

```

</fork>

<mule-send name="sendOrderToSalesForce"
            endpoint="jms://crm.customer.create"
            payload="incoming"
            exchange-pattern="request-response">
    <transition to="join"/>
</mule-send>

<mule-send name="sendOrderToNetSuite"
            endpoint="jms://erp.order.record"
            payload="incoming"
            exchange-pattern="request-response">
    <transition to="join"/>
</mule-send>

<join name="join">
    <transition to="dispatchOrderCompletedEvent"/>
</join>

<mule-send name="dispatchOrderCompletedEvent"
            endpoint="jms://topic:events.orders.completed"
            payload="incoming" exchange-pattern="one-way">
    <transition to="end"/>
</mule-send>

<end name="end"/>

</process>

```

Send message synchronously to crm.customer.create queue and wait for response on temporary queue

1 Send message synchronously to erp.order.record queue and wait for response on temporary queue

2 When responses from both queues received, proceed

3 Generate event message when process id completes

This process will orchestrate the dispatch of orders to Salesforce and NetSuite. A slight change will need to be made to the flows receiving messages from the `crm.customer.create` and `erp.order.record` queues. They'll need to change their `exchange-pattern` to `request-response` from `one-way`, and let Mule use temporary JMS queues to receive the response. They're using jBPM's fork/join feature to block the process until a response is received from both nodes. The final action of the process, at ③, dispatches a message to the `events.orders.completed` JMS topic.

`mule-send` is a jBPM action supplied by the jBPM module. As you can see at ① and ②, it can be used to send or dispatch an arbitrary variable in the process to a Mule endpoint. An analogous `mule-receive` action also exists to receive or wait for a message from Mule.²

Now that the process is defined, you need to configure Mule to use jBPM and trigger the process from a JMS queue.

Listing 14.2 Using jBPM as a BPM engine with Mule

```

<bpm:j bpm name="jbpm"/>
<flow name="orderProcessing">
    <jms:inbound-endpoint queue="order.submit">
        <jms:transaction action="ALWAYS_BEGIN"/>

```

1 Define jBPM as BPM engine

² It's also possible to advance or abort a process directly from a component by accessing the jBPM instance from Mule's registry. Consult the Javadoc for details.

```
</jms:inbound-endpoint>
<bpm:process processName="submitOrder"
  processDefinition="process/order-process.jpd1.xml"/>
</flow>
```

② Advance (begin submitOrder process)

You define the name of the jBPM instance at ①. The process message processor ② will advance the process with the given name in the given jBPM Process Definition Language (JPDL) file. In this case, the process is started. If the process was already running, then this message processor would advance the process. Mule tracks the process instance using the `MULE_BPM_PROCESS_ID` header property.³

Using a BPM engine with Mule is a good choice if your workflow requires human intervention, is particularly complex, might need to be defined by a business analyst, or requires more than basic operational management. State persistence of long-running operations is also a key criterion when considering a BPM. We'll now take a look at another piece of technology that's complementary to Mule's event-driven architecture: complex event processing.

ACTIVITI Mule support for Activiti, a competing BPM engine, is available from the community. To use it, you need to build and install the module locally. It's available on GitHub at <https://github.com/mulesoft/mule-module-activiti>.

14.2 Complex event processing

Complex event processing (CEP) is defined, per Wikipedia, as "...event processing that combines data from multiple sources to infer events or patterns that suggest more complicated circumstances" (see http://en.wikipedia.org/wiki/Complex_event_processing). Common use cases for CEP include fraud tracking by insurance and credit card companies, pattern detection for algorithmic trading engines, and the real-time correlation of geospatially tagged data. Mule, itself an event-driven system, is well suited for integration with CEP engines such as Drools Fusion and Esper. CEP itself is also well suited for monitoring event-driven, asynchronous systems such as Mule—as you'll see in this section.

14.2.1 Using CEP to monitor event-driven systems

Monitoring event-driven systems and processes can be difficult. With traditional TCP applications, like web servers, it's relatively easy to implement a health check to determine if the application is down. The same can't be said about streams of messages or events. This is particularly true when the sources of such events are remote systems over which you have no control. Consider a Mule application that asynchronously aggregates financial data, such as stock quotes and currency data, from various sources. Determining if this data has stopped flowing with a traditional approach might require repeatedly polling the store of the data. This is effective, but is tightly

³ Be sure your jBPM process file ends with the `.jpd1.xml` extension to avoid a hard-to-debug issue about the process not being defined.

coupled to the datastore (what happens when the schema changes?) and requires synchronization of the polling job between multiple nodes of the system.

Let's see how you can use Esper, an open source CEP engine, to monitor a similar stream of data for Prancing Donkey. The following listing uses the Esper module, available as a Mule connector, to subscribe to the event stream being published to the events.orders.completed JMS topic and generate another event when the number of orders falls within a standard deviation outside the average number of orders completed in a one-hour interval.

The Esper module can be installed via your project's Maven pom. Instructions on how to do this can be found here: <http://mng.bz/K26m>. Once that's done, the first thing you need to do is add an Esper configuration to the classpath and configure your Order completion event. Prancing Donkey's esper-config.xml is listed next.

Listing 14.3 The Esper configuration

```
<esper-configuration
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.espertech.com/schema/esper"
    xsi:schemaLocation="
        http://www.espertech.com/schema/esper
        http://www.espertech.com/schema/esper/esper-configuration-2.0.xsd">

    <event-type name="OrderCompletedEvent"
        class="com.prancingdonkey.model.Order"/>
    </event-type>
</esper-configuration>
```



For this example, you reuse the Order class from your canonical domain model to represent the OrderCompletedEvent. A more robust implementation would likely dedicate a class for the OrderCompletedEvent, containing various metadata about the Order's lifecycle. The event type could also be specified as XML or a Java Map. The latter is particularly useful when dealing with JSON messages in conjunction with Mule's built-in JSON transformers, which natively support transformation to and from Maps.

MESSAGE PROCESSOR REFERENCES The -ref in the Esper message processors is a little bit confusing and shouldn't be mistaken for the connector-ref you've seen elsewhere in this book. The -ref here is a DevKit artifact that indicates that the message processor is expecting an object value (usually the result of an expression).

Now let's configure Mule to perform complex event processing on the order completion event stream. You'll implement two flows. The first will consume messages off the events.orders.completed JMS topic and insert the payloads, which are instances of the Order class, into the event stream just defined. The next flow will define an EPL (event-processing language) statement against the event stream. The query will return

a result when less than a standard deviation of orders have completed within a time window of an hour.⁴

Listing 14.4 The Esper configuration

```
<flow name="orderCompletionEventInjection">
    <jms:inbound-endpoint topic="events.orders.completed" />
    <esper:send eventPayload-ref="# [message.payload]"
        eventName="OrderEvent" />
</flow>
<flow name="Event Listener Flow">
    <esper:listen statement=
        "select count(*), stddev(count(*))
         from OrderCompletedEvent.win:length(3600000)
         having stddev(count(*)) > 1" />
    <expression-transformer evaluator="map-payload"
        expression="count(*)" />
    <jms:outbound-endpoint topic="alerts.orders" />
</flow>
```

Inject events into OrderCompletedEvent stream

Generate composite event when less than a standard deviation of events are received in an hour

A composite event, in CEP terminology, is an event generated as a result of other events. In this case, the composite event is the alert you send on the `alerts.orders` topic in the case of a drop in order volume over the course of an hour.

Drools Fusion

The Drools Rules Engine, which ships with Mule, also contains support for CEP via Drools Fusion. See the Drools module documentation (<http://mng.bz/lMcj>) for more details.

Drools Fusion and Esper, despite both being CEP implementations, can complement each other well. Esper, via EPL, excels at event selection, whereas Drools Fusion is commonly used for streaming interpretation of these events. You can use Esper to make sense out of the “noise” of events, and then use Drools Fusion in streaming mode for reasoning over the composite events inferred by Esper.

Complex event processing is a powerful tool that can be applied beyond the realm of monitoring. Let’s see how you can use Esper to perform sentiment analysis from a series of Twitter status updates in real time.

14.2.2 Sentiment analysis using Esper and Twitter

The aggregate of individual social network activity on platforms like Facebook, Twitter, and Google+ can be used to sample the mood of a particular group of the population. A group of people complaining about a particular product on Twitter, for instance, might be an indication that some sort of customer service intervention is required. User-posted Likes on Facebook about a TV show can give television networks

⁴ EPL’s syntax is very similar to SQL.

insight into how popular a particular show or episode is. Twitter's "firehose" stream of tweets is being used by trading firms to predict changes in financial markets.

Mule's synergy of cloud connector support for social networking APIs, CEP support, and event-driven architecture makes it a natural platform to perform this sort of analysis. You've seen previously in this book how Prancing Donkey conducts some of its marketing activity over Twitter. They're also using Salesforce as their CRM. A marketing intern is currently responsible for monitoring Prancing Donkey's Twitter feed and manually creating cases in Salesforce based on who's talking about Prancing Donkey on Twitter. The marketing director wants to automate the triaging of cases based on tweets so that the marketing department can focus on taking the appropriate action. Someone might, for instance, authorize a credit to someone complaining about a shipping issue for an order.

To accomplish this, Prancing Donkey will write a Mule application that will subscribe to the Twitter status stream and select only tweets containing the #prancingdonkey hashtag. A case object will then be created on Salesforce based on the content of the tweet. The next listing shows the implementation.

Listing 14.5 Automatically creating Salesforce cases using Twitter

```
<flow name="StatusIngest" doc:name="StatusIngest">
    <twitter:sample-stream
        config-ref="Twitter"
        doc:name="Twitter Firehose"/>
    <esper:send config-ref="Esper"
        eventPayload-ref="#[payload]"
        eventName="Tweets"
        doc:name="Insert Status Event"/>
</flow>

<flow name="SentimentAnalysis" doc:name="SentimentAnalysis">
    <esper:listen config-ref="Esper"
        statement="select count(hashtagEntities.where
            (p => p.text = 'prancingdonkey')) as tagged
            from Tweets having count(hashtagEntities.where
            (p => p.text = 'prancingdonkey')) > 0"
        doc:name="Listen For Events"/>
    <sfdc:create-single config-ref="sfconfig"
        type="Case" doc:name="Create SFDC Case">
        <sfdc:object>
            <sfdc:object key="SuppliedName">
                #[payload['account']]</sfdc:object>
            <sfdc:object key="Description">
                #[payload['text']]</sfdc:object>
            <sfdc:object key="Subject">
                #[payload['text']]</sfdc:object>
        </sfdc:object>
    </sfdc:create-single>
</flow>
```

1 Consume tweets

Insert tweets into Esper event stream

Listen for tweet events that contain prancingdonkey hashtag

2 Generate SFDC case based on those tweets

The first flow subscribes to the Twitter event stream ① and injects Twitter status updates into the Tweets event stream. The second flow listens for Tweet events that contain the prancingdonkey hashtag. The content of these status updates is used to create Case objects in Salesforce ②.

Mule's typical role in an enterprise's architecture puts it in a good place to use CEP technology. As Mule applications are typically the mediators between different applications, they're in a unique position to make inferences from interapplication communication that is otherwise opaque. Esper, with its SQL-like query syntax for event streams, makes it a particularly good candidate to mine this information, as is evident from the preceding examples.

14.3 Using a rules engine with Mule

Business logic in Mule flows can be implemented with components, as you saw in chapter 6. Java and its derivatives (Groovy, MVEL, Jython, and so on) are the usual candidates for implementing this logic. Examples of such use cases include using a component to interact with an ORM, DAO, or service layer with the payload of a `MuleMessage`. There are, however, certain situations that are difficult to express in a declarative or object-oriented environment. Validation is a good example. A pharmacy, for instance, may need to perform dozens of evaluations on an order to ensure it can be filled. Such a prescription-filling application would need to validate that none of the medications in a prescription set interact adversely with each other, whether the customer is allergic to a particular medication, if a name-brand or generic medication is needed based on the customer's insurance plan, and so on.

Implementing such a use case is awkward in a language like Java. A typical implementation, at worst, would involve a cascading series of if-then-else blocks and, somewhat better, effective use of polymorphism to model the validation. There are other complications to this approach, beyond the developer headaches. These validation rules can change frequently, which can add a lot of development and operational overhead. Furthermore the “experts” in these validations are almost never the people implementing the code, introducing the very real possibility that things will get lost in translation as requirements trickle down to the development team.

Rule engines attempt to solve this problem by providing a framework for rules to be expressed and evaluated in a manner easy for nondeveloper business experts to understand. Rule definitions are typically stored outside of the project, usually on the filesystem or in a database, so they can be modified outside of the development cycle of an application. And, much like Mule Studio, most rules engines provide graphical tools to author and manage the rule definitions themselves.

Mule provides a generic framework for integrating with arbitrary rules engines and provides out-of-the-box support for Drools, a popular, open source rules engine developed by JBoss. In this section, you'll see how Prancing Donkey uses Drools for selective enrichment of messages, as well as for temporal, content-based routing.

14.3.1 Using Drools for selective message enrichment

As Prancing Donkey grows, its sales department decides to roll out a customer loyalty program. The goal is to reward customers who meet various criteria when they purchase beer from Prancing Donkey's web store. Since this is a new project that sales and marketing will want to tinker with, particularly in the beginning, Prancing Donkey's developers have decided to model the loyalty program using Drools. Their goal is to have Drools process Order messages and add rewards to the Order based on the following:

- The customer's birthday
- Whether the customer lives in a specific state that marketing is targeting
- Whether the customer has ordered more than 25 times or has spent over \$1,000

The following listing shows what this looks like when expressed using the Drools rules language.

Listing 14.6 Customer loyalty rules

```
package com.prancingdonkey

import com.prancingdonkey.model.*
import org.mule.MessageExchangePattern
import java.util.*

global org.mule.module.bpm.MessageService mule;
dialect "mvel"

rule "Purchasing Reward"
when
    $order : Order(customer.totalOrders > 25
    || customer.totalSpendForYear > 1000.00)
then
    $order.rewards.add("PURCHASE");
    mule.generateMessage("vm://loyalty",
        $order, null,
        MessageExchangePattern.ONE WAY);
end

rule "Purchasing Reward And In Free Shipping State"
when
    $order : Order( );
    HashMap(
        eval( ((List)this["states"])
            .contains($order.customer.address.state) ) )
then
    $order.rewards.add("STATE");
    mule.generateMessage("vm://loyalty",
        $order,
        null,
        MessageExchangePattern.ONE WAY);
end
```

- ① Define MessageService global
- ② Prior purchase amount rule
- ③ Add PURCHASE reward to the order
- ④ Resident state rule
- ⑤ Add STATE reward to the order

```

rule "Birthday Reward"
when
    $order : Order();
    eval(Date now = new Date();
        $order.customer.birthday.day == now.day
        and
        $order.customer.birthday.month == now.month)
then
    $order.rewards.add("BIRTHDAY");
    mule.generateMessage("vm://loyalty",
        $order,
        null, MessageExchangePattern.ONE_WAY);
end

```

6 Birthday rule

7 Add BIRTHDAY reward to the order

This file contains a series of conditions and consequences. The conditions at ②, ④, and ⑥ will “fire” the corresponding consequences defined at ③, ⑤, and ⑦. In each case, the consequence is to use the MessageService globally defined at ① to send the message to a new endpoint after adding the appropriate reward to the Order domain object.

DROOLS GUVRNOR At least for some developers, the Drools rule syntax can be pretty daunting. Luckily, JBoss has a complementary project, called Drools Guvnor, that provides a web-based interface for defining rules.

Now let’s look at how to configure this in Mule in the next listing.

Listing 14.7 Configuring Drools in Mule

```

<bpm:drools/>                                ← Define Drools as rules engine

<spring:bean id="states"
class="org.springframework.beans.factory.config.ListFactoryBean">
    <spring:property name="sourceList">
        <spring:list>
            <spring:value>NY</spring:value>
            <spring:value>NJ</spring:value>
            <spring:value>CT</spring:value>
            <spring:value>MA</spring:value>
        </spring:list>
    </spring:property>
</spring:bean>

<spring:beans>
    <util:map id="facts">
        <spring:entry
            key="states"
            value-ref="states"/>
    </util:map>
</spring:beans>

<flow name="customerLoyalty">
    <vm:inbound-endpoint path="order.processing.loyalty"/>
    <bpm:rules rulesDefinition="drools/loyalty.drl"
        initialFacts-ref="facts"/>
</flow>

```

Statically define list of states for use as initial facts

Invoke rules engine to process message

EXTERNAL FACT INITIALIZATION In a real application, the static list of initial facts would most likely be initialized externally from a database or properties file.

USING DROOLS FOR MESSAGE TRANSFORMATION Drools can't currently be used to transform a message, only to generate a new message and route it as a consequence.

14.3.2 Message routing with Drools

The choice router, which we discussed in chapter 5, can get you pretty far in terms of how messages are routed in Mule applications. Many of the same drawbacks we discussed in the beginning of this section, however, still apply. Neither a massive choice router block nor an overly verbose Java router are attractive options when the routing logic is complex or needs to be changed often.

LACK OF STUDIO SUPPORT FOR DROOLS There currently isn't support for the Drools module in Mule Studio. You'll need to rely on the XML configuration when using Drools in your Mule applications.

Let's see how Prancing Donkey leverages Drools to dynamically decide how monitoring alerts are routed. The next listing shows a domain object that models an Alert. This domain object will be passed as a JMS message payload that Mule will receive and route based on a Drools rules evaluation.

Listing 14.8 Java class for an Alert

```
public class Alert implements Serializable {
    private String application;
    private String severity;
    private String description;
    ...getters / setters omitted
}
```

The next listing shows a rules file that defines how alerts are routed.

Listing 14.9 The Alert routing DRL file

```
package com.prancingdonkey.domain;

import org.mule.MessageExchangePattern;
import java.util.Map;
import java.util.HashMap;

global org.mule.module.bpm.MessageService mule;           ← Define MessageService
rule "weekdays are high priority"
    calendars "weekday"
when
    $alert : Alert()
then
    Set a higher severity
    during week
```



```

    $a.setSeverity("HIGH")
end

rule "weekends are low priority"
    calendars "weekend"
when
    $alert : Alert()
then
    $a.setSeverity("LOW")
end

rule "Salesforce Alert"
when
    $alert : Alert( application == "salesforce" )
then
    mule.generateMessage("vm://alerts.salesforce",
        $alert, null, MessageExchangePattern.ONE WAY);
end

rule "Twilio Alert"
when
    $alert : Alert( application == "twilio" )
then
    mule.generateMessage("vm://alerts.twilio",
        $alert, null, MessageExchangePattern.ONE WAY);
end

rule "Mule Management Console Alert"
when
    $alert : Alert( application == "mmc", severity == "HIGH" )
then
    mule.generateMessage("vm://alerts.mmc", $alert, null,
        MessageExchangePattern.ONE WAY);
end

```

The diagram consists of five vertical red lines, each ending in a circular callout with a number and a descriptive text. Callout 1 points to the first rule. Callout 2 points to the second rule. Callout 3 points to the fourth rule. Callout 4 points to the fifth rule. Callout 5 points to the sixth rule.

The rules at ① and ② override the severity of the alert based on the day of the week. The rules at ③, ④, and ⑤ will route the alert to the appropriate VM queue. Now let's see how to wire this up into a Mule flow in the next listing.

Listing 14.10 Configuring Drools in Mule

```

<bpm:drools/>                                     ← Use Drools as BPM engine

<spring:beans>
    <spring:bean name="NoFactsBean"
        class="java.util.ArrayList"
        doc:name="Bean"/>
</spring:beans>

<flow name="routeAlerts">
    <jms:inbound-endpoint queue="alerts"/>
    <bpm:rules rulesDefinition="alerts.drl"
        initialFacts-ref="NoFactsBean"/>
</flow>

```

The diagram consists of two horizontal red lines, each ending in a circular callout with a number and a descriptive text. Callout 1 points to the 'NoFactsBean' bean definition. Callout 2 points to the 'rules' element within the 'flow' block.

For this example, you don't have an initial list of facts, so you define an empty list of initial facts ①. The rules engine is then invoked ② whenever a message is received in the alerts queue.

`generateMessage` is called on the `MuleService` when the appropriate rule is matched, routing the alert to the appropriate endpoint.

Drools provides a great way to decouple selective business logic that is either quick to change or requires an expert to define from the develop and deploy cycle of Mule applications. Now let's take a look at how Mule's support for polling and scheduling make it possible to develop batching applications.

14.4 Polling and scheduling

Many use cases in integration scenarios have a temporal component. Some resources, like a database table or filesystem, need to be polled at an interval to check if new data has been inserted. Other times you want to schedule a task to run at a certain time, such as to trigger a flow to process insurance claim data at the end of the month.

In this section, we'll look at Mule's polling and scheduling facilities. We'll start off by seeing how the poll message process facilitates the polling of arbitrary endpoints and cloud connectors. We'll then take a look at explicitly scheduling tasks with the Quartz transport.

14.4.1 Using the poll message processor

The majority of Mule's transports support polling where it makes sense. The file, HTTP, and FTP transports, for instance, provide polling out of the box by setting poll parameters on their endpoints. Some transports and cloud connectors don't provide such facilities. In those cases the poll message processor can be used to repeatedly invoke an arbitrary message processor at some interval.

The Twitter module is an example of a module that doesn't have built-in polling support. Let's take a look at how you can use the poll processor to repeatedly query the public timeline every five minutes. Prancing Donkey is using it in just this manner to insert tweets into an Esper stream. This allows the marketing group to get real-time alerts when certain hashtags (like Prancing Donkey's) appear on the stream.

Listing 14.11 The poll message processor repeatedly fetches Twitter's public timeline

```
<flow name="main">
    <poll frequency="300000">
        <twitter:get-public-timeline/>
    </poll>
    <collection-splitter/>
    <esper:send eventPayload-ref="#{message.payload}" />
</flow>
```

Invoke `get-public-timeline` processor
every five minutes

Now let's take a look at how the Quartz transport provides finer-grained scheduling.

14.4.2 Scheduling with the Quartz transport

The polling message processor, which we just examined, is handy to repeatedly execute a message processor at some interval. Often, though, you need to execute a job at some predetermined time. Mule's Quartz transport provides such a facility through the use of cron expressions. Cron expressions, whose format is detailed in figure 14.1, should be familiar to anyone who's ever administered a Unix system. It provides millisecond granularity to schedule jobs.

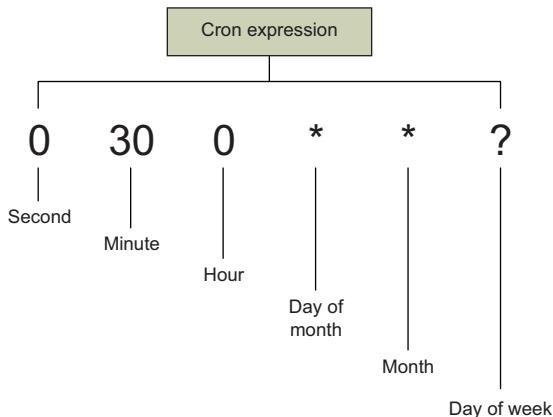


Figure 14.1 Cron expression format

Now let's take a look at how Prancing Donkey is using the Quartz transport. On the first day of the month, Prancing Donkey needs to generate reports for various stakeholders in the company. It also needs to perform some routing, warehousing, and invoicing tasks. These services are typically components hosted in Mule that are triggered by a message being dispatched to a JMS topic. The following listing demonstrates how the Quartz transport can be used to generate this message.

Listing 14.12 Using Quartz's cron expression to fire an event once per month

```

<flow name="firstOfMonthEventGenerator">
    <quartz:inbound-endpoint
        jobName="firstOfMonth"
        cronExpression="0 0 0 1 * ?">
        <quartz:event-generator-job>
            <quartz:payload>FIRST_OF_THE_MONTH_EVENT</quartz:payload>
        </quartz:event-generator-job>
    </quartz:inbound-endpoint>
    <jms:outbound-endpoint topic="events.trigger">
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:outbound-endpoint>
</flow>
  
```

- ① Schedule dispatch of message on a JMS topic on first day of every month
- ② Use event-generator-job to set payload of generated message

The cron expression defined at ① will trigger the Quartz job called `firstOfMonth` on the first day of every month. The job name is what Quartz uses internally to distinguish jobs from each other. The event generator at ② will generate a message with a

payload of FIRST_OF_THE_MONTH_EVENT. You can set the file attribute of this element to set the payload from a file on the filesystem or use a Spring bean reference as well (MEL expressions are currently unsupported).

Quartz and job store persistence

You might be wondering what happens if Mule crashes while jobs are queued. Since the Quartz connector uses an in-memory job store, by default the jobs will be lost. Thankfully, you can use Quartz's support for persistent job stores to overcome this limitation. One way to accomplish this is to create a quartz.properties file in Mule's CLASSPATH and set the org.quartz.jobStore.class with the scheduling factory appropriate for your needs. Full documentation is available on the Quartz website, but you will most likely be interested in the JDBC transactional JobStore, which will allow you to store your jobs in a database.

Quartz jobs are automatically made persistent and cluster aware if you're using Mule Enterprise Edition with HA clustering.

14.5 Summary

Tightly integrated technologies orthogonal to Mule, like CEP, BPM, rules evaluation, and scheduling, simplify the development and operation of Mule applications. Mule is internally architected to support these generically and provides default implementations, like jBPM and Drools to lower the burden of using them with your applications. In this chapter, you saw how Prancing Donkey was able to quickly use each of these technologies in their overall architecture as their business needs evolved, building on the work they've already done. These techniques, while possibly not immediately applicable to your projects, are good to know about when the problems they solve do arise, so that you're not stuck reinventing the wheel.

appendix A

Mule Expression Language

The Mule Expression Language (MEL) is based on the MVFLEX Expression Language (MVEL; see <http://mvel.codehaus.org/>), a hybrid dynamic/statically typed language. We strongly recommend you get acquainted with MVEL by reading its online documentation (<http://mvel.codehaus.org/Language+Guide+for+2.0>). MEL itself is also extensively documented online (<http://mng.bz/g8sM>). This appendix is a quick reference guide to Mule-specific features of MEL. It also provides a quick overview on how to customize MEL to implement custom needs.

A.1 **MEL quick reference**

Mule binds custom objects, variables, and functions within the MEL context to facilitate accessing Mule resources, processing messages, and so on.

TOP-LEVEL VARIABLES QUICK REFERENCE The four main context objects exposed by MEL are `server`, `mule`, `app`, and `message`. Additionally, `payload` is available as a shortcut to `message.payload`, `flowVars` and `sessionVars` are maps that give access to flow and session variables, and `exception` is bound if the current event carries an exception.

Before looking in detail at these context helpers, let's take a quick peek at MVEL. Take a look at this sample script from MVEL's documentation (<http://mvel.codehaus.org/Sample+Scripts>):

```
colors = ['red', 'green', 'blue'];
foreach (c : colors) {
    System.out.println(c + "!");
}
```

What should strike you is the following:

- MVEL has nice syntactic sugar for data structures. Lists and maps can be created as easily as arrays, for example, with `['key': 'value']` for a map.
- Any Java class can be reached. Classes not in the auto-imported list (see section A.1.4) must be referred to via their fully qualified names.

Of all the goodness MVEL provides, its unified support for property navigation (including null-safe navigation) is one of the most convenient features. For example, in the following snippet,

```
user.name
```

name can refer to either a getName accessor on a user JavaBean or the name key in a user hashmap.

Without further ado, let's go through the context objects Mule gives you access to.

A.1.1 Context objects

There are four context objects available: server, mule, app, and message. We'll review them in detail.

SERVER

The server context gives access to the properties of the hardware, operating system, virtual machine, user and network interface, and time-related functions. See table A.1.

Table A.1 Properties of the server context object

Name	Description
dateTime	Current system time via the <code>org.mule.el.datetime.DateTime</code> utility object, whose complete documentation is available online (http://mng.bz/LT3q). This object supports date and time zone manipulation methods as well as comparison and formatting ones.
fileSeparator	Character that separates components of a file path (/ on Unix and \ on Windows).
host	Fully qualified domain name of the server.
ip	The IP address of the server.
locale	Default locale (of type <code>java.util.Locale</code>) of the JRE (can access <code>server.locale.language</code> and <code>server.locale.country</code>).
javaVersion	JRE version.
javaVendor	JRE vendor name.
nanoTime	Current system time in nanoseconds.
osName	Operating system name.
osArch	Operating system architecture.
osVersion	Operating system version.
systemProperties	Map of Java system properties.
timeZone	Default <code>TimeZone</code> (<code>java.util.TimeZone</code>) of the JRE.
tmpDir	Temporary directory for use by the JRE.
userName	Username.
userHome	User home directory.
userDir	User working directory.

MULE

The `mule` context allows you to retrieve the properties of the Mule instance, which is either the standalone broker instance or the instance that's embedded in an application (for example, in a web application). See table A.2.

Table A.2 Properties of the `mule` context object

Name	Description
<code>clusterId</code>	Cluster ID
<code>home</code>	Filesystem path to the home directory of the Mule server installation
<code>nodeId</code>	Cluster node ID
<code>version</code>	Mule version

APP

The `app` context exposes the properties of the Mule application that the current expression is evaluated into. See table A.3.

Table A.3 Properties of the `app` context object

Name	Description
<code>encoding</code>	Application default encoding (read-only)
<code>name</code>	Application name (read-only)
<code>standalone</code>	True if Mule is running standalone (read-only)
<code>workdir</code>	Application work directory (read-only)
<code>registry</code>	Map representing the Mule registry (read/write)

MESSAGE

The message context gives access to the payload, attachments, and properties of the Mule message that's under processing. This context isn't available if the expression is evaluated outside the context of a Mule event. See table A.4.

Table A.4 Properties of the `message` context object

Name	Permissions
<code>id</code>	Read-only
<code>rootId</code>	Read-only
<code>correlationId</code>	Read-only
<code>correlationSequence</code>	Read-only

Table A.4 Properties of the message context object (continued)

Name	Permissions
correlationGroupSize	Read-only
replyTo	Read/write
dataType	Read-only
payload	Read/write
inboundProperties	Map (read-only)
inboundAttachments	Map (read-only)
outboundProperties	Map (read/write)
outboundAttachments	Map (read/write)

A.1.2 Context variables

Mule binds extra context entries that are similar to the objects we've just described but specific to the in-flight message flow and session variables (that is, respectively, *invocation*- and *session*-scoped message properties; refer to section 2.3.2 for more on this).

These entries are as follows:

- `flowVars`—A read/write Map of flow variables.
- `sessionVars`—A read/write Map of session variables.
- Any flow variable whose name doesn't conflict with other context entries and is a valid MVEL variable name. For example, the `currentUser` flow variable would be available both in the `flowVars` map, with either `flowVars['currentUser']` or `flowVars.currentUser`, and as a top-level context variable named `currentUser`. Conversely, a flow variable named `current-user` would only be available as `flowVars['current-user']` because it can't be used directly as a valid MVEL variable name.
- `exception`—A `java.lang.Exception` optionally bound if the current message carries an exception payload.

Turning autobinding off

If, for any reason, you don't want MEL to bind all flow variables as top-level variables in the evaluation context, add the following configuration element to your Mule configuration:

```
<configuration>
    <expression-language autoResolveVariables="false" />
</configuration>
```

A.1.3 Context functions

Mule binds helper functions in the MEL context. These functions simplify using XPath and regular expressions.

XPATH

- `xpath(xpathExpression)`—Applies the XPath expression to the in-flight message payload (which must be an XML document or a DOM instance).
- `xpath(xpathExpression, xmlElement)`—Applies the XPath expression to the XML element specified by the MEL expression appearing as the second argument. `xmlElement` can be an MVEL variable or another expression (for example, to retrieve XML out of a message property or attachment).

Be aware that the `xpath` function returns DOM nodes (elements and attributes, but also node lists). If you want to get the node string content instead, you'll need to specifically get it; for example, like this for an attribute:

```
xpath('/book/@id').value
```

Or like this for a text element:

```
path('/book/name').text
```

REGEX

- `regex(regularExpression)`—Applies the regular expression to the in-flight message payload
- `regex(regularExpression, melExpression)`—Applies the regular expression to the specified MEL expression
- `regex(regularExpression, melExpression, matchFlags)`—Applies the regular expression to the specified MEL expression using the flags defined in Java's Pattern documentation (<http://mng.bz/gyte>)

The `regex` function returns `null` if no match has been found and an array of results if more than one match was found. Otherwise, it returns the single matching value.

A.1.4 Imported classes

MEL automatically imports the following Java classes, so you can use them without using their fully qualified names:

- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.lang.Enum`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Math`

- `java.lang.Number`
- `java.lang.Object`
- `java.lang.Short`
- `java.lang.String`
- `java.lang.System`
- `java.util.Calendar`
- `java.util.Collection`
- `java.util.Date`
- `java.util.Map`
- `java.util.List`
- `java.util.Set`
- `java.util.regex.Pattern`
- `javax.activation.DataHandler`
- `javax.activation.MimeType`
- `org.mule.api.transformer.DataType`
- `org.mule.transformer.types.DataTypeFactory`

A.2 Customizing MEL

MEL supports multiple configuration-based customizations. For example, it is possible to do the following:

- Import additional classes in the context
- Alias these imported classes
- Alias complete MEL expressions under new names
- Define new globally bound functions

Take a look at the following Mule configuration fragment:

```
<configuration>
  <expression-language autoResolveVariables="false">
    <import class="org.mule.util.StringUtils" />
    <import name="rsu"
      class="org.apache.commons.lang.RandomStringUtils" />
    <alias name="appName" expression="app.name" />
    <global-functions>
      def reversePayload() { StringUtils.reverse(payload) }
      def randomString(size) { rsu.randomAlphanumeric(size) }
    </global-functions>
  </expression-language>
</configuration>
```

The noticeable aspects of this configuration are

- `org.mule.util.StringUtils` gets globally imported and used in a global custom function named `reversePayload` (of arity 0).

- `org.apache.commons.lang.RandomStringUtils` gets globally imported and aliased as `rsu`, and then used in a global custom function named `randomString` (of arity 1).
- The expression `app.name` gets globally aliased to `appName`.

Optionally, you can also specify a `file` attribute on the `global-functions` configuration element should you want to store your custom MEL functions in an external file. With this configuration in place, it's possible to use the declared global functions and variables, as shown in the following example:

```
<set-property propertyName="random-app-property"
              value="#[appName + ':' + randomString(20)]" />
<set-payload value="#[reversePayload()]" />
```

OR CODE IT... It's also possible to programmatically extend MEL by coding custom language extensions. This is beyond the scope of this appendix, but if you want to get started with that approach, take a look at `org.mule.module.xml.el.XMLExpressionLanguageExtension` from the `mule-module-xml` to have an idea of how to achieve this. Also take a look at `META-INF/services/org/mule/config/registry-bootstrap.properties` to see how the `XMLExpressionLanguageExtension` is globally registered so that it can be used throughout Mule.

appendix B

Component and transformer annotations quick reference

As introduced in section 6.1.4, Mule offers Java code-level annotations that can be used on components, transformers, and other classes loaded by Mule. These annotations cover a whole range of functionality. Table B.1 provides you with a quick reference for these annotations.

Most annotations are provided in the mule-module-annotations module, but some come from other modules. Be sure to have the necessary modules on your classpath if you're not deploying on Mule standalone.

Table B.1 Annotations reference

Annotation	Description	Type	Module
@ContainsTransformerMethods	Signals that a class contains transformer methods.	Method	annotations
@Transformer	Used to create a Mule transformer from a class method.	Method	annotations
@Schedule	Schedules a method for execution using either a simple frequency or cron expression.	Method	quartz
@IntegrationBean	A dependency injector annotation used to inject an iBean such as Facebook or Amazon S3 into your component.	Field	ibbeans
@MockIntegrationBean	Similar to the @IntegrationBean annotation, this one can be used in tests to create a mock instance of an iBean that's good for testing the bean without actually making requests to the external service.	Field	ibbeans
@Lookup	Dependency injection annotation used to retrieve objects from the registry.	Field, parameter	annotations

Table B.1 Annotations reference (continued)

Annotation	Description	Type	Module
@Payload	A parameter injection annotation that can be used on component entry points and transformer methods defined using the @Transformer annotation, this annotation controls how the current message payload is passed into a method by performing automatic transformation of the message payload to match the annotated parameter type.	Parameter	annotations
@InboundHeaders	Used on component entry points and transformer methods, this annotation controls how the current message inbound headers are passed into a method. The annotation supports Map, List, single headers, and wild-cards.	Parameter	annotations
@OutboundHeaders	Used on component entry points and transformer methods, this annotation controls how the current message outbound headers are passed into a method. Users can write to this Map to attach headers to the outgoing message.	Parameter	annotations
@InboundAttachments	Used on component entry points and transformer methods, this annotation controls how the current message inbound attachments are passed into a method. The annotation supports Map, List, single headers, and wild-cards.	Parameter	annotations
@OutboundAttachments	Used on component entry points and transformer methods, this annotation controls how the current message outbound attachments are passed into a method. Users can write to this Map to attach attachments to the outgoing message.	Parameter	annotations
@Xpath	This annotation can be used to execute an XPath expression on the message payload with the result being passed into the method.	Parameter	xml
@Groovy	This annotation can be used to execute a Groovy expression on the message payload with the result being passed into the method.	Parameter	scripting

Table B.1 Annotations reference (continued)

Annotation	Description	Type	Module
@Mule	A parameter injection annotation that can be used on component entry points and transformer methods, this annotation can be used to execute a Mule expression.	Parameter	annotations
@Function	A parameter injection annotation expression on the message payload with the result being passed into the method, this annotation exposes a common set of functions used in Mule such as a counter, UUID generator, date and timestamps, etc.	Parameter	annotations

appendix C

Mule Enterprise Edition

This appendix describes the differences between Mule Community Edition, which this book is based on, and Mule EE, the Enterprise Edition of Mule provided by MuleSoft Inc.

HIGH-AVAILABILITY CLUSTERING

Mule Enterprise Edition provides an HA solution that allows multiple Mule servers to form a distributed memory cluster. This enables stateful message processors, such as the idempotent message receiver, to transparently distribute their state in a cluster. It also allows some transports, such as the file transport, to parallelize work across clustered nodes. Message failover between nodes is also supported, provided that applications are structured with the reliability patterns discussed in chapter 7.

MULE MANAGEMENT CONSOLE

The Mule Management Console (MMC) is a web application deployed on premises to manage multiple Mule standalone and HA servers. In addition to monitoring and managing Mule servers, it handles the entire lifecycle of Mule applications, allowing the registration, deployment, starting, debugging, stopping, and monitoring of them at runtime. Business events, another feature of MMC, track messages and KPIs across Mule applications and flows.

ADDITIONAL AND ENHANCED TRANSPORTS

Mule EE provides additional and enhanced transports that aren't available in Mule CE. The WebSphere MQ connector, for instance, provides WSMQ-specific functionality not present in the JMS transport. The SAP transport, also not available in CE, simplifies integration with SAP systems. The JDBC transport is enhanced in Mule EE, offering support for large-dataset retrieval, batching, and outbound-stored procedures.

DATA MAPPER

DataMapper provides a graphical facility to define message and payload transformation; it currently supports XML, JSON, CSV, POJO, Excel, and fixed-width files. It provides a simpler alternative to implementing either programmatic Mule transformers or complex XSLT transformations.

ANYPOINT ENTERPRISE SECURITY

Anypoint Enterprise Security offers additional security features for Mule EE. These include the following:

- Mule Secure Token Service (STS) OAuth 2.0a Provider
- Mule Credentials Vault (encrypted values of properties in property files)
- Mule Message Encryption Processor
- Mule Digital Signature Processor
- Mule Filter Processor
- Mule CRC32 Processor
- IP White Listing

ANYPOINT SERVICE REGISTRY

Mule EE applications can take advantage of Anypoint Service Registry, MuleSoft’s SOA governance platform. Anypoint Service Registry lets users apply policies, such as throttling and security, to Mule flows at runtime.

HARDENED CODE LINE

Mule EE customers get access to incremental “dot” releases that contain bug and performance fixes that aren’t made available to CE users. These fixes are rolled into major releases (for example, 3.4.0) that are made available to CE and EE users on a less frequent basis.

SUPPORT AND SERVICES

Follow-the-sun support is available 24/7 to subscribers of Mule EE. EE subscribers also have access to MuleSoft’s services team, which can be used for architecture review, training, and tactical consulting engagements.

appendix D

A sample Mule application

This appendix presents an example of a Mule application. It demonstrates a legacy modernization use case faced by Prancing Donkey, and integration with a partner. Prancing Donkey has recently partnered with Balin's Stuff, Inc., a provider of accessories to facilitate and enhance the consumption of beer.

In order to sell Balin's products, Prancing Donkey must periodically pull an XML file from an FTP site. The XML file contains the inventory of Balin's products and their associated prices, shipping information, and so on.

To facilitate this, Prancing Donkey has developed a Mule application that does the following:

- Polls an FTP site periodically for product information
- Splits and transforms each product XML element to an instance of Product, part of a JPA domain model
- Persists each Product object to a relational database using the JPA connector
- Exposes the Products with a RESTful API using JAX-RS

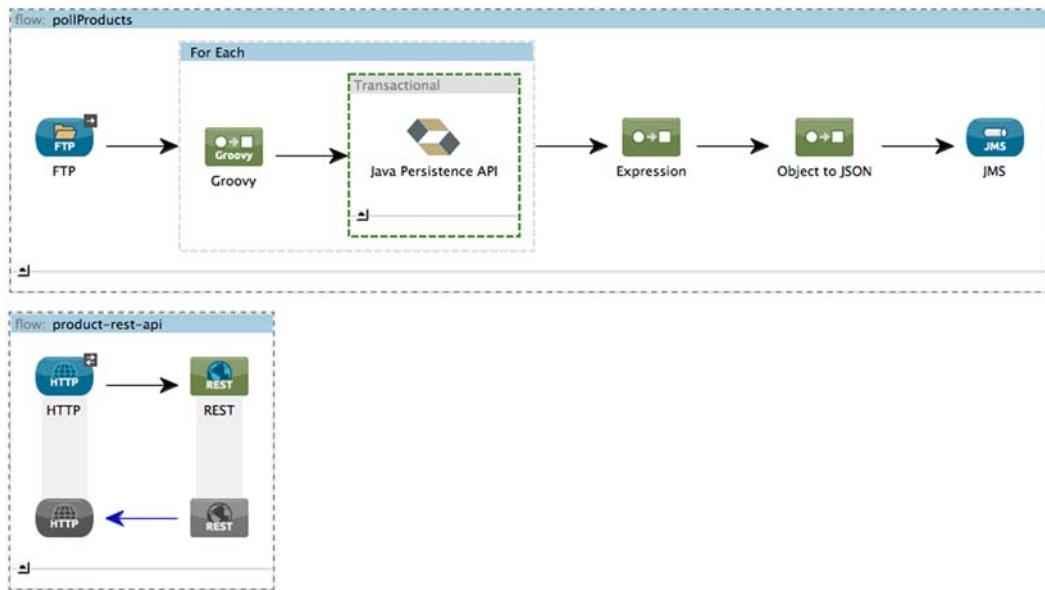


Figure D.1 The Mule application in Mule Studio

We'll now show the Mule configuration for this application as well as the supporting FunctionalTestCase implementation. The application is available in its entirety via GitHub at <https://github.com/ddosso/mule-in-action-2e>.

D.1 The Mule application configuration

The Mule Studio screenshot of the Mule application is shown in figure D.1, and the annotated listing of the Mule application is shown in listing D.1.

Listing D.1 The Mule application configuration

```
<mule xmlns:ftp="http://www.mulesoft.org/schema/mule/ftp"
      xmlns:jersey="http://www.mulesoft.org/schema/mule/jersey"
      xmlns:http="http://www.mulesoft.org/schema/mule/http"
      xmlns:scripting="http://www.mulesoft.org/schema/mule/scripting"
      xmlns:jms="http://www.mulesoft.org/schema/mule/jms"
      xmlns:json="http://www.mulesoft.org/schema/mule/json"
      xmlns:jpa="http://www.mulesoft.org/schema/mule/jpa"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:doc="http://www.mulesoft.org/schema/mule/documentation"
      xmlns:spring="http://www.springframework.org/schema/beans"
      version="CE-3.4.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-current.xsd
http://www.mulesoft.org/schema/mule/core
http://www.mulesoft.org/schema/mule/core/current/mule.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
http://www.mulesoft.org/schema/mule/ftp
http://www.mulesoft.org/schema/mule/ftp/current/mule-ftp.xsd
http://www.mulesoft.org/schema/mule/scripting
http://www.mulesoft.org/schema/mule/scripting/current/mule-scripting.xsd
http://www.mulesoft.org/schema/mule/jms
http://www.mulesoft.org/schema/mule/jms/current/mule-jms.xsd
http://www.mulesoft.org/schema/mule/json
http://www.mulesoft.org/schema/mule/json/current/mule-json.xsd
http://www.mulesoft.org/schema/mule/jpa
http://www.mulesoft.org/schema/mule/jpa/1.0/mule-jpa.xsd
http://www.mulesoft.org/schema/mule/http
http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
http://www.mulesoft.org/schema/mule/jersey
http://www.mulesoft.org/schema/mule/jersey/current/mule-jersey.xsd
http://www.mulesoft.org/schema/mule/ftp
http://www.mulesoft.org/schema/mule/ftp/current/mule-ftp.xsd">
```

Various
namespace
declarations
for modules
and
transports

Context property-
placeholder loads
configuration
at runtime
from mule-
app.properties at
root of classpath

Imports an external Spring
configuration file that
contains database
configuration for JPA module

```
<spring:beans>
    <spring:import
        resource="spring/applicationContext.xml"/>
</spring:beans>
<context:property-placeholder location="/mule-app.properties"/>
```

```

<jpa:config name="Java_Persistence_API"
    entityManagerFactory-ref="entityManagerFactory"
    doc:name="Java Persistence API"/>

```

Configuration for Java Persistence API module


```

<jms:activemq-connector name="jmsConnector"
    specification="1.1"
    brokerURL="${jms.url}"/>

```

Configuration of an ActiveMQ JMS broker


```

<flow name="pollProducts">
    <ftp:inbound-endpoint user="${ftp.user}"
        password="${ftp.password}"
        host="${ftp.host}"
        port="${ftp.port}"
        path="${ftp.path}"
        pollingFrequency="3600000"
        />
    <foreach collection="xpath('//product')">
        <scripting:transformer>
            <scripting:script engine="Groovy"
                file="scripts/productTransformer.groovy"/>
        </scripting:transformer>
        <transactional action="ALWAYS_BEGIN">
            <jpa:upsert config-ref="Java_Persistence_API"
                entity-ref="#{payload}"
                id-ref="#{payload.id}"
                fields="name,description,price,shippingCost"
                flush="true"
                />
        </transactional>
    </foreach>
    <expression-transformer
        expression="flowVars['products']"/>
    <json:object-to-json-transformer/>
    <jms:outbound-endpoint topic="events.product"
        connector-ref="jmsConnector"/>
</flow>

```

Poll FTP server once an hour for new files

Use xpath to split payload by product

Use JPA module's upsert processor to insert or update Product to database

Transform payload to JSON

Use Groovy script to transform XML to Product domain model instance

Transform payload to contents of products flow variable, which is being populated inside scripts/productTransformer.groovy

Dispatch JSON to JMS topic indicating that ingest is completed


```

<flow name="product-rest-api">
    <http:inbound-endpoint exchange-pattern="request-response"
        host="${http.host}"
        port="${http.port}"
        />
    <jersey:resources>
        <component
            class=
                "com.prancingdonkey.product.service.ProductService"/>
    </jersey:resources>
</flow>

```

Expose JAX-RS annotated ProductService using Jersey

Define HTTP endpoint to expose RESTful web service

D.2 ProductImportFunctionalTestCase

The following is the FunctionalTestCase implementation for the application.

Listing D.2 The ProductImportFunctionalTestCase implementation

```

public class ProductImportFunctionalTestCase extends FunctionalTestCase {
    private FakeFtpServer fakeFtpServer; // ← Mock FTP server
    JdbcTemplate jdbcTemplate;
    @BeforeClass
    public static void setupDirectories() throws Exception {
        File dataDirectory = new File("./data");
        if (dataDirectory.exists()) {
            FileUtils.deleteDirectory(dataDirectory);
        }
        dataDirectory.mkdirs();
        new File("./products").mkdirs();
        new File("./data/in").mkdirs();
    }
    @Override
    protected void doSetUp() throws Exception {
        super.doSetUp();
        startServer();
        jdbcTemplate = new JdbcTemplate(
            muleContext.getRegistry()
                .lookupObject(DataSource.class));
    }
    @Override
    protected String getConfigResources() {
        return "prancing-donkey-product-service.xml";
    }
    @Test
    public void testCanLoadAndQueryForProducts() throws Exception {
        MuleMessage completedEvent = muleContext.getClient()
            .request("jms://topic:events.product",
                     15000);
        assertNotNull(completedEvent); // Assert that completed event has been sent to JMS topic
        assertEquals(2,
                    jdbcTemplate.queryForInt(
                        "select count(*) from product")); // Invoke REST API and assert expected JSON result is received
        Map<String, Object> properties =
            new HashMap<String, Object>();
        properties.put("http.method", "GET");
        MuleMessage products =
            muleContext.getClient()
                .send("http://localhost:8081/products", "", properties); // Assert that appropriate number of rows exists in database
        assertNotNull(products);
        String expected = FileUtils.readFileToString(
            new File("src/test/files/expectedProducts.json"));
        assertEquals(expected, products.getPayloadAsString());
    }
}

```

```
void startServer() throws IOException {
    fakeFtpServer = new FakeFtpServer();
    fakeFtpServer.setServerControlPort(9879);
    fakeFtpServer.addUserAccount(
        new UserAccount("foo", "foo", "/"));

    FileSystem fileSystem = new UnixFakeFileSystem();
    fileSystem.add(new FileEntry("/products/file1.xml",
        FileUtils.readFileToString(
            new File(
                "src/test/files/suppliers/supplier1/products.xml"))));
    fakeFtpServer.setFileSystem(fileSystem);
    fakeFtpServer.start();
}
```


index

A

AbstractEntryPointResolver 144
AbstractPolicyTemplate 128, 223
acceptMuleMessage 104
ACID (atomicity, consistency, isolation, and durability) 232
acknowledgementMode property 67
ActiveMQ 316
Activiti defined 359
Mule module for 361
addresses, endpoints and 53
Advanced Message Queuing Protocol. *See* AMQP
aggregating messages 133–134
all router 73, 129–130
AMQP (Advanced Message Queuing Protocol)
asynchronous messaging support 182
extensions and 7
annotations components 148–149, 380
for custom cloud connectors 337–339
transformers 380
using for all parameters 149
Anypoint Enterprise Security 384
Anypoint Service Registry 384
AOP (aspect-oriented programming). *See* interceptor

Apache ActiveMQ 67
defined 15
downloading 16
external instance 68
sample application 387
Apache Commons Logging 200
Apache CXF connecting to SOAP web services 64–66
resources online 64
Apache Directory Server 252
Apache JMeter adjusting ramp-up period 324
custom samplers 327
HTTP and 323
JDBC and 323
JMS and 323
LDAP and 323
load testing 326
setting up JMS test 325
specifying test plan 324
stopping test 324
stopping thread 324
API Explosion 4
app context object 375
applications designing for high availability 213
execution context 46
apt command 336
array-entry-point-resolver 143
aspect-oriented programming. *See* interceptor
<async> blocks 34
async processor 130–131

asynchronous messaging 182–185
atomicity, consistency, isolation, and durability. *See* ACID
attachments, message defined 41
overview 45–46
autoDelete property 56, 58
automated integration testing 323
autoResolveVariables attribute 376
auto-transformer 99–100

B

backupEnabled property 72–73
backupFolder property 72–73
basic authentication 179, 254
behavior stubbing 319–323
binary property 75
binding of flow variables 376
BitTorrent 75
Boolean values operations using 124
returning from MEL expressions 48
bottlenecks best practices 281–284
profiler-based investigation 279–281
BPM (Business Process Management) 359–361
breakpoints 331–333
BreweryDB 350, 352

bridge, configuration
patterns 162–163
Business Process Management.
See BPM
byte transformers 92–94
byte-array-to-hex-string-
transformer 94
byte-array-to-object-transformer
92
byte-array-to-serializable-
transformer 94
byte-array-to-string-
transformer 94
BytesMessage 69

C

caching interceptor 304
Callable interface 141, 148, 289
callable-entry-point-
resolver 144
canonical data model 180–181
CAS (Central Authentication
Service) 250–251
catch-all routing 127
@Category annotation 349
ccAddress property 74
Central Authentication Service.
See CAS
CEP (complex event process-
ing)
defined 361
monitoring event-driven
systems 361–363
sentiment analysis using
Twitter 363–365
channel adaptors 178
checkFrequency property
72–73
choice router 30, 48, 115–118
classes imported by MEL
377–378
cloud connectors
custom
annotations for 337–339
configurable 339
DevKit and 335–337
integrating extension with
Mule Studio 349–350
intercepting message
processors 344–345
managing
connections 340–342
message processors
343–344

message sources 348–349
REST connector
example 350–355
REST consumers 345–347
transformers 347–348
defined 51
CloudHub 206–208
clusterId property 375
clustering
file transport and 58
idempotent-message-filter
126
internal state 214–216
Mule EE 315
comma-separated value. *See CSV*
complex event processing.
See CEP
Component interface 142
components
annotations for 148–149, 380
configuring 146–148
creation cost 149
custom 142
defined 36
executing business
logic 141–142
expression components 154
implementing Callable 141
injecting Mule objects 151
instantiation policy 142
lifecycle 155–157
Mule objects 146
overview 140–141
pooling
exhaustion of 150
overview 149–151
resolving entry point 143–146
role 140
scripting components
151–154
Spring beans 141, 147, 210
stateless 147
thread-safety 147, 149
components value 349
composite message source 36
compression transformers
compressing 94
overview 94–95
uncompressing 95
conditional breakpoints 332
<config> element 339
configElementName
attribute 338
@Configurable annotation 339,
352
@Configuration annotation
339
configuration patterns
bridge in 162–163
overview 157–158
reusing 170–172
simple service pattern
configuring with JAX-RS
160
configuring with JAX-WS
161
overview 158–160
using HTTP proxy 167–168
using WS proxy 168–170
validator in 164–167
configuration, Spring XML 209
@Connect annotation 338,
340
connecting to Mule
local Mule application
294–295
overview 293–294
remote Mule
application 295–298
using transports 298–300
@ConnectionIdentifier
annotation 338
@ConnectionKey
annotation 341
<connection-pooling-profile>
element 342
@Connector annotation 337
connectors
configuring
with Mule Studio 53–56
with XML 52–53
custom cloud connectors
340–342
databases
insertions 79–80
NoSQL with MongoDB
80–82
overview 77–78
queries 78–79
email
receiving with IMAP
transport 71–73
sending with SMTP
transport 73–75
failed 219
file transport 56–58
FTP transport
overview 75–76
receiving files 76–77
sending files 77

connectors (*continued*)
 HTTP transport
 REST services 62–64
 sending and receiving data 58–61
 SOAP services 64–66
 using web services 61–62
identifying in URIs 69
JMS transport
 overview 67–68
 receiving messages 69
 sending messages 68–69
 synchronous communication 70–71
 using selector filters 70
overview 51
Twitter connector 85–86
VM transport
 overview 82
 reliable flows using 83–85
Consumer Keys (Twitter) 85
@ContainsTransformer-
 Methods annotation 380
context functions 377
context objects 374–375
context variables 376
context, Mule
 accessing 289–290
 overview 288–289
 using 290–293
copy-properties element 43
core transformers
 auto-transformer 99–100
 byte transformers 92–94
 compression transformers 94–95
 flow variable
 transformers 95–97
 message enricher 98–99
 property transformers 95–97
 session variable transformers 95–97
 using expressions 97–98
correlationGroupSize
 property 376
correlationId property 375
correlationSequence
 property 375
cron expressions 371
cross-cutting concerns 178
Cryptix 261
CSV (comma-separated value) 110
curl command 19, 63
custom adapters 157

custom cloud connectors
 annotations for 337–339
 configurable 339
 DevKit and 335–337
 integrating extension with Mule Studio 349–350
intercepting message processors 344–345
managing connections 340–342
message processors 343–344
message sources 348–349
REST connector example 350–355
REST consumers 345–347
transformers 347–348
custom samplers 327
custom transformers 109–112
custom-entry-point-resolver 144
custom-interceptor 310
customizing MEL 378–379
cxf:proxy-service 186

D

data compression. *See* compression transformers
data persistence 310–315
data structures in MVEL 373
data transformations. *See* transformers
databases
 batch operations on 80
 JDBC transport
 insertions 79–80
 queries 78–79
 NoSQL with MongoDB 80–82
 overview 77–78
DataHandler class 45
DataMapper 383
dataSource-ref property 78
dataType property 376
dateTime property 374
DBObject class 81
debugging using Mule
 logging messages 328–330
 overview 328
 using breakpoints 331–333
decrypting messages with PGP 261–263
@Default annotation 343
default exception strategy 227
DefaultMuleContextFactory 209

deleteReadMessages property 72–73
demarcation, transactions 247–249
deployment
 embedding into existing application 208–210
 high availability clustering object stores 214–216
 designing applications for 213
 overview 210–213
 overview 23–26
standalone applications
 configuring logs 198–200
 deploying applications to Mule 196–198
 embedding web applications 202–204
inter-application communication 200–202
 overview 190–194
 packaging Mule applications 194–196
to CloudHub 206–208
to web container 204–206
description attribute 338
deserialization. *See* objects
development using Mule
 connecting to Mule local Mule application 294–295
 overview 293–294
 remote Mule application 295–298
 using transports 298–300
debugging
 logging messages 328–330
 overview 328
 using breakpoints 331–333
Mule API
 data persistence 310–315
 intercepting messages 303–307
 lifecycle methods 301–303
 listening to notifications 307–310
 overview 300–301
Mule context
 accessing 289–290
 overview 288–289
 using 290–293

development using Mule
(continued)
 testing
 behavior stubbing 319–323
 functional testing 315–318
 load testing 323–327

DevKit
 custom cloud connectors
 335–337
 endpoint URIs and 40
@Disconnect annotation 338, 340

dispatcher 293

dispatching, defined 35

Disposable interface.
See lifecycle

dispose method 155, 302

distributed memory grids 7

downloading
 ActiveMQ 16
 Mule Studio 9
 standalone server 23

Drools Guvnor 367

Drools Rules Engine
 CEP support 363
 creating rules 366–368
 message routing 368–370
 vs. Esper 363

durable property 67

dynamic ports 321

E

eBay 85

email transport
 receiving with IMAP
 transport 71–73
 sending with SMTP
 transport 73–75

embedding
 Mule, into existing application 208–210
 Mule, into web applications 204–206
 web applications in Mule 202–204

enableCookies property 59

encoding property 92, 375

encrypting messages
 decrypting with PGP 261–263
 password-based payload encryption 260

endpoints
 address attribute 53
 generic endpoints 53

inbound endpoint 53
 outbound endpoint 36, 53
 overview 53

URIs
 for messages 39–40
 specifying JMS destinations 69

endpoints value 349

Enterprise Integration Patterns 4–6

enterprise service bus. *See* ESB

entry point
 discovery 144
 exclude methods 145
 include methods 145
 resolving 143–146
 transformer 144
 void methods 144

EntryPointResolver 144

EnvelopeInterceptor. *See* interceptor

EPL (event-processing language) 362

error handling value 349

error handling. *See* exception handling

ESB (enterprise service bus)
 defined 5
 Mule as 6–7, 177–178

Esper Module
 installing 362
 sentiment analysis using Twitter 363–365
 vs. Drools Fusion 363

esper-config.xml file 362

event-driven systems 361–363

event-processing language.
See EPL

exception handling
 exception strategies
 configured on a per-flow basis 225
 default 225
 defining 225–227
 global 225
 messaging 225
 using 227–230

integration challenges 4

reconnection strategies
 creating 221–225
 overview 219–220

exception payload 41

exception variable 376

exception, defined 217

exchange patterns 13

exhaustedAction attribute 342

exhaustion of component pool 150

expression components 154

expression filters
 for payload 119
 overview 122–124

expressions (MEL)
 in XSL parameters 103
 using 47–49
 using with transformers 97–98

expression-transformer
 output 98
 overview 97–98

Extensible Markup Language.
See XML

Extensible Messaging and Presence Protocol. *See* XMPP

extensions
 DevKit and 336
 purpose of 6

F

Facebook 85, 363

failure expressions 137–138

fault tolerance. *See* topology

file attribute 379

File class 41

file connector
 autoDelete property 58
 configuration 52
 fileAge property 58
 overview 56–58

file endpoint 76

file transport 77

fileAge property 58

fileSeparator property 374

filters
 and filter 124
 Boolean evaluation 124
 expression filters 122–124
 header 119
 idempotent filter 125–126
 logical filters 124–125
 message filter 127
 not filter 124
 or filter 124
 payload type 119
 regex filter 120–121
 wildcard filter 120–121

filters value 349

firewall 67

first-successful router 136

Flow Analyzer 328
 flow control value 349
 flow variable transformers 95–97
flow. *See* thread pool
FlowConstructAware interface. *See* lifecycle
flow-ref element 32
flows
 asynchronous 13
 overview 13–16, 28–30
 private flows 34–35
 reliability, using VM
 transport 83–85
 response phase 30–31
 subflows 31–33
 testing 21–22
 VM transport and 83
flowVars 376
foreach message processor 80, 134–136
formats 4
friendlyName attribute 338
from property, SMTP
 transport 74
FTP transport
 ActiveMQ 388
 overview 75–76
 polling server 387
 receiving files 76–77
 sending files 77
@Function annotation 382
 functional test case 315, 317
 functional testing 315–318, 323
 functions, context (MEL) 377

G

generic endpoints 53
@GET annotation 63
 GET method 58
 get-public-timeline message
 processor 86
 getValue() method 48
 Git Cloud Connector 337
 global-functions element 379
 Gnu Privacy Guard. *See* GPG
 Google Chrome 63
 Google+ 363
 GPG (Gnu Privacy Guard) 261
 grid architecture 180
 GridFS 80
 Groovy
 builders 111
 sample application 387

@Groovy annotation 381
 groups of messages
 aggregating messages 133–134
 iterating with foreach message processor 134–136
 splitting up messages 132–133
 guaranteed routing
 failure expressions 137–138
 overview 136
 unreliable transports 136–137
Guerrilla SOA
 defined 5
 integration
 architecture 176–177
 gzip-compress-transformer 94
 gzip-uncompress-
 transformer 95

H

hash-square 47
 headers, filtering by 119
 hex-string-to-byte-array
 transformer 94
 Hibernate 77
 high availability
 clustering object stores 214–216
 designing applications for 213
 Mule EE 383
 overview 210–213
 home property 375
 honorQosHeaders property 68
 HornetQ 67
 host property 374
 FTP transport 76
 http connector 59
 hot deployment 211
HTTP (Hypertext Transfer Protocol) 75
 Apache JMeter and 323
 basic authentication 254
 chunking 283
 proxy in configuration
 patterns 167–168
 http connector
 configuration 52
 REST services 62–64
 sending and receiving
 data 58–61
 SOAP services 64–66
 using web services 61–62

HTTP endpoint
 adding to flow 13
 inbound 68
 polling thread model 273
HTTPS (Hypertext Transfer Protocol Secure) 255, 260
http-security-filter 254
 hub-and-spoke
 architecture 176–177
Hypertext Transfer Protocol.
 See HTTP

I

@Icons annotation 349
 id property 375
 idempotent filter 125–126, 214
IMAP connector 71–73
IMAPS (IMAP over SSL) 260
 imported classes 377–378
 inbound endpoint
 defined 51
 overview 53
 inbound scope 43
 inboundAttachments 45
@InboundAttachments
 annotation 381
 inboundAttachments
 property 376
@InboundHeaders
 annotation 148, 339, 343, 381
 inboundProperties
 property 376
Initialisable interface. *See* lifecycle
 initialise method 155, 302
 initialization interceptor 308
InputStream class 41
 installing
 Esper Module 362
 Mule Studio 9–12
 integration
 asynchronous
 messaging 182–185
 canonical data model 180–181
 Guerrilla SOA 176–177
 Mule as enterprise service bus 177–178
 Mule as mediation layer 178–180
 proxying SOAP requests with CXF 185–188
 technologies 3

integration testing 294
@IntegrationBean
 annotation 380
 inter-application
 communication 200–202
 interceptor
 component 303
 custom cloud
 connectors 344–345
 envelope 304
 Spring AOP 304
 stack
 defining 304
 using 306
@InvalidateConnectionOn
 annotation 338, 342
 invocation scope 43
 invocation styles 4
@InvocationHeaders
 annotation 339, 343
 ip property 374
 IP White Listing 384
 iterating messages 134–136

J

JAAS (Java Authentication and Authorization Service) 251
 JAR files 17
 Java API for RESTful Web Services. *See* JAX-RS
 Java API for XML Web Services.
See JAX-WS
 Java application
 deployment model 209
 Mule context 209
 pros and cons 210
 Spring parent context 210
 starting Mule 209
 stopping Mule 209
 Java Architecture for XML Binding. *See* JAXB
 Java Authentication and Authorization Service. *See* JAAS
 Java Business Integration. *See* JBI
 Java Message Service. *See* JMS
 Java Persistence API. *See* JPA
 Java service wrapper. *See* stand-alone server
 Java Transaction API. *See* JTA
`java.util.logging` 198
`javac` command 336
 JavaScript Object Notation. *See* JSON

javaVendor property 374
 javaVersion property 374
 JAXB (Java Architecture for XML Binding) 65, 105
 JAX-RS (Java API for RESTful Web Services)
 configuring simple service pattern 160
 connecting to REST web services 62–64
 JAX-WS (Java API for XML Web Services)
 configuring simple service pattern 161
 connecting to SOAP web services 64–66
 JBI (Java Business Integration) 8
 JBoss AS
 multiple resource transactions 242
 web service hosting 185
 JBossTS 242–244
 jBPM Process Definition Language. *See* JPDL
 JDBC endpoint
 Apache JMeter and 323
 inbound 78
 inbound querying and 78
 outbound 78–79
 query results as a Map 79
 single-resource transaction 232–234
 JDBC transport
 insertions 79–80
 queries 78–79
 JDBC-JobStoreTX 372
 JEE Application Server 242
 Jersey 388
 JMS (Java Message Service)
 1.0.2b 67–68
 1.1 67
 benefits for application integration 67
 brokers 68
 BytesMessage 69
DUPS_OK_ACKNOWLEDGE
 283
 durable subscription 69
 endpoints
 Apache JMeter and 323
 asynchronous messaging support 182
 configuring properties 15
 inbound 70
 outbound 68
 single-resource transaction 234–239
 URIs for 69
 highly-available provider 213
 JMSTimeToLive header
 property 238
 MapMessage 69
 ObjectMessage 69
 overview 67–68
 properties for 42
 queue 67
 receiving messages 69
 Reply-To 70
 selector 70
 sending messages 68–69
 StreamMessage 69
 synchronous communication 70–71
 temporary queue 70
 TextMessage 68
 threading and 273
 topic 67
 transacting message flows 238
 using selector filters 70
`jmx-default-config` 292
 Job Store persistence 372
 JPA (Java Persistence API) 387
 JPDL (jBPM Process Definition Language) 359, 361
`jpd़.xml` file 361
 JSON (JavaScript Object Notation)
 canonical data model 180
 MongoDB and 80
 native support 7
 object marshaling 107–109
 overview 105–106
 querying with MEL 106–107
 json-to-dbobject transformer 81
 JTA (Java Transaction API) 241–242, 244
 JUnit 21
 JVM (Java virtual machine) 82, 284

L

LDAP (Lightweight Directory Access Protocol)
 Apache JMeter and 323
 Spring Security 253–254
 SSO technologies 250–251
 legacy-entry-point-resolver-set 145

lifecycle
 of components 155–157
 custom adapters 157
 integration challenges 4
 interfaces 302
 methods in Mule API
 301–303
 Lightweight Directory Access Protocol. *See* LDAP
 listening to notifications 307–310
 load balancing 75
 load testing
 Apache JMeter and 326–327
 overview 323–325
 local Mule applications 294–295
 locale property 374
 log4j 198
 log4j.properties 198
 Logback 200
 logger message processor
 discovering properties 42
 expressions 48
 logging messages
 debugging using Mule 328–330
 performance
 considerations 282
 logical filters 124–125
 logs, deployment and 198–200
 @Lookup annotation 380

M

mailboxFolder property 72
 managers, security
 LDAP 253–254
 memory user-service 252–253
 overview 251–252
 security filters 254–255
 Map class 45, 79, 86
 MapMessage 69
 map-reduce operations 132, 180
 marshaling. *See* objects
 maxActive attribute 342
 maxIdle attribute 342
 maxOutstandingMessages
 property 84
 maxWait attribute 342
 MD5 file hashing 150, 306
 mediation layer 178–180
 MEL (Mule Expression Language)
 context functions 377

context objects 374–375
 context variables 376
 customizing 378–379
 defined 17
 documentation 373
 expression filters 122–124
 imported classes 377–378
 language reference 373–378
 Mule context and 290
 overview 46–47
 querying JSON with 106–107
 standardization using 7
 using expressions 47–49
 memory user-service 252–253
 MEPs (Message Exchange Patterns) 37–39
 message attribute 17
 message sources 348–349
 MessageProcessor class 37
 message-properties-transformer element 43
 messages
 attachments 45–46
 context object properties 375–376
 detaching 93
 encoding 92
 encryption
 decrypting with PGP 261–263
 password-based payload encryption 260
 endpoint URLs 39–40
 exceptions
 defining exception strategies 225–227
 using exception strategies 227–230
 execution context 46
 filter 127
 flows
 overview 28–30
 private flows 34–35
 response phase 30–31
 subflows 31–33
 groups of
 aggregating messages 133–134
 iterating with foreach message processor 134–136
 splitting up messages 132–133
 interactions with 35
 intercepting 303–307
 Java class of payload 41
 logging when debugging 328–330
 message enricher 98–99
 message exchange patterns 37–39
 message processors
 custom cloud connectors 343–344
 null values from 29
 overview 36–37
 poll message processor 370
 message sources 36
 overview 40–41
 performance considerations 283
 properties
 overview 42
 removing 96
 renaming 96
 scopes for 42–45
 transformer 88, 95
 routing with Drools 368–370
 thread safety of 41
 meta information. *See* properties
 method property 59
 method-entry-point-resolver 144
 Microsoft Active Directory 252
 MIME types 116
 minMuleVersion attribute 338
 MMC (Mule Management Console)
 debugging 328
 Mule EE 383
 @MockIntegrationBean annotation 380
 modular 175
 @Module annotation 337
 MongoDB 80–82
 monitoring
 event-driven systems using CEP 361–363
 integration challenges 4
 moveToDirectory property 57
 moveToPattern property 57
 Mule
 competition 7–8
 deploying applications 23–26
 Enterprise Integration Patterns 4–6
 as ESB 6–7
 flows
 overview 13–16
 testing 21–22

- Mule (*continued*)
- installing Mule Studio 9–12
 - mule object, execution context 46
 - name explained 6
 - running applications 17–20
 - testing applications 21–22
 - versions 2 and 3 30
 - web service hosting on 185
 - XML configuration 22–23
- @Mule annotation 49, 382
- Mule API 287
- data persistence 310–315
 - intercepting messages 303–307
 - lifecycle methods 301–303
 - listening to notifications 307–310
 - overview 300–301
- Mule client
- bootstrapping Mule 298
 - disposing 296, 298
 - in memory 294
 - instantiating 294, 296–298
 - module 295
 - role in testing 294
 - using transports directly 298
- Mule context
- accessing 289–290
 - how to get ahold of 289
 - object properties 375
 - overview 288–289
 - starting and disposing 290
 - statistics 291
 - system configuration 291
 - using 290–293
- Mule CRC32 Processor 384
- Mule Credentials Vault 384
- Mule Deployable Archive 24
- Mule DevKit 7
- Mule Digital Signature Processor 384
- Mule EE (Enterprise Edition)
- Anypoint Enterprise Security 384
 - Anypoint Service Registry 384
 - batch database operations 80
 - clustered object store 315
 - DataMapper 383
 - debugging in 328
 - frequent updates 384
 - high availability 383
 - MMC 383
 - Quartz job persistence 372
- support 384
- transports 383
- Mule Expression Language.
- See* MEL
- Mule Filter Processor 384
- Mule Management Console.
- See* MMC
- Mule Message Encryption Processor 384
- Mule Studio
- configuration patterns and 158
 - configuring connectors 53–56
 - creating projects 11
 - Drools and 368
 - installing 9–12
 - integrating extensions with 349–350
 - response phase in 30
 - subflows vs. private flows in 35
- MULE_BPM_PROCESS_ID header property 361
- MuleClient class
- addresses in 53
 - class 293
 - interface 293
- MuleConfiguration class 291
- MuleContextAware interface 290
- MULE_CORRELATION_GROUP_SIZE header property 133–134
- MULE_CORRELATION_ID header property 133–134
- MuleEvent class 40
- MuleMessage class 40
- mule-module-annotations module 380
- mule-module-client module 295
- mule-module-xml module 379
- mule-receive action 360
- MuleReceiverServlet 205
- MuleRESTReceiverServlet 206
- mule-send action 360
- MuleSoft Connectors Library 81
- MultiConsumerJmsMessage-Receiver 275
- multiple recipients, routing to all router
- overview 128
 - request-response exchange pattern with 129–130
 - async processor 130–131
- multiple-resource transactions
- overview 241–242
 - using JBossTS 242–244
 - using XA transactions in container 244–246
- MVEL (MVFLEX Expression Language) 46, 373
-
- ## N
- name attribute 338
- name property 375
- namespace attribute 186, 338
- namespaces, XML
- configuration 23
- nanoTime property 374
- network load balancer. *See* topology
- NFS (Network File System) 58
- no-arguments-entry-point-resolver 144
- nodeId property 375
- NoSQL
- connector for 80–82
 - extensions and 7
- notifications
- activating 308, 310
 - advanced features 310
 - framework 307
 - listening to 307–310
- null values 29
- NullPayload 29
- Nygard, Michael 3
-
- ## O
- Oasis Open Composite Services Architecture 8
- OAuth 346
- object stores, clustering 214–216
- ObjectMessage 69
- objects
- context objects (MEL) 374–375
 - deserialization
 - from bytes 92
 - from xml 104
 - JSON object
 - marshaling 107–109
 - serialization
 - to bytes 93
 - to xml 104–105
 - XML object marshaling 104–105

object-to-byte-array-transformer 93
 object-to-xml-transformer 104–105
 one-way MEPs 13, 37
 OpenAM 250
 OpenLDAP 252
 OpenPGP 261
 @Optional annotation 343
 org.apache.commons.lang
 .RandomStringUtils 379
 org.mule.module.xml.el
 .XMLExpressionLanguageExtension 379
 org.mule.util.StringUtils 378
 osArch property 374
 osName property 374
 osVersion property 374
 outbound endpoint
 defined 36, 51
 overview 53
 outbound scope 43
 @OutboundAttachments
 annotation 381
 outboundAttachments
 property 376
 @OutboundHeaders
 annotation 148, 339, 343, 381
 outboundProperties
 property 376
 outputAppend property 56
 outputPattern property
 file transport 57
 FTP transport 75

P

packaging Mule applications 194–196
 parallelization of tasks 38
 passing property 76
 @Password annotation 350
 password property
 FTP transport 76
 http connector 59
 passwords, payload encryption 260
 @Path annotation 63, 350
 path property
 http connector 59
 VM transport 82
 patterns. *See* configuration patterns

payload
 defined 41
 filtering by type of 119
 format transformer 88
 Java class of 41
 type transformer 88
 @Payload annotation 148, 339, 381
 payload attribute 187
 payload property 376
 peer-to-peer architecture 180
 performance
 improving 281
 processing strategies
 configuring 276–278
 synchronicity and 269–272
 profiler-based investigation 279–281
 SEDA and 265–267
 thread pools
 configuring 274–276
 overview 267–269
 transport peculiarities 272–274
 transports 281–284
 persistent property 84
 persistentDelivery property 67
 PGP (Pretty Good Privacy)
 credential accessor 262
 decrypting messages with 261–262
 key alias 262
 key ring management 262
 MuleHeaderCredentials-Accessor 262
 secret password 262
 SecurityManager and 251
 Unlimited Strength Jurisdiction Policy files
 requirement 262
 point-to-point integration 5
 POJOs (plain old Java objects) 66, 335
 policy template 223
 policyOK 222
 PolicyStatus 222–223
 poll message processor 370
 polling
 defined 35
 poll message processor 370
 See also scheduling
 pollingFrequency property
 file transport 56
 FTP transport 75
 JDBC transport 78
 pooling components 149–151
 pooling profile 149–150
 POP3S (POP3 Secure) 260
 port property
 FTP transport 76
 http connector 59
 ports, dynamic 321
 POST method 58–59
 Prancing Donkey Maltworks, Inc
 caching interceptor 304
 canonical data models 110
 company 8
 custom logging levels 199
 data coupling 237
 initialization interceptor 308
 JBossTS and 242
 lost messages 234
 MD5 file hashing 150, 306
 OpenLDAP and 253
 publishing of analytics data 242
 transactionally receiving billing data 242
 use of relational databases 233
 use of transactions 218, 232
 use of transactions with outbound endpoints 237
 preconfigured flows. *See* configuration patterns
 Pretty Good Privacy. *See* PGP
 private flows 34–35
 processing strategies
 configuring 276–278
 synchronicity and 269–272
 processingStrategy attribute 34
 @Processor annotation 338, 345
 processor-chain element
 encapsulating message processors 37
 subflows and 32
 processors, message
 null values from 29
 overview 36–37
 @Produces annotation 63
 ProductImportFunctionalTestCase, sample application 388–389
 -profile parameter 279
 Profiler Pack 279
 properties
 copying across scopes 43
 discovering in development mode 42

properties (*continued*)

- for messages
- defined 41
- overview 42
- scopes for 42–45
- JMS transport 42
- naming requirements for transports 42
- property placeholders 56
- property transformers 95–97
- property-entry-point-resolver 144
- protocols 4
- prototype-object 147
- proxyHostname property 59
- proxying SOAP requests with CXF 185–188
- proxyPassword property 59
- proxyPort property 59
- proxyUsername property 59
- public key encryption 261

Q

Quartz transport

- overview 371–372
- using with Twitter 86
- quartz.properties file 372
- queries, using JDBC transport 78–79
- queryKey property 78
- queue property 67
- queueTimeout property 82

R

ramp-up period 324

- readFromDirectory property 56
- receiver
 - idempotent 214
 - polling 273
- receiving, defined 35
- reconnection strategies
 - creating 221–225
 - custom cloud connectors 341
 - overview 219–220
- ref message processors (Esper) 362
- reflection-entry-point-resolver 144
- regex filter 120–121
- regex() method 47, 377

registry

- looking up flows 291
- looking up objects 292
- Spring application context 292
- storing objects 292
- registry property 375
- Release It!* 3
- reliable flows 83–85
- remote dispatcher
 - agent 295
 - security considerations 298
 - usage 296
 - wire format 295–296
- remote Mule applications 295–298
- replyTo property 376
- replyToAddresses property 74
- Representational State Transfer. *See REST*
- requesting
 - defined 35
 - synchronous 293
- request-response MEPs
 - BPM example 360
 - defined 13, 37
 - JMS transport 70
 - using all router 129–130
 - VM transport 84
- resolving entry point 143–146
- response phase for flows 30–31
- REST (Representational State Transfer)
 - connecting to services 62–64
 - custom cloud connectors
 - consumers for 345–347
 - example of 350–355
 - native support 7
 - REST Console 19, 63
 - @RestCall annotation 339, 346
 - @RestFailOn annotation 346
 - @RestHeaderParam annotation 339
 - RetryPolicy 221
 - reusing configuration patterns 170–172
 - reversePayload() function 378
 - rootId property 375
 - routers
 - defined 36, 113
 - selective consumer 151
 - routing data
 - choice router 115–118
 - filters
 - expression filters 122–124
 - header 119

idempotent filter 125–126

- logical filters 124–125
- message filter 127
- payload type 119
- regex filter 120–121
- wildcard filter 120–121
- groups of messages
 - aggregating messages 133–134
 - iterating with foreach message processor 134–136
 - splitting up messages 132–133
- guaranteed routing
 - failure expressions 137–138
 - overview 136
 - unreliable transports 136–137
- multiple recipients
 - all router 128
 - all router, request-response exchange pattern with 129–130
 - async processor 130–131
- rules engines
 - advantages of 365
 - Drools
 - creating rules 366–368
 - message routing 368–370
- running applications 17–20
- RuntimeException 222, 347

S

Salesforce CRM 3, 85, 180, 364

- sample application
 - application configuration 386–388
 - ProductImportFunctionalTestCase 388–389
- SAP 85
- @Schedule annotation 380
- scheduling 371–372
 - See also* polling
- schemaLocation attribute 338
- scopes for message properties 42–45
- scopes value 349
- SCP (Secure Copy) 75
- scripting components
 - custom transformers 109–112
- engine attribute 153

- scripting components
(continued)
- externally stored script 153
 - file attribute 153
 - inline script 151–152
 - JSR-223 compliant engine 152
 - refreshable scripts 153
 - script context 152–153
- Secure Copy. *See* SCP
- Secure Shell. *See* SSH
- Secure Socket Layer. *See* SSL
- Secure Token Service. *See* STS
- security
- message encryption
 - decryption with PGP 261–263
 - password-based payload encryption 260
 - security managers
 - authentication and 254
 - implementing
 - `org.mule.api.security.SecurityManager` 251 - LDAP 253–254
 - memory user-service 252–253
 - overview 251–252
 - security filters 254–255
 - swapping out delegate references 254
- SOAP services 258–259
- SSL
- client 256–258
 - HTTPS protocol 255
 - server 255–256
 - WS-* services 258–259
- security value 349
- SEDA (staged event-driven architecture) 265–267
- selector filters 70
- semicolon 47
- sending, defined 35
- sentiment analysis using Twitter 363–365
- serializable-to-byte-array-transformer 94
- serialization 61, 67
- server context object 46, 374
- Server Message Block. *See* SMB
- ServiceMix 8
- service-oriented architecture. *See* SOA
- services
- in Mule 3 30
 - supporting multiple versions 178
- servlet container. *See* web applications
- servlet transport 205
- ServletContextListener 204
- session scope 43
- session variables
- ownership of 45
 - transformers 95–97
- @SessionHeaders annotation 343
- sessionVars 376
- set-property element 43–44
- set-session-variable element 43
- set-variable element 43
- Simple Logging Facade for Java. *See* SLF4J
- Simple Mail Transfer Protocol Secure. *See* SMTPS
- simple service pattern
- configuring with JAX-RS 160
 - configuring with JAX-WS 161
 - overview 158–160
- SimpleRetryPolicy 223
- single sign-on. *See* SSO
- single-resource transaction
- overview 232
 - using JDBC endpoints 232–234
 - using JMS endpoints 234–239
- singletons 146, 156, 289
- sink, defined 51
- SLF4J (Simple Logging Facade for Java) 198
- SMB (Server Message Block) 58
- Smooks transformer 283
- SMTP transport
- outbound endpoint 73
 - overview 73–75
- SMTPS (Simple Mail Transfer Protocol Secure) 260
- SOA (service-oriented architecture) 4–6
- SOAP services
- advantages of 178
 - connecting to 64–66
 - proxying requests with CXF 185–188
 - security for 258–259
- SoapUI 65
- SocketTimeoutException 137
- @Source annotation 339, 348
- SourceCallback attribute 344, 348
- sources, message 36
- sourceTypes attribute 347
- Spaghetti integration 5
- specification property 68
- splitting up messages 132–133
- Spring
- JDBC datasource 79
 - JDBC template 77
 - parent context 210
- Spring Portfolio 251
- Spring Security
- { } evaluation 253
 - defining rootDN 253
 - group-search-base 253
 - memory user-service 252–253
 - MULE_USER for username propagation 253
 - OpenLDAP and 253
 - overview 251–252
 - prepending of ROLE_ 253
 - security filters 254–255
 - user-dn-pattern 253
- spring-object 147
- SpringSource 8
- SpringXmlConfiguration-Builder 209
- SSH (Secure Shell) 75
- SSL (Secure Socket Layer) 219, 260
- client 256–258
 - HTTPS protocol 255
 - server 255–256
- SSO (single sign-on) 250
- stack traces 331
- staged event-driven architecture. *See* SEDA
- standalone applications
- command line 191
 - configuring logs 198–200
 - deploying applications to Mule 196–198
 - directory structure 192
 - embedding web applications 202–204
- inter-application communication 200–202
- overview 190–194
- packaging Mule applications 194–196
- passing parameters 191
- pros and cons 193
- as service 191

standalone property 375
 standalone server
 downloading 23
 -profile parameter 279
 start method 155, 290, 302
 Startable interface. *See* lifecycle
 stateless components 147
 statements (MEL) 47
 statistics 291
 stop method 155, 290, 302
 Stoppable interface.
 See lifecycle
 streaming 283
 StreamMessage 69
 String values, evaluating as
 Booleans 48
 string-to-byte-array-transformer 94
 STS (Secure Token Service)
 384
 subflows 31–33
 subject property 74
 @Summary annotation 350
 Synapse 8
 synchronous communication
 disadvantages of 70
 processing strategies and
 269–272
 using JMS transport 70–71
 system exceptions
 creating reconnection
 strategies 221–225
 overview 219–220
 systemProperties property 374

T

TCK (Test Compatibility Kit)
 AbstractTransformerTestCase
 317
 functional test component
 319
 FunctionalTestCase 315–316
 @Test annotation 21
 testing
 Apache Derby and 316
 behavior stubbing 319–323
 flows 21–22
 functional testing 315–318
 GreenMail and 316
 HSQL and 316
 integration 294
 load testing 323–327
 @Text annotation 350
 TextMessage 69

thread pool
 asynchronous-synchronous
 271
 buffer 269, 273
 component pools vs. 268
 dispatcher 268
 exhausted 274
 flow 268
 fully asynchronous 269
 fully synchronous 271
 message processing and 269
 not handled by Mule 273
 performance 274–276
 polling receiver 273
 receiver 268
 synchronous-
 asynchronous 270
 transactions and 269
 transport peculiarities
 272–274
 VM transport 272
 thread safety
 components 147, 149
 initialise method and 156
 threading profile
 configuring 274
 exhausted action 274
 hierarchy of profiles 274
 performance 275
 service 274
 threads
 JDBC operations on separate 39
 message modification and 41
 Tibco EMS 67
 timeZone property 374
 TLS (Transport Layer Security) 260
 tmpDir property 374
 Tomcat 185
 topic property 67
 topology
 fault tolerance 213
 high availability 210
 load balancer 211
 transactions 213
 actions
 ALWAYS_BEGIN 234–235,
 239, 243
 ALWAYS_JOIN 236, 239,
 244
 atomicity 230
 component failures 235
 consistency 230
 databases and 232
 demarcation 247–249
 durability 231
 isolation 230
 JMS 234–235
 multicasting-router 239
 multiple-resource transactions
 overview 241–242
 using JBossTS 242–244
 using XA transactions in container 244–246
 MySQL 233
 outbound endpoint and 237
 overview 230–232
 real-world examples 218
 rollback 230, 234, 238
 single-resource transaction
 overview 232
 using JDBC endpoints
 232–234
 using JMS endpoints
 234–239
 synchronous operations 239, 269
 timeout 238
 XA
 HeuristicExceptions 241
 JDBC provider support
 241
 JMS provider support 241
 LookupFactory and 244
 requiring special drivers
 241
 rollback 244
 specifying JTA location
 with JNDI 245
 two-phase commit 241
 using in application container 244
 using with Resin JTA 244
 XA datasources and JMS and JDBC 241
 @Transformer annotation 339, 347, 380
 transformers
 annotations for 380
 behavior 88
 configuring 89–91
 core transformers
 auto-transformer 99–100
 byte transformers 92–94
 compression
 transformers 94–95
 flow variable
 transformers 95–97

transformers, core transformers
(*continued*)
message enricher 98–99
property transformers
95–97
session variable transformers 95–97
using expressions 97–98
custom cloud
connectors 347–348
custom, using scripting languages 109–112
defined 36
encoding 90
input type 90
JSON
 JSON object marshaling 107–109
 overview 105–106
 querying JSON with MEL 106–107
local 89
mime Type 90
overview 88–89
return class 88, 90, 99
round-trip 89
XML
 JAXB and 105
 XML object marshaling 104–105
 XPath and 101
 XSLT transformations 102–104
transformers value 349
Transport Layer Security. *See* TLS
transports
 connecting to 298–300
 defined 51
 Mule EE 383
 peculiarities of 272–274
 performance considerations 283
tuning. *See* performance
Tuscany 8
Twilio 85
Twitter
 connector for 85–86
 sentiment analysis 363–365
 uses for Mule 3
 using poll message processor with 370
two-phase commit (2PC) 241

U

unmarshaling. *See* objects
unreliable transports 136–137
until-successful router 136
upsert processor 387
URIs (Uniform Resource Identifiers)
 endpoints for messages 39–40
 identifying connector in 69
user property
 FTP transport 76
 http connector 59
userDir property 374
userHome property 374
userName property 374
UUID class 47

V

@ValidateConnection
 annotation 338, 342
validator pattern 48, 164–167
variables, context (MEL) 376
verbose stack traces 331
version property 375
VM endpoints 33
VM transport
 asynchronous messaging support 182
 default connector 52
 flows and 83
 MEPs types and 37
 overview 82
 persisted queues 213
 reliable flows using 83–85
 thread pool 272
 using in transaction 239

W

web applications
 deployment model 204
 embedding in Mule 202–204
 embedding Mule into 204–206
 interacting with Mule 205
 pros and cons 206
 service URIs 205
 starting Mule 204
 stopping Mule 204
Web Service Definition Language. *See* WSDL

web services

 hosting on Mule 185
 specifications 67
 using with HTTP transport 61–62
WebLogic 242
wildcard filter 120–121
wire format. *See* remote dis-patcher
workdir property 375
WorkingHoursAwareExhaustible-RetryPolicy 223
wrapper.conf 199
wrapper.logfile.maxfiles 199
wrapper.logfile.maxsize 199
writeToDirectory property 56
WS-* services
 proxy configuration pattern 168–170
 security for 258–259
WSDL (Web Service Definition Language) 64
wsdl2java 66, 185
wsdlLocation attribute 186

X

XA transactions
 HeuristicExceptions 241
 JDBC provider support 241
 JMS provider support 241
 LookupFactory and 244
 requiring special drivers 241
 rollback 244
 specifying JTA location with JNDI 245
 two-phase commit 241
 using in container 244–246
 using with Resin JTA 244
XA datasources and JMS and JDBC 241
XML (Extensible Markup Language)
 configuration files 22–23
 configuring connectors 52–53
 escaping characters 70, 79
 JAXB and 105
 object marshaling 104–105
 wrapping expression in 47
XPath and 101
XSLT transformations 102–104
xml-to-object-transformer 104

xml-wire-format. *See* remote dispatcher
XMPP (Extensible Messaging and Presence Protocol) 260
XPath
 overview 101
 routing based on 117

sample application 387
@Xpath annotation 381
xpath() method 46, 377
XSLT (XSL Transformation) 102–104
xslt-transformer 102–103
XStream 104

Y
YourKit 279
Z
ZeroMQ 7

Mule IN ACTION, Second Edition

Dossot • D'Emic • Romero

An enterprise service bus is a way to integrate enterprise applications using a bus-like infrastructure. Mule is the leading open source Java ESB. It borrows from the Hohpe/Woolf patterns, is lightweight, can publish REST and SOAP services, integrates well with Spring, is customizable, scales well, and is cloud-ready.

Mule in Action, Second Edition is a totally revised guide covering Mule 3 fundamentals and best practices. It starts with a quick ESB overview and then dives into rich examples covering core concepts like sending, receiving, routing, and transforming data. You'll get a close look at Mule's standard components and how to roll out custom ones. You'll also pick up techniques for testing, performance tuning, and BPM orchestration, and explore cloud API integration for SaaS applications.

What's Inside

- Full coverage of Mule 3
- Integration with cloud services
- Common transports, routers, and transformers
- Security, routing, orchestration, and transactions

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

David Dossot is a software architect and has created numerous modules and transports for Mule. **John D'Emic** is a principal solutions architect and **Victor Romero** a solutions architect, both at MuleSoft, Inc.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/MuleinActionSecondEdition

Free eBook
SEE INSERT

“Captures the essence of pragmatism that is the founding principle of Mule.”

—From the Foreword by Ross Mason, Creator of Mule

“A new, in-depth perspective.”

—Dan Barber, Penn Mutual

“Excellent topic coverage and code examples.”

—Davide Piazza, Thread Solutions srl, MuleSoft Partner

“This edition has grown, with more real-world examples and a thorough grounding in messaging.”

—Keith McAlister, CGI



MANNING

\$59.99 / Can \$62.99 [INCLUDING eBOOK]

ISBN 13: 978-1-617290-82-4
ISBN 10: 1-617290-82-3



9 781617 290824

