

## MULE: A CASE STUDY

### About the Author

Eugene Ciurana is the director of platform technology at Walmart.com Global and the enterprise architect at Walmart.com. He has led massive integration projects for some of the largest banks, telcos, and ecommerce companies worldwide since 1992. Eugene is also the author of the action thriller *The Tesla Testament* and is writing a professional book about Mule. You may reach him at various IRC development channels, including Freenode #esb, under the /nick pr3d4t0r.



The buzzword du jour in services oriented architectures is ESB. Enterprise service buses are the preferred tools for integrating systems with heterogeneous data interchange interfaces and based on a wide array of technologies, from COBOL to CORBA to JEE. This article is an introduction to ESBs and enterprise integration using Mule, the open-source ESB.

### WHY ARE ESBS NEEDED?

Major vendors first sold message queuing as the ultimate interoperability solution, then SOAP and REST, before realizing that multiple applications need to share data but had significant interface differences. Architects and vendors suggested many approaches to solving this issue, from writing wrappers using a common protocol to porting legacy systems to Java or .Net and, in the process, create a modern interface that fit in the enterprise architecture. None of these approaches is practical because they are code-intensive, expensive, and are coupled to specific systems, programming languages, and protocols.

Early attempts at solving this issue involved creating a “bus” using a common transport like MQ Series and defining a common message format (positional or XML). Systems participating in data exchanges would implement messages with specific attributes and place them in a queue. This soon becomes impractical because message formats need to be revised too often to accommodate new attributes and exponentially increases regression testing and debugging time and expense. Similar problems occur when using SOAP, REST, or almost any protocol.

The solution to this problem is elegant and obvious: let the applications communicate with one another in the protocols they already support, from EDI to SOAP, over a common transport aggregator independent of the native protocols, and adding application- or protocol-specific translation modules or message routing only where required at the endpoints. This approach allows a mainframe application written in COBOL to interoperate with a mobile device written with J2ME without either end knowing anything about the other’s characteristics.

Ross Mason, a leading Java engineer, identified these issues as early as 2001 and began working on what became the Codehaus open-source project Mule. Parallel development occurred at Progress Software who produced Sonic ESB and coined the term “Enterprise Service Bus” during the same period. Several open-source and commercial ESBs exist now. Each offers different features and caveats.

### COMMERCIAL AND OPEN-SOURCE ESBS

There are many commercial and open-source ESBs. Table 1 shows those with the most mature offerings, in alphabetical order. All of them, including the open-source products, have support from one or more companies.

table 1

Product	Vendor	Connects with
Matrix BusinessWorks	TIBCO	<ul style="list-style-type: none"> <li>• SOAP</li> <li>• EMS</li> <li>• JMS,</li> <li>• Rendezvous</li> <li>• MQ</li> <li>• BPEL</li> </ul>
Mule ESB	Open-source, MuleSource, Inc.	<ul style="list-style-type: none"> <li>• SOAP</li> <li>• REST</li> <li>• JMS</li> <li>• MQ</li> <li>• JBI</li> <li>• AQ</li> <li>• Caching</li> <li>• JavaSpaces</li> <li>• GigaSpaces</li> <li>• Email</li> <li>• IM</li> <li>• JCA</li> <li>• S400 Data Queues</li> <li>• System I/O</li> </ul>
OpenESB	Open-source, Sun Microsystems	<ul style="list-style-type: none"> <li>• JBI</li> <li>• JCA</li> <li>• JAX-RPC</li> <li>• JAX-WS</li> </ul>
Sonic ESB	Progress Software	<ul style="list-style-type: none"> <li>• JMS</li> <li>• SOAP</li> <li>• JMX</li> </ul>
Websphere ESB	IBM	<ul style="list-style-type: none"> <li>• JMS</li> <li>• MQ</li> <li>• SOAP</li> </ul> <p>Requires additional adapters to interface with other products and legacy protocols. Requires Websphere to work.</p>

Choosing an ESB weighs factors like existing technology, development and deployment roadmap, company policy, strategic partnerships (or lack of them), legacy systems, and so on. In a recent evaluation of all of these products, an enterprise's decision to use Mule over any of the others was based on these criteria:

- ▶ Active open-source community and commercial support available
- ▶ State of the art implementation and ability to run under Java 5/6
- ▶ Number of relevant features out of the box
- ▶ Response time from vendors within 48 hours (average time for the winners was 2 hours)
- ▶ Ease of installation and deployment
- ▶ Ease of configuration and expansion
- ▶ “Codeless” integration with legacy, third-party, and commercial products
- ▶ Ability to drop in/pull out without incurring in lock-in or ancillary product dependencies
- ▶ Low total cost of ownership

Mule ESB won the selection process by meeting or exceeding all these criteria; a commercial ESB from a Palo Alto-based company became a close second but it had a significant license cost in comparison with vendor subscriptions for Mule support. The other products either didn't meet the technological criteria, failed the vendor response time test (a vendor offered to send a blue-suited, clean-shaven sales engineer to talk about their product three weeks after the selection process was complete), involved unnecessary dependencies on a vendor's ancillary products, the product was immature, or all of the above.

## DEPLOYING MULE IN AN ENTERPRISE ENVIRONMENT

The availability of every common transport and protocol in an open-source package from a single download, at no cost, and with a rich community around it is one of the most compelling reasons for using Mule. The Mule community has demonstrated the software connecting a wide variety of mission-critical systems in financial institutions, airlines, commerce, and technology companies. Mule and all its bundled subcomponents are licensed under a variation of the well-known Mozilla Public License 1.1. Mule performs as well or better as commercial ESBs, and there is at least one company offering 7x24 support and indemnification, the last two requirements that many corporations demand before considering any open-source software for deployment. It seems like all the items in the corporate checklist can be marked off with minimal risk resulting from bringing this open-source product in-house. This ought to make IT management and legal departments sleep well at night when Mule becomes part of the enterprise architecture.

## WORKING WITH MULE

Getting and installing Mule is a simple process. The only prerequisites are a web browser and a functional Java run-time version 1.4.2 or later. A standard Mule download includes a number of packages, such as JMS, two web services platforms, adapters, translators, and everything one may need out of the box.

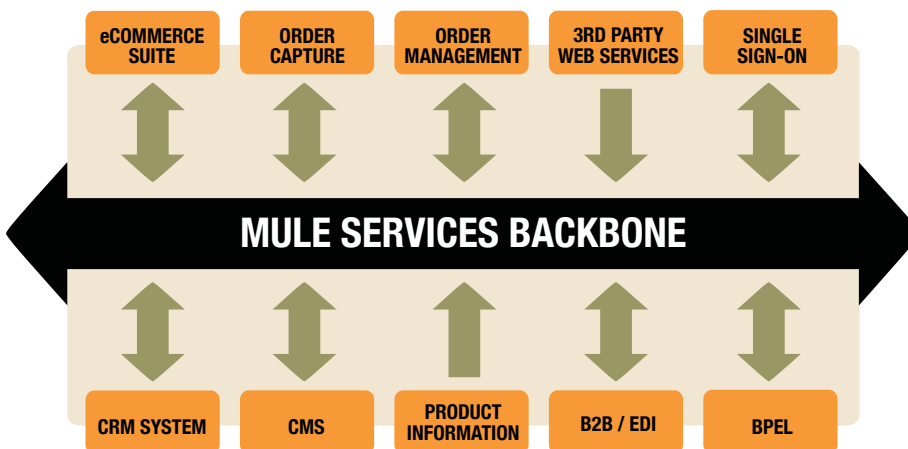
The configurations used in this article are:

- ▶ AMD-based 2.6 GHz 4 GB RAM, and Intel-based 3 GHz 4 GB RAM servers
- ▶ Solaris 10 and Ubuntu Linux Dapper Drake
- ▶ Java Run-time Environment 1.5.9
- ▶ Mule current general availability package (<http://www.mulesource.com/download/>)
- ▶ A commercial JMS provider
- ▶ Commercial applications for order capture, inventory management, order management, fulfillment, and reporting

For purposes of this article, assume a project that combines several enterprise-class (commercial and open-source) applications. Time-to-market pressure dictates a policy of “acquire-instead-of-build” and demands use of best of breed products for each subsystem. These products may come from competing vendors since one may offer an excellent product information manager but a lousy content management system. These products may have different data exchange interfaces as well, like the environment defined in Figure 1.

A useful ESB will allow near-codeless integration of all those subsystems, shifting the effort from programming interfaces to configuring transports, connectors, and filters. Mule allows that for most transports without modifications.

figure 1



## GETTING, INSTALLING AND TESTING MULE

There are two places for downloading the code:

- ▶ From the Mule project site at Codehaus (<http://mule.mulesource.org/display/MULE/Download>)
- ▶ From the MuleSource.com commercial support site (<http://www.mulesource.com/download/>)

Both downloads are identical. They consist of either a compressed tarball or a PKzipped file; download the appropriate one to your target installation environment.

Mule installation is a snap:

1. Download the compressed file
2. Decompress the mule file to its new home in your local server
3. Set the JAVA\_HOME and MULE\_HOME environment variables; MULE\_HOME must point to the directory where Mule was decompressed in the previous step
4. Add \$MULE\_HOME/bin to the PATH

Running the \$MULE\_HOME/bin/mule (or mule.bat) command from the console is the fastest way to test if the environment is set correctly. If all is well, you should see a message showing the mule command's usage; otherwise you'll see an exception indicating that MULE\_HOME isn't set, or other environmental error message.

Last, you may test your installation by running the Echo service, an example bundled with the Mule installation. Execute:

```
mule -config examples/maven/echo/conf/echo-config.xml
```

You will see this prompt:

```
Please enter something:
```

Type in any text and press the Return key. After a few seconds, Mule will display this message:

```
*****  
* Message received in component: EchoUMO. Content is:  
* 'Hello.'  
*****
```

The echo component takes input from the stdin console endpoint of the server where it runs, and sends its output to stdout. The echo component also defined a web service that you may access from:

```
http://muleserver.mycompany.com:8081/services/  
EchoUMO?method=echo&param=Hello.
```

Pointing the browser to that location and you should see:

```
<soap:Envelope>  
  <soap:Body>  
    <echoResponse>  
      <out>Hello.</out>  
    </echoResponse>  
  </soap:Body>  
</soap:Envelope>
```

Congratulations! Your Mule installation is working well. You can view the users guide for an explanation of what just went on. The rest of this article introduces the Mule terminology and explains how to define endpoints and components

## LEARNING MULE

The best way to understand how Mule works is by doing, studying the code examples, reading the documentation, and configuring new services. The documentation available from either Codehaus or MuleSource is sufficient but incomplete. An effort from the Mule community to improve the user and developers guides is in progress. Like many open-source projects, you will need to inspect the sample code, configuration files, API documentation, users and programmer's guides, and perhaps visit the Codehaus #mule or Freenode #esb IRC channels.

The Mule web site provides an excellent introduction to the software, its design principles, uses, and terminology in a thorough architecture guide (<http://www.eaipatterns.com/>). You may also view Mule as a real-life implementation of the enterprise application integration discussions by Gregor Hohpe and Bobby Wolf from their Enterprise Integration Patterns book.

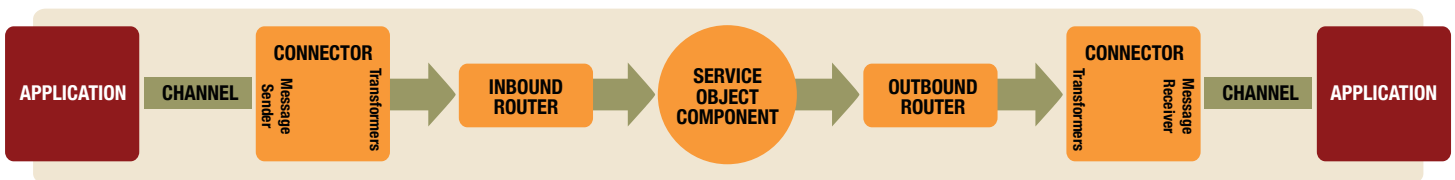
The key to using Mule is to learn how it enables two or more applications to communicate. Mule implements elegant protocols and standards based on well-defined patterns. Applications communicate with one another by implementing these patterns with a set of predefined components, as illustrated in figure 2.

Applications interface with Mule over a transport. A transport carries service objects between components (some of the current documentation refers to service objects as "UMOs" or universal message objects; that term is deprecated now). The transport is any mechanism for exchanging data between two points. Mule doesn't implement or mandate use of particular transports. It offers instead a number of transport providers and the integration engineers or developers get to choose the appropriate one for a given application.

A transport provider implements a message receiver or dispatcher, a connector that implements a protocol like JMS, TCP, etc., and an optional transformer. The transformer may be used to convert service objects from one format to another (XML to native Java, for example). Mule routes inbound and outbound data between transports through the service object component, which is responsible for managing component events to and from the component, and for managing pooled resources.

Let's review a case study now that the architecture and terminology are out of the way.

figure 2



## ENVIRONMENT SETUP

This setup describes large enterprise Mule implementation that's a mashup of commercial and open-source component applications. Each application was designed to communicate over the vendor's pre-defined protocol (JMS, SOAP, whatever). Applications are optimized for communicating easily with other applications from the same vendor (lock-in!) so applications from assorted vendors normally requires agreeing on a protocol and a document format. Mule helps interoperability by freeing the integration team from worrying about protocols altogether. An application that only supports JMS may communicate with one that only supports REST web services by using Mule's transports, routers, and transformers. Neither application needs modifications. Point their transports at Mule and dispatch the data. Mule will do the rest.

Mule runs on redundant, high-performance servers that will propel messages between endpoints. These servers may become a single point of failure. Take special care in selecting hardware and operating systems designed for handling large loads, easy and quick failover, and fast throughput.

## CASE STUDY: MULE AS A COMMON JMS TRANSPORT

The applications integration team had to deal with six different vendors. JMS and SOAP emerged as the common transports for all applications; 90% of communications could take place over JMS.

A normal JMS configuration would require each application to create inbound and output queues for every single application that it needs to interact with. This "point-to-point" configuration would explode complexity because more vendor applications would later be added to the mix, and each application would have to be revised to add new interface definitions every single time. The integration team decided instead to leverage Mule's capabilities to simplify this by creating a single, common queue where all messages are sent. Every message has an attribute that indicates its type and routes it to the appropriate application. The application captures the message in its private input queue, process it, and replies by putting a response in the common queue. Mule routes the response to the appropriate applications.

Listing 1 is a Mule configuration file for handling the interactions between the application components communicating over JMS. The colored regions match each of the next paragraphs.

The grayed parts of the listing are redundancies or require no explanation.

Download Mule configuration file:

<http://www.theserverside.com/tt/articles/content/CaseStudyMule/listing-1.zip>

The first step consists of defining the connector name and type. Any JMS provider, open-source or commercial, can be defined here as long as Mule may reach the server across the network. Additional properties such as credentials or naming should be defined here as well. Mule will use this connector for all JMS traffic. More than one connector of the same type may be defined, if required. Connectors are identified by name; these names are used Mule to determine which one handles a given interaction. Connectors for other transports would be defined in a similar manner.

Next, define the global endpoints. These endpoints are the interfaces between each application and Mule. A service object component routes communications between endpoints; the endpoints may be defined at the service object component level or they may be defined globally. It seems easier to define them globally so that the Mule model and router configuration is less verbose. Use the approach that works best for your application.

Mule is rather silent by design. It just sits there and waits for traffic to go by, dutifully writing status messages to its logs. Some times it's convenient to define console output queues for debugging purposes since not everyone may have access to a JMX console. Consoles display ASCII or UTF-8 output, but the traffic sent through this Mule configuration is all JMS. Thus, two transformers are defined. One converts JMS messages to Mule objects, the other converts these objects to strings; the console can display the diagnostic strings without problems. These messages could also be routed via email to each vendor using Mule's Java Mail transport without having to extract the text from the logs, etc.

The Mule model manages the run-time behavior of a server instance and encapsulates all its functionality. The model is responsible for maintaining the service object's configuration and instances.

The model for this example has one descriptor, the `JMSMessageSwitchboard`, which will route all messages between different applications via their endpoints. This contains all the routers, filtering rules, and endpoint definitions to ensure that messages travel within applications accurately.

Since the integrators decided that all messages are input through a single delivery point, they defined a single inbound router that listens on a specific JMS connector. As long as every application defines its message outputs to the outJMSBitco endpoint, Mule will be able to route them to the appropriate destination.

Output routing definitions require two parts: first, we define a rule that matches all JMS messages and sends them to specific filters for each application's router definition; that's what the match-all="true" attribute is for. Next, each endpoint corresponding to a given application contains a filter that matches a specific attribute defined in the JMS message header such as OrderHistoryRequest. Notice that, in this case, communications are asynchronous. Applications put requests in the common queue and move on to other tasks. The responses will come back in their application-specific queues. Each response will also have an identifying JMS message header; parsing the validity of the header and the data payload are each application's responsibility.

Every message circulating through the JMSMessageSwitchboard must match a filtered rule. If it doesn't, it means that an integrator forgot to update the Mule configuration file, a new service is now on-line and using the input queue, or there is a misspelling. A quick way of detecting this is the catchall outbound router defined toward the end of the Mule descriptor tags. If none of the rules matches, Mule will route the message to stdout after converting it to a string (remember those transformers that we defined earlier?).

Adding a new application to this mashup becomes a trivial exercise: ensure that it uses JMS for outbound and inbound communications, define its routing attribute, and update the Mule configuration file. If the new application communicates over a different protocol, like SOAP, define the connector, endpoints, and a transformer to convert JMS-to-SOAP and back. The whole integration process will take five minutes, at most. That is the power of Mule.

## EXTENDING AND CUSTOMIZING MULE

Mule may interface with service providers or transports written in any programming language as long as it can connect to them over the network and transform data using standard protocols. On occasion, however, an application may demand the creation of additional transports, transformers, connectors, service objects, or other features. Mule is written in Java, may be built with Java 1.4.2, Java 5 or Java 6, and it offers a rich API (<http://mule.mulesource.org/docs/apidocs/>) that may be customized for building specific applications. Programmers may build Mule from scratch using Maven 2, and may embed it in applications using the Spring framework as a container. Last, a number of Eclipse-friendly tools are either available now or will become available in the first half of 2007.

A programmer with working knowledge of the JEE stack, Java programming techniques, and understanding of enterprise integration issues may extend the system after a short learning curve. From a programmer's perspective, the only shortcoming in working with Mule is a slight lack of formal documentation. At the time this article was written, the best way to learn how to extend and program Mule is by reading the documentation, studying the code examples, and learning the quirks by implementing the APIs. Like many open-source projects, the prime directive is to Use the Force – read the Source! Fortunately, the members of the Mule community are very helpful in assisting newcomers to learn the technology.

## OPEN-SOURCE IS FINE, BUT MY BOSS WANTS COMMERCIAL SUPPORT...

Support is the daily reality of any software product, whether it's a commercial or open-source application. Many large companies, the ones that would benefit the most from using a well-designed open-source product, won't deploy it or even evaluate it unless there is a company providing customer support. The smart companies combine open-source community participation with a subscription to a commercial support provider.

MuleSource.com is a company established by the same team that created Mule. They offer support subscriptions at three different levels. The company has offices in San Francisco and London to support their US and European clients. They offer 24x7 support, problem resolution, performance tuning, and configuration advice. The company organized the first Mule International Users Group gathering in London in November of 2006 where they announced the Mule roadmap (commercial and open-source). The highlights from that meeting include:

- ▶ All Mule licensing is Mozilla-like, and covers Mule and all its component parts to reduce the number of licensing headaches that a company's legal department must engage in order to allow Mule to run in-house
- ▶ 24x7 support expanded to include phone resolution
- ▶ MuleSource absorbs indemnification for users
- ▶ The 2007 launch of MuleForge, a community site where commercial and open-source users will gather to exchange tips and techniques for designing, developing, and deploying Mule components and applications
- ▶ A certification program for engineers

A number of technical announcements by the Mule open-source team included:

- ▶ Auto-discovery of new services
- ▶ Hot patching
- ▶ OSGi for components deployed through Mule, and eventually for Mule itself
- ▶ Enhancements to the Mule IDE and additional Eclipse-ready modules

Based on a recent experience at a large company where Mule was deployed, MuleSource's support offerings are in par or better than those from purely commercial ESB products, at a fraction of the cost, and with a solid and enthusiastic community that balances the volunteerism of open-source projects with sensible business decisions and offerings from MuleSource. A company, large or small, may deploy the product and services knowing that commercial support and the community involvement of users and developers from over 100 production sites (Fortune 500, large financial institutions, airlines, etc.) are available.

## CONCLUSION

Mule is the best of breed open-source enterprise service bus. It does the same things as any commercial offering with similar or better results. It's free. An open-source community thrives around it. Mule is in production in many large companies worldwide, from financial institutions to large e-commerce applications. The product is easy to install, deploy, maintain, and extend. Anyone with some understanding of enterprise integration and a text editor may configure it. While it lacks (at this time) some of the flashy features from its commercial cousins like drag and drop configuration and pretty manuals, it outshines other systems by the sheer number of adapters and extensions already available that your company can use for solving tough integration problems with no more investment than a click and a download.

Consider Mule seriously if you are in the design, evaluation, or development phases of a massive enterprise integration effort for your company. Mule can do everything the commercial ESBs can, at a lower TCO, it's more complete than other open-source offerings like OpenESB, it complies with more standards (de facto or formal) than commercial products, and it's ideal for preventing vendor lock-in. Check it out!

---

## ABOUT MULESOURCE

MuleSource is the leading provider of open source service oriented architecture (SOA) infrastructure software. Founded by the creators of the Mule project, the world's most reliable and widely used open source enterprise service bus (ESB) and integration platform, MuleSource delivers enterprise class support and services to the hundreds of organizations that have downloaded the open source project worldwide. Founded in 2006 and backed by investors Hummer Winblad Venture Partners, Lightspeed Venture Partners and Morgenthaler Ventures, MuleSource is headquartered in San Francisco with offices worldwide.

**For more information: [www.mulesource.com](http://www.mulesource.com), or email [info@mulesource.com](mailto:info@mulesource.com).**