Accelerating Integration with SaaS, Social Media, and Open APIs

Getting Started with

Mule Cloud Connect

O'REILLY®

Ryan Carter

Getting Started with Mule Cloud Connect
by Ryan Carter

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Andy Oram and Mike Hendrickson
Production Editor: Kara Ebrahim
Proofreader: Kara Ebrahim
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Kara Ebrahim

Revision History for the First Edition:
2012-12-19 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449331009* for release details.

ISBN: 978-1-449-33100-9

[LSI]

1356358928

CHAPTER 1
Getting Started

It all starts with a simple API that publishes someone's status to Facebook, sends a Tweet, or updates a contact in Salesforce. As you start to integrate more and more of these external services with your applications, trying to identify the tasks that one might want to perform when you're surrounded by SOAP, REST, JSON, XML, GETs, PUTs, POSTs, and DELETEs, can be a real challenge.

Open APIs are all about endpoints. Most services follow the current trend of providing a RESTful endpoint, others use older RPC-based protocols such as SOAP or XML-RPC, some use newer "real-time", push-focused endpoints like WebSockets or HTTP Streaming, others may offer a number of different endpoints to meet different requirements, and some just use what seems to be best for a specific job, which might mean not strictly following protocol rules. This is one of the biggest challenges with open APIs: inconsistency. Figure 1-1 shows the estimated popularity of different styles of APIs.

Each API is different, with different data formats and authorization mechanisms. One API's interpretation of REST may even differ from another. One reason for this is the nature of REST itself. The RESTful principles come from a paper published by Roy Fielding in 2000 and since then RESTful services have dominated SOAP-based services on the web year after year. Although REST services have many advantages over SOAP-based services, the original paper only included a set of constraints and provides no specification about how to define a RESTful API and handle things like URI schemes, authentication, error handling, and more.

By observing the vastly different opinions out there, there is no one right way to define a RESTful API, which has resulted in many inconsistencies, even between APIs from the same service provider. Top that off with the remaining SOAP services and newer technologies such as HTTP Streaming and you're left with a lot of different API styles and protocols to learn. Working with all these APIs can just be too damn hard, and this is where Mule Cloud Connect comes in. Mule Cloud Connect is a powerful, lightweight toolset providing a consistent interface to a large number of cloud, SaaS, social media, and Web 2.0 APIs.

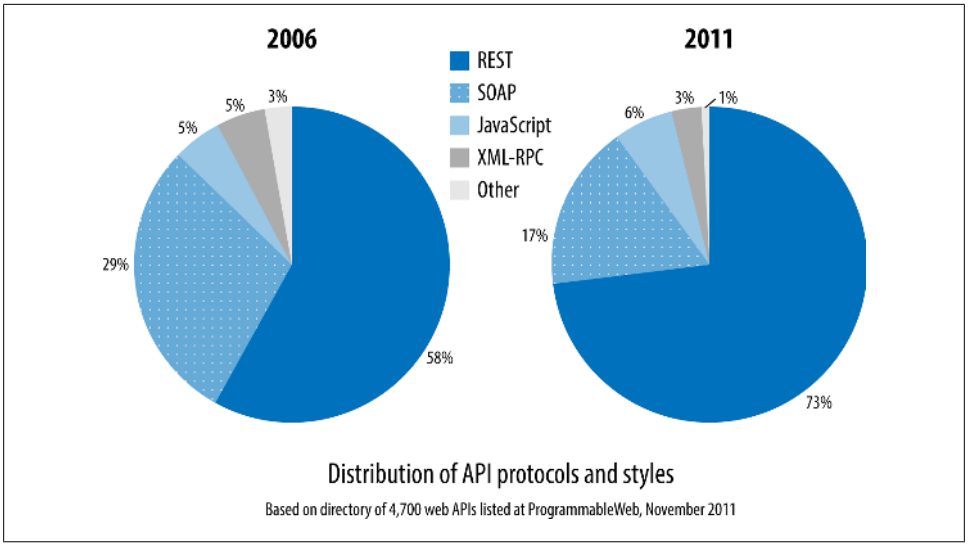


Figure 1-1. Distribution of API protocols

Cloud Connectors versus the REST of the World

There are many different levels of working with APIs. To put Cloud Connectors into context, let's first look at some other approaches to integrating APIs.

To demonstrate, we will use the GeoNames API as our external service. I tend to use GeoNames as the API equivalent of the Northwind database, because it's easy to consume (providing both XML and JSON formats) and does not require any account setup for demo purposes.

GeoNames is a worldwide geographical database that contains over 10 million geographical names and consists of 7.5 million unique features, of which 2.8 million are populated places and 5.5 million are alternate names. All features are categorized into one out of nine feature classes and further subcategorized into one out of 645 feature codes. In addition to listing names of places in various languages, data stored by GeoNames includes latitude, longitude, elevation, population, administrative subdivision, and postal codes. GeoNames features include direct and reverse geocoding, finding places through postal codes, finding places next to a given place, and finding Wikipedia articles about neighboring places.

Transport-Specific Clients

Transport-specific clients deal directly with APIs over the wire. These clients deal with the actual bytes that pass between your application and the external API. For a RESTful service, it requires you to build a URL and associate it with the correct URI parameters

and HTTP headers. For a SOAP-based service, it requires you to build the contents of the HTTP POST yourself, including the `SOAP:Envelope` and any `WS-*` content. Example 1-1 shows a very simple Java snippet for constructing a simple client for a RESTful service using Java's HTTP packages.

Example 1-1. RESTful Java client with java.net URL

```
URL url = new URL("http://api.geonames.org/findNearbyJSON" +
                  "?lat=37.51&amp;lng=-122.18&amp;username=demo");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setRequestProperty("Accept", "application/json");

if (conn.getResponseCode() != 200) { throw new RuntimeException("Failed :
                                     HTTP error code : " + conn.getResponseCode());
}
BufferedReader br = new BufferedReader(new InputStreamReader(
                    (conn.getInputStream())));

String output;
System.out.println("Output from Server .... \n");
while ((output = br.readLine()) != null) { System.out.println(output);
}

conn.disconnect();
```

This is the most abstract way of working with APIs. The semantics of HTTP libraries match the HTTP protocol and not REST or SOAP APIs specifically. This leaves it up to you to construct URLs, build up request structures, and write them to and from input and output streams, requiring you to know the API very well.

When you start working with more complex APIs that require connection or state management, you're left to do this manually, which is error prone and requires far more effort to handle reliably.

Language-Specific Clients

Language-specific libraries, such as Jersey clients for Rest APIs or Apache CXF for SOAP APIs, wrap the underlying protocols in methods that are more familiar and comfortable for programmers in that language. For example, Example 1-2 shows a very simple code snippet for using Jersey to invoke RESTful service.

Example 1-2. Jersey REST client

```
WebResource webResource = client.resource("http://api.geonames.org/findNearbyJSON");
MultivaluedMap queryParams = new MultivaluedMapImpl();
queryParams.add("lat", "lat");
queryParams.add("lng", "-122.18");
queryParams.add("username", "demo");
String s = webResource.queryParams(queryParams).get(String.class);
```

Using this example, the Jersey client libraries abstract away a lot of the HTTP specifics and make API clients a lot clearer by providing short code that helps express the semantics of the particular API protocol. This is one advantage over using transports, but you're still left importing WSDLs for SOAP services and object binding to and from request structures. If you're using multiple protocols, you may have to learn and maintain multiple libraries. Because they are generic and not specific to any particular API, you will still have to write custom code to work with each API's little idiosyncrasies or custom features such as session-based authentication and OAuth.

Service-Specific Client Libraries

A client library specifically developed for a particular API, such as Twitter4j for the Twitter APIs, makes things easier by extracting away a lot of the protocol and transport specifics. Example 1-3 shows an example of working with GeoNames' Java library.

Example 1-3. Service-specific client library

```
WebService.setUserName("demo");
ToponymSearchCriteria searchCriteria = new ToponymSearchCriteria();
searchCriteria.setQ("zurich");
ToponymSearchResult searchResult = WebService.search(searchCriteria);
for (Toponym toponym : searchResult.getToponyms()) {
   System.out.println(toponym.getName()+" "+ toponym.getCountryName());
}
```

Convenient as these are, because they fit the semantics of the service closely, they are typically developed by the individual service providers or developer communities. Therefore, there is no consistency between implementations.

Cloud Connectors

Mule Cloud Connect offers a more maintainable way to work with APIs. Built on top of the Mule and CloudHub integration platforms, Cloud Connectors are service-specific clients that abstract away the complexities of transports and protocols. Many complex but common processes such as authorization and session management work without you having to write a single line of code. Although service-specific, Cloud Connectors all share a common and consistent interface to configure typical API tasks such as OAuth, WebHooks, and connection management. They remove the pain from working with multiple, individual client libraries. Example 1-4 shows a really basic example of configuring a Cloud Connector to access the GeoNames API, which will be covered in more detail shortly.

Example 1-4. Cloud Connector configuration

```
<geonames:config username="demo" />

<geonames:find-nearby-pois-osm latitude="37.451"
            longitude="-127" />
```

Cloud Connectors are essentially plain old Java objects (POJOs) developed by Mule and the community using the Cloud Connect SDK called the DevKit. The DevKit is the successor to the original Cloud Connect SDK, which was developed with just external APIs in mind but has since been opened up to create any manner of Mule extension such as transformers or pretty much anything. The DevKit uses annotations that mimic typical integration tasks to simplify development, and when processed, are converted into fully featured components for the Mule ESB and CloudHub integration platforms.

Mule Cloud Connect supports many of the most widely-used open APIs from SaaS to social media, with more being developed every day. Current Connectors include Twitter, Facebook, LinkedIn, Salesforce, Amazon WebServices, Twillio, and many more. A full categorized list of available connectors and what they offer can be found here.

Mule: A Primer

Before diving straight into configuring Cloud Connectors, it's important to understand some basic concepts. After this short overview, you'll be ready to build your first application and start taking advantage of Mule Cloud Connectors. To begin, we will first build a simple Mule application that we can use as the base of our examples and introduce some core concepts for those unfamiliar with Mule.

As mentioned previously, Mule is an integration platform that allows developers to connect applications together quickly and easily, enabling them to exchange data regardless of the different technologies that the applications use. It is also at the core of CloudHub, an Integration Platform as a Service (IPaaS). CloudHub allows you to integrate cross-cloud services, create new APIs on top of existing data sources, and integrate on-premise applications with cloud services.

Later in the book we will look at specific connectors, but to start let's take a look at a simple API proxy that can be used to mediate an external service and introduce some transformation and some routing between the two. This application will expose a simple RESTful interface that can be invoked through a browser or HTTP client, contact an external service, and transform the returned response to the browser.

Mule Configuration

XML is the format for the files that control Mule, and it uses schemas and namespaces to provide a dynamic schema language (DSL) authoring environment. Example 1-5 shows the finished application.

Example 1-5. Simple Mule API proxy application

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:http="http://www.mulesoft.org/schema/mule/http"
```

```
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/current/mule.xsd
        http://www.mulesoft.org/schema/mule/http
        http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <flow name="main">
        <http:inbound-endpoint host="localhost" port="8080" path="geonamesproxy"
            exchange-pattern="request-response" />

        <!--
            TODO add your service component here. This can also be a Spring bean
            using <spring-object bean="name"/>
        -->

        <echo-component />

        <http:outbound-endpoint
           address="http://api.geonames.org/findNearbyPOIsOSM?lat=37.451
           &amp;lng=-122.18&amp;username=demo" method="GET" />
    </flow>
</mule>
```

Inspecting this configuration, we can see that it is an XML document with a root element of `mule`. This element is the key element and must always be included. It is this element that contains references to specific Mule modules, via schema and namespace declarations, to provide the DSL authoring environment. The most important of these is the `core` namespace, `xmlns="http://www.mulesoft.org/schema/mule/core"`, which allows you to use all the Mule core components such as flows, routers, transformers, and filters. The core namespace is then followed by subsequent namespace declarations that represent individual Mule modules, such as the HTTP module represented by `xmlns:http="http://www.mulesoft.org/schema/mule/http"` and the Spring module represented by `xmlns:spring="http://www.springframework.org/schema/beans"`.

Flows

Within the `mule` root element is a critical child element: `flow`. Flows are underlying configurations for your Mule or CloudHub integration and are the default constructs for orchestrating message processing. Each flow has a `name` attribute, which must be a unique identifier within your configuration. The flow then consists of a *message source* followed by a sequence of *message processors*. Flows are executed from top to bottom, just like any imperative programming language. Example 1-6 shows the flow we have created with the unique ID: `main`.

Example 1-6. A Mule flow

```
<flow name="main">
    <http:inbound-endpoint host="localhost" port="8080"
```

```
        path="geonamesproxy" exchange-pattern="request-response" />

    <!--
        TODO add your service component here. This can also be a Spring bean
        using <spring-object bean="name"/>
    -->

    <echo-component />

    <http:outbound-endpoint
        address="http://api.geonames.org/findNearbyPOIsOSM?lat=37.451
        &amp;lng=-122.18&amp;username=demo" method="GET" />
</flow>
```

Message Sources

A message source appears at the beginning of a flow. It receives or generates messages and forwards them on to a set of message processors to start working with the message. The message source is typically an inbound endpoint, such as HTTP or JMS, which can listen or poll on a certain address. The flow in the previous example has an HTTP message source for listening on a specific HTTP port, as shown in Example 1-7.

Example 1-7. HTTP message source

```
<http:inbound-endpoint host="localhost" port="8080"
    path="geonamesproxy" exchange-pattern="request-response" />
```

In this case, we have added a `host` attribute with the value `localhost`, a `port` attribute with the value `8080`, and a `path` attribute with the value `geonamesproxy`. This flow, when run, will create a web server that will listen on *http://localhost:8080/geonamesproxy*.

Message Processors

With the message source in place, we now need some message processors to actually do something with the received message. A message processor is used by Mule to process any messages received by a message source. Each processor can be a transformer, a Java component, or an outbound endpoint to forward on the message to an external system or to another flow.

In this case, we want to forward the message on to the GeoNames API. The GeoNames API is a simple HTTP RESTful API, so we can create an HTTP outbound endpoint similar to that of our message source to forward on the message:

```
    <http:outbound-endpoint
            address="http://api.geonames.org/findNearbyPOIsOSM?lat=37.451
            &amp;lng=-122.18&amp;username=demo" method="GET" />
```

As you can see, this is very similar to the message source, with the most noticeable difference being that we have changed the element name from `-inbound-endpoint` to `-`

`outbound-endpoint`. In this element we have then specified an `address` attribute with the value of one of the GeoNames APIs and some hard-coded query parameters:

```
http://api.geonames.org/findNearbyPOIsOSM?
lat=37.451&amp;lng=-122.18&amp;username=demo
```

The GeoNames API also requires the `GET` HTTP method, so we have included the `method` attribute on the endpoint and set its value to `GET`.

Variables and Expressions

To support the work of message processors, Mule provides the *Mule Expression Language* (MEL) to access, manipulate, and consume information from the message and its environment. Mule makes this data available via the following four contexts:

Server
: The operating system on which the message processor is running

Mule
: The Mule instance on which the application is running

Application
: The user application within which the current flow is deployed

Message
: The package (payload, attachments, properties) that the message processor is processing

These contexts are at the heart of most MEL expressions. A typical MEL expression combines one of these contexts with one or more operands and zero or more operators in a Java-like syntax and returns the resulting value. For example, to access the payload of the message, we can use the expression `#[message.payload]`, where `message` represents the message context and `payload` represents the payload property within the specified context. The syntax consists of a preceding `#[` followed by the expression to execute and a terminating `]` character.

In most cases, MEL expressions work within message processors to modify the way those processors do their main jobs, such as routing and filtering based on the message content. The following sections will focus on using the message context and cover some of the main use-cases that will be used throughout the book.

Message properties

Aside from the payload of the message, which is typically the main body of a message, message processors such as inbound and outbound endpoints add additional headers to a message called *message properties*. Message properties are defined within the following two scopes:

Inbound properties
: Inbound properties are placed on a message receiving a request on an inbound endpoint or a response from an outbound endpoint. For example, if a message to an inbound endpoint is called via HTTP with a `Content-Type` header, this property will be placed as a property within the inbound scope.

Outbound properties
: Outbound properties are set on a message to be sent via an outbound endpoint. For example, if a message with an outbound property `Content-Type` is sent via HTTP, the `Content-Type` property will be placed as an HTTP header on the outbound message.

MEL expressions allow you to refer to these message properties via a `java.util.Map` interface. For each property scope, Mule associates a map containing each property with the current message. You can refer to these maps using the following syntax:

```
#[message.inboundProperties['someProperty']]
#[message.outboundProperties['someProperty']]
```

where `inboundProperties` and `outboundProperties` are the maps within the `message` context and *`someProperty`* is they key of the property you want to retrieve from the map. Example 1-8 amends our GeoNames example to extract the `latitude` query parameter from the incoming request to use as an input to the original GeoNames request URL.

Example 1-8. Using message properties

```
<flow name="main">
    <http:inbound-endpoint host="localhost" port="8080"
    path="geonamesproxy" exchange-pattern="request-response" />

    <!--
        TODO add your service component here. This can also be a Spring bean
        using <spring-object bean="name"/>
    -->

    <echo-component />

    <http:outbound-endpoint
        address="http://api.geonames.org/findNearbyPOIsOSM
        ?lat=#[message.inboundProperties['latitude']]
        &amp;lng=-122.18&amp;username=demo" method="GET" />
</flow>
```

With the amended configuration in place, if you execute the flow with your browser using the URL *http://localhost:8080/geonamesproxy?latitude=37.451*, Mule will now propagate the `latitude` parameter to the `lat` argument in the GeoNames request URL.

Additional variables

Typically, message properties should be reserved for the Mule message for things such as HTTP headers or JMS headers. To store additional information during the execution of a flow, like variables in Java, Mule provides two more types of scoped variables:

Flow variables
: Flow variables are global to the current flow. They retain their values as control passes from one message processor to another. Thus, you can set them in one message processor and use them in another.

Session variables
: Session variables are essentially the same as flow variables, but in addition, when one flow calls another one via a Mule endpoint, they are propagated and are available in the subsequent flow.

As with message properties, flow and session variables are available via a `java.util.Map` interface. This map data can be referenced using the following syntax:

```
#[flowVars['someProperty']]
#[sessionVars['someProperty']]
```

Storing variable data

In order to store variable data, Mule provides a set of message processors to simplify working with each property or variable scope.

Setting properties. To set a message property, Mule provides the `set-property` message processor. This message property works only with outbound scoped properties as the inbound scoped properties are immutable. The following example shows how to set the `Content-Type` property on a message using this message processor:

```
<set-property propertyName="Content-Type" value="text/plain"/>
```

This message processor takes two mandatory arguments: `propertyName` and `value`. `propertyName` is the name of the property to set and `value` is the value of the property. Either of these arguments' values can also be expressions themselves. For example, to copy the `Content-Type` property from the inbound scope to the outbound scope, you could use the following example:

```
<set-property propertyName="Content-Type"
          value="#[message.inboundProperties['Content-Type']]"/>
```

Setting variables. As with properties, similar message processors are available for both flow and session variables. `set-variable` sets a flow variable and `set-session-variable` sets a session variable. The syntax for these message processors are very similar as the previous `set-property` message processor, with `variableName` being the name of the variable to set and `value` being the value of the variable. The following example demonstrates setting both flow and session variables:

```
<set-variable variableName="myFlowVariable" value="some data"/>

<set-session-variable variableName="mySessionVariable" value="some data"/>
```

Enrichment. Another way of setting message properties or variables is via enrichment. Mule provides an `enricher` element to enrich the current message with extra information. It allows you to call out to another resource and set extra information on the message without overriding the current payload of the message. For example, you can call out to another endpoint or message processor and store its return value in a message property or variable. The following example demonstrates this effect, using the enricher to call the GeoNames service and store the response in a message property:

```
<flow name="main">
    <http:inbound-endpoint host="localhost" port="8080"
        path="geonamesproxy" exchange-pattern="request-response" />


    <enricher target="#[message.outboundProperties['response']]">
        <http:outbound-endpoint
            address="http://api.geonames.org/findNearbyPOIsOSM
            ?lat=#[message.inboundProperties['latitude']]
            &amp;lng=-122.18&amp;username=demo" method="GET" />
    </enricher>
</flow>
```

The `target` attribute defines how the current message is enriched by using expressions to define where the value is stored on the message. Here we are using standard MEL syntax to refer to an outbound property using `#[message.outboundProperties['response']]`. This will add or overwrite the specified message property with the result of the outbound endpoint. The main difference between using the enricher and the `set-property` message processor is that the enricher supports setting the value of the property via a nested message processor such as an outbound endpoint, whereas the `set-property` and other associated message processors only support setting the value's `value` attribute. This just demonstrates the broad strokes of the procedure. More information on enrichment can be found here.

Functions

In addition to getting or setting information within a specific context, Mule also provides an expression syntax for executing certain functions. Functions provide a way of extracting information that doesn't already exist as a single value within a particular context. For example, if you have an XML document and care only about a particular node or value within that document, you can use the `xpath` function to extract that particular value. Or if you want extract a specific part of a string, you can use the `regex` function, and so on.

Xpath is a closely related sister specification of the XML document specification and provides a declarative query language for addressing parts of an XML document.

Our current configuration will return an XML-formatted document representing the GeoNames response. Example 1-9 demonstrates using a simple `xpath` expression to log the name of the root element.

Example 1-9. Using functions

```
<flow name="main">
    <http:inbound-endpoint host="localhost" port="8080"
       path="geonamesproxy" exchange-pattern="request-response" />

    <!--
        TODO add your service component here. This can also be a Spring bean
        using <spring-object bean="name"/>
    -->

    <echo-component />

    <http:outbound-endpoint
        address="http://api.geonames.org/findNearbyPOIsOSM
        ?lat=#[message.inboundProperties['latitude']]
        &amp;lng=-122.18&amp;username=demo" method="GET" />

    <logger level="INFO" message="#[xpath('local-name(/*)')]" />
</flow>
```

Routing

Mule has always had support for many routing options. Routers in Mule implement the Enterprise Integration Patterns (EIP). They are message processors that determine how messages are directed within a flow. Some of the most common routers are:

all
: Sends the message to each endpoint

choice
: Sends the message to the first endpoint that matches

recipient-list
: Sends the message to all endpoints in the expression evaluated with the given evaluator

round-robin
: Each message received by the router is sent to alternating endpoints.

wire-tap
: Sends a copy of the message to the supplied endpoint, then passes the original message to the next processor in the chain

first-successful
: Sends the message to the first endpoint that doesn't throw an exception

splitter
: Splits the current message into parts using a MEL expression, or just splits elements of a list

aggregator
: Combines related messages into a message collection

Alongside MEL, routers can decide on a course of action based on the contents, properties, or context of a message. Example 1-10 demonstrates using the `choice` router. It builds upon Example 1-8 to call the GeoNames API only if the `latitude` property is sent in the request.

Example 1-10. Choice router with expressions

```
<flow name="main">
    <http:inbound-endpoint host="localhost" port="8080"
    path="geonamesproxy" exchange-pattern="request-response" />

    <!--
        TODO add your service component here. This can also be a Spring bean
        using <spring-object bean="name"/>
    -->

    <echo-component />

    <choice>
        <when expression="#[message.inboundProperties['latitude']] != null]">
            <http:outbound-endpoint
                address="http://api.geonames.org/findNearbyPOIsOSM
                ?lat=#[message.inboundProperties['latitude']]
                &amp;lng=-122.18&amp;username=demo" method="GET" />
        </when>
    </choice>
</flow>
```

Summary

This chapter has offered a primer on Mule. You have been introduced to some its core features, started your first working Mule application, and connected your first API. But you have merely scratched the surface of Mule, and there are many more features for you to explore. But you're now ready to delve into Mule Cloud Connect.

CHAPTER 2

Cloud Connectors

As with transports, Cloud Connectors can process messages, communicate with a remote system, and be configured as part of a Mule flow. They can take full advantage of Mule's DSL authoring environment for autocompletion in your favorite IDE or XML editor, offering context-sensitive documentation and access to lists of default and valid values. The main purpose of Mule Cloud Connect is to provide you with an easy way to connect to the thousands of open APIs out there without having to work with transports or dealing with the different protocols that each API uses. Over the following sections we will start to replace transports with Cloud Connectors and discuss in detail how to get up and running with some of the most popular APIs.

Installing Cloud Connectors

To get started with Mule Cloud Connect, you will first need to download the connector you want to use. Most Mule modules, such as the HTTP module we used earlier, are prepackaged with Mule and do not require downloading, but you'll have to download and install the Cloud Connectors yourself. Each connector and its associated documentation is available at MuleForge, but the steps to download a connector differ slightly depending on your development environment. The following sections detail the most common approaches.

Maven

If you use Apache Maven to build your Mule projects, you can install Cloud Connectors by adding dependency entries for each connector you will be using to your Maven *pom.xml* file. Each connector's documentation page provides you with Maven dependency XML snippets that you can simply copy and paste. To install a connector via Maven, you first need to add the Mule repository to your Maven *pom.xml* file, as shown in Example 2-1.

Example 2-1. Mule repository configuration

```
<repositories>
    <repository>
        <id>mulesoft-releases</id>
        <name>MuleSoft Releases Repository</name>
        <url>http://repository.mulesoft.org/releases/</url>
        <layout>default</layout>
    </repsitory>
</repositories>
```

Once the repository is defined, add a dependency for each connector you want to use—in Example 2-2, it's GeoNames.

Example 2-2. Connector dependency configuration

```
<dependency>
    <groupId>org.mule.modules</groupId>
    <artifactId>mule-module-geonames</artifactId>
    <version>1.0</version>
</dependency>
```

With your configuration in place, recompiling will download the required connector and its dependencies. After downloading finishes, the connector will be available to your Mule application.

Update Sites

If you are using MuleStudio, you can take advantage of the Cloud Connector's Update site, shown in Figure 2-1.

Use the Update site as follows:

1. Click Help → Install New Software on the Mule menu bar.
2. After the Install window opens, click Add, which is located to the right of the Work with field.
3. Enter the unique name of choice for the update site in the Name field (for example, "Connector Updates").
4. In the Location field, enter **`http://repository.mulesoft.org/connectors/releases/3.3.1`**, which points to the Cloud Connector Update site for the current version. In this case `3.3.1`.
5. A table will appear displaying the available connectors under community and standard categories, the newest version, and the connector name.
6. Click the available version, then click Next, and finally click Finish. The connector will now be available to import into your project.

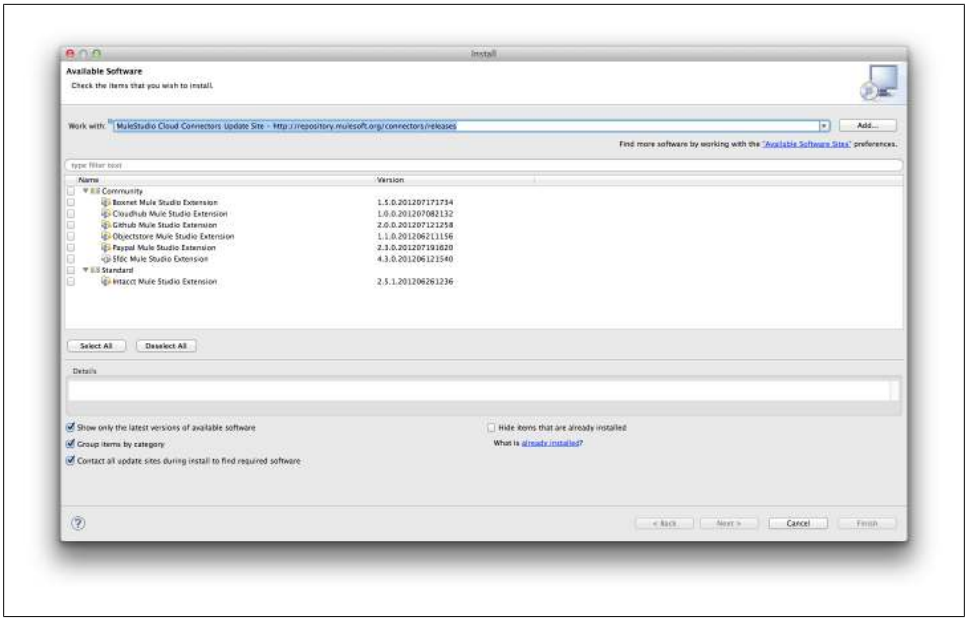
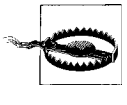


Figure 2-1. MuleStudio Update site

After following the onscreen instructions, you will be asked to restart your IDE. After that completes, the connector will be available to all your Mule applications.

Manual Installation

If you're not using Maven or Update sites, another option is to download the connector and directly add it to the build path of your project. Each connector hosted on MuleForge has a download link that will provide you with the connector of choice as a JAR file.



Be careful when using this method for installing connectors, as there is no automatic dependency management. If the connector library is reliant on other libraries, which the majority are, you will have to manually add them yourself, which can be time-consuming and error-prone.

If you are using Mule Studio, you can add the connector JAR file and other dependencies to a particular project as follows:

1. Create a *src/main/app/lib* directory in your Studio project.
2. Copy the downloaded JAR file to the *src/main/app/lib* directory.
3. Right click, or select your project and navigate to Project → Properties from the respective menu.

4. Choose Java Build Path from the left-hand menu and then click the Libraries tab in the subsequent view.
5. Click Add JARs..., then use the directory view to navigate through your project and select the JAR files in the *src/main/app/lib* directory.
6. Click OK on the resulting screens to save the changes and go back to your project.

Alternatively, if you're using a stand-alone Mule instance, you can then drop the downloaded connector JAR file into the *lib/user* directory of your Mule distribution.

Namespace and Schema Declarations

The programming model for Mule is XML, and it uses schemas and namespaces to provide a DSL authoring environment. To utilize a connector from a Mule project, you must first include the namespace and schema location declarations within your Mule configuration files.

Each connector's documentation page will provide you with namespace and schema snippets that you can simply copy and paste. Example 2-3 demonstrates the configuration for the namespace and schema locations for the GeoNames connector.

Example 2-3. Connector namespace declarations

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
      xmlns:http="http://www.mulesoft.org/schema/mule/http"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xmlns:geonames="http://www.mulesoft.org/schema/mule/geonames"
      xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/current/mule.xsd
        http://www.mulesoft.org/schema/mule/http
        http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
        http://www.mulesoft.org/schema/mule/geonames
        http://www.mulesoft.org/schema/mule/geonames/current/mule-geonames.xsd">
</mule>
```

Global Configuration

After the namespace and schema locations are defined, every Cloud Connector must define a `config` element. This element is used for setting global service properties such as credentials, security tokens, and API keys. This configuration then applies to all the operations supported by the connector, and once defined, cannot be overridden within a flow.

Each connector's `config` element provides a `name` attribute, which adds an identifier to each configuration so it can be referenced from each connector operation to let Mule know which service configuration to use. Other attributes then differ between each

connector. In this case, the GeoNames connector requires that you configure the `username` attribute that maps to the username parameter of the service:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
    xmlns:http="http://www.mulesoft.org/schema/mule/http"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:geonames="http://www.mulesoft.org/schema/mule/geonames"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/current/mule.xsd
        http://www.mulesoft.org/schema/mule/http
        http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
        http://www.mulesoft.org/schema/mule/geonames
        http://www.mulesoft.org/schema/mule/geonames/current/mule-geonames.xsd">

    <geonames:config username="demo" />

    <flow name="main">
      ...
    </flow>

</mule>
```

As you can see here, we are defining the `config` element within our `mule` configuration, but outside of any `flow`.

Multiple Connector Configurations

Each global `config` element has a `name` attribute and each connector operation has a corresponding `config-ref` attribute that associates the operation with the specific configuration to use. If only one `config` element per connector is present within your app, it is not necessary to explicitly reference a specific configuration, as Mule will default to the only one available. However, if you have multiple configurations per connector, you must explicitly reference the configuration via the `config-ref` attribute on each connector operation.

As you can see from Example 2-4, we have two GeoNames connector configurations. Each of them has a unique `name` attribute that adds an identifier to each configuration. Underneath the covers, Mule will instantiate two copies of your connector and register them within its registry with the name supplied.

Example 2-4. Referencing connector configurations

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
    xmlns:http="http://www.mulesoft.org/schema/mule/http"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
```

```
    xmlns:geonames="http://www.mulesoft.org/schema/mule/geonames"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/current/mule.xsd
        http://www.mulesoft.org/schema/mule/http
        http://www.mulesoft.org/schema/mule/http/current/mule-http.xsd
        http://www.mulesoft.org/schema/mule/geonames
        http://www.mulesoft.org/schema/mule/geonames/current/mule-geonames.xsd">

    <geonames:config username="demo" name="config1" />

    <geonames:config username="anotherUser" name="config2" />

    <flow name="main">
        <http:inbound-endpoint host="localhost" port="8080"
           path="geonamesproxy" exchange-pattern="request-response" />

        <geonames:find-nearby-pois-osm
            latitude="37.451"
            longitude="-122.18"
            config-ref="config1" />

        <geonames:find-nearby-pois-osm
            latitude="37.451"
            longitude="-122.18"
            config-ref="config2" />
    </flow>

</mule>
```

Operations are then defined as usual, with the difference that each operation has an attached `config-ref` attribute that will signal the configuration to use it for that particular operation.

Connector Operations

Connector operations wrap up connectivity to external systems or some logic into a simple call that an application can make within its flow. Each operation typically represents a particular API or function that the service provides. Connector operations can be used anywhere in a flow and can be used in a similar way to transports to invoke a remote service:

```
    <geonames:find-nearby-pois-osm ... />
```

Each operation is composed of the namespace we previously bound for the connector (`geonames:`) and the operation name (in this case, `find-nearby-pois-osm`). All available operations for a connector are accessible via content assist in your IDE or via the connector's documentation.

Simple Arguments

Instead of using URL query and path parameters, any basic arguments to the API are represented as attributes on the operation that map message payload and properties directly to API arguments. Attributes can be optional or mandatory and can provide content assisted values for enumerations and default values for properties that are not specified.

The following example uses the `find-nearby-pois-osm` operation, which represents the GeoNames Find Nearby Points of Interest API, to find the nearest points of interests for a given lat/lng pair:

```
<geonames:find-nearby-pois-osm latitude="37.451" longitude="-127" />
```

The operation accepts multiple arguments, some required and some optional. The first required arguments, `latitude` and `longitude`, are basic `java.lang.String` parameters that represent the specific coordinates.

Most simple arguments are represented as a `java.lang.String` type, but some arguments that require more specific types, such as `java.util.Date`, need to be constructed as the specific type. Using `java.util.Date` as an example, you may be inclined to configure a date argument as follows, which would lead to an error:

```
<someconnector:operation date="1984-06-03" />
```

This configuration would pass the date to the connector as a `String`, when the connector needs a `java.util.Date` type. The correct way to map the date value to the argument is to generate a `java.util.Date` object. One convenient way is to use the `groovy` expression evaluator, like so:

```
<someconnector:operation
    date="#[groovy:Date.parse("yyyy-MM-dd", "1984-06-03")]" />
```

The same also goes for `boolean` values. If you need to pass in a `boolean` value, construct it as the correct type using a `groovy` expression again:

```
<someconnector:operation true-or-false="#[groovy:true]" />
```

Information on each argument's type can be found at the connector's documentation page online and also as part of content assist. Connectors and the DevKit have support for the following types:

- `int`
- `float`
- `long`
- `byte`
- `short`
- `double`
- `boolean`

- char
- java.lang.Integer
- java.lang.Float
- java.lang.Long
- java.lang.Byte
- java.lang.Short
- java.lang.Double
- java.lang.Boolean
- java.lang.Character
- java.lang.String
- java.math.BigDecimal
- java.math.BigInteger
- java.util.Date
- java.lang.Class
- java.net.URL
- java.net.URI

Collections and Structured Arguments

Complex types

More complex arguments, such as collections or structured objects that can't really be expressed as simple attributes, can instead be represented as complex types as child elements within the operation itself. Example 2-5 demonstrates using the GeoNames `astergdem` operation, which accepts a set of lists for latitudes and longitudes as input.

Example 2-5. Collections configuration

```
<geonames:astergdem>
    <geonames:latitudes>
        <geonames:latitude>37.451</geonames:latitude>
        <geonames:latitude>37.450</geonames:latitude>
    </geonames:latitudes>
    <geonames:longitudes>
        <geonames:longitude>-122.18</geonames:longitude>
        <geonames:longitude>-122.18</geonames:longitude>
    </geonames:longitudes>
</geonames:astergdem>
```

The example first defines the root element for each list (for example, `geonames:latitudes`). Secondly, the list's root element contains an array of child elements representing the list items (for example, `geonames:latitude`).

As with collections, any complex types such as custom Java classes can be passed to the operation via child elements. Let's take a look at the following example from the GetSatisfaction connector. One of the connectors operations, `getsatisfaction:create-topic-at-company`, has a method signature that requires a custom Java class: `org.mule.module.getsatisfaction.model.Topic`. Inspecting this class you will see that it's a simple POJO with some fields for subject, content, products, etc., similar to the following snippet:

```
public class Topic extends Post {
    ...
    private String subject;
    private String content;
    private Style style;
    private List<Product> products;
    private List<String> keywords;
    ...
}
```

Any custom classes like this are automatically deconstructed and reconstructed as complex types within the schemas themselves, enabling them to be defined easily as child elements of the operation.

As you can see in Example 2-6, the topic class is now constructed directly using XML via the `getsatisfaction:topic` element. Any simple properties of the class (as documented in the earlier list of supported types) are represented as normal (attributes on the element directly), such as `subject`, `content`, and `style`. And any complex properties such as custom classes and collections are represented as further nested complex types within the element, as demonstrated by the `getsatisfaction:product` property.

Example 2-6. Complex type configuration

```
<getsatisfaction:create-topic-at-company companyId="mulesoft">
    <getsatisfaction:topic
        subject="test for product affiliate"
        content="additional detail goes here"
        style="PRAISE">
        <getsatisfaction:products>
            <getsatisfaction:product name="muleion"/>
        </getsatisfaction:products>
        <getsatisfaction:keywords>
            <getsatisfaction:keyword>keyword</getsatisfaction:keyword>
        </getsatisfaction:keywords>
    </getsatisfaction:topic>
</getsatisfaction:create-topic-at-company>
```

Passing by reference

For everything else that is not in the supported list of types, the DevKit allows the information to be passed along using references. As we saw in Example 2-5 and Example 2-6, we can build up our objects using the strongly typed schemas that represent the object itself. However, in previous versions of the DevKit, this wasn't possible, and

even now you may want to reference an object or collection already constructed elsewhere. To allow this, connector operations that require structured arguments also allow you to reference preconstructed arguments via an attribute on the child element named `ref`.

Using the Collections example in Example 2-5, we can instead build our list outside of our operation and refer to it as follows:

```
<spring:bean id="latitudeA" class="java.lang.String">
    <spring:constructor-arg value="37.451" />
</spring:bean>

<spring:bean id="latitudeB" class="java.lang.String">
    <spring:constructor-arg value="37.451" />
</spring:bean>


<spring:bean id="list"
    class="org.springframework.beans.factory.config.ListFactoryBean">
    <spring:property name="sourceList">
        <spring:list>
            <spring:ref bean="latitudeA" />
            <spring:ref bean="latitudeB" />
        </spring:list>
    </spring:property>
</spring:bean>

...

<geonames:astergdem>
    <geonames:latitudes ref="list" />
</geonames:astergdem>
```

This example uses `spring` to manually build our list from two Strings and then reference the list from a `ref` attribute using the `id` of the `spring:bean` (in this case, `list`).

The same functionality we've shown for collections can be applied to custom classes. Using the complex type example in Example 2-6, we can instead build the object outside of the operation and then reference it as follows:

```
<spring:bean id="keywordA" class="java.lang.String">
    <spring:constructor-arg value="muleion" />
</spring:bean>

<spring:bean id="product" class="org.mule.module.getsatisfaction.model.Product">
    <spring:constructor-arg value="muleion" />
</spring:bean>

<spring:bean id="keywords" class="org.springframework.beans.factory.config.
                                  ListFactoryBean">
    <spring:property name="sourceList">
        <spring:list>
            <spring:ref bean="keywordA"/>
        </spring:list>
    </spring:property>
```

```
</spring:bean>


<spring:bean id="topic"
    class="org.mule.module.getsatisfaction.model.Topic">
    <spring:property name="subject" value="test for product affiliate" />
    <spring:property name="content" value="additional detail goes here" />
    <spring:property name="keywords" ref="keywords" />
    <spring:property name="product" ref="product" />
</spring:bean>

...

<getsatisfaction:create-topic-at-company companyId="mulesoft">
    <getsatisfaction:topic ref="topic" />
</getsatisfaction:create-topic-at-company>
```

This example uses `spring` to manually build our `org.mule.module.getsatisfaction.model.Topic` object and then reference it from the `ref` attribute using the `id` of the `spring:bean` (in this case, `topic`).

Expression Evaluation

Previous examples use static values as inputs to operation arguments, but in real life you will probably want to use variable values extracted from requests, responses, or properties files. To support this, each connector operation can handle full expression evaluation and argument transformation as shown in "Variables and Expressions" on page 8.

The expression evaluation performed by Example 2-7 allows us to parameterize values to operations from a variety of sources. This example uses the MEL to extract the parameters from the header of an incoming HTTP request, but if the source of the message is XML, JSON, or pretty much anything, there's an expression evaluator for it. More information on expressions can be found here.

Example 2-7. Connector operation with expressions

```
<geonames:find-nearby-pois-osm
    latitude="#[message.inboundProperties['latitude']]"
    longitude="#[message.inboundProperties['longitude']]" />
```

Parsing the Response

The response format from each operation can differ between connectors. Most connectors provide the raw response from the service provider's API and will also provide a choice between multiple response formats, if available. Take GeoNames for example. GeoNames offers both XML and JSON formatted responses. The configuration in Example 1-4 would result in the operation returning the default response format, XML. Once invoked, it will return an XML response similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<geonames>
    <poi>
        <name>Cook's Seafood</name>
        <typeClass>amenity</typeClass>
        <typeName>restaurant</typeName>
        <lng>-122.1795529</lng>
        <lat>37.4516093</lat>
        <distance>0.08</distance>
    </poi>
    <poi>
        <name>Starbucks</name>
        <typeClass>amenity</typeClass>
        <typeName>cafe</typeName>
        <lng>-122.1803386</lng>
        <lat>37.452055</lat>
        <distance>0.12</distance>
    </poi>
    <poi>
        <name>Safeway</name>
        <typeClass>shop</typeClass>
        <typeName>supermarket</typeName>
        <lng>-122.1787081</lng>
        <lat>37.4507461</lat>
        <distance>0.12</distance>
    </poi>
    <poi>
        <name>Akasaka</name>
        <typeClass>amenity</typeClass>
        <typeName>restaurant</typeName>
        <lng>-122.1809239</lng>
        <lat>37.4524367</lat>
        <distance>0.18</distance>
    </poi>
</geonames>
```

As with many other connectors, the GeoNames connector allows you to switch between the available response formats by specifying an optional argument either on the operation itself or via the connector's `config` element. In the case of the GeoNames connector, you can specify the argument at the operation level through the attribute named `type`:

```
<geonames:find-nearby-pois-osm latitude="37.51"
          longitude="-122.18" type="json"/>
```

Here we have added the optional `type` argument and set it's value to `json`. With this new configuration in place, instead of seeing XML, you should now see a JSON-formatted response, similar to the following:

```
{"poi":[
    {"typeName":"restaurant","distance":"0.08","name":"Cook's Seafood",
        "lng":"-122.1795529","typeClass":"amenity","lat":"37.4516093"},
    {"typeName":"cafe","distance":"0.12","name":"Starbucks",
        "lng":"-122.1803386","typeClass":"amenity","lat":"37.452055"},
    {"typeName":"fire_hydrant","distance":"0.14","name":"",
```

```
        "lng":"-122.1784682","typeClass":"amenity","lat":"37.4510495"},
    {"typeName":"fire_hydrant","distance":"0.19","name":"",
        "lng":"-122.1800146","typeClass":"amenity","lat":"37.4493291"},
    {"typeName":"restaurant","distance":"0.18","name":"Akasaka",
        "lng":"-122.1809239","typeClass":"amenity","lat":"37.4524367"},
    {"typeName":"fast_food","distance":"0.16","name":"Rubios",
        "lng":"-122.1784509","typeClass":"amenity","lat":"37.4501701"},
    {"typeName":"cinema","distance":"0.21","name":"Guild",
        "lng":"-122.1812488","typeClass":"amenity","lat":"37.4525935"},
    {"typeName":"cafe","distance":"0.22","name":"Pete's Coffee",
        "lng":"-122.1780217","typeClass":"amenity","lat":"37.4498336"},
    {"typeName":"fast_food","distance":"0.23","name":"Applewood2Go",
        "lng":"-122.1816743","typeClass":"amenity","lat":"37.4526078"},
    {"typeName":"fire_hydrant","distance":"0.23","name":"",
        "lng":"-122.178034","typeClass":"amenity","lat":"37.449606"}
]}
```


If you are unsure what response formats are available, or at what level they are configured, they can be found by using content assist or by reading the connector documentation. Additional information on particular responses can also be found using the service provider's documentation.

Summary

In summary, Cloud Connectors make the simple tasks easy and the hard tasks possible by taking a step back from HTTP and protocols to provide a higher level of abstraction and a consistent interaction model with APIs, allowing developers to concentrate on the task at hand.

The examples in this chapter are reasonably straightforward: read-only operations, no complex authorization mechanisms, etc. But even these types of APIs can be difficult to work with, the ones that make you jump through hoops in order to perform a task that should be dead simple to do.

The upcoming chapters will take a look at how Cloud Connectors can simplify even more complex APIs and deal with authorization, events, connection management, and more.