DZone Refcardz

Get More Refcardz! Visit refcardz.com

www.dzone.com

Core Mule

CONTENTS INCLUDE:

- About Mule Configuration
- Mule Architecture in a Nutshell
- Configuring Mule
- Mule Transformers, Filters, Routers and Components
- Mule Entry Point Resolving
- Hot Tips and more...

Core Mule

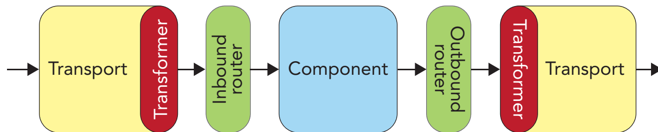
By Jos Dirksen

ABOUT MULE CONFIGURATION

Mule is one of the most mature open source enterprise service busses (ESBs) out there. It provides an easy to use, lightweight ESB that can easily be integrated with a large amount of technologies. Mule also provides a rich set of routers, transformers, and filters which you can use in your own integration flows. This reference card will provide an overview of the architecture of Mule and show the different routers, transformers, and filters that are available, and will show how to use them by using example configurations.

MULE ARCHITECTURE IN A NUTSHELL

To make it easier to understand Mule, let's first have a quick look at Mule's architecture.



As you can see, the basic concepts of Mule are pretty straight forward. In Mule's architecture we have the following main parts:

Component	Contains the business logic. For instance this could be a spring bean, a REST service, a POJO, etc.
Transport	Handles connectivity with a specific technology or application (e.g. JMS, SAP, FTP, etc.).
Transformers	Transforms the data to the format the next component expects and can work with.
Inbound Routers	Determines what to do with the received message before it's sent to the service.
Outbound Routers	Determines where a message needs to be sent to after it's been processed by the service.

Basically what happens is:

1. A transport receives a message. For instance a message has been put on a JMS queue the transport is listening on.
2. Before the message is sent to the inbound router, it's first transformed (if needed) to the required format.
3. Then the message is processed by the inbound router. For instance we could have a "selective consumer" which only accepts messages that are sent by applications we trust.
4. After the inbound router, the message is sent to the component, which applies its business logic to it.
5. After the service is done processing, the message is sent to the outbound router. This router determines where to next send the message. We could for instance split this message into multiple parts and send those to different targets.
6. And finally we can transform the message once again, and let the transport handle all the connectivity details.

CONFIGURING MULE

Mule's configuration is based on Spring and uses XML schemas to provide code completion. This makes it very easy to write your integration flows. Let's start with a very basic Mule configuration:

```
<mule xmlns="http://www.mulesource.org/
schema/mule/core/2.1"   xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"   xmlns:vm="http://
www.mulesource.org/schema/mule/vm/2.1"
xmlns:file="http://www.mulesource.org/schema/mule/
file/2.1"   xsi:schemaLocation="
        http://www.mulesource.org/schema/mule/
        core/2.1
        http://www.mulesource.org/schema/mule/
        core/2.1/mule.xsd
        http://www.mulesource.org/schema/mule/vm/2.1
        http://www.mulesource.org/schema/mule/
        vm/2.1/mule-vm.xsd
        http://www.mulesource.org/schema/mule/
        file/2.1
        http://www.mulesource.org/schema/mule/
        file/2.1/mule-file.xsd">

<model name="refcheat-model">
   <service name="basic-service">
      <inbound>
         <file:inbound-endpoint name="example-in"
            path="work/example/in" />
      </inbound>
      <component>
         <singleton-object class="dzone.Reverser" />
      </component>
      <outbound>
         <pass-through-router>
            <file:outbound-endpoint name="example-out"
```

Get More Refcardz
(They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!
Refcardz.com

Configuring Mule, continued

```
            path="work/example/out" />
          </pass-through-router>
        </outbound>
      </service>
    </model>
</mule>
```

You can see that at the top of the file we've declared a number of Mule specific namespaces. Mule provides XML schemas for all its features. We'll focus on Mule core sheets, but also show you a couple of features from the vm and file schemas. After the namespaces declaration we define a `<model>` element. A model in Mule is a container element for a number of services. In the model element you can see that we've defined a single service where the parts we've discussed in the introduction appear again. In this case we've configured a file transport which will read messages from the file system, pass it on to a component which will reverse the content of the input file, and finally use an outbound router, with a single transport to write the now reversed string back to the file system. We didn't specify an inbound-router. If we don't specify one, all the messages are simply processed by the specified component.

In this example the inbound endpoint uses the File transport. Mule provides several standard transports you can use, as described in the next section. For details on a specific transport, see http://www.mulesource.org/display/MULE2USER/Available+Transports

We won't go into the details of all the endpoints. We'll just provide an overview of the transports available.

Mule Transports

Here is an overview of the transports Mule provides.

Namespace	Description
file	Provides endpoints which allow you to read and write to the file system
axis	Allows you to consume and provide webservices using axis
jbpm	Adds functionality to interact with jBPM
cxf	Allows you to consume and provide webservices using CXF
ejb	Using the endpoints from this transport you can connect to EJBs
email	The email namespace provides functionality to connect to POP3, SMTP and IMAP servers
ftp	Provides endpoints to read and write to ftp servers
http	Allows you to receive and send information using HTTP
jdbc	With the JDBC endpoints you can interact with databases using SQL
jms	Provides endpoints to connect to JMS queues and topics
multicast	Provides an UDP multicast endpoint
quartz	Allows you to control the quartz job manager from Mule
rmi	Provides inbound and outbound endpoints for RMI
stdio	Allows you to send messages to mule using stdio
tcp	Provides endpoints for tcp connectivity
udp	Provides endpoints for udp connectivity
vm	The vm endpoint can be used for internal communication
xmpp	The XMPP endpoint can be used to connect to XMPP compliant instant messaging servers

Mule Expressions

With expressions Mule allows you to access certain properties from the message or from the payload, and based on these

Mule Expressions, continued

properties, execute certain actions. There are, for instance, routers, filters, and transformers that work based on these expressions. Below are a couple of examples of how these expressions can be configured. The first one shows how to use an expression on a filter.

```
<expression-filter evaluator="header"
expression="priority=1"/>
```

This one shows how you can use expressions for routing.

```
 <expression-recipient-list-router
evaluator="xpath"
         expression="/header/routing/recipient" />
```

Available Evaluators

To use an expression, you specify an evaluator (the expression type) and the expression itself.

Evaluator	Description
attachment	Allows you to access an attachment of a message
attachments	Returns a java.util.Map of attachments
attachments-list	Returns a java.util.List of attachments objects
bean	With this property you can access the message using a javabean style
endpoint	Allows you to access endpoint information
exception-type	Allows you to match the type of an exception
function	Performs a function: now, date, datestamp, systime, uuid, hostname, ip, or count. Not supported by expression filters.
groovy	Evaluates the expression using the Groovy language
header	Evaluates the specified part of the message header
headers	Returns all the headers as a java.util.Map
headers-list	Returns all the headers as a java.util.List of header values
jxpath	Allows you to specify an XPath expression that works on XML and javabeans
map-payload	Returns a single value from a Map
message	Gives you access to various message properties: id, correlationId, correlationSequence, correlationGroupSize, replyTo, payload, encoding, exception
mule	Allows access to certain Mule properties: serviceName, modelName, inboundEndpoint, serverId, clusterId, domainId, workingDir and homeDir
ognl	Allows you to use OGNL to access the message
payload	If expression is provided, it will be a class to be class loaded. The class will be the desired return type of the payload.
payload-type	Allows you to check the payload-type of the message
regex	Allows you to use a regular expression to access data
wildcard	You can use a wildcard expression to determine whether a filter matches
xpath	Allows you to use an XPath expression

Most elements allow you to configure the expression using the evaluator and expression attributes. For properties, you can specify multiple expressions using `#[<evaluator>:<expression>]` in Mule 2.1 or `${<evaluator>:<expression>}` in Mule 2.0. For example:

```
<message-properties-transformer>
    <add-property name="GUID" value="#[xpath:/
msg/header/ID]-#[xpath:/msg/body/@ref]"/>
</message-properties-transformer>
```

For more information on expressions you can look at http://www.mulesource.org/display/MULE2USER/Expressions+Configuration+Reference.

TRANSFORMERS

Mule provides a number of transformers which you can use in your own integration flows. Before we look at the transformers provided by Mule, let's first look at how you configure transformers. In the following listings you can see the different ways we can configure and reference a transformer:

```
<custom-transformer class="dzone.CustomTransformer"
         name="myCustomTransformer"/>
<xml:xslt-transformer name="xsltTransformer"
         xsl-file="resources/xslt/transform.xslt"/>
<file:file-to-string-transformer name="fileToString"/>
...
<file:inbound-endpoint name="example-in"
         path="work/example/in"
     transformer-refs="fileToString
         myCustomTransformer xsltTransformer"/>
```

You can add transformers as a transformers-refs attribute to any endpoint. If you want to do this you first have to make sure you've already defined them. The transformers will be executed in the same sequence as they are listed in the attribute. Note that most of the transports have their own default transformer which is executed if you don't specify transformers yourself. If you do specify transformers yourself you have to make sure you also add the default one, which in this case is the `fileToString` transformer.

```
<file:inbound-endpoint name="example-in" path="work/
 example/in">
     <transformer ref="fileToString"/>
     <custom-transformer class="dzone.
      CustomTransformer"/>
     <transformer ref="xsltTransformer"/>
</file:inbound-endpoint>
```

In the previous listing we added the transformers as child elements of the endpoint. This has the same effect as the previous configuration, but now we don't have to define all the transformers before hand, but can define them inline.

```
<file:inbound-endpoint name="example-in" path="work/
 example/in">
   <transformers>
     <transformer ref="fileToString"/>
     <transformer ref="myCustomTransformer"/>
     <custom-transformer class="dzone.
      CustomTransformer"/>
   </transformers>
   <response-transformers>
     <base64-encoder-transformer/>
     <transformer ref="stringToFile"/>
   </response-transformers>
</file:inbound-endpoint>
```

This last configuration is the same as the previous one, but the transformers are now wrapped in a `<transformers>` element. What you also see is that we've added a response-transformers element (which is also available as an attribute `<response-transformers-refs>`). A response-transformer does the same as a normal transformer, but is applied specifically on the response to a synchronous call.

Synchronous or asynchronous

Starting from Mule 2.1 you need to explicitly define whether a message is processed synchronously or asynchronously on both the inbound and outbound endpoints. You can do this by using the synchronous attribute on an endpoint. If you specify `synchronous="true"` Mule will return a result from the call. If you specify `synchronous="false"` no result will be returned. This value defaults to false. So by default Mule operates asynchronously. You can, however, override this by adding `<configuration defaultSynchronousEndpoints="true"/>` to your configuration file.

Available Transformers

The following table lists all the transformers from the Mule core and the Mule XML namespace. They can be used in the manner explained earlier.

Name	Description
`<append-string-transformer/>`	A transformer that appends a string to a string payload
`<auto-transformer>`	A transformer that uses the transform discovery mechanism to convert the message payload
`<custom-transformer>`	Allows you to create a custom transformer
`<message-properties-transformer>`	A transformer that can add or delete message properties
`<no-action-transformer>`	A transformer that does nothing
`<base64-encoder-transformer>`	Transforms a string or byte array to base64
`<base64-decoder-transformer>`	Transforms a base64 message to an array of bytes
`<xml-entity-encoder-transformer>`	A transformer that encodes a string using XML entities
`<xml-entity-decoder-transformer>`	A transformer that decodes a string containing XML entities
`<gzip-compress-transformer>`	A transformer that compresses a byte array using gzip
`<gzip-uncompress-transformer>`	A transformer that uncompresses a byte array using gzip
`<byte-array-to-hex-string-transformer>`	A transformer that converts a byte array to a string of hexadecimal digits
`<hex-string-to-byte-array-transformer>`	A transformer that converts a string of hexadecimal digits to a byte array
`<byte-array-to-object-transformer>`	A transformer that converts a byte array to an object
`<object-to-byte-array-transformer>`	A transformer that serializes all objects
`<object-to-string-transformer>`	A transformer that gives a human-readable description of various types
`<byte-array-to-serializable-transformer>`	A transformer that converts a byte array to an object (deserializing the object)
`<serializable-to-byte-array-transformer>`	A transformer that converts an object to a byte array (serializing the object)
`<byte-array-to-string-transformer>`	A transformer that converts a byte array to a string
`<string-to-byte-array-transformer>`	A transformer that converts a string to a byte array
`<encrypt-transformer>`	A transformer that encrypts a message
`<decrypt-transformer>`	A transformer that decrypts a message
`<expression-transformer>`	A transformer that evaluates one or more expressions on the current event
`<xml:xml-to-dom-transformer>`	Transforms an XML message payload to an org.w3c.dom.Document
`<xml:xml-to-object-transformer>`	Converts XML to Java bean graphs using Xstream
`<xml:xslt-transformer>`	Transformer that uses XSLT to transform the message payload

MULE FILTERS

Mule provides a set of default filters you can use to determine whether a message should be sent to a destination or whether it's read from a destination. Defining a filter works in the same manner as defining a transformer. You can define them globally and reference them from an endpoint.

```
<regex-filter name="regex" pattern="(^my)(.*)(txt$
 )"/>
<custom-filter name="custom" class="dzone.
 CustomFilter"/>
....
<file:inbound-endpoint name="example-in"
 path="work/example/in-1">
       <filter ref="regex"/>
</file:inbound-endpoint>
<file:inbound-endpoint name="example-in-2"
 path="work/example/in-2">
       <payload-type-filter expectedType="java.
        lang.String"/>
</file:inbound-endpoint>
```

In the previous example the message will only be received if it passes the filter. Mule also provides a set of logical filters which you can use to combine filters using **NOT**, **AND** and **OR** semantics.

```
<not-filter>
  <filter ref="custom"/>
</not-filter>
```

```
<and-filter>
  <payload-type-filter expectedType="java.lang.
   String"/>
  <filter ref="regex"/>
</and-filter>
```

```
<or-filter>
  <payload-type-filter expectedType="java.lang.
   String"/>
  <payload-type-filter expectedType="java.lang.
   StringBuffer"/>
</or-filter>
```

Available Filters

The following table shows an overview of all the filters from the core and the XML schema. Note that certain transports have their own custom filters you can use.

Name	Description
<not-filter>	Invert the enclosed filter
<and-filter>	Return true only if all the enclosed filters return true
<or-filter>	Return true if any of the enclosed filters returns true
<wildcard-filter>	Matches String messages against a number of wildcards. For example order.* would match order.line, order.total etc.
<expression-filter>	A filter that evaluates whether a specific expression is valid
<regex-filter>	A filter that matches the message against a regular expression
<exception-type-filter>	A filter that matches the type of an exception
<payload-type-filter>	A filter that matches whether the payload is of the correct class
<custom-filter>	Allows you to implement your own custom filter
<xml:is-xml-filter>	Checks whether the message is an XML message
<xml:jxpath-filter>	Checks the message against an XPath expression using JXPath
<xml:jaxen-filter>	Checks the message against an XPath expression using Jaxen

MULE ROUTERS

Routers are used in Mule to determine how messages are received by a component and to where they are sent after the component has processed them. Mule implements most of the patterns from the *Enterprise Integration Patterns* book (Addison-Wesley), and for most uses the same names. We have inbound routers and outbound routers. In this section, we'll first look at the inbound routers, how to configure them, and which ones are available. After that, we'll look at the outbound routers and do the same thing. First let's look at how to configure an inbound router.

```
<inbound>
  <file:inbound-endpoint path="work/test/in"/>
  <idempotent-secure-hash-receiver-router/>
</inbound>
```

You define the inbound router on the inbound element in the Mule service configuration. This means that every message that is received on any of the inbound endpoints is processed by the inbound router, before it's processed by the configured component.

Available Inbound Routers

Name	Description
<collection-aggregator-router>	Configures a Collection Response Router. This will return a MuleMessageCollection message type that will contain all messages received for each correlation group.
<custom-correlation-aggregator-router>	Allows you to create a custom correlation implementation.
<custom-inbound-router>	With this element you can configure your own custom router.
<forwarding-router>	Forwards a message directly to the outbound router without invoking the component.
<idempotent-receiver-router>	This router makes sure that only unique messages are received. This is done by checking the unique message ID of the message.
<idempotent-secure-hash-receiver-router>	This router generates a hash of the message and uses that to determine whether a message has already been received.
<message-chunking-aggregator-router>	Combines two or more messages into a single message by matching messages with a given Correlation ID
<selective-consumer-router>	Applies one or more filters to the incoming message. If the filters match, the message is forwarded to the component.
<wire-tap-router>	This router allows you to send a copy of a specific message to a certain destination.

Outbound routers are configured on the outbound element:

```
<outbound>
  <static-recipient-list-router>
    <file:outbound-endpoint path="work/example/
     out" />
    <vm:outbound-endpoint path="example.out" />
  </static-recipient-list-router>
</outbound>
```

In this example we define an outbound router on the outbound element, and defined a static-recipient-list-router which sends the message that is received from the component to all the specified endpoints.

Available Outbound Routers

Name	Description
pass-through-router	This router always matches and simply sends or dispatches the message via the endpoint that is configured.

Available Outbound Routers, continued

Name	Description
filtering-router	Uses filters to determine whether the message matches a particular criteria, and if so, will route the message to the endpoint configured on the router.
template-endpoint-router	Allows endpoints to be altered at runtime based on properties set on the current message, or fallback values, set on the endpoint properties.
chaining-router	Sends the message through multiple endpoints using the result of the first invocation as the input for the next.
exception-based-router	Sends a message over an endpoint by selecting the first endpoint that can connect to the transport.
multicasting-router	Sends the same message over multiple endpoints.
endpoint-selector-router	Selects the outgoing endpoint based on the evaluation of an expression.
list-message-splitter-router	Accepts a list of objects that will be routed to different endpoints. The actual endpoint used for each object in the list is determined by a filter configured on the endpoint itself.
expression-splitter-router	Splits the message based on an expression. The expression must return one or more message parts in order to be effective.
message-chunking-router	Allows you to split a single message into a number of fixed-length messages that will all be routed to the same endpoint.
static-recipient-list-router	Sends the same message to multiple endpoints.
expression-recipient-list-router	Sends the same message to multiple endpoints. The destination is determined based on the evaluation of an expression.
custom-outbound-router	This router allows you to define your own custom outbound router.

So far we've seen all the various parts that make up a Mule service except the component which contains the business logic. For this, Mule provides a number of options.

Reuse existing spring configurations

Since Mule is based on Spring it's very easy to reuse your existing spring beans. If you've already got an applicationcontext, and want to reuse those beans from Mule, you can very easily import them. All you have to do is declare the spring namespace, and add the following to your configuration: `<spring:import resource="applicationContext.xml"/>`. Now you can use all the beans defined in that file directly in Mule.

MULE COMPONENTS

There are a number of different ways to configure Mule components. Here, we'll show you, and also explain how Mule determines which method to call on your component. Mule provides two types of elements to use in your configuration to specify the component you want to use. The first one is the `<component>` element:

```
<component class="dzone.Reverser"/>
```

If you use this configuration, Mule will create a new instance of this class for each request which is received. You can also configure Mule to create objects that can be pooled. For this, don't use the `<component>` element, but use the `<pooled-component>` element:

```
<pooled-component class="dzone.Reverser"/>
```

In the previous examples we directly specified the class as an attribute on the elements. We can also use a different way to specify the implementation of the component. You can do this by using any of the following elements inside the `<component>` or the `<pooled-component>` element:

Mule Components, continued

```
<component class>
  <prototype-object class="dzone.Reverser"/>
</component>
<component class>
  <singleton-object class="dzone.Reverser"/>
</component>
<component class>
  <spring-object name="springBean"/>
</component>
```

The first two of these elements allow you to specify whether you want a new object for each message (the `<propotype-object>` element), or whether you want to create an object to be a singleton (`<singleton-object>` element), and reused for all the messages. The final option you can use to specify the implementation of the component is the `<spring-object>` element. Here you can directly reference a spring-bean from the application context.

MULE ENTRY POINT RESOLVING

One thing we haven't discussed yet is how Mule can determine which method to call on your component. Your component often is just a simple spring bean or POJO, which has multiple methods. The default configuration for Mule is to use a set of entry point resolvers to determine which method to call on your bean. Mule uses the following steps to determine which method to invoke on your POJO.

1.	If a property with the name "method" is specified, the value of that property is used to determine the method to invoke on your component. So if you set this (message) property to helloWorld, Mule will look for a method with that name on your bean. This makes use of the `MethodHeaderPropertyEntryPointResolver`.
2.	Mule provides an interface, `org.mule.api.lifecycle.Callable`, you can implement. If Mule finds this interface on your POJO it will invoke the `onCall()` method of this interface, when a message is received for this component. This uses the `CallableEntryPointResolver`.
3.	If there is a transformer configured, Mule will use the return type of this transformer to try and determine if there is a method which accepts this type. If this is found Mule will invoke that method. This uses the `ReflectionEntryPointResolver`.
4.	If there is still no unique match Mule will check the type of the payload to see if that matches any of the methods in the bean. This also uses the `ReflectionEntryPointResolver`.

If the previous steps don't result in a single method, Mule will throw an exception. Beside the ones already mentioned, you can configure your own set of entry point resolvers, should the default configuration be insufficient. The following example shows a custom configuration, which you can configure on the model or on a component.

```
<entry-point-resolver-set>
  <array-entry-point-resolver
      acceptVoidMethods="true"
transformFirst="true"/>
  <callable-entry-point-resolver/>
  <method-entry-point-resolver
      acceptVoidMethods="true"/>
</entry-point-resolver-set>
```

If you create a custom entry-point resolver, you can easily add to this entry point resolver set.

Entry Point Resolvers

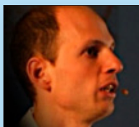
The following table shows an overview of the entry point resolvers which are provided by Mule.

Name	Description
<callable-entry-point-resolver>	An entry point resolver for components that implement the Callable interface.
<custom-entry-point-resolver>	Use to create your own custom implementation.
<property-entry-point-resolver>	Uses a property to determine which method on your component to invoke.
<method-entry-point-resolver>	This uses the "method" property to determine which method to invoke.
<reflection-entry-point-resolver>	Tries to determine the method to invoke based on the payload of the message.
<array-entry-point-resolver>	Checks whether there is a method available which takes a single array as its parameter.
<no-arguments-entry-point-resolver>	Calls a method which has no arguments.

Resources

Open Source Mule site	http://www.mulesource.org
Commercial Mule site	http://www.mulesource.com
Open Source ESB in action website	http://www.esbinaction.com

ABOUT THE AUTHOR



Jos Dirksen

Jos Dirksen is a software architect for Atos Origin, where he has been the architect for a number of large integration projects over the last couple of years. Jos has worked with various integration products, commercial and open source, for the last five years. He co-authored the book *Open Source ESBs in Action*, and regularly presents on topics ranging from enterprise integration patterns to JavaFX, at such conferences as Javapolis and JavaOne.

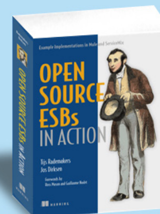
Publications

Open Source ESBs in Action, co-author with Tijs Rademakers, Manning Publications

Website

www.esbinaction.com

RECOMMENDED BOOK



Open-Source ESBs in Action will help you to learn open-source integration technologies quickly and will provide you with knowledge that you can use to effectively work with Mule and ServiceMix.

BUY NOW

books.dzone.com/books/opensource-esb

Get More FREE Refcardz. Visit refcardz.com now!

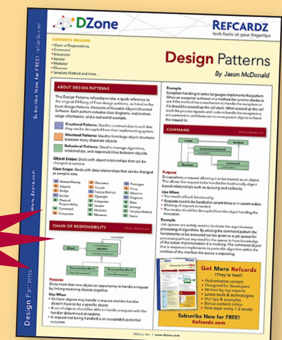
Upcoming Refcardz:

- Getting Started with Equinox OSGi
- SOA Patterns
- Essential EMF
- Windows Presentation Foundation
- HTML and XHTML
- SOA Governance
- Agile Methodologies

Available:

- Core Mule
- Using XML in Java
- Core CSS: Part III
- Essential JSP Expression Language
- Getting Started with Hibernate Search
- Core Seam
- Essential Ruby
- Essential MySQL
- JUnit and EasyMock
- Spring Annotations
- Getting Started with MyEclipse
- Core Java
- Core CSS: Part II
- PHP
- Getting Started with JPA
- JavaServer Faces
- Core CSS: Part I
- Struts2
- Core .NET
- Very First Steps in Flex
- C#

Visit **refcardz.com** for a complete listing of available Refcardz.



Design Patterns
Published June 2008

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-42-4
ISBN-10: 1-934238-42-2
50795
9 781934 238424
$7.95

Copyright © 2008 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference: Open Source ESBs in Action, Tijs Rademakers and Jos Dirksen, Manning Publications, September 2008

Version 1.0