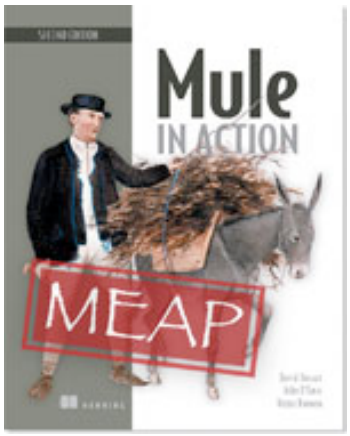
MANNING PUBLICATIONS



Mule in Action, Second Edition
By David Dossot and John D'Emic

Ideally, your Mule applications should exhibit the properties of any other well-architected software application: it should be easy to refactor and test, modular, and decoupled. It should also make some attempt to foresee future needs, making it easy to modify as requirements inevitably change. Mule can function anywhere from a one of integration applications to the central integration fabric for a Fortune 50 enterprise. Architecture of these applications, whether implicit or explicit, will impact their success.In this article based on chapter 7 of Mule in Action, Second Edition, the authors look at some patterns that simplify the implementation of these architectures.

You may also be interested in…

Mule Implementation Patterns

In this article, we'll take a look at patterns that simplify the composition of integration applications. These aren't explicit features of Mule but rather approaches that take advantage of Mule's functionality to ease integration application development. We'll see how adopting a canonical domain model enables you to decouple your business logic from your integration operations, like routing and transformation. We'll then take a look at how to use asynchronous messaging to build resiliency into your Mule flows.

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 into which Mule transforms message payloads prior to any further processing. Canonical data models are typically Java domain objects, an XML schema, or an agreed upon JSON format.

Let's consider one of our sample company 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 if necessary or augment the CRM functionality with another application.

A canonical domain model avoids this coupling. Listing 1 shows the `Customer` domain object that is part of the canonical model introduced by Prancing Donkey.

Listing 1 The customer canonical data model

```
mlRootElement
@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.

Listing 2 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>                                                    #1
           <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"/>          #2
</flow>
```

#1 Use the Mule Expression Language to create the SalesForce contact object
#2 A custom transformer is used to convert the response from the SalesForce connector, a Map, to an instance of Customer.

You can see that the create flow uses Mule Expression Language (MEL) to populate the appropriate fields in the SalesForce contact object on #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 article uses a POJO data model, but other domain models like JSON and XML are just 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 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 we can leverage asynchronous message to reliably deliver messages with Mule.

Reliability patterns with asynchronous messaging

Asynchronous messaging providers, like 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," 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 into 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 that the order has been 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 to submit 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 on to track the status of the order. The front-end part of this flow is below.

Listing 3 Front-end order submission

```
<flow name="orderSubmission">
    <http:inbound-endpoint exchange-pattern="request-response"
                           host="localhost" port="8081"
                           path="order"
            />
    <cxf:jaxws-service
 serviceClass="com.prancingdonkey.service.OrderSubmissionService"
                    />
    <component
class="com.prancingdonkey.service.OrderSubmissionServiceImpl"/>
    <jms:outbound-endpoint queue="order.submit">                        #1
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:outbound-endpoint>
</flow>
```

#1 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 Prancing Donkey's JMS infrastructure is robust and the backend 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 backend of the flow.

Listing 4 Backend order processing

```
<flow name="orderProcessing">
    <jms:inbound-endpoint queue="order.submit">

        <jms:transaction action="ALWAYS_BEGIN"/>                        #1
    </jms:inbound-endpoint>
    <all>
        <jms:outbound-endpoint queue="crm.customer.create">

            <jms:transaction action="ALWAYS_JOIN"/>                     #2
        </jms:outbound-endpoint>
        <jms:outbound-endpoint queue="erp.order.record">

            <jms:transaction action="ALWAYS_JOIN"/>                     #3
        </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"/>                      #4
    <jms:outbound-endpoint topic="events.orders.completed">             #5
        <jms:transaction action="ALWAYS_BEGIN"/>
    </jms:outbound-endpoint>
</flow>
```

#1 Transactionally accept the Offer off a JMS queue
#2 Join in the previous transaction and submit to the crm.customer.create queue
#3 Join in the previous transaction and submit to the erp.order.record queue
#4 Wait and aggregate responses from SalesForce and Netsuite
#5 Dispatch order completion events to the events.orders.completed JMS topic

These two flows handle the asynchronous routing and aggregation of backend order processing. The "orderProcessing" flow accepts a message and submits it to two queues in a single transaction. The crm.customer.create flow submits the Order object to a SalesForce flow similar to the one we saw above. The erp.order.record flow submits the Order object to a flow that uses Mule's NetSuite Cloud connector.

Prancing Donkey is using ActiveMQ's message redelivery feature to redelivery messages up to a certain point in the event of a transactional rollback. In this case if there is 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 servies, like 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" flow. Mule's collection-aggregator will wait for both responses for an order to arrive before dispatching the Order on the "events.order.completed."

Prancing Donkey uses JMS topics to generate events when certain business phenomena occur.

Service hosting with Mule

Web service hosting has typically been the domain of dedicated application or web servers like JBoss AS or Tomcat. Both platforms provide varying degrees of support for the JEE stack and multi-tenancy. For web services, particularly those built on top of JAX-RS and JAX-WS, they were natural choices as host platforms.

While such applications could be hosted on Mule, this was typically a difficult affair—particularly since the standalone server could only host one application a time and application deploys required a restart of the server.

Things have changed in Mule 3. Mule fully supports the JAX-RS and JAX-WS specifications via Jersey and Apache CXF. The standalone server is now fully multi-tenant. 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.

Summary

Using Mule as a decoupling middleware platform, as we saw in this article, lets you leverage reliable messaging to make otherwise transient, fatal errors in a non-reliable transport recoverable.

Mule's agnosticism to both the payload of message and the architecture of integration applications make it easy to implement patterns like the canonical data model and decoupling middleware. It also makes it possible to leverage technologies complementary to the features supported directly by Mule.

Here are some other Manning titles you might be interested in:

Spring in Action, Third Edition
Craig Walls

Spring Batch in Action
Thierry Templier, Arnaud Cogoluegnes, Gary Gregory, and Olivier Bazoud

Spring Integration in Action
Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld

Last updated: December 18, 2012