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
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. 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**

```xml
<process name="submitOrder"
   xmlns="http://jbpm.org/4.3/jpdl">
  <start name="start">
    <transition to="fork"/>
  </start>

  <fork name="fork">
    <transition to="sendOrderToSalesForce"/>
    <transition to="sendOrderToNetSuite"/>
  </fork>
</process>
```

1 Users wishing to implement support for another BPM engine need to implement the org.mule.module.bpm.BPMS interface.
Send message synchronously to crm.customer.create queue and wait for response on temporary queue

Send message synchronously to erp.order.record queue and wait for response on temporary queue

When responses from both queues received, proceed

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

```xml
<bpm:jbpm name="jbpm"/>
<flow name="orderProcessing">
   <jms:inbound-endpoint queue="order.submit">
      <jms:transaction action="ALWAYS_BEGIN"/>
   </jms:inbound-endpoint>
</flow>
```

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.
Complex event processing

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 .jpdl.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 syn-
chronization 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 com-
pleted 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"/>

</esper-configuration>
```

For this example, you reuse the Order class from your canonical domain model to re-
present the OrderCompletedEvent. A more robust implementation would likely dedi-
cate 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 lat-
ter 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 comple-
tion 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"/>
  </esper:listen>
</flow>
```

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**

```xml
<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>
```
The first flow subscribes to the Twitter event stream 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.

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

```java
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
```

Define MessageService

1. Global MessageService

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

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 GUVNOR 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

```xml
<bpm:drools/>
<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>
```
**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**

```java
public class Alert implements Serializable {
    private String application;
    private String severity;
    private String description;
    ...
}
```

The next listing shows a rules file that defines how alerts are routed.

**Listing 14.9  The Alert routing DRL file**

```java
package com.prancingdonkey.domain;
import org.mule.MessageExchangePattern;
import java.util.Map;
import java.util.HashMap;

global org.mule.module.bpm.MessageService mule;
rule "weekdays are high priority"
    calendars "weekday"
    when
        $alert : Alert()
    then
```

1. **Define MessageService**

1. **Set a higher severity during week**

...
Using a rules engine with Mule

```
$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 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/>

<brp: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>
```
For this example, you don’t have an initial list of facts, so you define an empty list of initial facts. The rules engine is then invoked 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.

```xml
<flow name="main">
  <poll frequency="300000">
    <twitter:get-public-timeline/>
  </poll>
  <collection-splitter/>
  <esper:send eventPayload-ref="#[message.payload]"/>
</flow>
```

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.

![Cron expression](image-url)

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.

```xml
<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>
```

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.
A n 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