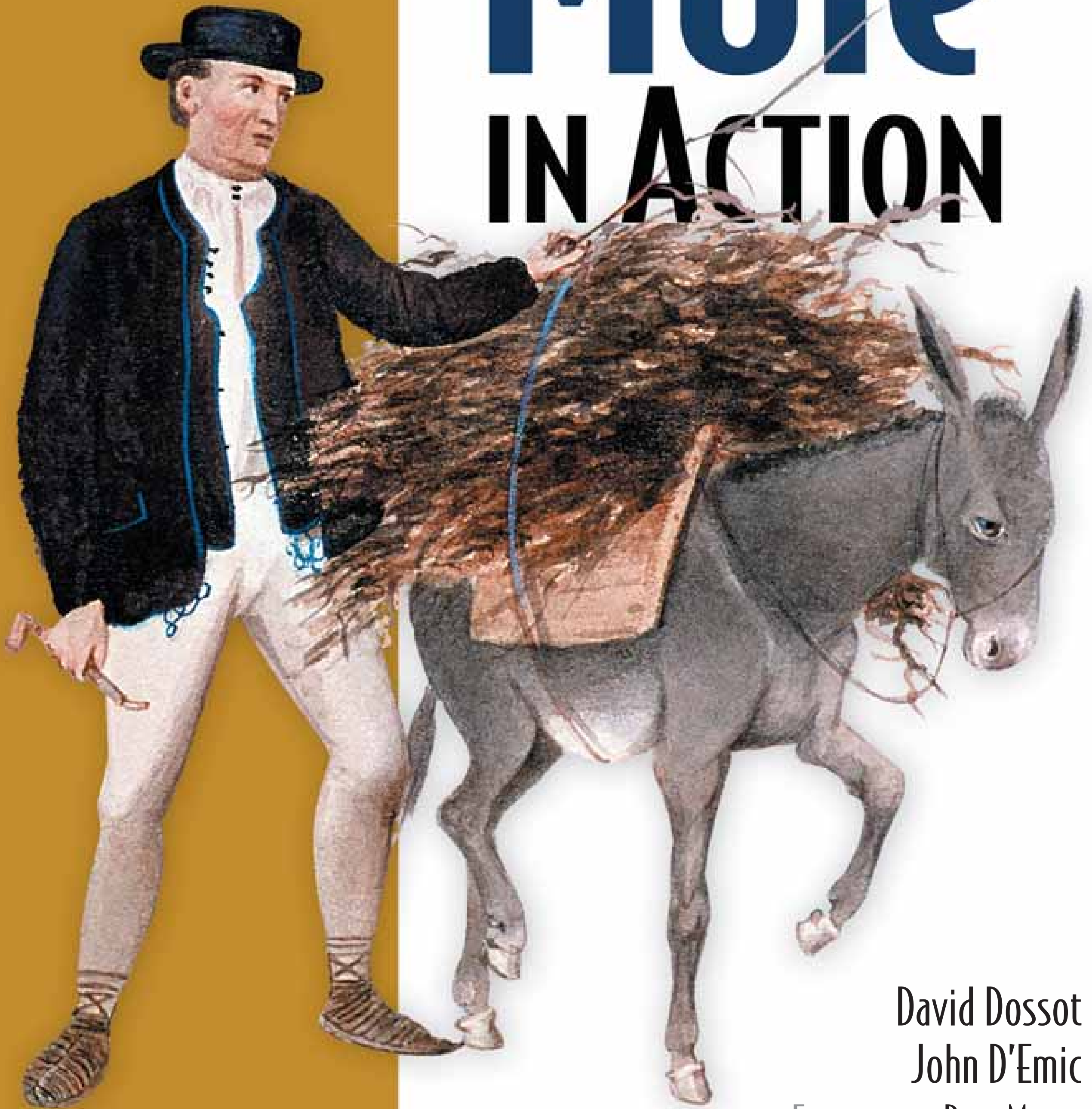
MANNING

SAMPLE CHAPTER

Mule IN ACTION



David Dossot
John D'Emic

FOREWORD BY ROSS MASON

Mule in Action

David Dossot

John D'Emic

Chapter 8

Copyright 2010 Manning Publications

brief contents

PART 1 CORE MULE 1

1 ■ Discovering Mule 3
2 ■ Configuring Mule 21
3 ■ Sending and receiving data with Mule 39
4 ■ Routing data with Mule 82
5 ■ Transforming data with Mule 108
6 ■ Working with components 139

PART 2 RUNNING MULE 165

7 ■ Deploying Mule 167
8 ■ Exception handling and logging 196
9 ■ Securing Mule 218
10 ■ Using transactions with Mule 233
11 ■ Monitoring with Mule 251

PART 3 TRAVELING FURTHER WITH MULE 271

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

8 Exception handling and logging

In this chapter

- Managing exceptions with exception strategies
- Using retry policies
- Logging with Mule

Dealing with the unexpected is an unfortunate reality when writing software. Through the use of exceptions, the Java platform provides a framework for dealing with events of this sort. Exceptions occur when unanticipated events arise in a system. These are things such as network failures, I/O issues, and authentication errors. When you control a system, you can anticipate these events and provide a means to recover from them. This luxury is often absent in a distributed integration environment. Remote applications you have no control over will fail for no apparent reason or supply malformed data. A messaging broker somewhere in your environment might begin to refuse connections. Your mail server's disk may fill up, prohibiting you from downloading emails. Your own code might even have a bug that causes your data to be routed improperly. In any case, it's undesirable for your entire application to fail because of a single unanticipated error.

Logging is closely related to exception recovery. You naturally want to know when error conditions occur. This enables you to identify where the issue is and recover from it. If you have a bunch of data-type exceptions on one of your endpoints, for instance, you'll want to know where they're coming from—even if you're correctly ignoring them. Logging also aids in debugging—giving you insight into what your system's doing.

Mule's exception handling and logging functionality recognize these facts. They let you plan for, react to, and log errors that would otherwise bring your integration process to a screeching halt. You'll find yourself leveraging Mule's exception handling ability to identify and troubleshoot failures in your endpoints, components, and routers.

In this chapter we'll be examining how Mule implements exception handling and logging. We'll first consider exception strategies, where we'll see how Mule lets you react to errors on your connectors and components. We'll see how you can use Mule's routing capabilities to control where exceptions are sent after they're generated. We'll then take a look at how Mule uses the SLF4J logging facade and log4j to simplify logging configuration. Finally, we'll see how you can use Apache Chainsaw as a graphical front end to view Mule's logging data.

8.1 Exception strategies

Mule uses *exception strategies* to handle failures in connectors and components. Runtime exceptions thrown in connectors and components have the potential to "trickle up" and wreak havoc. This could cause core parts of Mule and even Mule itself to fail. Exception strategies prohibit this—they catch an exception and perform an action as a result. The appropriate response might be as simple as logging the exception and moving on, or as complex as rolling back a transaction.

While you're free to implement your own exception strategies, Mule supplies default exception strategies that are flexible enough to handle basic exception handling requirements. We'll start off this section by examining these exception strategies. We'll then see how you can use Mule's routing capabilities in conjunction with exception strategies to intelligently route and handle errors.

8.1.1 Positioning exception strategies

Mule provides exception strategies for connectors and services. The default exception strategy for connectors is responsible for handling transport-related exceptions—such as SSL errors on an HTTPS endpoint, or a connection failure on a JMS endpoint. The default exception strategy for services handles exceptions that occur in components. As components generally host your custom code, exceptions thrown here will usually be related to your business logic. By default, both of these strategies handle exceptions in the same way—they'll log the exception and Mule will continue execution.

Being able to define separate exception strategies for connectors and components lets you handle each sort of error independently. This is often desirable. You may want connector-level exceptions logged at a higher level than component-level exceptions,

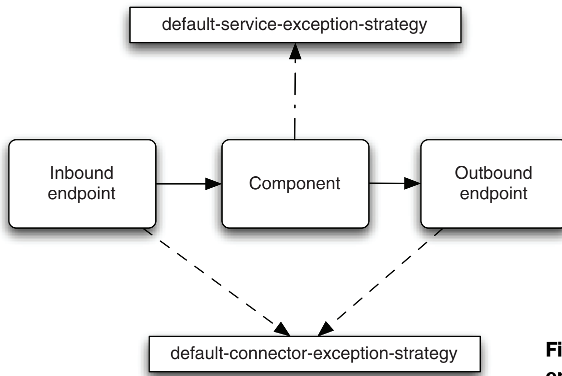


Figure 8.1 Handling exceptions on endpoints and components

for instance. As we'll see in the next chapter, this also gives you the flexibility to handle certain transaction-related responses, such as rollbacks, differently, based on where an exception occurs. The default exception strategies are illustrated in figure 8.1

You have the option of explicitly defining exception strategies in multiple places in your Mule configurations. Exception strategies can be configured on a *per-model* basis, before any service definitions. In this case, the exception strategy, either service or connector, will be applied to all subsequent services and connectors defined in the configuration. You can additionally define exception strategies on a *per-service* basis. This is done by defining the default-connector exception strategy or the default-service exception strategy at the end of each service definition.

While Mule will implicitly configure the default exception strategies for you, in order to override the defaults it's useful to see how to manually configure them. We'll demonstrate the placement of exception strategies in this section by showing where we can place the default exception strategies. You'll need this information in the next section, where the placement of an exception strategy will dictate how errors are routed out of a model or service. This will also be useful when you implement and place custom exception strategies.

The default exception strategy for connectors is configured by defining a `default-connector-exception-strategy` element on either a model or on a service. Defining the default-connector exception strategy on a model will cause all connectors used in that model to be handled by the defined exception strategy. Let's revisit listing 3.7 from chapter 3 and explicitly define the default-connector exception strategy for the model. The result is shown in listing 8.1.

Listing 8.1 Configuring the default-connector exception strategy on a model

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

```
    </file:connector>

<model name="smtpModel">
    <default-connector-exception-strategy/>        <!-- 1 Configure default-connector exception strategy -->

    <service name="smtpService">
        <inbound>
            <file:inbound-endpoint path="./data/invoice">        <!-- 2 File inbound endpoint -->
                <file:file-to-string-transformer/>
            </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <smtp:outbound-endpoint host="mail.clood.com"
                                        from="mule@clood.com"
                                        subject="Accounting Invoice"
                                        to="accounting@clood.com">        <!-- 3 SMTP outbound endpoint -->
                    <email:string-to-email-transformer/>
                </smtp:outbound-endpoint>
            </pass-through-router>
        </outbound>
    </service>

    <service name="fileService">
        <inbound>
            <file:inbound-endpoint path="./data/snapshot">        <!-- 4 File inbound endpoint -->
                <file:filename-wildcard-filter pattern="SNAPSHOT*.xml"/>
                </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
              <file:outbound-endpoint
                 path="./data/archive"
                 outputPattern=
                 "#[header:originalFilename]-#[function:dateStamp].xml"/>        <!-- 5 File outbound endpoint -->
            </pass-through-router>
        </outbound>
    </service>
</model>
```

The default-connector exception strategy is configured after the model definition and before any service definitions, as we see in ❶. This default-connector exception strategy will now handle all transport-related exceptions for the endpoints defined on ❷, ❸, ❹, and ❺. We can also define the default-connector exception strategy on a per-service basis as well. This is done by defining the default-connector exception as the last element of a service, as we see in listing 8.2.

Listing 8.2 Configuring the default-connector exception strategy on a service

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

<model name="smtpModel">

    <service name="smtpService">
        <inbound>
```

```
            <file:inbound-endpoint path="./data/invoice">
                <file:file-to-string-transformer/>
            </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <smtp:outbound-endpoint host="mail.clood.com"
                                        from="mule@clood.com"
                                        subject="Accounting Invoice"
                                        to="accounting@clood.com">
                    <email:string-to-email-transformer/>
                </smtp:outbound-endpoint>
            </pass-through-router>
        </outbound>
    <default-connector-exception-strategy/>          ❶
    </service>

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

The `default-connector-exception-strategy` elements are defined on ❶ and ❷. Defining them here will cause connector-level exceptions thrown by the `smtpService` and the `fileService` to be handled independently of each other.

Now let's turn our attention to the default-service exception strategy, which defines how exceptions on our components are handled. Let's look at listing 6.8 from chapter 6 and see how to explicitly configure the default-service exception strategy to handle errors thrown by the `RandomIntegerGenerator`. The result is shown in listing 8.3.

Listing 8.3 Configuring the default-service exception strategy on a particular service

```
<service name="RandomIntegerGenerator">

  <inbound>
    <vm:inbound-endpoint path="RIG.In" />
  </inbound>

  <component>
    <no-arguments-entry-point-resolver>
```

```
      <include-entry-point method="nextInt"/>
    </no-arguments-entry-point-resolver>

    <singleton-object class="java.util.Random" />
  </component>
                                                    ❶ Define default-service
  <default-service-exception-strategy/>               exception strategy
</service>
```

The default-service exception strategy configured on ❶ will ensure all exceptions thrown by the `RandomIntegerGenerator` will be handled by the default-service exception strategy. You can also define a default-service exception strategy that all the services in a model use. The configuration is analogous to the global default-connector exception strategy we saw previously. Listing 8.4 illustrates how to accomplish this. We introduce the `SeededRandomIntegegerGenerator` from chapter 6 and explicitly configure a default-service exception strategy that both will share.

Listing 8.4 Configuring the default-service exception strategy on all services

```
<model name="randomGeneratorModel">                 ❶ Define default-service
                                                      exception strategy
    <default-service-exception-strategy/>

    <service name="RandomIntegerGenerator">           Define
      <inbound>                                       RandomIntegerGenerator
        <vm:inbound-endpoint path="RIG.In" />       ❷ service
      </inbound>

      <component>
        <no-arguments-entry-point-resolver>
          <include-entry-point method="nextInt"/>
        </no-arguments-entry-point-resolver>

        <singleton-object class="java.util.Random" />
      </component>                                                Define ❸
    </service>                                SeededRandomIntegerGenerator
                                                                 service
    <service name="SeededRandomIntegerGenerator">
       <inbound>
           <vm:inbound-endpoint path="SRIG.In" />
       </inbound>
       <component>
           <no-arguments-entry-point-resolver>
               <include-entry-point method="nextInt"/>
           </no-arguments-entry-point-resolver>
           <singleton-object class="java.util.Random">
               <property key="seed" value="${seed}" />
           </singleton-object>
       </component>
    </service>
</model>
```

The default-service exception strategy defined on ❶ will now handle all exceptions thrown by the `RandomIntegerGenerator` defined on ❷ and the `SeededRandomIntegerGenerator` defined on ❸.

Let's now see how we can leverage the explicit exception strategy configuration with Mule's outbound routing capabilities. This will enable us to route exceptions to different endpoints for further processing and response.

8.1.2 Exceptions and routing

As we mentioned before, the default exception strategies will simply log exceptions and move on. This is often the right action to take, but sometimes you'll want to take more elaborate measures when an exception occurs. This is especially true in a large or distributed environment. Equally true in such an environment is the inevitability that a remote service will be unavailable. When such a service is the target of an outbound endpoint, you might want Mule to attempt to deliver the message to a different endpoint. We'll consider both scenarios in this section. Let's start by looking at how we can use routers in conjunction with exception strategies, which will allow us to do more than simply log exceptions as they occur. We'll then look at using exception-based routing. This will enable us to send data to alternate endpoints if one is unavailable.[1]

While it's easy to keep an eye on log files on a handful of Mule instances, it becomes increasingly challenging when the number of Mule instances explodes or becomes distributed across a variety of locations. We'll discuss ways to mitigate this later on in this chapter when we discuss the Chainsaw tool, but you might need to distribute errors to someone who can't access or read the log data. Additionally, some errors are so critical that you want to be notified immediately when they occur. This can be difficult to accomplish by parsing log data alone. For situations like these, Mule allows you to route exceptions in much the same way we routed messages in chapter 4. This is accomplished by adding outbound endpoints to an exception strategy. This causes the exception strategy to send the exceptions through the endpoint as a message—in much the same way that a component does. Figure 8.2 illustrates how this works.

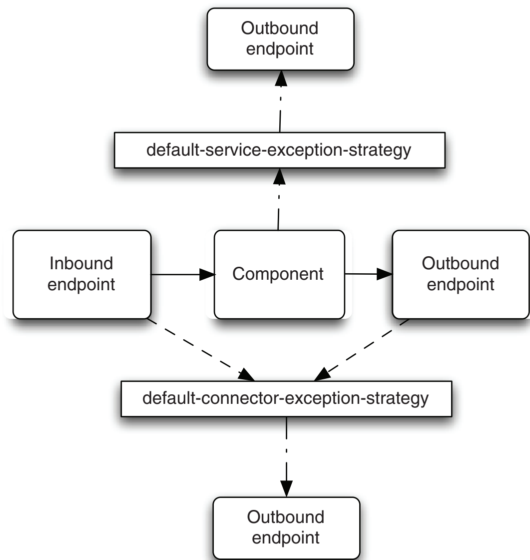


Figure 8.2 Routing exceptions with outbound endpoints

[1] We'll see later on in this chapter how to control logging in Mule.

Now let's see how we can configure Mule to do this. Listing 8.5 modifies listing 8.4 to send service exceptions and connector exceptions to separate JMS queues.

Listing 8.5 Sending all service-level exceptions to a JMS queue

```
<model name="randomGeneratorModel">
    <default-connector-exception-strategy>
       <jms:outbound-endpoint queue="transport-errors"/>        ①
    </default-connector-exception-strategy>

    <default-service-exception-strategy>
       <jms:outbound-endpoint queue="business-errors"/>         ②
    </default-service-exception-strategy>

    <service name="RandomIntegerGenerator">
      <inbound>
        <vm:inbound-endpoint path="RIG.In" />
      </inbound>

      <component>
        <no-arguments-entry-point-resolver>
          <include-entry-point method="nextInt"/>
        </no-arguments-entry-point-resolver>

        <singleton-object class="java.util.Random" />
      </component>
     </service>

    <service name="SeededRandomIntegerGenerator">
       <inbound>
           <vm:inbound-endpoint path="SRIG.In" />
       </inbound>
       <component>
            <no-arguments-entry-point-resolver>
                 <include-entry-point method="nextInt"/>
            </no-arguments-entry-point-resolver>
            <singleton-object class="java.util.Random">
                <property key="seed" value="${seed}" />
            </singleton-object>
       </component>
    </service>
</model>
```

① Send connector exceptions to JMS queue

② Send business exceptions to JMS queue

Define SeededRandimIntegerGenerator service

The JMS outbound endpoint defined on ② will route all exceptions thrown by `RandomIntegerGenerator` and `SeededRandomIntegerGenerator` to the JMS queue named `business-errors`. The outbound endpoint defined on ① will send all connector exceptions to the queue called `transport-errors`. Receivers on these queues can take some sort of action when a particular exception occurs. Let's look at how we can use Jabber to send messages to different parties depending on which queue an exception arrives on.

Listing 8.6 implements two services to handle errors on each of these queues.

Listing 8.6 Handling transport and business exceptions differently

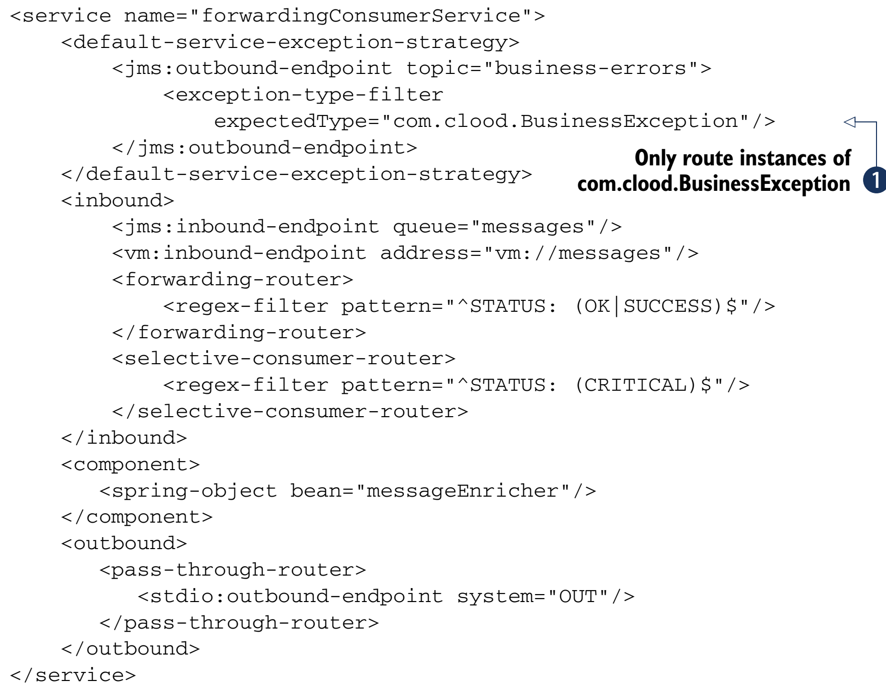
```
<service name="transportErrorService">
    <inbound>
        <jms:inbound-endpoint topic="transport-errors"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <xmpp:outbound-endpoint recipient="ops-on-duty" />
        </pass-through-router>
    </outbound>
</service>

<service name="businessErrorService">
    <inbound>
        <jms:inbound-endpoint topic="business-errors"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <xmpp:outbound-endpoint recipient="engr-on-duty" />
        </pass-through-router>
    </outbound>
</service>
```

- Accept messages off transport-errors queue (at `<jms:inbound-endpoint topic="transport-errors"/>`)
- Send exception contents to ops-on-duty (at `<xmpp:outbound-endpoint recipient="ops-on-duty" />`)
- Accept messages off business-errors queue (at `<jms:inbound-endpoint topic="business-errors"/>`)
- Send exception contents to engr-on-duty (at `<xmpp:outbound-endpoint recipient="engr-on-duty" />`)

The configuration is fairly straightforward. Exceptions that arrive on the `transport-errors` topic will be sent to an XMPP outbound endpoint where they arrive as Jabber-messages to the `ops-on-duty` user. Exceptions that arrive on the `business-errors` queue are delivered to the `engr-on-duty` user. Using topics to dispatch the errors ensures multiple parties can receive the error messages. For instance, there might be another service that subscribes to these errors and sends them as email alerts or forwards them to a logging database. As you saw in chapter 3, subscriptions to topics can also be made durable, ensuring that the message reaches the subscriber.

You might have a situation where you're only interested in routing certain types of exceptions through an outbound endpoint. Let's reconsider listing 4.9 from chapter 4 in the context of our friends at Clood, Inc. If you recall, listing 4.9 illustrated using a forwarding router to bypass component processing for messages containing a certain payload. Messages that didn't contain this payload were sent to the `messageEnricher` component for processing. Let's assume that Clood, Inc., is using this service to correct messages that aren't in an OK or SUCCESS state. If the messages aren't successfully put into the OK or SUCCESS state by the `messageEnricher` service, then an instance of `com.clood.BusinessException` is thrown. Since this is presumably a rare event, we want to send these exceptions to a JMS topic for further processing. We're satisfied with simply logging other types of exceptions. The exception-type filter is useful in cases like this—it'll cause the default exception strategy in question to only route exceptions that match the type in question. Listing 8.7 modifies listing 4.9 to accomplish this.

Listing 8.7 Using the exception-type router



```
<service name="forwardingConsumerService">
    <default-service-exception-strategy>
        <jms:outbound-endpoint topic="business-errors">
            <exception-type-filter
                expectedType="com.clood.BusinessException"/>
        </jms:outbound-endpoint>
    </default-service-exception-strategy>
    <inbound>
        <jms:inbound-endpoint queue="messages"/>
        <vm:inbound-endpoint address="vm://messages"/>
        <forwarding-router>
            <regex-filter pattern="^STATUS: (OK|SUCCESS)$"/>
        </forwarding-router>
        <selective-consumer-router>
            <regex-filter pattern="^STATUS: (CRITICAL)$"/>
        </selective-consumer-router>
    </inbound>
    <component>
       <spring-object bean="messageEnricher"/>
    </component>
    <outbound>
       <pass-through-router>
          <stdio:outbound-endpoint system="OUT"/>
       </pass-through-router>
    </outbound>
</service>
```


By adding the exception-type filter on ❶, we're ensuring that only instances of `com.clood.BusinessException` are being routed to the `business-errors` topic. All other exceptions will be logged by the default-service exception strategy.

Routing exceptions using the default exception strategies is often a good idea, particularly in distributed environments where Mule usually runs. Routing all exceptions to a single JMS queue, for instance, aggregates errors in a central place for later triage, reporting, and management. More complex error handling can be accomplished by extending `org.mule.service.DefaultServiceExceptionStrategy` and overriding the default behaviors. You could, for instance, override the `routeException()` method to change how the exception message is sent. This would allow you to send the original message to the outbound endpoints instead of just the exception payload, serving the basis for a dead-letter queue strategy allowing you to retry failed message delivery at a later date.

BEST PRACTICE Override the default exception strategy routing to facilitate centralized error management and dead-letter queue functionality.

The default exception strategies provide a powerful means for error notification. We'll see in chapter 10 how we can further leverage exception strategies in the context of transactions—this will enable us to perform actions such as rolling back a transaction when an exception occurs. In addition to exception strategies, Mule provides another facility for reacting to errors—exception-based routing. Let's look at that now.

It's often useful in the context of outbound routing to provide multiple endpoints for a service to try in the event one of the endpoints is unavailable. The exception-based router exists for this purpose—it allows you to specify a list of endpoints that'll be tried in sequence until a message is accepted. Figure 8.3 illustrates this.

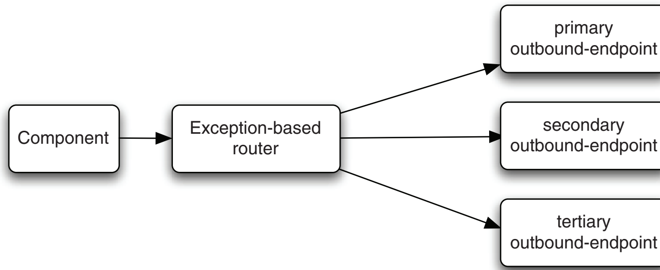


Figure 8.3 Using an exception-based router to send a message in sequence to multiple endpoints, halting when the message is successfully delivered

In the figure, the exception-based router will attempt to send the message to each endpoint in sequence until one succeeds. Let's reconsider listing 3.15 from chapter 3 to see how this works.

In listing 3.15 we were examining how one of Clood, Inc.'s partners might post reporting data to an HTTP inbound endpoint. In that example, Clood supplied the provider with a URL to post data to. Let's assume that Clood, Inc., has a new requirement to accept backup data from globally distributed partners. As such, Clood has made an effort to set up redundant endpoints for this data in its data centers in North America as well as Europe. A backup provider can choose to try the endpoint closest to it first, then fall back on a secondary endpoint in the event the first is unavailable. Listing 8.8 demonstrates how a provider in North America can use exception-based routing to accomplish this.

Listing 8.8 Falling back on multiple endpoints using an exception-based router

```
<model name="httpOutboundModel">
    <service name="httpInboundService">
        <inbound>
            <file:inbound-endpoint path="./data/provider"/>
        </inbound>
        <outbound>
            <exception-based-router>
                <http:outbound-endpoint
                    address=
                    "http://services.nyc.clood.com/backup-reporting"/>
                <http:outbound-endpoint
                    address=
                    "http://services.dub.clood.com/backup-reporting"/>
            </exception-based-router>
        </outbound>
    </service>
</model>
```

In this example, Mule would attempt to post the data from the file inbound endpoint to Clood's data center in New York first. In the event this failed, the exception-based router would attempt to post the data to Clood's data center in Dublin. If this failed as well, an exception would be thrown and handled by the exception strategy configured for the connector.

While the exception-based router is useful in failover scenarios, it might be preferable to retry the same resource repeatedly in the event it's unavailable. Let's look at how we can use retry policies to intelligently try to recover from connector failures.

8.2 Using retry policies

It's an unfortunate reality that services, servers, and remote applications are occasionally unavailable. Thankfully, though, these outages tend to be short-lived. Network routing issues, a sysadmin restarting an application, or a server rebooting all represent common scenarios that typically don't take long to recover from. Nonetheless, such failures can have a drastic impact on applications dependent on them. In order to mitigate such failures, Mule provides a retry policy mechanism to dictate how connectors-deal with failed connections. We'll start off this section by examining Mule's retry policy support. We'll implement a simple retry policy that'll indefinitely attempt to connect to a failed resource. We'll then see how to use this policy with the JMS transport. Finally we'll show how we can use retry policies to allow Mule instances to start independently of remote services they may depend on.

Mule Enterprise Edition ships with a set of retry policies along with an associated XML schema. This saves you the effort of rolling your own retry policies and using the Spring injection of the retry policy template that we'll see in this section. If you're a Mule Enterprise user, you're encouraged to check the relevant documentation for your Mule Enterprise version and use the supplied retry policies and templates. You can use the information in this section to implement your own retry policies and template when the ones supplied by Mule Enterprise aren't sufficient.

8.2.1 Implementing a retry policy

A *retry policy* dictates how a connector should attempt to reconnect to a failed resource. You may want a connector to attempt to reconnect to the resource indefinitely every 5 seconds. In other scenarios you may want to connect to a resource every 2 minutes for 10 times and then stop. In order to define such behavior, you'll need to roll up your sleeves and work with the Mule API. Retry policies are defined by implementing the `RetryPolicy` interface. Listing 8.9 demonstrates a simple retry policy that'll instruct a connector to reconnect to the failed resource every 5 seconds.

Listing 8.9 An indefinitely reconnecting retry policy

```
public class SimpleRetryPolicy implements RetryPolicy {
    public PolicyStatus applyPolicy(Throwable throwable) {
        try {
            Thread.sleep(5000);                   ❶
```

```
        } catch (InterruptedException e) {
            throw new RuntimeException(e);          ②
        }
        return PolicyStatus.policyOk();
    }
}
```

This retry policy will sleep for 5 seconds on ①; if not interrupted it'll return a `PolicyStatus` of OK. This informs the connector to attempt to retry again. If the thread is interrupted, a `RuntimeException` will be thrown on ②. In addition to returning a `policyOK` state, `PolicyStatus` can also return an exhausted state. This is useful if you want to limit the number of retry attempts to a particular resource and is demonstrated in listing 8.10.

Listing 8.10 An exhaustible retry policy

```
public class ExhaustingRetryPolicy implements RetryPolicy {
    private static int RETRY_LIMIT = 5;
    private int retryCounter = 0;

    public PolicyStatus applyPolicy(Throwable throwable) {
        if (retryCounter >= RETRY_LIMIT) {              ①
          return PolicyStatus.policyExhausted(throwable);      ②
        } else {
              try {
                  Thread.sleep(5000);
              } catch (InterruptedException e) {
                  throw new RuntimeException(e);
              }
              retryCounter++;                   ③
              return PolicyStatus.policyOk();
        }
    }
 }
```

The `ExhaustingRetryPolicy` will attempt to connect to the remote resource five times. This is handled by incrementing the `retryCounter` on ③. If the `retryCounter` exceeds the `RETRY_LIMIT` on ①, then a `PolicyStatus` of `exhausted` is returned on ③, along with the cause of the retry. This causes the connector to stop retrying to connect to the failed resource.

Before we can use either of these retry policies, we must implement a *policy template*. This is accomplished by extending the `AbstractPolicyTemplate` class. Listing 8.11 illustrates a policy template for the `ExhaustingRetryPolicy`.

Listing 8.11 An exhaustible retry policy

```
public class ExhaustingRetryPolicyTemplate
               extends AbstractPolicyTemplate {

    public RetryPolicy createRetryInstance() {
        return new ExhaustingRetryPolicy();
    }
}
```

We simply need to implement the `createRetryInstance` method of `AbstractPolicy-Template` to return an instance of the retry policy we wish to use. In this case, we'll return an `ExhaustingRetryPolicy`, which will cause the connector to attempt to connect to the failed resource a specified amount of times before giving up.

Now that we've defined our retry policies, let's see how to configure them for use in a connector.

8.2.2 Using the SimpleRetryPolicy with JMS

An endpoint will become unavailable if the underlying connector fails. Using the JMS transport with an external JMS broker is a good example. If the ActiveMQ instance the connector is configured on goes down, the connector will fail and JMS messages will stop being delivered—even if the ActiveMQ instance recovers. A retry policy enables the JMS connector to reconnect to the ActiveMQ instance. If it can reconnect, endpoints on the connector can begin sending and receiving JMS messages again. This allows your Mule instances to automatically recover from remote service failures without administrative assistance.

Let's return our attention to Clood, Inc., and see how they might use a retry policy. In listing 8.7 we reconsidered how Clood's backup partners might publish reports to globally distributed endpoints using the exception-based router. We discussed the receiving end of this publishing in listing 3.17, where Clood, Inc., publishes the reports to a JMS topic. To continue our example, let's assume Clood, Inc., is sharing an ActiveMQ infrastructure between its New York and Dublin POPs—this ensures reports will be received at both locations when published to a JMS topic. In order to increase resiliency for its JMS connections, Clood uses the `SimpleRetryPolicy` from listing 8.9 to automatically recover from ActiveMQ failures. Listing 8.12 illustrates how they do this.

Listing 8.12 Using the `SimpleRetryConnectionStrategy`

```
<jms:activemq-connector
           name="jmsConnector"
           specification="1.1"
           brokerURL="${jms.url}">
    <spring:property name="retryPolicyTemplate">
        <spring:bean class="SimpleRetryPolicyTemplate"/>
    </spring:property>
</jms:activemq-connector>

<model name="jmsOutboundModel">
    <service name="jmsOutboundService">
        <inbound>
            <http:inbound-endpoint
                 address="http://${http.host}:${http.port}/backup-reports"
                 synchronous="true">
                <byte-array-to-string-transformer/>
            </http:inbound-endpoint>
        </inbound>
        <outbound>
```

(1) Define RetryPolicyTemplate to use

```
            <pass-through-router>
                <jms:outbound-endpoint topic="backup-reports"/>
            </pass-through-router>
        </outbound>
    </service>
</model>
```

We inject the `RetryPolicyTemplate` we wish to use on the ActiveMQ connector on (1). The JMS connector will now invoke the `SimpleRetryPolicy` when its connection to the ActiveMQ broker is interrupted.

In the event of an ActiveMQ failure, the JMS connector will attempt to reconnect to the ActiveMQ instance every 5 seconds for an indefinite amount of time. JMS messaging will fail during this time, but the Mule instance will still be up. Once ActiveMQ recovers, the JMS connector will reconnect to the endpoint and JMS messaging will continue as usual. A side effect of this behavior is that it allows Mule to start up when external dependencies are unavailable. Let's see how to do this.

8.2.3 Starting Mule with failed connectors using the Common Retry Policies

Mule will fail to start if an external dependency, such as a JMS broker or SMTP server, is unavailable. The `SimpleRetryPolicy` we implemented earlier doesn't handle this situation well—it'll block indefinitely until the remote resource becomes available, prohibiting Mule from starting up. Having the retry attempts occur in their own thread is one way to circumvent this issue. Such functionality is available to Mule EE users but is currently not available in the community Mule 2 release. Fortunately a MuleForge project exists to fill this gap—the Common Retry Policies

The Common Retry Policies Mule module is available on MuleForge at this address: http://www.mulesource.org/display/COMMONRETRYPOLICIES/Home. In addition to containing an implementation of the `SimpleRetryPolicy` we saw previously, the Common Retry Policies package also provides a multithreaded retry policy template. This template, called the *adaptive retry policy*, will cause retry attempts to occur in a thread separate from the main Mule thread when Mule is starting up. This allows Mule to start even if some connectors are unavailable. You'll need to download the JAR file from the Common Retry Policies home and make it available to your Mule installation prior to being able to use the policies.[2]

Let's see how we can have listing 8.11 start up when the JMS broker is unavailable. Listing 8.13 illustrates using the adaptive retry policy to accomplish this.

Listing 8.13 Allowing Mule to start when a connector is failed

```
<spring:beans>
    <spring:bean id="foreverRetryPolicyTemplate"
        class=
"org.mule.modules.common.retry.policies.ForeverRetryPolicyTemplate"/>
```

(1) Define retry policy template

[2] The Common Retry Policies is a MuleForge project maintained by the community. It's not part of the mainstream Mule distribution or maintained by MuleSource.

```
    <spring:bean id="threadingPolicyTemplate"
            class=
"org.mule.modules.common.retry.policies.AdaptiveRetryPolicyTemplateWrapper">
       <spring:property name="delegate" ref="foreverRetryPolicyTemplate"/>
    </spring:bean>
 </spring:beans>
                                                   Define the adaptive
                                                   retry policy template 2
  <jms:activemq-connector
          name="jmsConnector"
          specification="1.1"
          brokerURL="${jms.url}">
       <spring:property name="retryPolicyTemplate"
              ref="threadingPolicyTemplate"/>
                                                   3 Inject template
                                                     into ActiveMQ
                                                     connector
  </jms:activemq-connector>

  <model name="jmsThreadingRetryModel">
      <service name="jmsThreadingRetryService">
          <inbound>
              <http:inbound-endpoint
                   address="http://${http.host}:${http.port}/backup-reports"
                 synchronous="true">
                   <byte-array-to-string-transformer/>
              </http:inbound-endpoint>
          </inbound>
          <outbound>
              <pass-through-router>
                  <jms:outbound-endpoint topic="backup-reports"/>
              </pass-through-router>
          </outbound>
    </service>
 </model>
```

We start off by defining the retry policy we want to use on ❶. In this case, we're going to use the `ForeverRetryPolicyTemplate` supplied by the Common Retry Policies. The `ForeverRetryPolicyTemplate` behaves much like the `SimpleRetryPolicy` we looked at before. It'll attempt to reconnect to a failed connector every 5 seconds. The `AdaptiveRetryPolicyTemplateWrapper` defined on ❷ is what enables the `Forever-RetryPolicy` behavior to occur in its own thread when Mule starts. We finally inject the `threadingPolicyTemplate` into the ActiveMQ connector on ❸. If the JMS broker is down when Mule is started, the `AdaptiveRetryPolicyTemplateWrapper` will use the `ForeverRetryPolicyTemplate` to attempt to reconnect to the broker independently of Mule bootstrapping. This will allow other services that aren't dependent on that JMS broker to start and behave normally.

In this section we saw how connection strategies enable us to tolerate failures in remote services without restarting Mule. We saw how we can automatically reconnect to failed services as well as start up Mule when remote dependencies are unavailable. Let's turn our attention now to Mule's logging facilities.

8.3 Logging with Mule

We've spent a lot of time discussing different ways to deal with errors that crop up in Mule deployments. One of the most common ways you'll deal with errors and other diagnostic events is by logging them. Mule uses the Apache Commons Logging package along with the SLF4J logging facade (`http://www.slf4j.org/`) to allow you to plug and play logging facilities. By default, Mule will use the popular log4j logging library without any intervention on your part. We'll see later on how we can change this behavior to allow Mule to use other logging implementations, such as `java.util.logging`. Let's start off with a look at how logging works in a freshly installed, standalone Mule instance.

8.3.1 Using log4j with Mule

Mule uses log4j as its default logging implementation. Log4j is a robust logging facility that's commonly used in many Java applications. Full documentation for using log4j is available on the project's web site at `http://logging.apache.org/log4j/`. Mule provides a default log4j configuration in the `$MULE_HOME/conf/log4j.properties` file. Let's look at this file now; listing 8.14 shows the default `log4j.properties` file.

Listing 8.14 The default `log4j.properties` file

```
#Default log level
log4j.rootCategory=INFO, console                    <-- 1 Specify default log level

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-5p %d [%t] %c: %m%n   <-- 2 Specify logging format

#################################################
# You can set custom log levels per-package here
#################################################

# Apache Commons tend to make a lot of noise which can clutter the log.
log4j.logger.org.apache=WARN                 <-- 3 Specify logging level of org.apache package

#Mule classes
log4j.logger.org.mule=INFO                   <-- 4 Specify logging level of org.mule

# Your custom classes
log4j.logger.com.mycompany=DEBUG             <-- 5 Specify logging level for your packages
```

If you've worked with log4j at all before, this should seem familiar to you. Log4j supports the concept of log levels for packages. The log levels available are DEBUG, INFO, WARN, ERROR, and FAIL, in ascending order of severity. The default log level is specified on 1, which is logging messages of level INFO to the console. The format of the log output is specified on 2. You can tweak this to customize how logging is output (see the log4j documentation for more information on how to do this). The log definitions on a per-package basis start on 3, where the libraries for the `org.apache` project (which are used extensively by Mule and Spring) are set at a logging level of WARN. The logging level for the Mule packages is specified next on 4. The default level is INFO, but you'll soon find it's convenient to set this to DEBUG for troubleshooting and

general insight into how Mule is behaving. Finally, you can change `com.mycompany` to your company's package prefix in order to set the debugging level for your custom components, transformers, routers, and so forth. For instance, in order to set DEBUG logging for Clood, Inc.'s custom classes we'd change ❺ to this:

```
log4j.logger.com.clood=DEBUG
```

By default, Mule will write to a log file called `mule.log`. This file is located in `$MULE_HOME/logs`. You can change this location by editing the `wrapper.logfile` variable in `$MULE_HOME/conf/wrapper.conf`. Mule will write 1 megabyte of data to the `mule.log` file before automatically rotating it. It'll archive up to 10 rotations with the stock configuration. This behavior is configured in `wrapper.conf` as well, by tuning the `wrapper.logfile.maxsize` and `wrapper.logfile.maxfiles` variables.

NOTE *Getting diagnostics from log4j* You occasionally need to gather debugging information from log4j itself. This can be accomplished by setting the `log4j.debug` property. If you're launching Mule in the standalone fashion discussed in chapter 7, this can be accomplished by appending the string `-M-Dlog4j.debug` at the end of the command to launch Mule. The following code shows how to do this:

```
mule -config my-config.xml -M-Dlog4j.debug
```

It's also possible to specify an alternative `log4j.properties` file by specifying the URL to the file. The following illustrates how to do this.

```
mule -config my-config.xml
-M-Dlog4j.configuration=file:///path/conf/log4j.properties
```

Now that you've seen how to configure log4j and SLF4J, let's look at how we can use Chainsaw to make it easier to view and analyze log entries.

NOTE You might want to use a logger other then log4j with Mule. This is an easy task provided you're using a logging implementation supported by SLF4J. SLF4J supports JDK 1.4 logging, Logback, and the Apache Commons Logging project. Simply download the SLF4J implementation for your version of Mule, remove the existing SLF4J bridge and place the appropriate SLF4J bridge in `$MULE_HOME/lib/boot`. On your next restart, Mule should log using the new implementation.

8.3.2 Using Apache Chainsaw with log4j

Dealing with text logs can be difficult at times. Even in a Unix-like environment, where a plethora of excellent text manipulation tools are available, it's occasionally useful to have a way to graphically look at and compare log data. This is especially true when log data must be analyzed from multiple sources, a common scenario when dealing with distributed environments and applications. Apache Chainsaw is an attempt to provide such a tool for log4j. In this section we'll investigate how to use Apache Chainsaw to view log data from Mule.[3]

[3] You must be using log4j as your SLF4J logging implementation to use Apache Chainsaw.

Let's start by installing Apache Chainsaw. You can download Chainsaw from the project's web site here: http://logging.apache.org/chainsaw. There are three options for installation: running as a Java Webstart application, installing as on OS/X application, or running from the command line. The GUI is functionally identical between the three, so pick the means most convenient for you and start Chainsaw up. You should see a screen that looks something like figure 8.4.

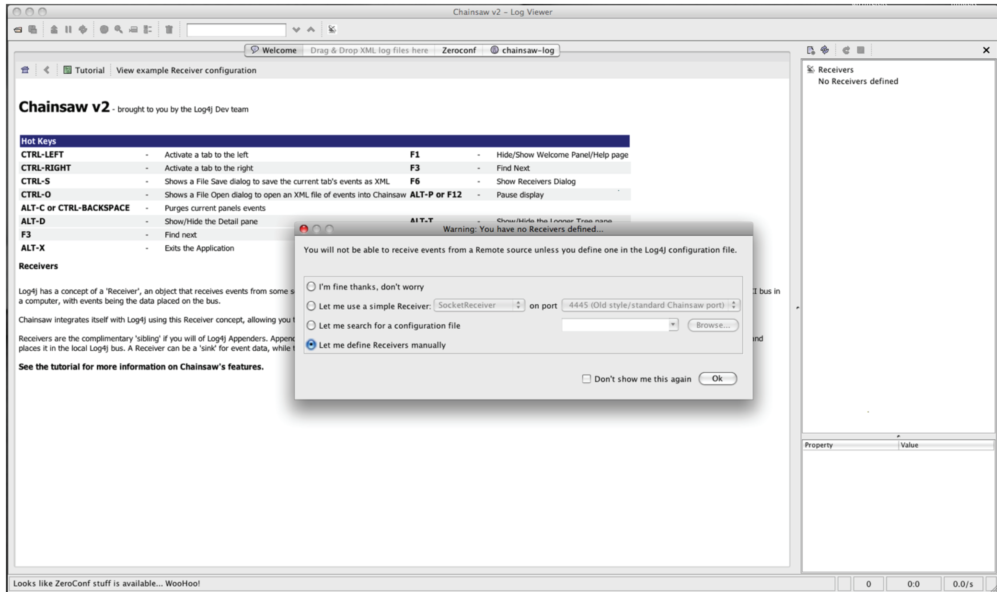


Figure 8.4 Running Chainsaw for the first time

For now, select *Let me define Receivers manually* and then click OK to continue. You should now be on the Welcome tab. Feel free to click around and explore the UI, and then we'll investigate how we can modify Mule's default logging configuration in order to see logs in Chainsaw.

Chainsaw uses log4j's receiver framework to accept remote logging events. One such receiver it supports is the `SocketHubAppender`. The `SocketHubAppender` is enabled in the log4j configuration to listen on a socket and publish logging events to connected clients. Listing 8.15 demonstrates how to modify the default `log4j.properties` file in `$MULE_HOME/conf` to get the `SocketHubAppender` going.

Listing 8.15 Configure a `SocketHubAppender` in the default `log4j.properties`

```
# Default log level
log4j.rootCategory=INFO, console, sockethub

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%-5p %d [%t] %c: %m%n
```

Send logs to SocketHubAppender

```
log4j.appender.sockethub=org.apache.log4j.net.SocketHubAppender
log4j.appender.sockethub.port=9999

###############################################
# You can set custom log levels per-package here
###############################################

# Apache Commons tend to make a lot of noise which can clutter the log.
log4j.logger.org.apache=WARN

# Mule classes
log4j.logger.org.mule=INFO

# Your custom classes
log4j.logger.com.mycompany=DEBUG
```

Define SocketHubAppender

Set which port SocketHubAppender will listen on

This configuration will set up a `SocketHubAppender` to listen for connections on port 9999. Clients connected to this socket will receive logging data from log4j. When you start Mule you should be able to connect to this port, using a tool such as netcat or telnet, and you'll see logging data sent to the socket. Let's now see how to get Chainsaw to connect to this port and receive logging events from Mule.

To receive logging events in Chainsaw we'll need to configure it to connect to the `SocketHubAppender` we just configured. The receiver is configured in the right panel of the Chainsaw GUI. You can configure a new receiver by clicking on the New Receiver button in the upper left side of the panel. The screenshot in figure 8.5 illustrates this.

After you click on this, you'll be presented with options to configure the receiver. We're going to set the host to `localhost`, the name to `Mule-1`, and the port to `9999`. Once this is done, click the Refresh button and you should see the Mule-1 receiver appear in the right panel along with a localhost tab on the top of the middle panel. This should look something like figure 8.6.

Assuming your Mule instance is generating logging events, you should start seeing these appear in the `localhost` tab. As you can see, you can now filter through the logging events by ID, timestamp, level, logger, message, and thread. The panel in the bottom center of the screen displays the full content of the logged message. On the right panel is a tree you can use to filter messages based on package hierarchy. This allows you to ignore or focus on messages logged by classes in a certain package. You can even change the color of log entries based on package or severity.

NOTE *The log component* You saw in chapter 6 how you can use the log component to send message payloads to the Mule log. We'll see further use of this component in chapter 11 when we begin talking about logging in the context of monitoring Mule instances.

The real value of a tool such as Chainsaw comes into play when you're dealing with several deployed Mule instances. Manually watching the log files on 10 Mule instances, for instance, quickly becomes unmanageable. With Chainsaw, you can set up a `SocketHubAppender` on each instance and monitor all 10 from a single window

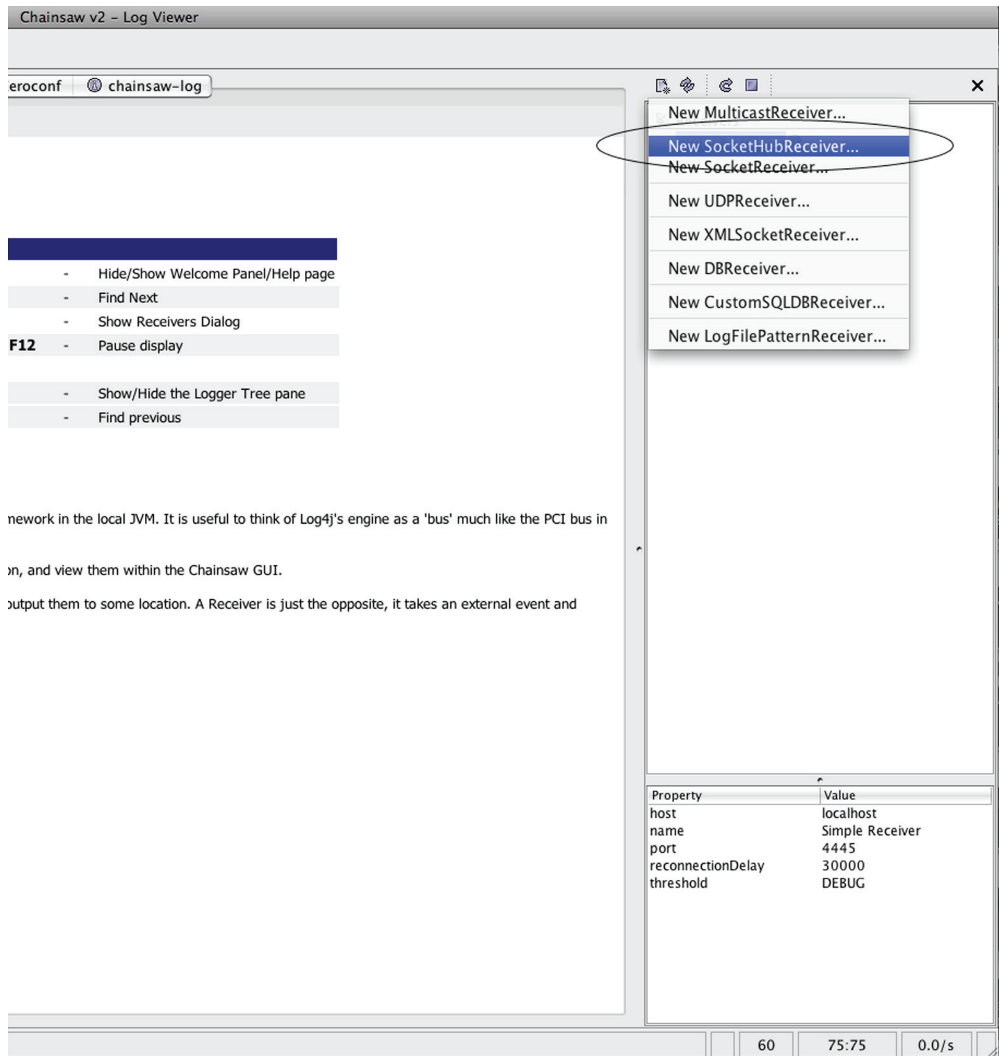


Figure 8.5 Creating a new `SocketHubReceiver`

on your desktop. Even with a small number of Mule instances, it's easy to see how this can be useful. You'll see more uses of Chainsaw in chapter 11, where we discuss how it can be useful in the context of monitoring Mule instances.

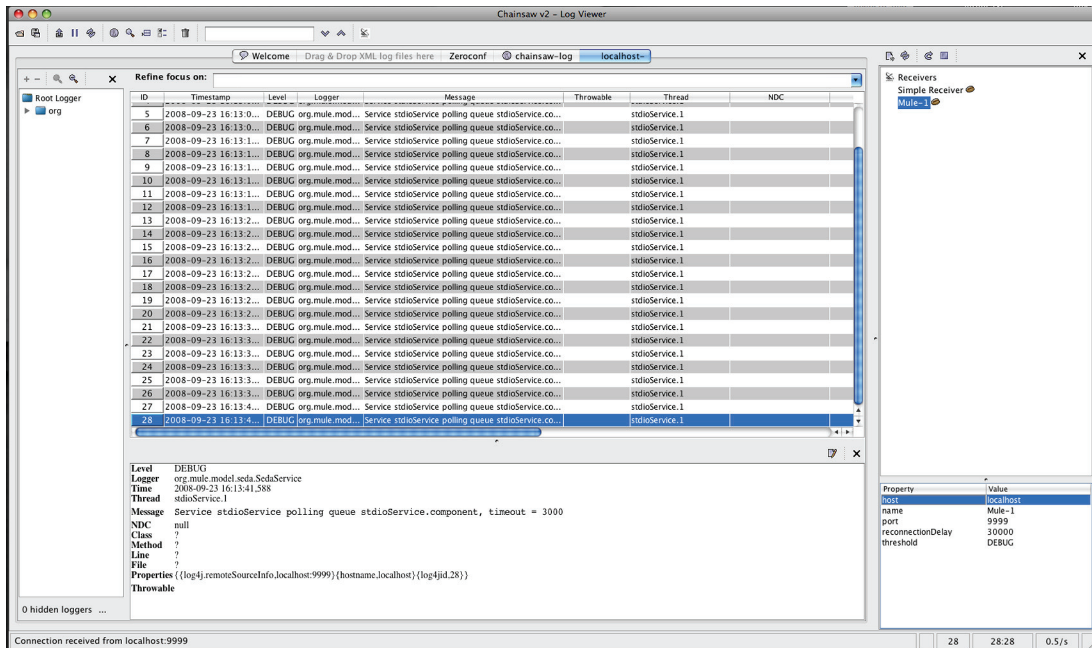


Figure 8.6 Connect Chainsaw to `SocketHubAppender` to receive Mule logging events.

8.4 Summary

In this chapter we investigated Mule's error handling capabilities. We saw how to use exception strategies to define how errors are handled on the connector and service levels. We then saw how to leverage exception strategies with outbound routing to manage what happens after an exception occurs. In the event of an endpoint failure, you saw how exception-based routing enables you to send the same message to multiple endpoints, trying each in succession. We turned our attention then to logging, investigating how Mule uses the SLF4J logging facade to support multiple logging implementations. We looked at how to configure Mule's default logger, log4j, as well as replace it with JDK 1.4 logging. Finally, we saw how to use Chainsaw as a graphical front end to Mule's logging data.

In the next chapters we'll be discussing other aspects of running Mule. We'll continue an aspect crucial to Mule deployments: security.

ENTERPRISE JAVA

Mule IN ACTION

David Dossot • John D'Emic

Foreword by Ross Mason, Creator of Mule

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

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

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

What's Inside

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

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

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

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

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

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

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

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

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

MANNING $44.99 / Can $56.99 [INCLUDING eBOOK]