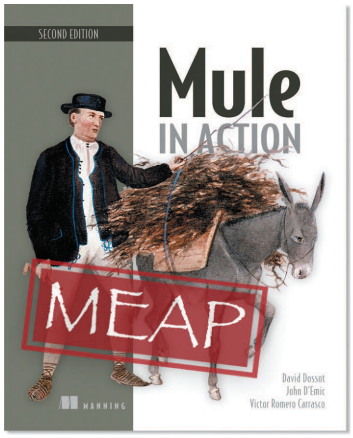
MANNING PUBLICATIONS



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

Mule is a workhorse whose sole purpose in life is to move your messages around. It actually does way more than just moving messages but is also able to transform, enrich, and smartly route them. In this article based on chapter 2 of Mule in Action, Second Edition, the authors look at details of message interactions Mule supports and the set of abstractions that enables these interactions.

Interacting with Messages

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

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

To support these interactions, Mule relies on a set of core abstractions that act as a foundation for all the other aspects of message processing. In this article, we will detail the following abstractions whose understanding is fundamental to a successful usage of Mule:

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

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

Message sources

In a Mule configuration, message sources usually manifest themselves as inbound endpoints. It's as simple as that.

Only a single message source is allowed in a flow. To allow multiple inbound endpoints to feed messages in the same flow, a composite message source must be used. As you've guessed it, this composite message source is also a message source! Listing 1 shows a few valid message sources.

Listing 1 Simple and composite message sources

```
<vm:inbound-endpoint path="payment-processor" />

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

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

Message processors

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

Let's list the main features provided by message processors:

- As *outbound endpoints*, they take care of dispatching messages to whatever destination you want.
- As *transformers*, they are able to modify messages.
- As *routers*, they ensure messages are distributed to the right destinations.
- As *components*, they perform business operations on the messages.

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

Shoehorn message processors with a chain

When only a single message processor is allowed in a configuration spot, use a `processor-chain` element, which is a message processor that can encapsulate several message processors that will be called one after the other (as if they were chained).

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

The next abstraction we're about to explore deals with the time aspect of message interactions.

Message exchange patterns

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

Currently, Mule supports only two MEPs:

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

Both MEPs can be used on inbound and outbound endpoints but some transports can apply restrictions to the MEPs they actually support (for example, POP3 is one-way only). On inbound endpoints, one-way means Mule will

not return a response to the caller, while request-response means it will. On outbound endpoints, one-way means Mule will not wait for a response from the callee, disregarding one if there is, while request-response means it will.

Say something!

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

Let's illustrate this with an example. Look at listing 2 two see two different HTTP endpoints, each configured with a different MEP (configured by the exchange-pattern attribute). The inbound endpoint uses the request-response MEP, thus it will provide synchronous responses to clients that send HTTP requests to it.

On the other hand, the outbound endpoint is one-way, which means that Mule (acting as client in this case) will not wait for any response from the remote server, whether there is one or not.

Listing 2 HTTP endpoints with different exchange patterns

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

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

To illustrate this, let's take a look at the configuration in listing 3.

Listing 3 A simple flow where execution doesn't happen as expected

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

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

To answer this question, let's look at the log file recorded while a message goes through the flow of listing 3:

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

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

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

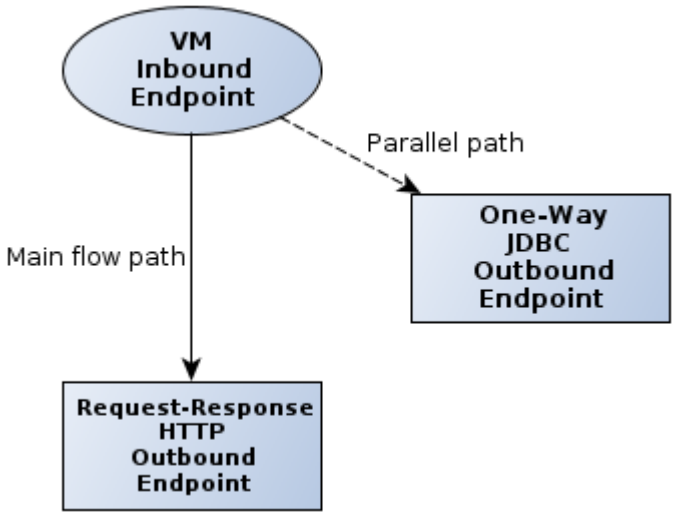


Figure 1 Message exchange patterns influence a flow's threading model.

Figure 1 illustrates how a one-way MEP initiates a parallel processing of a message interaction.

No guessing game

Make your flows predictable: don't rely on transport defaults and use explicit MEPs on your endpoints.

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

Endpoint URIs

Mule uses uniform resource identifiers (URIs) as the unified representation for all the resources it exposes or accesses via its endpoints. In other words, inbound and outbound endpoints are internally configured by a URI. Externally, in the XML configuration, they are configured by specific elements with transport specific attributes: behind the scene, all this information is folded into an endpoint URI.

Don't skip this part thinking that you will never have to deal with endpoint URIs since they're internal. The Mule client uses endpoint URIs to interact with Mule or through Mule.

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

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

Listing 4 shows a Mule configuration exposing resources at the URIs.

Listing 4 Mule endpoints exposing different types of resources

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

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

Summary

We're done with our tour of Mule message processing capacities. We've learned about controlling message paths with flows. We've also discovered the abstractions that sit at Mule's core.

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: March 25, 2012