

# Ivan's Mule Examples and Recipes

by

Ivan A Krizsan

Version: April 23, 2012

Copyright 2011-2012 Ivan A Krizsan. All Rights Reserved.

# Table of Contents

<u>Table of Contents</u> .....	2
<u>Purpose</u> .....	9
<u>Structure</u> .....	9
<u>Licensing</u> .....	9
<u>Disclaimers</u> .....	9
<u>Thanks</u> .....	9
<u>Prerequisites</u> .....	10
<u>Part One – The Examples</u> .....	11
<u>1. Standalone Mule Exposing a Web Service</u> .....	12
<u>1.1. Create the Project</u> .....	12
<u>1.2. Create the Web Service Implementation Class</u> .....	12
<u>1.3. Create the Starter Class</u> .....	13
<u>1.4. Create the Mule Configuration Files</u> .....	14
<u>Create the Mule 2.x Configuration File</u> .....	14
<u>Create the Mule 3.x Configuration File</u> .....	16
<u>1.5. Run the Example Starter Program</u> .....	19
<u>Run the Mule 3.x Version</u> .....	19
<u>Run the Mule 2.x Version</u> .....	21
<u>1.6. Run the Mule Configuration Files</u> .....	23
<u>2. Mule in Web Applications</u> .....	24
<u>2.1. Prepare Tomcat for Mule Development</u> .....	25
<u>Add the Necessary Libraries to the Tomcat Server</u> .....	25
<u>Configure a Tomcat Server in Eclipse</u> .....	27
<u>Configuring a Standalone Tomcat Server</u> .....	30
<u>2.2. Tomcat Mule Example Web Application</u> .....	32
<u>Create the Project</u> .....	32
<u>Create the Service Implementation Class</u> .....	33
<u>Create the Mule Configuration Files</u> .....	34
<u>Create the Deployment Descriptor</u> .....	36
<u>Deploy the Web Application</u> .....	38
<u>Test the Service</u> .....	39
<u>3. Modular Mule Configuration</u> .....	40
<u>3.1. Create the Project</u> .....	40
<u>3.2. Create the Service Implementation Class</u> .....	41
<u>3.3. Create the Custom Exception Handler/Listener Classes</u> .....	42
<u>Create the Mule 2.x Custom Exception Listener</u> .....	42
<u>Create the Mule 3.x Custom Exception Handler</u> .....	44
<u>3.4. Create the Mule Configuration Files</u> .....	46
<u>Create the Mule 2.x Configuration Files</u> .....	46
<u>Create the Mule 3.x Configuration Files</u> .....	50
<u>3.5. Run the Example Program</u> .....	53
<u>4. Insert File Data into a Database</u> .....	56
<u>4.1. Create the Project</u> .....	57
<u>4.2. Add Dependencies</u> .....	58
<u>4.3. Database Table Creation Script</u> .....	59
<u>4.4. Create the Spring Bean Configuration File</u> .....	62
<u>4.5. Create the Mule Configuration Files</u> .....	63
<u>Create the Mule File Connector Configuration Files</u> .....	63

<a href="#">Create the Mule JDBC Connector Configuration Files</a>	65
<a href="#">Create the Main Mule Configuration Files</a>	67
4.6. Run the Example Program	71
4.7. Mule 3.x Version with Flow	73
5. Validate XML Data	75
5.1. Create the Project	76
5.2. Create the XML Schemas	76
5.3. Create XML Data Files	78
5.4. Create the Mule Configuration Files	79
<a href="#">Create the Mule File Connector Configuration Files</a>	79
<a href="#">Create the Main Mule Configuration Files</a>	80
<a href="#">The Mule 2.x Configuration File</a>	80
<a href="#">The Mule 3.x Configuration File</a>	84
5.5. Run the Example Program	88
5.6. Validation and XML Schema Imports	90
<a href="#">Modify the XML Schema</a>	90
<a href="#">Implement a Resource Resolver</a>	91
<a href="#">Modify the Schema Validators</a>	93
<a href="#">Run the Example Program</a>	98
6. Extract XML Message Payload with XPath	99
6.1. Create the Project	99
6.2. Create XML Data Files	99
6.3. Create the Mule Configuration Files	100
<a href="#">Create the Mule File Connector Configuration Files</a>	100
<a href="#">Create the Main Mule Configuration Files</a>	101
<a href="#">The Mule 2.x Configuration File</a>	101
<a href="#">The Mule 3.x Configuration File</a>	104
<a href="#">The XPath Expression</a>	106
6.4. Run the Example Program	107
6.5. Exercises	108
7. Monitoring Mule	109
7.1. Create the Project	109
7.2. Create the Web Service Implementation Class	109
7.3. Create the Mule Configuration Files	110
7.4. Run the Example Program	112
<a href="#">Run the Mule 2.x Example Program</a>	112
<a href="#">Run the Mule 3.x Example Program</a>	113
<a href="#">Test the Running Example Program</a>	114
7.5. Managing a Mule Instance Using JMX	115
<a href="#">Run JConsole</a>	115
<a href="#">Generate Some Statistics</a>	116
<a href="#">Starting and Stopping a Mule Instance Using JMX</a>	117
7.6. MX4J and Mule	118
7.7. Monitoring Mule in Web Applications	119
8. Mule Notifications	120
8.1. Create the Project	120
8.2. Create the Service Implementation Class	120
8.3. Create the Notification Listeners	121
<a href="#">Create the Common Notification Listener Base Class</a>	121
<a href="#">Create the Mule 2.x Notification Listener</a>	124

<u>Create the Mule 3.x Notification Listener</u>	125
<b>8.4. Create the Mule Configuration Files</b>	127
<u>Create the Mule 2.x Configuration File</u>	127
<u>Create the Mule 3.x Configuration File</u>	130
<b>8.5. Run the Example Program</b>	133
<u>Run the Mule 2.x Example Program</u>	133
<u>Create a soapUI Client</u>	134
<u>Examine the Mule 2.x Example Program Result</u>	135
<u>Run the Mule 3.x Example Program</u>	137
<u>Examine the Mule 3.x Example Program Result</u>	138
<b>8.6. Additional Exercises</b>	139
<b>9. Exception Handling in Mule</b>	140
<u>9.1. Mule 2.x Configuration Structure</u>	141
<u>9.2. Mule 3.x Configuration Structure</u>	142
<u>9.3. Create the Project</u>	142
<u>9.4. Create the Service Implementation Classes</u>	143
<u>Create the Exception Service Implementation Class</u>	143
<u>Create the Hello Service Implementation Class</u>	144
<u>Create the Logging Service Implementation Class</u>	145
<u>9.5. The Callable Interface</u>	147
<u>9.6. Create the Starter Classes</u>	148
<u>Create the Mule 2.x Starter Class</u>	148
<u>Create the Mule 3.x Starter Class</u>	150
<u>9.7. Create the Exception Listeners</u>	152
<u>Create the Mule 2.x Exception Listener</u>	152
<u>Create the Mule 3.x Exception Listener</u>	154
<u>9.8. Create the Mule Configuration Files</u>	157
<u>Create the Mule 2.x Configuration Files</u>	157
<u>Create the Mule 3.x Configuration File</u>	162
<u>9.9. Run the Example Program</u>	166
<u>Run the Mule 2.x Version of the Example Program</u>	166
<u>Send a Message to the First Endpoint</u>	166
<u>Examine the Output</u>	166
<u>Sending Message to the Second Endpoint</u>	169
<u>Examine the Output</u>	169
<u>Run the Mule 3.x Version of the Example Program</u>	172
<u>Send a Message to the First Endpoint</u>	172
<u>Examine the Output</u>	173
<u>Sending Message to the Second Endpoint</u>	174
<u>Examine the Output</u>	174
<u>9.10. Exercises</u>	176
<b>10. Mule Programmatic Use and Message Properties</b>	177
<u>10.1. Introduction to Message Properties</u>	177
<u>10.2. Create the Project</u>	178
<u>10.3. Create the Mule Configuration Files</u>	178
<u>Mule 2.x Configuration Files</u>	178
<u>Mule 3.x Configuration Files</u>	180
<u>10.4. Create the Starter Classes</u>	182
<u>Create the Mule 2.x Starter Class</u>	182
<u>Create the Mule 3.x Starter Class</u>	186

<u>10.5. Run the Example Program.....</u>	191
<u>Run the Mule 2.x Version of the Example Program.....</u>	191
<u>Running the Mule 3.x Version of the Example Program.....</u>	193
<u>10.6. Exercises.....</u>	194
<u>11. Testing Mule Configurations.....</u>	195
<u>11.1. Create the Project.....</u>	195
<u>11.2. Create the Tests.....</u>	195
<u>Create the Common Test Class.....</u>	195
<u>Create the Mule 2.x Test.....</u>	197
<u>Create the Mule 3.x Test.....</u>	198
<u>11.3. Create the Mule Configuration Files.....</u>	199
<u>Create the Mule 2.x Configuration File.....</u>	199
<u>Create the Mule 3.x Configuration File.....</u>	200
<u>11.4. Create the Service Implementation Classes.....</u>	201
<u>Create the Mule 2.x Service Implementation Class.....</u>	201
<u>Create the Mule 3.x Service Implementation Class.....</u>	202
<u>11.5. Run the Example Program.....</u>	203
<u>Run the Mule 2.x Example Program.....</u>	203
<u>Run the Mule 3.x Example Program.....</u>	204
<u>11.6. Additional Exercises.....</u>	204
<u>12. Create Mule Projects with Maven.....</u>	205
<u>12.1. Prerequisites.....</u>	206
<u>12.2. Create the Project.....</u>	207
<u>12.3. Import Project Into Eclipse.....</u>	208
<u>12.4. Configure Project in Eclipse.....</u>	209
<u>12.5. Use Maven in Eclipse.....</u>	212
<u>Maven Goals.....</u>	212
<u>Create an Eclipse Maven Run Configuration.....</u>	213
<u>13. Mule Configuration Patterns.....</u>	218
<u>13.1. Create the Project.....</u>	218
<u>13.2. The Bridge Pattern.....</u>	219
<u>Synchronous Bridge.....</u>	220
<u>Create Inbound and Outbound Services.....</u>	220
<u>Bridge the Services.....</u>	222
<u>Run the Synchronous Bridge.....</u>	223
<u>Asynchronous Bridge.....</u>	225
<u>Create the Outbound Service.....</u>	225
<u>Bridge the Services.....</u>	226
<u>Run the Asynchronous Bridge.....</u>	227
<u>13.3. The Simple Service Pattern.....</u>	228
<u>JAX-RS Simple Service.....</u>	229
<u>Create the Component Implementation Class.....</u>	229
<u>Modify the Mule Configuration File.....</u>	230
<u>Run the JAX-RS Simple Service.....</u>	231
<u>Simple Services and Inheritance.....</u>	232
<u>Modify the Mule Configuration File.....</u>	232
<u>Run the Simple Service with Inheritance.....</u>	233
<u>JAX-WS Simple Service.....</u>	235
<u>Modify the Mule Configuration File.....</u>	235
<u>Run the JAX-WS Simple Service.....</u>	236

<u>JAXB Simple Service</u>	237
Create JAXB Classes	237
Create Example Data	237
Create the Component Implementation Class	238
Modify the Mule Configuration File	239
Run the JAXB Simple Service	239
<u>XPath Simple Service</u>	241
Create the Component Implementation Class	241
Modify the Mule Configuration File	242
Run the XPath Simple Service	243
<b>13.4. The Validator Pattern</b>	244
First Validator Example	246
Modify the Mule Configuration File	246
Run the First Validator Example	247
Second Validator Example	248
Modify the Mule Configuration File	248
Run the Second Validator Example	250
Third Validator Example	251
Modify the Mule Configuration File	251
Run the Third Validator Example	253
<b>13.5. The Web Service Proxy Pattern</b>	254
Example Preparations	254
Create the Mule Configuration File	254
Implement a Custom Logging Transformer	255
Create the Service WSDL	256
Create the Mock Service in soapUI	258
First Web Service Proxy Example	260
Modify the Mule Configuration File	261
Run the Web Service Proxy Example	262
Second Web Service Proxy Example	265
Create the Double Sum XSL Transform	266
Modify the Mule Configuration File	267
Modify the WSDL File	269
Run the Web Service Proxy Example	269
<b>Part Two – Recipes and Reference</b>	272
<b>1. Message Routing</b>	272
1.1. Selecting Outbound Endpoint Depending on the Message	272
1.2. Routing a Message Depending on a Single Filter	273
1.3. Exception-Dependent Message Routing	274
The Exception-Based Router	274
The First-Successful Message Processor	275
<b>2. Filtering</b>	277
2.1. Validating XML Message Payload	277
2.2. Combining Filters	278
The AND-filter	278
The OR-filter	278
The NOT-filter	278
2.3. Implementing Custom Filters	279
<b>3. Transforming</b>	280
3.1. Extract Part of an XML Message with XPath	280

<a href="#">3.2. Transform XML Data Using XSL</a>	281
<a href="#">3.3. Pack or Unpack Message Payload Data</a>	282
<a href="#">3.4. Custom Transformers</a>	283
<a href="#">Mule 2.x Custom Transformers</a>	283
<a href="#">Mule 3.x Custom Transformers</a>	284
<a href="#">4. Message Properties</a>	285
<a href="#">4.1. Retrieving Message Properties</a>	285
<a href="#">4.2. Setting Message Properties</a>	285
<a href="#">4.3. Removing Message Properties</a>	285
<a href="#">4.4. Renaming Message Properties</a>	286
<a href="#">4.5. Reading and Writing Message Properties to Different Scopes</a>	286
<a href="#">5. Expressions</a>	287
<a href="#">5.1. Evaluators</a>	287
<a href="#">Attachment Evaluator</a>	287
<a href="#">Attachments Evaluator</a>	287
<a href="#">Attachments-List Evaluator</a>	288
<a href="#">Bean Evaluator</a>	288
<a href="#">Endpoint Evaluator</a>	289
<a href="#">Exception-Type Evaluator</a>	289
<a href="#">Function Evaluator</a>	290
<a href="#">Groovy Evaluator</a>	290
<a href="#">Header Evaluator</a>	291
<a href="#">Headers Evaluator</a>	291
<a href="#">Headers-List Evaluator</a>	291
<a href="#">JSON Evaluator</a>	292
<a href="#">JSON-Node Evaluator</a>	292
<a href="#">JXPath Evaluator</a>	292
<a href="#">Map-Payload Evaluator</a>	293
<a href="#">Message Evaluator</a>	293
<a href="#">OGNL Evaluator</a>	294
<a href="#">Payload Evaluator</a>	294
<a href="#">Payload-Type Evaluator</a>	294
<a href="#">Processor Evaluator</a>	294
<a href="#">Regex Evaluator</a>	295
<a href="#">String Evaluator</a>	295
<a href="#">Variable Evaluator</a>	295
<a href="#">Wildcard Evaluator</a>	296
<a href="#">XPath Evaluator</a>	296
<a href="#">XPath-Node Evaluator</a>	297
<a href="#">6. Notifications</a>	298
<a href="#">6.1. Notification Event Types</a>	298
<a href="#">6.2. Notification Listener Interfaces</a>	299
<a href="#">6.3. Notification Events</a>	301
<a href="#">6.4. Listening to Notifications</a>	302
<a href="#">6.5. Listening to Notifications from a Specific Component</a>	303
<a href="#">6.6. Disabling Notifications</a>	304
<a href="#">6.7. Registering a Notification Listener Programmatically</a>	305
<a href="#">7. Mule JMX Management</a>	306
<a href="#">Mule 2.x JMX Management</a>	306
<a href="#">Mule 2.x Server Global Configuration</a>	306

<u>Mule 2.x Connectors Configuration</u>	308
<u>Mule 2.x Endpoint Configuration</u>	309
<u>Mule 2.x Model Configuration</u>	311
<u>Mule 2.x Context Configuration</u>	312
<u>Mule 2.x Notification Configuration</u>	314
<u>Mule 2.x Service Configuration</u>	316
<u>Mule 2.x Statistics Configuration</u>	319
<b>Mule 3.x JMX Management</b>	320
<u>Mule 3.x Application Statistics</u>	321
<u>Mule 3.x Server Global Configuration</u>	322
<u>Mule 3.x Connectors Configuration</u>	323
<u>Mule 3.x Endpoint Configuration</u>	324
<u>Mule 3.x Flow Configuration</u>	325
<u>Mule 3.x Model Configuration</u>	326
<u>Mule 3.x Context Configuration</u>	327
<u>Mule 3.x Statistics Configuration</u>	327
<u>Mule 3.x Notification Configuration</u>	328
<b>8. Package a Mule Application</b>	329
<u>8.1. Package Mule 2.x Applications</u>	329
<u>8.2. Package Mule 3.x Applications</u>	329
<b>9. Testing</b>	330
<u>9.1. Exception Component</u>	330
<u>9.2. Return Mock Data from a Component</u>	332
<u>9.3. Logging Message Details</u>	332
<u>9.4. Retain a Message History</u>	333
<u>9.5. Introduce a Delay</u>	334
<u>9.6. Append Text to Received Messages</u>	334
<b>Appendix A – Prepare for Mule Development</b>	335
<u>1. Download and Install Mule</u>	335
<u>2. Install the Eclipse Mule Plugin</u>	335
<u>3. Configure the Mule Plugin in Eclipse</u>	336
<b>Appendix B – Create a Mule Project</b>	337
<u>1. Create the Project</u>	337
<u>2. Switch off the Mule 3 Hot Deployment Builder</u>	339
<u>3. Create Mule Configuration Files</u>	340
<u>4. Create the Log4J Configuration File</u>	342
<u>5. Change the Mule Distribution of a Project</u>	343
<b>Appendix C – Enabling Maven Dependency Management for an Eclipse Project</b>	345
<b>Appendix D – Mule Standalone Server</b>	348
<u>1. Mule Standalone Server on OS X</u>	348
<u>2. Mule Standalone Server Basic Management</u>	349
<u>2.1. Start and Stop a Mule Server</u>	349
<u>2.2. Deploy and Undeploy a Mule Application</u>	350
<u>Mule 2.x Deployment</u>	350
<u>Mule 3.x Deployment</u>	350
<u>2.3. Undeploy a Mule Application</u>	350
<b>Appendix E – Database Access from within Eclipse</b>	351
<u>1. Data Source Creation</u>	351
<u>2. Data Access</u>	355

# Purpose

This book aims to fill two purposes: first to give an introduction to using the community version of Mule versions 2.x and 3.x, which, as of writing, are versions 2.2.1 and 3.2.0.

Second, this book also aim to serve as a reference containing small examples to serve as solutions to specific problems.

Differences between the two versions of Mule will also be pointed out and discussed.

# Structure

The first part of this book contains complete examples of how to use Mule with brief explanations. The second part of the book contains smaller, incomplete, examples of how to accomplish particular tasks. I have called these examples recipes. Part two also contains some reference information on certain features I have found useful.

Finally there are the appendices that contains information that didn't fit in either of the first two parts, such as how to set up the IDE for Mule development, how to create IDE projects for Mule development etc.

# Licensing

This book is licensed under a [Creative Commons Attribution-Noncommercial-No Derivative Works 3.0](#) license. In short this means that:

- You may share this book with others.
- You may not use this book for commercial purposes.
- You may not create derivate works from this book.

# Disclaimers

Though I have done my best to avoid it, this book might contain errors. I cannot be held responsible for any effects caused, directly or indirectly, by the information in this book – you are using it on your own risk. I cannot make any guarantees concerning the completeness of the information contained in this book.

Submitting any suggestions related to this book the information submitted becomes my property and you give me the right to use the information in whatever way I find suitable, without compensating you in any way. With that said, I will happily credit contributors.

All trademarks are properties of their respective owner and do not imply endorsement of any kind. This book has been written in my spare time and has no connection whatsoever with my employer.

# Thanks

Many thanks to the people on the Mule Users forum for posting and answering questions! I am also very grateful to all the people behind LibreOffice, which I exclusively use when writing my books. Thanks also to the people putting effort in developing Gimp – my tool of choice for editing graphics in this book.

## Prerequisites

This book assumes the following prerequisites:

- The Java 1.6 JDK/JRE or later.
- Experience with the Eclipse IDE or the SpringSource Tool Suite.  
All the examples in this book have been developed in the [SpringSource Tool Suite](#), though a recent version of [Eclipse](#) should suffice.
- Some experience with the [Spring dependency injection framework](#) is assumed.
- Some experience with web services in general and JAX-WS in particular is beneficial.
- Basic experience of testing SOAP web services using [soapUI](#).

This book also assumes that you have prepared for Mule development as described in the appendix [Preparing for Mule Development](#).

## **Part One – The Examples**

The first part of this book contains complete examples of basic Mule usage. These examples are to provide a starting-point for people new to Mule.

The examples just show one way of, for instance, constructing Mule 2.x and Mule 3.x configuration files. This does not mean that this is the only alternative – on the contrary, there are often several ways to accomplish one and the same thing.

Mule configuration files refers to a number of Mule XML schemas. One version of the Mule implementation cannot use Mule configuration files that refer to Mule XML schemas belonging to another version. If you use another version of Mule, you must modify the Mule configuration files accordingly.

## 1. Standalone Mule Exposing a Web Service

In this first example, we'll create a very simple Mule configuration that exposes a SOAP web service using the Apache CXF web service stack. Mule will be run embedded in a standalone Java program.

We will take a first look at Mule configuration files for both the 2.x and 3.x versions of Mule.

### 1.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “SOAPWebServiceInMule”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded.

### 1.2. Create the Web Service Implementation Class

The web service endpoint implementation class implements a service that extends greetings.

- In the source root, create the package `com.ivan.mule`.
- In the new package, create the class `HelloService` with the following contents:

```
package com.ivan.mule;

import java.util.Date;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

/**
 * SOAP web service endpoint implementation class that implements
 * a service that extends greetings.
 */
@WebService
public class HelloService
{
    /**
     * Default constructor. Logs creation of service instances.
     */
    public HelloService()
    {
        System.out.println("***** HelloService instance created.");
    }

    /**
     * Greets the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    @WebResult(name="greeting")
    public String greet(@WebParam(name="name") String inName)
    {
        return "Hello " + inName + ", the time is now " + new Date();
    }
}
```

Note that:

- The `HelloService` class contains a default constructor.  
This is used to show us how many instances of the class is created and not necessary.
- The `HelloService` class contains a number of JAX-WS annotations.  
These are used exactly as when developing JAX-WS web services deployed outside of Mule.

### 1.3. Create the Starter Class

The starter class shows how to programmatically start an embedded instance of Mule. In our case it is not strictly necessary, as it is possible to launch the Mule configuration files directly, as we will see in subsequent chapters.

- In the package *com.ivan.mule*, create a class named *SOAPWebServiceInMuleStarter* with the following contents:

```
package com.ivan.mule;

import org.mule.api.MuleContext;
import org.mule.config.spring.SpringXmlConfigurationBuilder;
import org.mule.context.DefaultMuleContextFactory;

/**
 * Reads Mule configuration file and starts an instance of Mule that is
 * configured accordingly.
 *
 * @author Ivan A Krizsan
 */
public class SOAPWebServiceInMuleStarter
{
    public final static String[] MULE2_CONFIG_FILES =
    {
        "mule-config2.xml"
    };
    public final static String[] MULE3_CONFIG_FILES =
    {
        "mule-config3.xml"
    };

    public static void main(String[] args) throws Exception
    {
        /*
         * Change here to use different configuration file for the
         * Mule context.
         * Remember to change the Mule libraries accordingly!
         */
        String[] theConfigFiles = MULE3_CONFIG_FILES;

        DefaultMuleContextFactory theMuleContextFactory =
            new DefaultMuleContextFactory();
        SpringXmlConfigurationBuilder theSpringConfigBuilder =
            new SpringXmlConfigurationBuilder(theConfigFiles);
        MuleContext theMuleContext =
            theMuleContextFactory.createMuleContext(theSpringConfigBuilder);
        theMuleContext.start();
    }
}
```

Note that:

- There are two constants specifying Mule configuration files.  
As indicated by the names, one is for the 2.x version of Mule and the other for the 3.x version.
- The creation of a Mule context follows these steps:
  - Create a configuration builder.  
The configuration builder is responsible from reading the Mule configuration file(s) needed when creating the Mule context.
  - Create a Mule context factory.
  - Create a Mule context.  
The context factory can, using the configuration builder, create the context.

## 1.4. Create the Mule Configuration Files

Mule configuration files are files in which Mule is told how to receive messages, how to route them to different components and how to transform messages etc.

The main difference between the two versions of Mule we use in this example is the Mule configuration files; one for each version of Mule.

### Create the Mule 2.x Configuration File

The Mule 2.x configuration file is to be located in the root of the source directory and is to be named “mule-config2.xml”. It has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://www.mulesource.org/schema/mule/cxf/2.2"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/cxf/2.2
          http://www.mulesource.org/schema/mule-cxf.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule-stdio.xsd">

    <!--
        A model contains a set of services.
    -->
<model name="GreetingModel">
    <!--
        A service specifies how messages are received, to which
        component the messages are delivered and how the result
        is handled.
    -->
    <service name="GreetingService">
        <!--
            The <inbound> element specifies how the service receives
            messages. In this case, we use Apache CXF on the address
            specified.
        -->
        <inbound>
            <cxf:inbound-endpoint
                address="http://localhost:8182/services/GreetingService" />
        </inbound>

        <!--
            Specifies the component that will be invoked when there is
            an incoming message.
            The component can be one of the following types:
            singleton-object: one single instance handles all messages.
            prototype-object: each instance handles a single message.
            spring-object: messages handled by specified Spring bean.
        -->
        <component>
            <prototype-object class="com.ivan.mule.HelloService" />
        </component>

        <!--
            Specifies where to direct responses.
            Responses are results of incoming messages having been
            processed by the above component.
        -->
        <outbound>
            <pass-through-router>
                <!--
                    Logs messages to the console.
                -->
                <stdio:outbound-endpoint system="OUT" />
            </pass-through-router>
        </outbound>
    </service>
</model>
```

```

</outbound>
</service>
</model>
</mule>

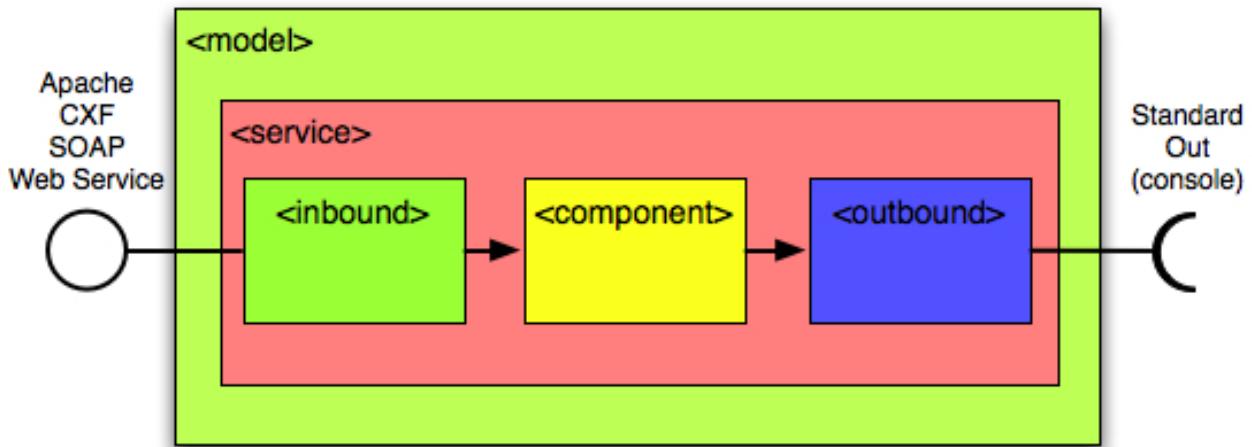
```

Note that:

- The <mule> element contains several XML namespace prefix declarations. These are extensions to the standard Mule XML schema that allows us to declare, in this case, web service endpoints implemented using Apache CXF and endpoints using standard IO (console).
- The <mule> element contains a *schemaLocation* attribute in which the links between namespaces and actual XML schemas are specified. The namespaces and/or the XML schema locations depend on the version of Mule used. If you migrate a Mule configuration file from one version of Mule to another, you will need to modify this section (and perhaps more).
- The <mule> element being the root of the Mule configuraion contains a <model> element. The <model> element contains a set of services. In this example, it only contains a single service.
- The <model> element contains a <service> element. The <service> element specifies how messages are received, a component that is to process messages and how any results are to be handled. The service in this example is to receive messages using an Apache CXF web service, send the messages to an instance of the *HelloService* class for processing and finally send the result to the console. The result from the *HelloService* instance will also become the result of the web service request, despite no explicit configuration stating this.
- The <inbound> element in the <service> element specifies how the service receives messages.
- The <cxft:inbound-endpoint> element in the <inbound> element causes a JAX-WS (SOAP) web service to be exposed by the Apache CXF web service stack. The address at which the service is exposed is specified using the *address* attribute. The default value of the *frontend* attribute is “jaxws”, so we need not specify that this is the one we want to use.
- The <component> element in the <service> element specifies the component that will process received messages and its instantiation-policy. The component can be one of three types:
  - <singleton-object> - one single object processes all messages.
  - <prototype-object> - each message is processed by a dedicated instance.
  - <spring-object> - messages are processed by specified Spring bean.
- The <outbound> element in the <service> element specifies where processed messages are sent. In this example processed messages are printed on the console.
- The <outbound> element contains a <pass-through-router> element. As we saw earlier, the <inbound> element did not contain a router element. It may, but is not required to. The <outbound> element must, however, contain at least one router. The <pass-through-router> routes all messages it receives to the outbound endpoint configured inside the element.

- The <stdio:outbound-endpoint> element directs processed messages to the standard I/O console.

A figure describing the above Mule configuration may look like this:



Visual representation of the Mule 2.x configuration file in this example.

Messages are received by the web service endpoint, passed on to the component for processing. The result is then passed on to the console, passing through an outbound router.

### Create the Mule 3.x Configuration File

The Mule 3.x configuration file is to be located in the root of the source directory and is to be named “mule-config3.xml”. It has the following contents:

```
<?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:cxf="http://www.mulesoft.org/schema/mule/cxf"
    xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/cxf
        http://www.mulesoft.org/schema/mule/cxf/3.2/mule-cxf.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <flow name="GreetingFlow">
        <!--
            An inbound endpoint receives incoming messages on a
            specified address, using the the specified exchange
            pattern.
        -->
        <inbound-endpoint
            address="http://localhost:8182/services/GreetingService"
            exchange-pattern="request-response"/>
        <!--
            Incoming requests are received using a JAX-WS web service
            using the Apache CXF web service stack.
            The CXF jaxws-service honors JAX-WS annotations in the
            service class.
            Note that the serviceClass attribute must specify the
        -->
    </flow>
</mule>
```

```

        class implementing the service.
-->
<cxw:jaxws-service serviceClass="com.ivan.mule.HelloService" />

<!--
    Despite having specified the service class in the above
    element, we must supply a <component> specifying the
    object to be invoked when having received a message.
-->
<component>
    <!--
        The component implementation object can be specified
        either using a Spring bean, a singleton object or
        a prototype object.
        The component can be one of the following types:
        singleton-object: one single instance handles all requests.
        prototype-object: each instance handles a single request.
        spring-object: requests handled by specified Spring bean.
    -->
    <spring-object bean="helloService" />
</component>

<!--
    The log component can be inserted in a flow to log
    the content of a message being passed through the
    log component.
-->
<log-component/>
</flow>

<!--
    In the <spring:beans> element we can declare Spring beans
    and other things that can be done in a regular Spring
    configuration file.
-->
<spring:beans>
    <!--
        This Spring bean supplies the implementation of the
        Hello service.
    -->
    <spring:bean id="helloService" class="com.ivan.mule.HelloService" />
</spring:beans>
</mule>
```

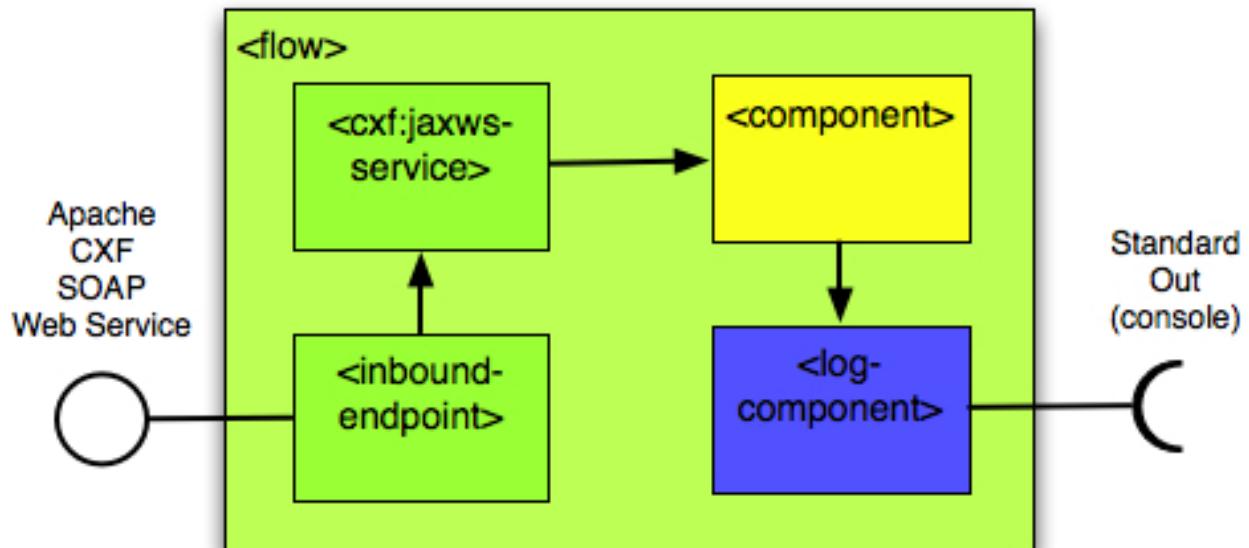
Note that:

- The `<mule>` element contains several XML namespace prefix declarations. These are extensions to the standard Mule XML schema that allows us to declare, in this case, web service endpoints implemented using Apache CXF, endpoints using standard IO (console) and Spring beans.
- The `<mule>` element contains a *schemaLocation* attribute in which the links between namespaces and actual XML schemas are specified. The XML schema locations depend on the version of Mule used. If you migrate a Mule configuration file from one version of Mule to another, you will need to modify this section (and perhaps more).
- The `<mule>` element being the root of the Mule configuration contains a `<flow>` element. A `<flow>` element describes a message processing flow; an optional source from which the flow receives messages, a number of message processors and finally an optional exception handling strategy (not present in this example).
- The `<inbound-endpoint>` element specifies how the flow receives messages. The *address* attribute of the `<inbound-endpoint>` element contains an URI that specifies the protocol and address of the endpoint. The *exchange-pattern* attribute specifies the exchange pattern used when interacting with

the endpoint; either request-response or one-way.

- The `<cxf:jaxws-service>` is used to specify properties of a message processor that is a JAX-WS web service using the Apache CXF web service stack.  
In this example, we have only used the `serviceClass` attribute to specify the annotated JAX-WS endpoint implementation class, but it is also possible to, for instance, configure the use of MTOM, add one or more interceptors etc.
- The `<component>` element specifies a message processor.  
Despite having specified an endpoint implementation class, we must specify a component that processes messages.  
The `<cxf:jaxws-service>` element specifies the class that defines the JAX-WS endpoint and from which a WSDL is generated, but the `<component>` element specifies the class, which in this example is the same class as the service class of the JAX-WS endpoint, that processes messages.
- The `<spring-object>` element in the `<component>` element specifies the Spring bean that is to process messages.
- After the end of the `<flow>` element there is a `<spring:beans>` element.  
This element enables us to embed Spring configuration in a Mule configuration file, as is done with the `helloService` bean.

A figure visualizing the above Mule 3.x configuration file may look like this:



Visual representation of the Mule 3.x configuration file in this example.

## 1.5. Run the Example Starter Program

With the two configuration files in place, we are now ready to run the example starter program. In its original configuration, we will run the example with Mule 3.x.

### Run the Mule 3.x Version

The project in its original incarnation should be configured for using Mule 3.x. To make sure you can use the following checklist:

- The Mule 3.x libraries should be on the project's Java Build Path.
- The starter class *SOAPWebServiceInMuleStarter* should use the Mule 3.x configuration file, as shown in the following code snippet (highlighted in red):

```
...
public static void main(String[] args) throws Exception
{
    /*
     * Change here to use different configuration file for the
     * Mule context.
     * Remember to change the Mule libraries accordingly!
     */
    String[] theConfigFiles = MULE3_CONFIG_FILES;

    DefaultMuleContextFactory theMuleContextFactory =
        new DefaultMuleContextFactory();
...
}
```

To start the program, simply run the *SOAPWebServiceInMuleStarter* class as a Java Application in Eclipse. Log output similar to the following should appear in the console (portions have been excluded to conserve space):

```
...
[10-10 06:45:33] INFO FlowConstructLifecycleManager [main]: Initialising flow: GreetingFlow
[10-10 06:45:33] INFO DefaultMessagingExceptionStrategy [main]: Initialising exception listener: org.mule.exception.DefaultMessagingExceptionStrategy@c4a3158
[10-10 06:45:33] INFO SedaStageLifecycleManager [main]: Initialising service: GreetingFlow.stage1
[10-10 06:45:33] INFO WebServiceFactoryBean [main]: Built CXF Inbound MessageProcessor for service class com.ivan.mule.HelloService
[10-10 06:45:34] INFO ComponentLifecycleManager [main]: Initialising component: component.1753593456
[10-10 06:45:34] INFO ComponentLifecycleManager [main]: Initialising component: component.2142386190
***** HelloService instance created.
[10-10 06:45:34] INFO AutoConfigurationBuilder [main]: Configured Mule using "org.mule.config.spring.SpringXmlConfigurationBuilder" with configuration resource(s): "[ConfigResource{resourceName='mule-config3.xml'}]"
...
[10-10 06:45:34] INFO HttpConnector [main]: Registering listener: GreetingFlow on endpointUri: http://localhost:8182/services/GreetingService
...
[10-10 06:45:35] INFO DefaultMuleContext [main]:
*****
* Mule ESB and Integration Platform
* Version: 3.2.0 Build: 22917
* MuleSoft, Inc.
...
```

There is a lot that can be seen from the above log, but items of main interest for us are:

- An instance of the *HelloService* bean is created.  
The corresponding log output has been highlighted in green.  
Since no scope has been supplied for the Spring bean, it defaults to singleton scope – thus one single instance of the bean will process all messages received.
- The Mule instance was configured using the *SpringXmlConfigurationBuilder* and the configuration file “mule-config3.xml”.  
The corresponding log output has been highlighted in orange.
- An endpoint of the GreetingFlow is registered on the URI  
`http://localhost:8182/services/GreetingService`.  
Thus, the WSDL of the GreetingService endpoint can be found at  
<http://localhost:8182/services/GreetingService?wsdl>  
The corresponding log output has been highlighted in blue.
- 



**Important note!**

When accessing the WSDL of a web service exposed using Mule, the “wsdl” part of the URL, after the question mark, must always be written using lowercase only!

If we use soapUI to send a request to the endpoint of the GreetingService, a response similar to the following should be received:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:greetResponse xmlns:ns2="http://mule.ivan.com/">
      <greeting>Hello Ivan, the time is now Thu Mar 03 07:08:34 CET 2011</greeting>
    </ns2:greetResponse>
  </soap:Body>
</soap:Envelope>
```

In the log, output similar to the following should appear:

```
...
[03-03 07:08:34] INFO LogComponent [connector.http.0.receiver.2]:
*****
* Message received in service: GreetingFlow. Content is: 'Hello Ivan, the
* time is now Thu Mar 03 07:08:34 CET 2011'
```

Subsequent requests causes output similar to the above to be output to the console. We note that no new instances of the *HelloService* are created. This is due to the Spring bean implementing the service being scoped as singleton.

We can see that Mule 3.x does what it is configured to do!

In the next section we will reconfigure the project to use Mule 2.x and run the example program.

## Run the Mule 2.x Version

To reconfigure the project to use the Mule 2.x configuration file and runtime, we need to:

- The Mule 2.x libraries should be on the project's Java Build Path.  
See the section on [Changing the Mule Distribution of a Project](#) in the appendix [Create a Mule Project](#).
- The starter class *SOAPWebServiceInMuleStarter* should use the Mule 2.x configuration file, as shown in the following code snippet (highlighted in red):

```
...
public static void main(String[] args) throws Exception
{
    /*
     * Change here to use different configuration file for the
     * Mule context.
     * Remember to change the Mule libraries accordingly!
     */
    String[] theConfigFiles = MULE2_CONFIG_FILES;

    DefaultMuleContextFactory theMuleContextFactory =
        new DefaultMuleContextFactory();
...
}
```

To start the program, simply run the *SOAPWebServiceInMuleStarter* class as a Java Application in Eclipse. Log output similar to the following should appear in the console (portions have been excluded to conserve space):

```
[03-03 17:12:45] INFO MuleApplicationContext [main]: Refreshing
org.mule.config.spring.MuleApplicationContext@e49d67c:
display name [org.mule.config.spring.MuleApplicationContext@e49d67c]; startup date [Thu
Mar 03 17:12:45 CET 2011]; root of context hierarchy
[03-03 17:12:46] INFO MuleApplicationContext [main]: Bean factory for application
context [org.mule.config.spring.MuleApplicationContext@e49d67c]:
org.springframework.beans.factory.support.DefaultListableBeanFactory@6f57b46f
[03-03 17:12:47] INFO CxfConnector [main]: Initialising: CxfConnector{this=30dc9065,
started=false, initialised=false, name='connector.cxf.0', disposed=false,
numberOfConcurrentTransactedReceivers=4, createMultipleTransactedReceivers=true,
connected=false, supportedProtocols=[cxf, cxf:http, cxf:https, cxf:jms, cxf:vm],
serviceOverrides=null}
...
[03-03 17:12:48] INFO AutoConfigurationBuilder [main]: Configured Mule using
"org.mule.config.spring.SpringXmlConfigurationBuilder" with configuration resource(s):
"[ConfigResource{resourceName='mule-config2.xml'}]"
...
[03-03 17:12:48] INFO CxfConnector [main]: Registering listener: GreetingService on
endpointUri: http://localhost:8182/services/GreetingService
***** HelloService instance created
Mar 3, 2011 5:12:48 PM org.apache.cxf.service.factory.ReflectionServiceFactoryBean
buildServiceFromClass
INFO: Creating Service {http://mule.ivan.com/}HelloServiceService from class
com.ivan.mule.HelloService
Mar 3, 2011 5:12:48 PM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be
http://localhost:8182/services/GreetingService
...
[03-03 17:12:48] INFO HttpConnector [main]: Registering listener:
_cxfServiceComponent{http://mule.ivan.com/}HelloServiceService254355256 on endpointUri:
http://localhost:8182/services/GreetingService
[03-03 17:12:48] INFO HttpMessageReceiver [main]: Connected:
http://localhost:8182/services/GreetingService
*****
* Mule ESB and Integration Platform
* Version: 2.2.1 Build: 14422
* MuleSource, Inc.
* For more information go to http://mule.mulesource.org
...
```

The information of main interest for us in the above log is:

- The Mule instance was configured using the *SpringXmlConfigurationBuilder* and the configuration file “mule-config2.xml”.  
The corresponding log output has been highlighted in orange.
- An endpoint of the GreetingFlow is registered on the URI  
`http://localhost:8182/services/GreetingService`.  
Thus, the WSDL of the GreetingService endpoint can be found at  
<http://localhost:8182/services/GreetingService?wsdl>  
The corresponding log output has been highlighted in blue.  
If we compare the WSDL of the v3.x and v2.x versions of the exposed service, we can see that they are functionally identical.
- An instance of the *HelloService* bean is created.  
The corresponding log output has been highlighted in green.

If we use soapUI to send a request to the endpoint of the GreetingService, a response similar to the following should be received:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:greetResponse xmlns:ns2="http://mule.ivan.com/">
      <greeting>Hello Ivan, the time is now Thu Mar 03 17:35:38 CET 2011</greeting>
    </ns2:greetResponse>
  </soap:Body>
</soap:Envelope>
```

In the log, output similar to the following should appear:

```
...
***** HelloService instance created.
[03-03 17:35:38] INFO StdioMessageDispatcher [connector.stdio.0.dispatcher.1]:
Connected: endpoint.outbound.stdio://system.out
Hello Ivan, the time is now Thu Mar 03 17:35:38 CET 2011
```

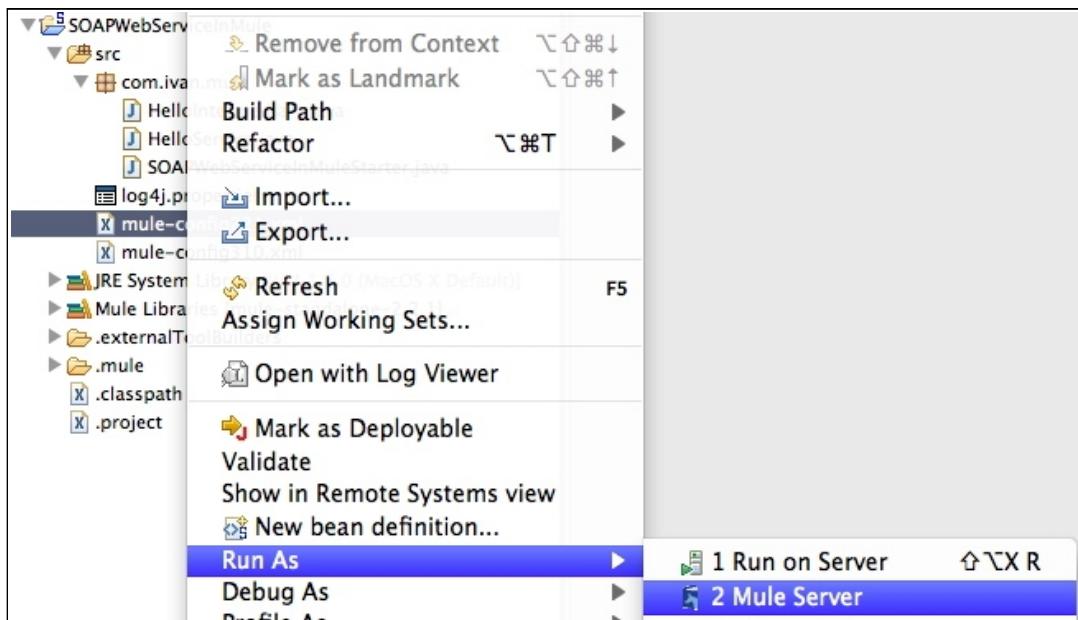
Subsequent requests will cause new instances of *HelloService* to be created and output similar to the above to be output to the console. Since we use a prototype-object for the component processing messages, a new instance will be created for each message received.

We can see that Mule 2.x also does what it is configured to do!

## 1.6. Run the Mule Configuration Files

With the Mule IDE Eclipse plugin installed, we do not need a starter program – it is possible to “run” the Mule configuration files directly in Eclipse.

- Depending on the Mule distribution on the project's classpath, select a Mule configuration file.  
That is, if you have Mule 3.x libraries on the classpath, then select the mule-config3.xml configuration file, if you have Mule 2.x libraries on the classpath, then use the mule-config2.xml file.
- Right-click the selected Mule configuration file and select Run As -> Mule Server.



Running a Mule configuration file in Eclipse without a starter program.

- Observe the console.  
There should be console output similar to that we saw when starting Mule using the starter program.
- Try the web service using soapUI.  
The SOAP web service should, of course, be up and running as earlier.

This concludes the example showing how to run Mule in a standalone program. As a bonus we also saw how to run Mule in the Eclipse development environment.

## **2. Mule in Web Applications**

In this chapter we will look at how to run a Mule instance embedded in the Tomcat web container. Since I began writing this chapter just before the release of Mule 3.2.0 community version, I tried the examples on three versions of Mule; 2.2.1, 3.1.0 and 3.2.0. Since Mule 3.1.0 proved to require special treatment, I decided to keep the notes about Mule 3.1.0 in this chapter.

There are three ways Mule can be used in a container:

- Embedded in a Tomcat instance.

All Mule configuration files from different WARs use a common Mule codebase.

Requests over HTTP can be directed to pass through a servlet that routes requests to the appropriate service. This prevents us from starting a Mule server within Tomcat.

One Mule context is created per web application that contains one or more Mule configuration files.

- Embedded in an instance of a JCA 1.5 compliant container.

- Embedded in a web application.

The web application contains the necessary Mule libraries. Each such application runs its own Mule codebase. This use case is very similar to using Mule in a standalone application and so no example will be given.

This chapter's examples will use Tomcat 7.0. In addition to the server and the appropriate Mule distributions, we will also need the following libraries:

- Mule 2.x only:

“commons-logging-1.1.jar” JAR file. It can be downloaded from [here](#).

- Mule 3.1.0 only:

The Jackson distribution enclosed with Mule seems too old and must be replaced with a newer version. Use version 1.8.0 or later. Download Jackson [here](#).

I will use version 1.8.6, which is the latest stable version of the 1.8 branch, in this chapter.

- Mule 2.x only:

The mule-module-tomcat-2.2.9-SNAPSHOT.jar JAR file can be downloaded [here](#).

## 2.1. Prepare Tomcat for Mule Development

The following steps assume that you have downloaded and installed [Tomcat](#). As before, I have chosen to use Tomcat 7.0, but any fairly recent version should work too.

If you intend to work through both the Mule 2.x and Mule 3.x examples of this chapter, you should first prepare Tomcat for Mule 2.x development according to the instructions below and then come back for a second iteration for Mule 3.x.

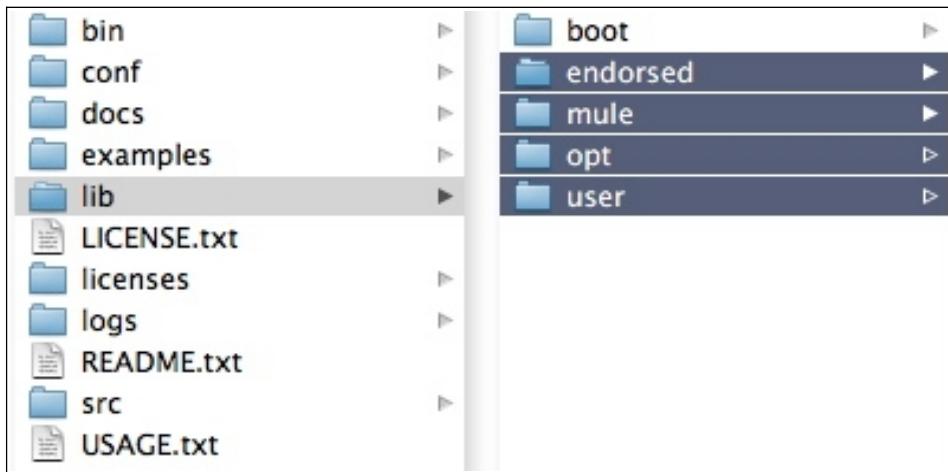
### Add the Necessary Libraries to the Tomcat Server

First we'll add the necessary libraries to the Tomcat instance in which we are to develop this chapter's example program. This is a common step that needs to be undertaken regardless of whether you are running Tomcat from within Eclipse or standalone.

In the following instructions, \${TOMCAT\_HOME} refers to the Tomcat installation directory and \${MULE\_HOME} refers to the root directory of the Mule binary distribution.

- In the \${TOMCAT\_HOME} directory, create a directory named “mule-libs”.
- Copy the contents of the \${MULE\_HOME}/lib directory, excluding the “boot” directory to the \${TOMCAT\_HOME}/mule-libs directory created in the previous step.

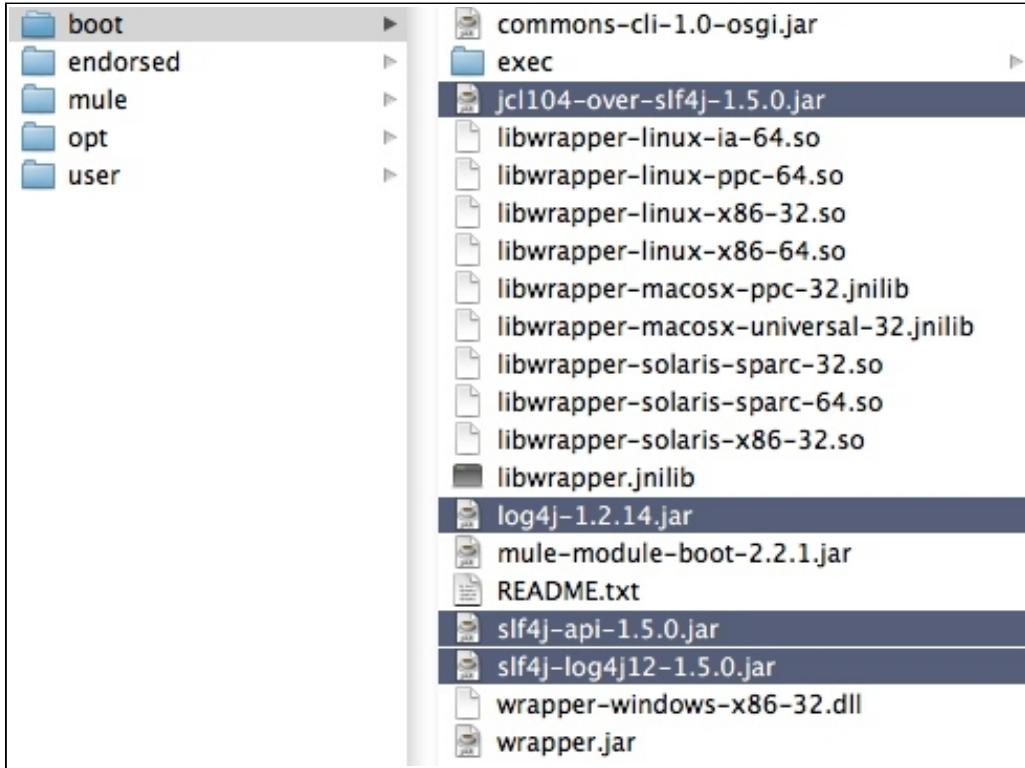
The picture below shows the directories to copy.



Copy the directories in the Mule distribution to the mule-libs directory in the Tomcat installation.

- Mule 2.x only:  
Copy the mule-module-tomcat-2.2.9-SNAPSHOT.jar JAR file to the \${TOMCAT\_HOME}/mule-libs/mule directory.
- Mule 3.x only:  
Copy the wrapper-3.2.3.jar JAR file from the \${MULE\_HOME}/lib/boot directory to the \${TOMCAT\_HOME}/mule-libs/opt directory.
- Mule 3.1.0 only:  
Delete the following JAR files from the \${TOMCAT\_HOME}/mule-libs/opt directory:  
jackson-core-asl-1.3.1.jar, jackson-jaxrs-1.3.1.jar, jackson-mapper-asl-1.3.1.jar  
Copy the following JAR files to the \${TOMCAT\_HOME}/mule-libs/opt directory:  
jackson-core-asl-1.8.6.jar, jackson-jaxrs-1.8.6.jar, jackson-mapper-asl-1.8.6.jar, jackson-xc-1.8.6.jar.

- Mule 2.x and 3.1.0:  
From the \${MULE\_HOME}/lib/boot directory, copy the following JAR files to the \${TOMCAT\_HOME}/mule-libs/opt directory:  
jcl104-over-slf4j-1.5.0.jar, log4j-1.2.14.jar, slf4j-api-1.5.0.jar, slf4j-log4j12-1.5.0.jar.



Copy the above JAR files from the Mule distribution to the mule-libs/opt directory in the Tomcat installation.

- Mule 2.x only:  
Copy the commons-logging-1.1.jar JAR file to the \${TOMCAT\_HOME}/mule-libs/opt directory.

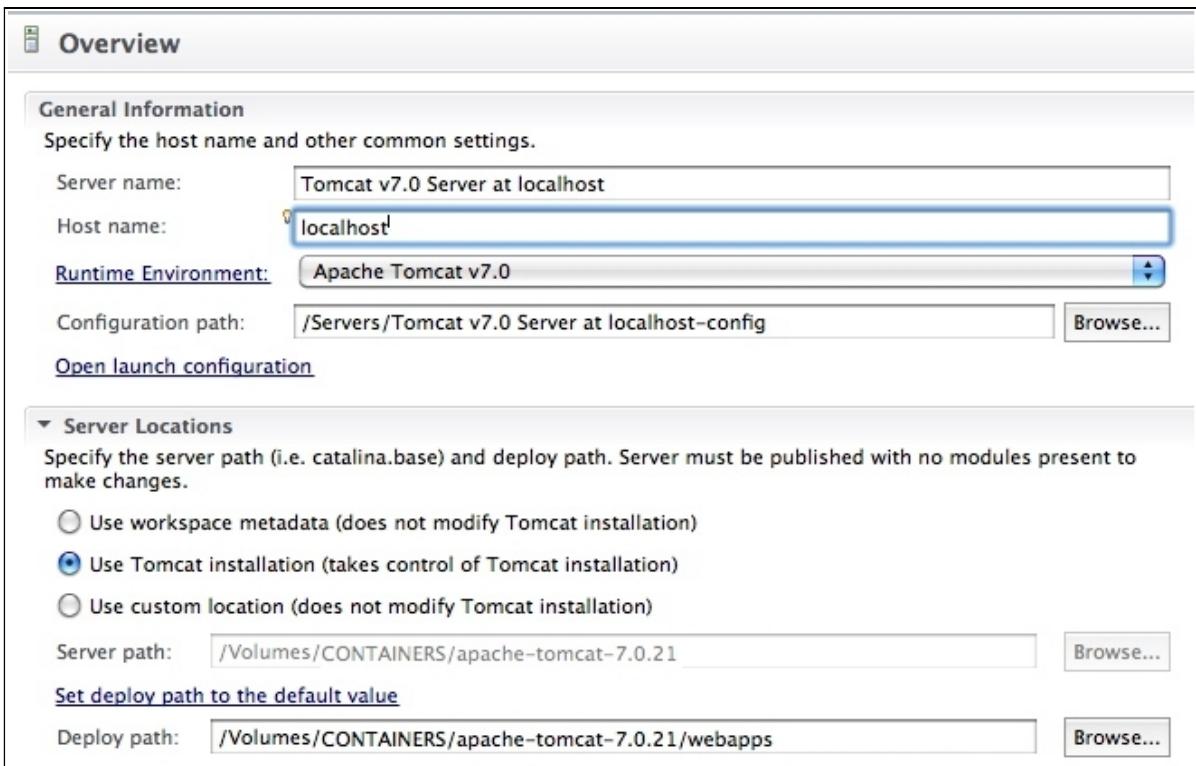
The necessary libraries are now in place and we are ready to configure the Tomcat instance.

## Configure a Tomcat Server in Eclipse

First we'll configure the Tomcat server instance for use from within Eclipse.

The following steps can be skipped if you do not intend to run Tomcat from within Eclipse.

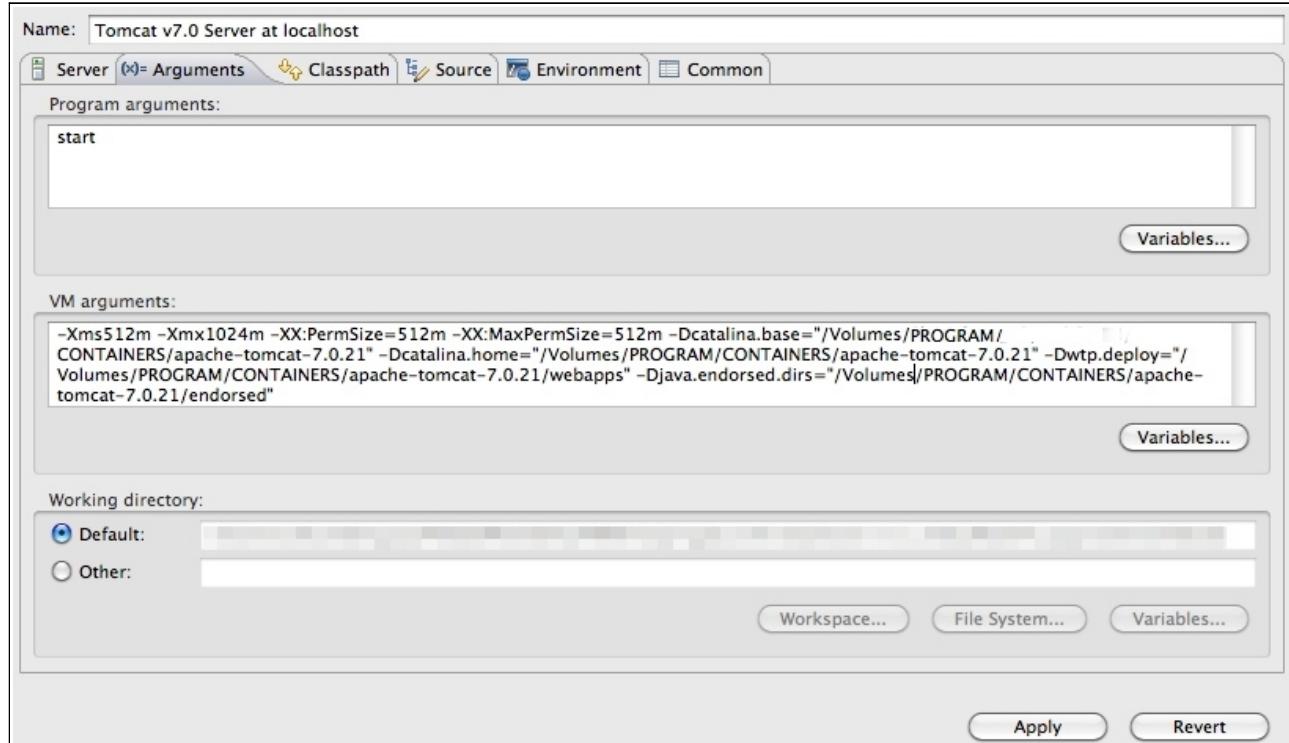
- If you haven't already created a Server in the Servers view in Eclipse, create one.
- Double-click on the Tomcat Server in the Servers view in Eclipse.
- This step is optional and its purpose is to configure Eclipse to deploy web applications to the Tomcat installation folder instead of maintaining an Eclipse-specific deployment directory. In the Server Locations, select the radio-button “Use Tomcat installation (takes control of Tomcat installation)” and set the Deploy path to the “webapps” directory in the Tomcat installation directory.



Tomcat server configuration in Eclipse.

- In the Timeouts section enter 80 in the field “Start (in seconds)”. If the Tomcat server times out when starting up, this value have to be increased.
- Save the server configuration.
- Right-click the Tomcat instance in the Servers view and select Publish.
- In the Tomcat server configuration, click the underlined text “Open launch configuration”. We need to allocate more memory, otherwise web applications using Mule 3.x will run out of memory.
- In the launch configuration window, click the Arguments tab.

- In the “VM arguments” field, insert the following parameters first in the parameter string:  
`-Xms512m -Xmx1024m -XX:PermSize=512m -XX:MaxPermSize=512m`  
If any of the above parameters are already present, remove the old value.



Adding memory configuration VM parameters in the Tomcat launch configuration.

- In the Package Explorer, locate the Servers folder and expand the node for the Tomcat instance you intend to use for Mule development.



Locating the server settings files for the Tomcat instance in Eclipse.

(continued on next page)

- Double-click the file server.xml and add the <Listener> child element below to the <Server> element in the file.

```
<Listener className="org.mule.module.tomcat.MuleTomcatListener" />
```

The relevant segment of the server.xml file should look like this, with the new <Listener> element highlighted:

```
...
<Server port="8005" shutdown="SHUTDOWN">
    <!-- Security listener. Documentation at /docs/config/listeners.html
    <Listener className="org.apache.catalina.security.SecurityListener" />
    -->
    <!--APR library loader. Documentation at /docs/apr.html -->
    <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
    <!--Initialize Jasper prior to webapps are loaded. Documentation at /docs/jasper-
howto.html -->
    <Listener className="org.apache.catalina.core.JasperListener" />
    <!-- Prevent memory leaks due to use of particular java/javax APIs-->
    <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
    <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
    <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

    <!-- Mule loader. -->
    <Listener className="org.mule.module.tomcat.MuleTomcatListener" />

    <!-- Global JNDI resources
        Documentation at /docs/jndi-resources-howto.html
    -->
    <GlobalNamingResources>
    ...

```

- Save the modified server.xml file.
- Double-click the catalina.properties file and append the following string to the value of the common.loader property (note the comma first in the string):  
,\$ {catalina.home}/mule-libs/user/\*.jar,\$ {catalina.home}/mule-libs/mule/\*.jar, \$ {catalina.home}/mule-libs/opt/\*.jar

The result should look like this, with the modified property highlighted:

```
...
#
# List of comma-separated paths defining the contents of the "common"
# classloader. Prefixes should be used to define what is the repository type.
# Path may be relative to the CATALINA_HOME or CATALINA_BASE path or absolute.
# If left as blank, the JVM system loader will be used as Catalina's "common"
# loader.
# Examples:
#     "foo": Add this folder as a class repository
#     "foo/*.jar": Add all the JARs of the specified folder as class
#                   repositories
#     "foo/bar.jar": Add bar.jar as a class repository
common.loader=${catalina.base}/lib,$ {catalina.base}/lib/*.jar,$ {catalina.home}/lib,$
{catalina.home}/lib/*.jar,$ {catalina.home}/mule-libs/user/*.jar,$ {catalina.home}/mule-
libs/mule/*.jar, $ {catalina.home}/mule-libs/opt/*.jar
...
```

- Save the modified catalina.properties file.

The configuration of the Tomcat server, when run from within Eclipse, is now finished.

## Configuring a Standalone Tomcat Server

Next we'll configure the Tomcat instance for use from outside of Eclipse. Examples of such use is when Tomcat is started as a service or from the terminal.

The following steps can be skipped if you only intend to run the Tomcat instance from within Eclipse.

In the following instructions, \${TOMCAT\_HOME} refers to the Tomcat installation directory.

- Open the file \${TOMCAT\_HOME}/conf/server.xml and add the <Listener> child element below to the <Server> element in the file.

```
<Listener className="org.mule.module.tomcat.MuleTomcatListener" />
```

The result should look like this (the new <Listener> element highlighted):

```
...
<Server port="8005" shutdown="SHUTDOWN">
    <!-- Security listener. Documentation at /docs/config/listeners.html
    <Listener className="org.apache.catalina.security.SecurityListener" />
    -->
    <!--APR library loader. Documentation at /docs/apr.html -->
    <Listener className="org.apache.catalina.core.AprLifecycleListener" SSLEngine="on" />
    <!--Initialize Jasper prior to webapps are loaded. Documentation at /docs/jasper-
howto.html -->
    <Listener className="org.apache.catalina.core.JasperListener" />
    <!-- Prevent memory leaks due to use of particular java/javax APIs-->
    <Listener className="org.apache.catalina.core.JreMemoryLeakPreventionListener" />
    <Listener className="org.apache.catalina.mbeans.GlobalResourcesLifecycleListener" />
    <Listener className="org.apache.catalina.core.ThreadLocalLeakPreventionListener" />

    <!-- Mule loader. -->
    <Listener className="org.mule.module.tomcat.MuleTomcatListener" />

    <!-- Global JNDI resources
        Documentation at /docs/jndi-resources-howto.html
    -->
    <GlobalNamingResources>
    ...

```

- Save the modified server.xml file.
- Open the file \${TOMCAT\_HOME}/conf/catalina.properties and append the following string to the value of the common.loader property:  
,\$ {catalina.home}/mule-libs/user/\*.jar,\$ {catalina.home}/mule-libs/mule/\*.jar, \$ {catalina.home}/mule-libs/opt/\*.jar

The result should look like this (modified property highlighted):

```
...
#
#
# List of comma-separated paths defining the contents of the "common"
# classloader. Prefixes should be used to define what is the repository type.
# Path may be relative to the CATALINA_HOME or CATALINA_BASE path or absolute.
# If left as blank, the JVM system loader will be used as Catalina's "common"
# loader.
# Examples:
#     "foo": Add this folder as a class repository
#     "foo/*.jar": Add all the JARs of the specified folder as class
#                 repositories
#     "foo/bar.jar": Add bar.jar as a class repository
common.loader=${catalina.base}/lib,$ {catalina.base}/lib/*.jar,$ {catalina.home}/lib,$
{catalina.home}/lib/*.jar,$ {catalina.home}/mule-libs/user/*.jar,$ {catalina.home}/mule-
libs/mule/*.jar, $ {catalina.home}/mule-libs/opt/*.jar
...
```

- Save the modified catalina.properties file.

Mule 3 consumes more memory than Mule 2, so prior to launching Tomcat we must give it more memory. Do this by issuing the, depending on your operating system, appropriate terminal command:

- Unix and similar, including OS X:

```
export CATALINA_OPTS="-Xms512m -Xmx1024m -XX:PermSize=512m  
-XX:MaxPermSize=512m"
```

- DOS-based operating systems:

```
set CATALINA_OPTS="-Xms512m -Xmx1024m -XX:PermSize=512m  
-XX:MaxPermSize=512m"
```

We are now finished setting up the Tomcat server for Mule development. In the next section we'll develop a web application that contains a Mule configuration file and deploy it to the Tomcat server.

## 2.2. Tomcat Mule Example Web Application

In this section we will look at how to develop a web application that use Mule to expose a SOAP web service and deploy it to a Tomcat instance prepared as described [earlier](#).

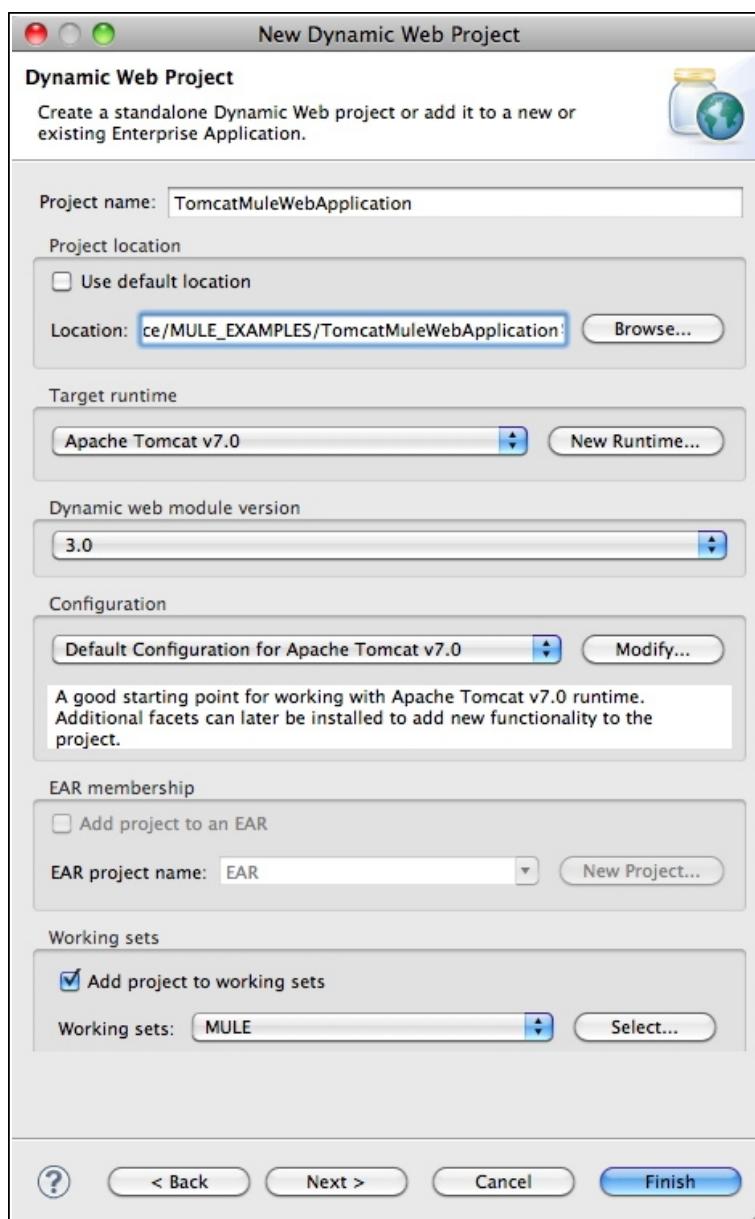
Mule will be configured to receive requests to the web service over HTTP using a servlet that runs in Tomcat, that is, no additional server will be started.

### Create the Project

The kind of Eclipse project used in this example is a regular dynamic web project project with a regular web.xml deployment descriptor.

- In Eclipse, create a Dynamic Web Project.

I call my project “TomcatMuleWebApplication”. It is to be targeted at the Apache Tomcat instance that we prepared for Mule development earlier.



Creating a dynamic web project in Eclipse.

- Click the Finish button in the lower right corner.

That is actually all we need to do when creating the project.

There are some special configuration required in the web.xml deployment descriptor that we will look at later.

### **Create the Service Implementation Class**

The service implementation class is the service implementation class for the HelloService developed in chapter 1. No special considerations need to be made when deploying the service implementation class to a Tomcat server.

```
package com.ivan.mule;

import java.util.Date;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

/**
 * SOAP web service endpoint implementation class that implements
 * a service that extends greetings.
 *
 * @author Ivan A Krizsan
 */
@WebService
public class HelloService
{
    /**
     * Default constructor.
     * Logs creation of service instances.
     */
    public HelloService()
    {
        System.out.println("***** HelloService instance created.");
    }

    /**
     * Greets the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    @WebResult(name="greeting")
    public String greet(@WebParam(name="name") String inName)
    {
        return "Hello " + inName + ", the time is now " +
               new Date();
    }
}
```

## Create the Mule Configuration Files

The Mule configurations files used in this example are identical to those used in the [previous chapter](#), except for one small modification we need to do when deploying to a web application. In this example, I have placed the Mule configuration files in the package com.ivan.mule, next to the service implementation class in order to show how the location of the Mule configuration file can be configured when deploying to Tomcat.

- In the package com.ivan.mule, create a file named “mule-config2.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://www.mulesource.org/schema/mule/cxf/2.2"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/cxf/2.2
          http://www.mulesource.org/schema/mule/cxf/2.2/mule-cxf.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd">

    <!--
        A model contains a set of services.
    -->
<model name="GreetingModel">
    <!--
        A service specifies how messages are received, to which
        component the messages are delivered and how the result
        is handled.
    -->
<service name="GreetingService">
    <!--
        The <inbound> element specifies how the service receives
        messages.
        Note that the address of the inbound endpoint has been changed.
        This has been done in order to use the server this web application
        is deployed to and avoid starting another server.
        To access the WSDL of the greeting service, use the
        following URL:
        http://localhost:8080/MuleWebApplication/mule/services/GreetingService?
wsdl
    -->
    <inbound>
        <cxf:inbound-endpoint address="servlet://GreetingService"/>
    </inbound>

    <!--
        Specifies the component that will be invoked when there is
        an incoming message.
        The component can be one of the following types:
        singleton-object: one single instance handles all messages.
        prototype-object: each instance handles a single message.
        spring-object: messages handled by specified Spring bean.
    -->
    <component>
        <prototype-object class="com.ivan.mule.HelloService"/>
    </component>

    <!--
        Specifies where to direct responses.
        Responses are results of incoming messages having been
        processed by the above component.
    -->
    <outbound>
        <pass-through-router>
            <!--
                Logs messages to the console.
            -->
    </outbound>
</model>
```

```

-->
<stdio:outbound-endpoint system="OUT" />
</pass-through-router>
</outbound>
</service>
</model>
</mule>

```

- In the package com.ivan.mule, create a file named “mule-config3.xml” with the following contents:

```

<?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:cxf="http://www.mulesoft.org/schema/mule/cxf"
    xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/cxf
        http://www.mulesoft.org/schema/mule-cxf.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <flow name="GreetingFlow">
        <!--
            The <inbound> element specifies how the service receives
            messages.
            Note that the address of the inbound endpoint has been changed.
            This has been done in order to use the server this web application
            is deployed to and avoid starting another server.
            To access the WSDL of the greeting service, use the
            following URL:
            http://localhost:8080/MuleWebApplication/mule/services/GreetingService?wsdl
        -->
        <inbound-endpoint
            address="servlet://GreetingService"
            exchange-pattern="request-response"/>
        <!--
            Incoming requests are received using a JAX-WS web service
            using the Apache CXF web service stack.
            The CXF jaxws-service honors JAX-WS annotations in the
            service class.
            Note that the serviceClass attribute must specify the
            class implementing the service.
        -->
        <cxf:jaxws-service serviceClass="com.ivan.mule.HelloService" />

        <!--
            Despite having specified the service class in the above
            element, we must supply a <component> specifying the
            object to be invoked when having received a message.
        -->
        <component>
            <!--
                The component implementation object can be specified
                either using a Spring bean, a singleton object or
                a prototype object.
                The component can be one of the following types:
                singleton-object: one single instance handles all requests.
                prototype-object: each instance handles a single request.
                spring-object: requests handled by specified Spring bean.
            -->
            <spring-object bean="helloService" />
        </component>
    <!--

```

```

The log component can be inserted in a flow to log
the content of a message being passed through the
log component.
-->
<log-component/>
</flow>

<!--
In the <spring:beans> element we can declare Spring beans
and other things that can be done in a regular Spring
configuration file.
-->
<spring:beans>
<!--
This Spring bean supplies the implementation of the
Hello service.
-->
<spring:bean id="helloService" class="com.ivan.mule.HelloService"/>
</spring:beans>
</mule>

```

Note that:

- The *address* attribute of the `<cxfrs:inbound-endpoint>` element in both the configuration files has the value “`servlet://GreetingService`”.
- The prefix “`servlet://`” tells Mule that we want to use the servlet transport and registers the endpoint with the servlet integrated with the servlet container.
- This facilitates integration with the servlet container, Tomcat in our case, and prevents the opening of an additional port for communication with the Mule endpoint.
- It is possible to use the same configuration for the endpoint address as in a standalone application.
- If you want to deploy the Mule configuration file to Mule 3.1.0, you need to replace occurrences of “`3.2`” in the *schemaLocation* attribute with “`3.1`”.

## Create the Deployment Descriptor

When deploying to a web container like Tomcat, we need to configure the web application deployment descriptor in order to start Mule and to tell Mule where our Mule configuration file(s) are located.

- In the Eclipse project, in the WebContent/WEB-INF directory, create a file named “`web.xml`” with the following contents:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="
        http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    id="WebApp_ID" version="3.0">

    <display-name>TomcatMuleWebApplication</display-name>

    <!--
        Tell Mule where the Mule configuration file is located.
        More than one configuration file can be supplied by separating
        the filenames with commas.
    -->
    <context-param>
        <param-name>org.mule.config</param-name>
        <param-value>com/ivan/mule/mule-config2.xml</param-value>
    </context-param>

```

```

<!--
    Servlet context listener that loads the Mule configuration file(s)
    and creates a MuleContext.
-->
<listener>
    <listener-class>
        org.mule.config.builders.MuleXmlBuilderContextListener
    </listener-class>
</listener>

<!--
    Receives HTTP requests via a servlet and routes them to services
    that has an address starting with servlet://, that is that uses
    the servlet transport.
    This approach facilitates closer integration with the servlet
    container, Tomcat in our case.
-->
<servlet>
    <servlet-name>muleServlet</servlet-name>
    <servlet-class>org.mule.transport.servlet.MuleReceiverServlet</servlet-class>
    <load-on-startup>100</load-on-startup>
</servlet>

<!--
    Set the URL pattern used to access exposed Mule services.
    The URL pattern is appended to the server address and the web
    application context.
-->
<servlet-mapping>
    <servlet-name>muleServlet</servlet-name>
    <url-pattern>/mule/services/*</url-pattern>
</servlet-mapping>

<welcome-file-list>
    <welcome-file>index.html</welcome-file>
</welcome-file-list>
</web-app>

```

Note that:

- The deployment descriptor contains a context parameter with the name “org.mule.config” and the value “com/ivan/mule/mule-config2.xml”.  
This context parameter is used to inform Mule about the name and location of our configuration file.  
Multiple Mule configuration files may be specified by separating the names with commas. When running the Mule 3.x version of the example, change the value of this context parameter to “com/ivan/mule/mule-config3.xml”. This is the only change required to switch between the two versions, apart from the libraries in Tomcat.
- There is a <listener> element in the deployment descriptor that contains a <listener-class> element with the value “org.mule.config.builders.MuleXmlBuilderContextListener”.  
This servlet context listener is responsible for finding Mule configuration files, using for instance the context parameter discussed above, and creating a Mule context using these configuration files.
- There is a <servlet> element that configures a servlet named “muleServlet”.  
This configuration is optional and only necessary if any endpoints in our Mule configuration file uses the servlet transport discussed above in connection to the Mule configuration file.
- There is a <servlet-mapping> element that maps the “muleServlet” to the URL pattern “/mule/services/\*”.  
This servlet mapping specifies the common URL prefix for endpoints using the servlet transport. We will look more closely at the URL of the service in our example after having deployed the web application.

## **Deploy the Web Application**

With the Mule configuration file, the service implementation class and the web application deployment descriptor in place, we are now ready to deploy the web application.

- Start the Tomcat server which has been prepared for Mule development.  
In Eclipse, right-click the server in the Servers view and select Start.  
In a standalone Tomcat server, navigate to the “bin” directory in the Tomcat installation directory and execute the script “startup.sh”, or “startup.bat” if you are using a DOS-based operating system.
- Deploy the web application to the Tomcat server.  
In Eclipse, right-click the project in the Package or Project explorer view and select Run As->Run On Server. In the dialog that appears, make sure the appropriate Tomcat server is selected and click the Finish button.  
If you are asked whether to restart the server, select “Continue without restarting”.  
Console output similar to the following should be generated:

```
...
Sep 30, 2011 3:46:41 PM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deploying web application directory TomcatMuleWebApplication
Sep 30, 2011 3:46:54 PM org.apache.cxf.bus.spring.BusApplicationContext
getConfigResources
INFO: No cxf.xml configuration file detected, relying on defaults.
***** HelloService instance created.
Sep 30, 2011 3:46:56 PM org.apache.cxf.service.factory.ReflectionServiceFactoryBean
buildServiceFromClass
INFO: Creating Service {http://mule.ivan.com/}HelloServiceService from class
com.ivan.mule.HelloService
Sep 30, 2011 3:46:56 PM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be servlet://GreetingService
...
...
```

Note that an instance of the HelloService endpoint implementation class was created as part of deploying the web application.

In addition, an error may be logged:

```
...
[09-30 15:46:56] ERROR ServiceService [Thread-9]: Error post-registering the MBean
javax.management.MalformedObjectNameException: Invalid character ':' in value part of
property
...
...
```

This error occurs when trying to register an MBean for JMX management with an illegal character in one of the property names. The error does not affect the functionality of our example program and can be ignored.

## **Test the Service**

With the web application successfully deployed, we should now be able to view the WSDL of the service exposed using Mule.

- In a web browser, open the following URL:  
<http://localhost:8080/TomcatMuleWebApplication/mule/services/GreetingService?wsdl>  
The WSDL should appear.
- Note how the URL of the service is created by appending “mule/services” to the web application URL and then the name of the service, “GreetingService”, as specified in the Mule configuration file.
- Using soapUI, test the service and make sure you receive a greeting in the following format from the service:

```
Hello Ivan, the time is now Mon Oct 03 05:33:33 CEST 2011
```

The service has an URL that “is part of” the web application, that is uses the same port and context root as the web application. In addition, and also significant to some extent, the service answers requests as expected.

If you want to embed Mule in the web application, instead of in the Tomcat server, then copy the Mule libraries mentioned [above](#) to the library folder of the web application (WEB-INF/lib) instead of copying them to the Tomcat server. Also remember that no configuration of the Tomcat server is necessary in this case.

If you wish to repeat the exercise for Mule 3.x, do the following:

- Stop the Tomcat server if it is running.
- Remove the Mule 2.x libraries from Tomcat.
- Add the Mule 3.x libraries to Tomcat, as described [above](#).
- Change the Mule configuration file used in the web.xml deployment descriptor.
- Restart Tomcat and redeploy the web application.

This concludes the example showing how to develop and deploy a web application that uses Mule to a Tomcat server.

### **3. Modular Mule Configuration**

The example program in this chapter will prompt the user for console input, reverse the entered string and print it on the console. Occasionally, which in this case means randomly, the string processing will throw an exception.

We will look at the following in various degree of detail:

- Modularizing Mule configuration files.  
Not entirely obvious in such a small example, but dividing Mule configuration in multiple files can help us organize things.
- Define parent-child relationships between models.  
As in object-oriented programming, this technique can be used to refactor out properties common to several modules and avoid duplication.
- Implement custom exception handling.  
In Mule 2.x, we will use a custom exception listener, and in Mule 3.x we will use a custom exception handler.  
The example program will give a brief introduction to exception handling in Mule. This subject has a chapter of its own devoted to it [later](#) in this book.

We will use the technique of running the Mule configuration files, described in a [previous chapter](#), so we will not need a starter class.

#### **3.1. Create the Project**

Create the project as described in the appendix [Create a Mule Project](#), naming it “MuleModularConfiguration”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded.

### 3.2. Create the Service Implementation Class

The web service endpoint implementation class implements a service that extends greetings.

- In the source root, create the package *com.ivan.mule*.
- In the new package, create the class *ReverseStringService* with the following contents:

```
package com.ivan.mule;

/**
 * Service that reverses strings.
 *
 * @author Ivan A Krizsan
 */
public class ReverseStringService
{
    /**
     * Default constructor.
     * Logs creation of service instances.
     */
    public ReverseStringService()
    {
        System.out.println("***** ReverseStringService instance created.");
    }

    /**
     * Reverses the string with the supplied name.
     *
     * @param inString String to reverse.
     * @return Reversed string.
     */
    public String reverse(String inString)
    {
        /* Sometimes an error occurs. */
        if (Math.random() > 0.8)
        {
            throw new Error("An exception occurred in the reverse-service");
        }

        StringBuffer theBuf = new StringBuffer(inString);
        return theBuf.reverse().toString();
    }
}
```

Note that:

- The *ReverseStringService* class is completely decoupled from anything related to Mule. This gives us the freedom to, using Mule, expose the service in different ways, without having to change the service implementation.

### 3.3. Create the Custom Exception Handler/Listener Classes

There are two different custom exception handler/listener classes in the example program; one for Mule 2.x and one for Mule 3.x. This is due to the fact that custom exception handlers/listeners implement different interfaces in Mule 2.x and Mule 3.x.

Both the exception listener/handler classes have a property that is set at creation time and printed whenever there is an exception. This is just to show how exception handlers/listeners can be customized with properties and how these are configured in the Mule configuration file.

When switching to the Mule 2.x runtime libraries, there will be compilation errors in the Mule 3 exception handler class. This is normal and should be ignored.

#### Create the Mule 2.x Custom Exception Listener

The Mule 2.x exception listener class is named *MyMule2ExceptionListener* and implemented as follows:

```
package com.ivan.mule;

import java.beans.ExceptionListener;

/**
 * Custom Mule 2 exception listener.
 *
 * @author Ivan A Krizsan
 */
public class MyMule2ExceptionListener implements ExceptionListener
{
    private String mListenerProperty;

    /**
     * Default constructor.
     * Logs creation of instances of the exception listener.
     */
    public MyMule2ExceptionListener()
    {
        System.out.println("**** MyMule2ExceptionListener created");
    }

    /* (non-Javadoc)
     * @see java.beans.ExceptionListener#exceptionThrown(java.lang.Exception)
     */
    @Override
    public void exceptionThrown(final Exception inException)
    {
        System.out.println("**** MyMule2ExceptionListener.exceptionThrown: " +
                           inException.getLocalizedMessage());
        System.out.println("    Listener property: " + mListenerProperty);
    }

    public String getListenerProperty()
    {
        return mListenerProperty;
    }

    public void setListenerProperty(String inListenerProperty)
    {
        System.out.println(
            "**** Setting MyMule2ExceptionListener.listenerProperty: " +
            inListenerProperty);
        mListenerProperty = inListenerProperty;
    }
}
```

Note that:

- The class implements the *ExceptionListener* interface.  
This interface is part of the Java SE API and contains one single method - the *exceptionThrown* method.
- The *exceptionThrown* method takes a single parameter and has a void return type.  
The parameter is the exception that caused the exception listener to be invoked.
- The *AbstractExceptionListener* class adds several callback methods that are invoked depending on what kind of exception is thrown.
- The class has an instance variable; *mListenerProperty* and associated getter and setter methods.  
As with any other Java bean, this exposes a property named “*listenerProperty*”. We use this property to identify a certain instance of the exception listener.
- The class contains a default constructor.  
While a default constructor is not required, we implement one to see when the listener is instantiated.
- For a more in-depth discussion on implementing custom Mule 2.x exception listeners, please refer to [this](#) section in the chapter on [Exception Handling in Mule](#).

## Create the Mule 3.x Custom Exception Handler

The Mule 3.x exception listener class is named *MyMule3ExceptionHandler* and implemented as follows:

```
package com.ivan.mule;

import org.mule.api.MuleEvent;
import org.mule.api.exception.MessagingExceptionHandler;

/**
 * A custom Mule 3 exception handler.
 *
 * @author Ivan A Krizsan
 */
public class MyMule3ExceptionHandler implements MessagingExceptionHandler
{
    private String mListenerProperty;

    /**
     * Default constructor.
     * Logs creation of instances of the exception handler.
     */
    public MyMule3ExceptionHandler()
    {
        System.out.println("**** MyMule3ExceptionHandler created");
    }

    /* (non-Javadoc)
     * @see
     */
    @Override
    public MuleEvent handleException(Exception inException, MuleEvent inEvent)
    {
        System.out.println("**** MyMule3ExceptionHandler.exceptionThrown: " +
                           inException.getLocalizedMessage());
        System.out.println("    Listener property: " + mListenerProperty);

        return inEvent;
    }

    public String getListenerProperty()
    {
        return mListenerProperty;
    }

    public void setListenerProperty(String inListenerProperty)
    {
        System.out.println(
            "**** Setting MyMule3ExceptionHandler.listenerProperty: " +
            inListenerProperty);
        mListenerProperty = inListenerProperty;
    }
}
```

Note that:

- The class implements the *MessagingExceptionHandler* interface.  
This interface is part of the Mule 3.x API and contains one single method - the *handleException* method.  
Having your exception listener implement this interface is the most basic approach to implementing an exception listener.
- The *handleException* method takes two parameters..  
The first parameter is the exception that caused the exception listener to be invoked.  
The second parameter is a Mule event object representing the Mule event that was processed

when the exception occurred.

- The *handleException* method returns a Mule event object.  
The event object returned is the event that is to continue to be routed through the remaining part of the flow.
- The exception handler is invoked when exceptions occur during processing of a message.  
Compare this to the Mule 2.x exception listener and the Mule 3.x interface *SystemExceptionHandler*. The latter is to be implemented by Mule 3.x exception handlers that are to handle exceptions that occur when a message is not processed.
- The class has an instance variable; *mListenerProperty* and associated getter and setter methods.  
As with any other Java bean, this exposes a property named “listenerProperty”.
- The exception listener class has a default constructor.  
It is not required, but implement to see when the listener is instantiated.
- For a more in-depth discussion on implementing custom Mule 3.x exception listeners, please refer to [this](#) section in the chapter on [Exception Handling in Mule](#).

### 3.4. Create the Mule Configuration Files

This example will, as mentioned earlier, modularize the Mule configuration. Thus, we will create several configuration files; three for Mule 2.x and two for Mule 3.x. The configuration files will have the following contents:

- Main configuration file.  
Includes the other configuration file(s) and defines the main module/flow.
- Global connector configuration file.  
Configures the STDIO transport connector.
- Parent model configuration file (Mule 2.x only in this example).  
Configures an abstract parent model with custom exception handling. In a larger system, this module could be the parent of all modules that wanted to have custom exception handling. Technically, this is also possible with Mule 3.x, but we use a flow instead of a module in the Mule 3.x configuration and flows do not support inheritance like modules do.

#### Create the Mule 2.x Configuration Files

There are three Mule 2.x configuration files, all located in the root of the source directory, named “mule-config2.xml”, “mule2-global-config.xml” and “mule2-parentmodel-config.xml”.

The “mule-config2.xml” file imports the two other files and defines the model:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd
          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--
        Import configuration files that this configuration depends on.
        Note that the imports must be wrapped in a <spring:beans> element.
        Also note that it is possible to import Mule configuration files,
        not only Spring bean configuration files.
    -->
    <spring:beans>
        <spring:import resource="mule2-global-config.xml"/>
        <spring:import resource="mule2-parentmodel-config.xml"/>
    </spring:beans>

    <model name="echoModel" inherit="true">
        <service name="echoServiceWithReverse">
            <inbound>
                <!--
                    Inbound endpoint receiving text from the standard
                    IO console.
                    Use a connector defined in a separate configuration file.
                -->
                <stdio:inbound-endpoint
                    system="IN"
                    connector-ref="SystemStreamConnector"/>
            </inbound>

            <component>
                <singleton-object class="com.ivan.mule.ReverseStringService"/>
            </component>
        </service>
    </model>

```

```

<outbound>
    <pass-through-router>
        <stdio:outbound-endpoint system="OUT" />
    </pass-through-router>
</outbound>
</service>
</model>
</mule>

```

Note that:

- The configuration file is a regular Mule 2.x configuration file.
- Spring <import> elements are used to import the other two configuration files. Both Mule configuration files as well as Spring bean configuration files can be imported.
- The Spring <import> elements are wrapped in a Spring <beans> element.
- The <model> element has, as we will later see, the same name as the parent <model> element it inherits from.  
In addition, the child <model> element also sets the *inherit* attribute to true.
- The <model> element contains a <stdio:inbound-endpoint> element.  
This element is used to specify how the flow receives messages.
- The <stdio:inbound-endpoint> element contains a *connector-ref* attribute.  
An endpoint is a concretization of a connector. The *connector-ref* attribute is used to specify the connector which the endpoint concretizes. If there is only one connector for a particular protocol, Mule will automatically find the appropriate connector. Thus, in this example, it is not strictly necessary to specify the connector reference.
- The connector *SystemStreamConnector* is not defined in the above configuration file.  
This connector is, as we will shortly see, defined in a separate configuration file which could be included by several configuration files.  
This allows us to avoid repetition and make modifications to this particular connector easier.

The next file, “mule2-global-config.xml”, contains the definition of the *SystemStreamConnector* that we saw in the above configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule
    xmlns="http://www.mulesource.org/schema/mule/core/2.2"
    xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.mulesource.org/schema/mule/core/2.2
        http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

        http://www.mulesource.org/schema/mule/stdio/2.2
        http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd">

    <!--
        Configure the STDIO transport connector to use a specific
        prompt when reading data and to output a message to the console
        when a message, the read data, is sent.
    -->
<stdio:connector
    name="SystemStreamConnector"
    promptMessage="Please enter text: "
    messageDelayTime="200"
    outputMessage="STDIO connector sending string. "/>
</mule>
```

Note that:

- The `<stdio:connector>` element is an immediate child of the `<mule>` element.  
Mule allows us to configure connectors, filters, transformers etc in separate configuration files that can later be reused.
- The `<stdio:connector>` element contains the *name* attribute with the value “SystemStreamConnector”.  
As might be suspected, this gives the connector the name by which it may be referred to (as seen earlier with the *connector-ref* attribute).
- The `<stdio:connector>` element contains a *promptMessage* attribute.  
As we will see when we run the example program, this attribute can be used to specify a message that is to be displayed when the connector asks for input.
- The `<stdio:connector>` element contains a *messageDelayTime* attribute.  
This attribute specifies the time in milliseconds that the connector will wait before printing the prompt message.
- The `<stdio:connector>` element contains an *outputMessage* attribute.  
This attribute specifies a message that will be printed to the console when, in the case of this example, the input entered by the user is sent to the next processing stage.

The final Mule 2.x configuration file, “mule2-parentmodel-config.xml”, contains the declaration of a model that specifies a custom exception-handling strategy:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule
    xmlns="http://www.mulesource.org/schema/mule/core/2.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.mulesource.org/schema/mule/core/2.2
        http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

        http://www.mulesource.org/schema/mule/stdio/2.2
        http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--
        Model specifying a custom exception handling strategy.
        This model can be used as a parent model which other models
        can inherit from. They will thus all have the same custom
        exception handling strategy.
    -->
    <model name="echoModel">
        <custom-exception-strategy class="com.ivan.mule.MyMule2ExceptionListener">
            <spring:property name="listenerProperty" value="SomeData"/>
        </custom-exception-strategy>
    </model>
</mule>
```

Note that:

- The name of the model is “echoModel”.  
All child models must have the same name, which we saw in the first Mule 2.x configuration file.
- The model does not contain a `<service>` element.  
Since this model is not intended to be used on its own, it only needs to contain elements specifying what is to become common properties with its child-models.
- The model contains a `<custom-exception-strategy>` element.  
This element tells Mule to use the custom exception-listener class that we implemented [earlier](#).
- The `<custom-exception-strategy>` element contains a `<spring:property>` element.  
The `<spring:property>` element allows us to inject a value into an instance field of the exception-listener class whenever an instance is created.

## Create the Mule 3.x Configuration Files

There are two Mule 3.x configuration files, located in the root of the source directory, named “mule-config3.xml” and “mule3-global-config.xml”.

The “mule-config3.xml” file imports the other configuration file and defines a flow:

```
<?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:cxf="http://www.mulesoft.org/schema/mule/cxf"
    xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/cxf
        http://www.mulesoft.org/schema/mule/cxf/3.2/mule-cxf.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!--
        Import configuration files that this configuration depends on.
        Note that the imports must be wrapped in a <spring:beans> element.
        Also note that it is possible to import Mule configuration files,
        not only Spring bean configuration files.
    -->
    <spring:beans>
        <spring:import resource="mule3-global-config.xml"/>
    </spring:beans>

    <flow name="GreetingFlow">
        <!--
            Inbound endpoint receiving text from the standard
            IO console.
            Use a connector defined in a separate configuration file.
        -->
        <stdio:inbound-endpoint system="IN" connector-ref="ConsoleInputConnector"/>

        <component>
            <singleton-object class="com.ivan.mule.ReverseStringService"/>
        </component>

        <stdio:outbound-endpoint system="OUT"/>

        <!--
            Flows cannot inherit from other flows, so we specify the
            custom exception handling strategy for the flow.
        -->
        <custom-exception-strategy class="com.ivan.mule.MyMule3ExceptionHandler">
            <spring:property name="listenerProperty" value="SomeData"/>
        </custom-exception-strategy>
    </flow>
</mule>
```

(continued on next page)

Note that:

- The configuration file is a regular Mule 3.x configuration file.
- Spring <import> elements are used to import the other configuration file.  
Both Mule configuration files as well as Spring bean configuration files can be imported.
- The Spring <import> element is wrapped in a Spring <beans> element.
- The <flow> element contains a <stdio:inbound-endpoint> element.  
The inbound endpoint element specifies how the flow receives messages.
- The <stdio:inbound-endpoint> element contains a *connector-ref* attribute.  
An endpoint is a concretization of a connector. The *connector-ref* attribute is used to specify the connector which the endpoint concretizes. If there is only one connector for a particular protocol, Mule will automatically find the appropriate connector. Thus, in this example, it is not strictly necessary to specify the connector reference.
- The connector *ConsoleInputConnector* is not defined in the above configuration file.  
This connector is, as we will shortly see, defined in a separate configuration file which could be included by several configuration files.
- The <flow> element contains a <custom-exception-strategy> element.  
Since one flow cannot inherit from another flow, we have to specify the exception strategy like this.
- The <custom-exception-strategy> element contains a <spring:property> element.  
The <spring:property> element allows us to inject a value into an instance field of the exception-listener class whenever an instance is created.

The next file, “mule3-global-config.xml”, is identical to the Mule 2.x counterpart except for the namespace declarations and contains the definition of the *SystemStreamConnector* that we saw in the above configuration file:

```
<?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:stdio="http://www.mulesoft.org/schema/mule/stdio"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

          http://www.mulesoft.org/schema/mule/stdio
          http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd">

    <!--
        Configure the STDIO transport connector to use a specific
        prompt when reading data and to output a message to the console
        when a message, the read data, is sent.
    -->
    <stdio:connector
        name="ConsoleInputConnector"
        promptMessage="Please enter text: "
        messageDelayTime="200"
        outputMessage="STDIO connector sending string. "/>
</mule>
```

Note that:

- The `<stdio:connector>` element is an immediate child of the `<mule>` element.  
Mule allows us to configure connectors, filters, transformers etc in separate configuration files that can later be reused.
- The `<stdio:connector>` element contains the *name* attribute with the value “*SystemStreamConnector*”.  
As might be suspected, this gives the connector the name by which it may be referred to (as seen earlier with the *connector-ref* attribute).
- The `<stdio:connector>` element contains a *promptMessage* attribute.  
As we will see when we run the example program, this attribute can be used to specify a message that is to be displayed when the connector asks for input.
- The `<stdio:connector>` element contains a *messageDelayTime* attribute.  
This attribute specifies the time in milliseconds that the connector will wait before printing the prompt message.
- The `<stdio:connector>` element contains an *outputMessage* attribute.  
This attribute specifies a message that will be printed to the console when, in the case of this example, the input entered by the user is sent to the next processing stage.

### 3.5. Run the Example Program

We are now ready to run the example program. We'll use the technique of running the Mule configuration files as described in the [earlier example](#).

It is assumed that the project is configured with the Mule 3.x distribution on the classpath when starting this section.

- In the Package or Project Explorer, right-click the “mule-config3.xml” file and select Run As -> Mule Server.
- The Mule 3.x server should start up and display something similar to this on the console (some output omitted to conserve space):

```
...
INFO 2011-03-17 17:36:47,595 [main] org.mule.lifecycle.AbstractLifecycleManager:
Initialising connector: ConsoleInputConnector
*** MyMule3ExceptionHandler created
*** Setting MyMule3ExceptionHandler.listenerProperty: SomeData
INFO 2011-03-17 17:36:47,748 [main] org.mule.construct.FlowConstructLifecycleManager:
Initialising flow: GreetingFlow
...
INFO 2011-03-17 17:36:48,017 [main] org.mule.component.ComponentLifecycleManager:
Starting component: component.1300307500
***** ReverseStringService instance created.
INFO 2011-03-17 17:36:48,029 [main] org.mule.transport.stdio.PromptStdioConnector:
Registering listener: GreetingFlow on endpointUri: stdio://system.in
...
INFO 2011-03-17 17:36:48,242 [main] org.mule.DefaultMuleContext:
*****
* Mule ESB and Integration Platform *
* Version: 3.2.0 Build: 22917 *
...
* Agents Running: *
* JMX Agent *
*****
Please enter text:
```

Note that:

- An instance of the custom exception handler class is created and the value of the property *listenerProperty* is set to “SomeData” (highlighted in yellow).
- An instance of the *ReverseStringService* class is created (highlighted in green).
- The GreetingFlow is registered on an endpoint with the URI “stdio://system.in” (highlighted in turquoise).

The endpoint providing input to the flow will thus take its input from the System.in stream.

(continued on next page)

Continuing with the example:

- Enter some text at the prompt and press return.
- Output similar to the following should be generated on the console:

```
...
Please enter text: SomeText
INFO 2011-03-17 17:40:01,324 [ConsoleInputConnector.dispatcher.1]
org.mule.lifecycle.AbstractLifecycleManager: Initialising:
'ConsoleInputConnector.dispatcher.599179652'. Object is: StdioMessageDispatcher
INFO 2011-03-17 17:40:01,325 [ConsoleInputConnector.dispatcher.1]
org.mule.lifecycle.AbstractLifecycleManager: Starting
: 'ConsoleInputConnector.dispatcher.599179652'. Object is: StdioMessageDispatcher
STDIO connector sending string. txeTemoS
Please enter text:
...
```

Note that:

- The string “STDIO connector sending string” is output on the console (highlighted in red), indicating that the connector passes on a message to the next stage in the flow.
- The reversed string I entered (highlighted in green) is output on the console.

Again, continuing with the example:

- Enter more text at the prompt and press return.  
Subsequent output will only consist of the message from the connector and the reversed string entered.
- Continue to enter text at the prompt until a message from the custom exception handler is shown:

```
...
Please enter text: MoreText
*** MyMule3ExceptionHandler.exceptionThrown: Component that caused exception is:
org.mule.component.DefaultJavaComponent component for:
SimpleFlowConstruct{GreetingFlow}. Message payload is of type: String
Listener property: SomeData

Please enter text:
...
```

Note that:

- The custom exception handler is invoked and outputs a message (highlighted in green).
- The value contained in the property of the custom exception handler is output to the console (highlighted in red).  
As expected, it contains the value with which it was initialized.

The example program works as we hoped it would, with multiple configuration files, a custom exception handler and a customized connector.

When you are done inputting text, terminate the program.

We now go on to run the Mule 2.x version of the example program:

- Change the Mule distribution of the project, as described in the appendix [Create a Mule Project](#).

As before, this will result in a compilation error in the *MyMule3ExceptionHandler* class but this should be ignored.

- Run the “mule-config2.xml” configuration file by right-clicking it and selecting Run As -> Mule Server.  
Ignore any warnings about errors in the project and proceed with launching the program.
- In the console, we should see result similar to that of the Mule 3.x version, with some variations:

```
INFO 2011-03-17 18:22:20,555 [main] org.mule.MuleServer: Mule Server initializing...
...
INFO 2011-03-17 18:22:23,792 [main] org.mule.DefaultExceptionStrategy: Initialising exception listener: org.mule.DefaultExceptionStrategy@2825491d
*** MyMule2ExceptionListener created
*** Setting MyMule2ExceptionListener.listenerProperty: SomeData
***** ReverseStringService instance created.
INFO 2011-03-17 18:22:23,845 [main] org.mule.component.DefaultJavaComponent: Initialising: org.mule.component.DefaultJavaComponent component for: SedaService{null}
...
INFO 2011-03-17 18:22:23,886 [main] org.mule.MuleServer: Mule Server starting...
*** MyMule2ExceptionListener created
*** Setting MyMule2ExceptionListener.listenerProperty: SomeData
***** ReverseStringService instance created.
INFO 2011-03-17 18:22:23,910 [main] org.mule.transport.stdio.PromptStdioConnector: Connected: PromptStdioConnector{this=578b1f8f, started=false, initialised=true, name='SystemStreamConnector', disposed=false, numberOfConcurrentTransactedReceivers=4, createMultipleTransactedReceivers=true, connected=true, supportedProtocols=[stdio], serviceOverrides=null}
...
INFO 2011-03-17 18:22:24,007 [main] org.mule.util.queue.TransactionalQueueManager: Started ResourceManager
INFO 2011-03-17 18:22:24,056 [main] org.mule.DefaultMuleContext:
*****
* Mule ESB and Integration Platform *
* Version: 2.2.1 Build: 14422 *
...
* Agents Running: None *
*****
Please enter text: SomeText
INFO 2011-03-17 18:22:42,411 [SystemStreamConnector.dispatcher.1]
org.mule.transport.stdio.StdioMessageDispatcher: Connected:
endpoint.outbound.stdio://system.out
STDIO connector sending string. txeTemoS
Please enter text: MoreText
INFO 2011-03-17 18:22:50,599 [SystemStreamConnector.dispatcher.2]
org.mule.transport.stdio.StdioMessageDispatcher: Connected:
endpoint.outbound.stdio://system.out
STDIO connector sending string. txeTeroM
Please enter text: MoreText
*** MyMule2ExceptionListener.exceptionThrown: Component that caused exception is: SedaService{echoServiceWithReverse}. Message payload is of type: String
    Listener property: SomeData
Please enter text:
```

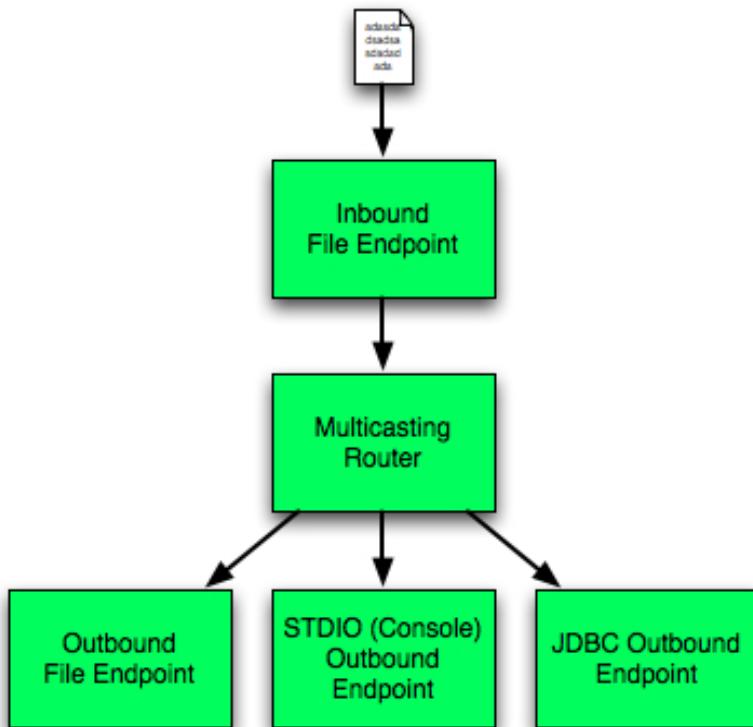
This concludes the example showing, among other things, how to modularize Mule configuration files and how to customize exception handling.

## 4. Insert File Data into a Database

In this chapter we will look at an example showing how to insert data from XML files into a table in a database using Mule. We will also look very briefly at message routing.

Earlier examples have had distinctly different Mule 2.x and Mule 3.x configuration files; usually using a model with Mule 2.x and a flow with Mule 3.x. In this chapter we will use configuration files that are identical, except for the XML namespace declarations to show that it is also possible to configure Mule 3.x in a manner identical to that used with Mule 2.x.

A graphical representation of this chapter's example program may look like this:



This chapter's example program consists of one inbound endpoint, a router and three outbound endpoints.

Again, we will use the technique of running the Mule configuration files, described in the previous chapter, so we will not need a starter class.

The example program will utilize a PostgreSQL database with the default account “postgres” with the password “adminadmin” (password chosen during installation) that has all privileges. You are free to use any database setup you desire, but must adapt the example accordingly.

If you haven't already, download and install the latest version of the PostgreSQL database from <http://www.postgresql.org/>.

We will use the Eclipse Data Source Explorer to examine data in the database. Configuration and basic use of the Data Source Explorer is described in the appendix [Database Access from within Eclipse](#).

## 4.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “MuleFileToDatabase”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded. In addition to the basic project setup, we will perform some additional setup for this particular project:

- In the project root, create a directory named “file-input-directory”.  
This is the directory in which files are to be placed when their contents is to be inserted into the database.
- In the project root, create another directory and name it “file-output-directory”.  
This is the directory to which we will instruct Mule to copy files that has been processed.
- Enable Maven dependency management for the new project according to instructions in the appendix [Enabling Maven Dependency Management for an Eclipse Project](#).

Maven dependency management will be used to include some additional libraries needed for the example program.

## 4.2. Add Dependencies

The Maven pom.xml file is used to manage the additional dependencies of the project. The Mule libraries will still be included on the project's build path in the regular fashion in order to enable us to easily switch between using the Mule 2.x runtime and the Mule 3.x runtime.

A pom.xml file should already have been created when we enabled Maven dependency management for the project. We just need to add dependencies to the libraries we need.

- Open the pom.xml file.
- Add the dependencies to the file so that it looks like this:

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ivan.mule</groupId>
    <artifactId>MuleFileToDatabase</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <!--
            Includes the Spring framework's JDBC module.
        -->
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>org.springframework.jdbc</artifactId>
            <version>3.0.2.RELEASE</version>
        </dependency>
        <!--
            Includes the Apache Commons data source implementation.
        -->
        <dependency>
            <groupId>commons-dbcp</groupId>
            <artifactId>commons-dbcp</artifactId>
            <version>1.4</version>
            <type>jar</type>
            <scope>compile</scope>
        </dependency>
        <!--
            Includes the PostgreSQL JDBC connector.
            Change this if you use another database.
        -->
        <dependency>
            <groupId>postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>9.0-801.jdbc4</version>
        </dependency>
    </dependencies>
</project>
```

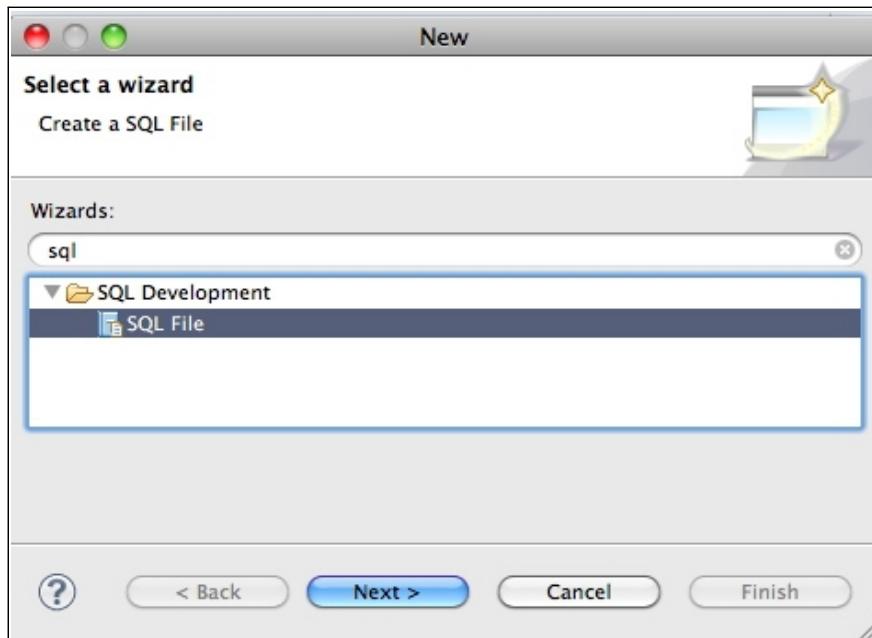
Note that:

- If you want to use another database, you need to replace the PostgreSQL JDBC connector library with the appropriate JDBC connector library for the database in question.

### 4.3. Database Table Creation Script

In order for Mule to be able to store data into a database, there must be a schema and a table in which to insert the data. In this section we'll write and execute the SQL script that creates the database schema and table used by our example program.

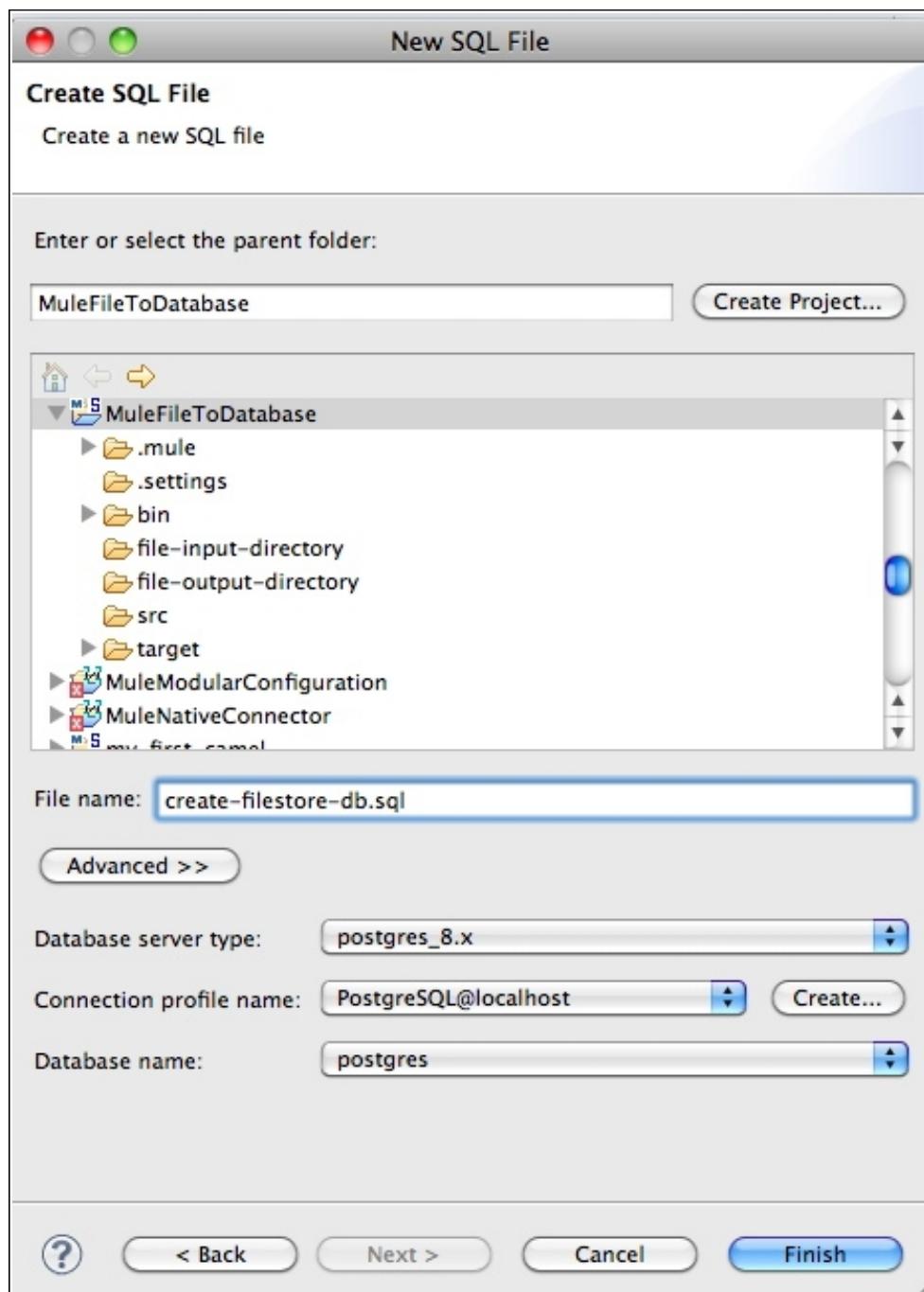
- In the File menu, select New -> Other.
- In the wizard dialog, select SQL File and click Next.



Selecting the SQL File wizard in Eclipse.

(continued on next page)

- In the Create SQL File dialog, select the MuleFileToDatabase project as the parent folder.



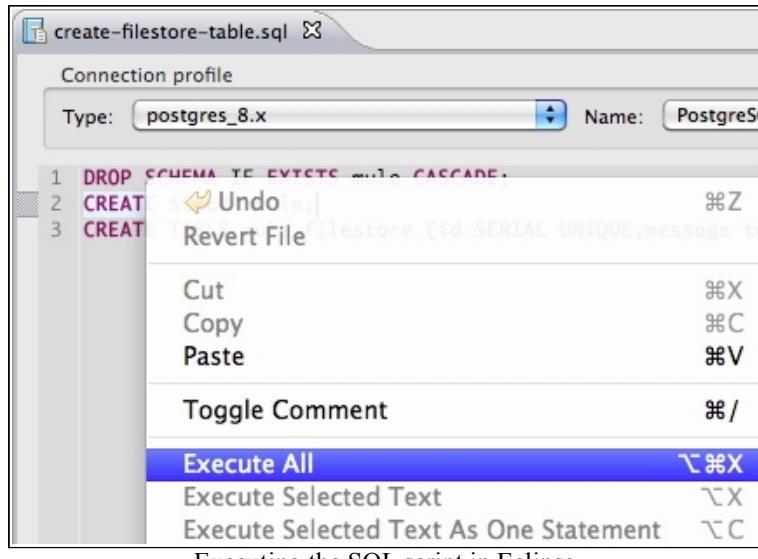
The Create SQL File wizard in Eclipse.

- Still in the Create SQL File dialog, enter “create-filestore-db.sql” as name of the SQL file.
- Verify the database server type, connection profile name and database name.
- Click Finish.

- In the new SQL file window, enter the following SQL statements:

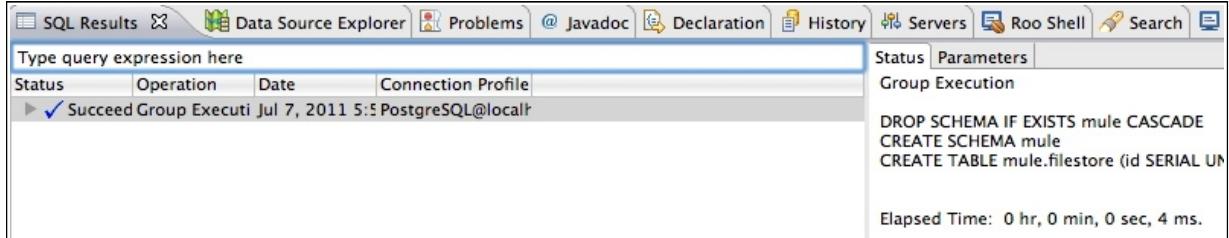
```
DROP SCHEMA IF EXISTS mule CASCADE;
CREATE SCHEMA mule;
CREATE TABLE mule.filestore (id SERIAL UNIQUE,message text NOT NULL,time_stamp timestamp NOT NULL,PRIMARY KEY (id));
```

- Right-click in the SQL file window and select Execute All.



Executing the SQL script in Eclipse.

- In the SQL Results view that is opened, verify the success of the SQL script execution.



The result of executing the SQL script show in Eclipse.

- Verify that the “filestore” table in the “mule” database is empty.  
Use the procedure described in the [Data Access](#) section of appendix E.

Note that:

- In the part of the script that creates the table in the database, the id column is specified as having the type SERIAL.  
The SERIAL type is PostgreSQL's way of defining an unique identifier column that is of the type integer and which default values will be assigned from a sequence generator.  
Please refer to the PostgreSQL documentation for details.

## 4.4. Create the Spring Bean Configuration File

Both versions of the example program uses one and the same Spring bean configuration file to configure a Spring bean that acts as a data-source. This data-source will be used by Mule when inserting data into the database.

In the root of the project source directory, create a file named “spring-config.xml” with the following contents:

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

    <!--
        Data source Spring bean used by both the Mule 2.x and Mule 3.x
        versions of the example.
        Should be changed if another database is used etc.
        Note that before being able to run the example, the database
        must be set up using the enclosed script.
    -->
    <bean
        id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="org.postgresql.Driver"
        p:url="jdbc:postgresql://localhost:5432/postgres"
        p:username="postgres"
        p:password="adminadmin"/>
</beans>
```

Note that:

- The p namespace is used to specify the database properties.  
Please refer to the Spring reference documentation for details.
- The username and password are, as before, “postgres” and “adminadmin” respectively.

## 4.5. Create the Mule Configuration Files

The Mule configuration of this chapter's example program is split into three different files; one file containing the file connector definition, one file containing the JDBC connector definition and one file containing the main Mule model/flow.

### ***Create the Mule File Connector Configuration Files***

The Mule file connector specifies how files should be read or written from or to a directory in the local file system.

In this example, we specify how files in the input directory are to be handled.

The only difference between the Mule 2.x and Mule 3.x file connector configuration files are the namespaces. Both are located in the root of the project's source directory.

The Mule 2.x configuration file is named “mule2-fileconnector.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/file/2.2
          http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd">

    <!--
        File connector specifying:
        - How file content should be passed along.
        The streaming attribute specifies whether a stream (value true) or
        the contents of the file (value false) is to be passed around as
        the incoming message.
        - How often the input directory should be checked for new files.
        The pollingFrequency attribute specifies the interval in
        milliseconds between checks for files in the input directory.

        The default setting is that files in the input directory,
        once read, should be deleted.
    -->
    <file:connector name="fileConnector" streaming="false" pollingFrequency="1000">
        <!--
            The expression-filename-parser creates a filename for
            arrived messages. The filename is placed in a header passed
            along with the message which will be used later when writing
            the file contents to an output file.
        -->
        <file:expression-filename-parser/>
    </file:connector>
</mule>
```

The Mule 3.x file connector configuration file is named “mule3-fileconnector.xml”:

```
<?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:file="http://www.mulesoft.org/schema/mule/file"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/file
          http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd">

    <!--
        File connector specifying:
        - How file content should be passed along.
        The streaming attribute specifies whether a stream (value true) or
        the contents of the file (value false) is to be passed around as
        the incoming message.
        - How often the input directory should be checked for new files.
        The pollingFrequency attribute specifies the interval in
        milliseconds between checks for files in the input directory.

        The default setting is that files in the input directory,
        once read, should be deleted.
    -->
    <file:connector name="fileConnector" streaming="false" pollingFrequency="1000">
        <!--
            The expression-filename-parser creates a filename for
            arrived messages. The filename is placed in a header passed
            along with the message which will be used later when writing
            the file contents to an output file.
        -->
        <file:expression-filename-parser/>
    </file:connector>
</mule>
```

Note that:

- The *streaming* attribute of the `<file:connector>` element specifies whether a stream from which a file can be read (true) or whether the entire contents of a file (false) is to be passed around with a message.
- The *pollingFrequency* attribute of the `<file:connector>` element specifies the how frequently an input directory is to be checked for new files. Time is in milliseconds.
- The `<file:expression-filename-parser>` element in the `<file:connector>` element associates a filename parser with the file connector.  
In this example, the parser will extract the name of an incoming file and place it in a header in the message associated with the incoming file.  
The parser can also be used to construct the name of a file to be written.
- As per default, incoming files are deleted after having been read.  
This behaviour can be changed using the *autoDelete* attribute of the `<file:connector>` element.

## Create the Mule JDBC Connector Configuration Files

The JDBC connector indirectly specifies which database to be used by referencing the data source bean defined in the Spring configuration file [earlier](#).

A JDBC connector contain one or more SQL queries used to read from or write to a database. In this example, there is one single query used to write file data to the database table defined in the database script we wrote [earlier](#).

Again, the Mule configuration files for the different versions of Mule are identical, except for the namespaces. Both configuration files are located in the root of the project's source directory.

The Mule 2.x configuration file is named “mule2-jdbcconnector.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jdbc="http://www.mulesource.org/schema/mule/jdbc/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/jdbc/2.2
          http://www.mulesource.org/schema/mule/jdbc/2.2/mule-jdbc.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd">

    <!--
        JDBC connector specifying which data source to use when
        interacting with the database. The data source is a Spring bean.
        A JDBC connector contain one or more queries.
    -->
    <jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
        <!--
            A JDBC query is a database query that can be used by, for
            example, an inbound or outbound endpoint to read
            data from or write data to a database.

            This query inserts the payload of a message received
            along with a timestamp into the table FILESTORE.
        -->
        <jdbc:query
            key="fileInsert"
            value="INSERT INTO mule.filestore VALUES
(DEFAULT,#[payload:java.lang.String],CURRENT_TIMESTAMP)"/>
    </jdbc:connector>
</mule>
```

(continued on next page)

The Mule 3.x configuration file is named “mule3-jdbcconnector.xml”:

```
<?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:jdbc="http://www.mulesoft.org/schema/mule/jdbc"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/jdbc
          http://www.mulesoft.org/schema/mule/jdbc/3.2/mule-jdbc.xsd
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd">

    <!--
        JDBC connector specifying which data source to use when
        interacting with the database. The data source is a Spring bean.
        A JDBC connector contain one or more queries.
    -->
    <jdbc:connector name="jdbcConnector" dataSource-ref="dataSource">
        <!--
            A JDBC query is a database query that can be used by, for
            example, an inbound or outbound endpoint to read
            data from or write data to a database.

            This query inserts the payload of a message received
            along with a timestamp into the table FILESTORE.
        -->
        <jdbc:query
            key="fileInsert"
            value="INSERT INTO mule.filestore VALUES
(DEFAULT,#[payload:java.lang.String],CURRENT_TIMESTAMP)" />
    </jdbc:connector>
</mule>
```

Note that:

- Which data-source to be used by the JDBC connector is specified using the *dataSource-ref* attribute of the `<jdbc:connector>` element.  
The name refers to the Spring bean “*dataSource*” defined in the Spring configuration file in the [previous section](#).
- The `<jdbc:connector>` element contains a `<jdbc:query>` element.  
A `<jdbc:connector>` element contain one or more queries, each having a name specified by the *key* attribute of the `<jdbc:query>` element.
- The query specified by the *value* attribute of the `<jdbc:query>` element contains the string “*DEFAULT*”.  
Using PostgreSQL, this will cause the next default value to be inserted in the corresponding column in the database table.
- The query specified by the *value* attribute of the `<jdbc:query>` element contains the string “*#*[*payload:java.lang.String*]”.  
This string an expression that specifies that the message payload should be inserted as a Java String into the query.  
The “#” is referred to as the placeholder prefix.
- The query specified by the *value* attribute of the `<jdbc:query>` element contains the string “*CURRENT\_TIMESTAMP*”.  
Using PostgreSQL, this will cause the date and time of the start of the current transaction to be inserted into the corresponding column in the database table.

## Create the Main Mule Configuration Files

The main Mule configuration files are responsible for putting the file processing model together using the connectors and Spring configuration created earlier in this chapter. As in the introduction of this chapter, we will deliberately use a model in both the configuration files to show that Mule 2.x configuration can, with minimal modifications, be run on Mule 3.x as well.

These Mule configuration files are also located in the root of the source directory of the project.

The Mule 2.x configuration file is named “mule-config2.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
      xmlns:jdbc="http://www.mulesource.org/schema/mule/jdbc/2.2"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/file/2.2
          http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

          http://www.mulesource.org/schema/mule/jdbc/2.2
          http://www.mulesource.org/schema/mule/jdbc/2.2/mule-jdbc.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <spring:beans>
        <spring:import resource="spring-config.xml"/>
        <spring:import resource="mule2-fileconnector.xml"/>
        <spring:import resource="mule2-jdbcconnector.xml"/>
    </spring:beans>

    <model name="fileProcessingModel">
        <service name="fileToDBService">
            <inbound>
                <!--
                    Input endpoint receiving files in a directory,
                    applying filtering by filename.
                -->
                <file:inbound-endpoint
                    connector-ref="fileConnector"
                    path=".//file-input-directory">
                    <!--
                        Only process files with the .xml suffix.
                    -->
                    <file:filename-wildcard-filter pattern="*.xml"/>
                </file:inbound-endpoint>
            </inbound>

            <outbound>
                <!--
                    The multicast router sends the received message to all
                    contained endpoints. In this example, the contents
                    of the file dropped in the input-directory will be
                    written to a file in the output-director as well as
                    printed to the console.
                -->
                <multicasting-router>
                    <!--
                        Endpoint writing message to file.
                        The name of the output file contains the name of
                        the input file.
                        The name of the file written contains a timestamp
                        showing the time the file was written.
                        The timestamp is obtained from the function "dateStamp"
                        and formatted according to the string that follows.
                    -->
                </multicasting-router>
            </outbound>
        </service>
    </model>

```

```

-->
<file:outbound-endpoint
    path=".//file-output-directory"
    outputPattern=
        "processed-[header:originalFilename]-
#[function:dateStamp:yyyy-MM-dd_hh:mm:ss]" />

<!--
      Endpoint printing message to the console.
-->
<stdio:outbound-endpoint system="OUT" />

<!--
      Endpoint inserting messages into database using the
      "fileInsert" query in the JDBC connector "jdbcConnector".
-->
<jdbc:outbound-endpoint connector-ref="jdbcConnector"
queryKey="fileInsert"/>
    </multicasting-router>
    </outbound>
</service>
</model>
</mule>

```

The Mule 3.x configuration file is named “mule-config3.xml”:

```

<?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:file="http://www.mulesoft.org/schema/mule/file"
    xmlns:jdbc="http://www.mulesoft.org/schema/mule/jdbc"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/file
        http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

        http://www.mulesoft.org/schema/mule/jdbc
        http://www.mulesoft.org/schema/mule/jdbc/3.2/mule-jdbc.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <spring:beans>
        <spring:import resource="spring-config.xml" />
        <spring:import resource="mule3-fileconnector.xml" />
        <spring:import resource="mule3-jdbcconnector.xml" />
    </spring:beans>

    <model name="fileProcessingModel">
        <service name="fileToDBService">
            <inbound>
                <!--
                    Input endpoint receiving files in a directory,
                    applying filtering by filename.
                -->
                <file:inbound-endpoint
                    connector-ref="fileConnector"
                    path=".//file-input-directory">
                    <!--
                        Only process files with the .xml suffix.
                    -->
                    <file:filename-wildcard-filter pattern="*.xml" />
                </file:inbound-endpoint>
            </inbound>

            <outbound>
                <!--

```

```

The multicast router sends the received message to all
contained endpoints. In this example, the contents
of the file dropped in the input-directory will be
written to a file in the output-director as well as
printed to the console.
-->
<multicasting-router>
<!--
    Endpoint writing message to file.
    The name of the output file contains the name of
    the input file.
    The name of the file written contains a timestamp
    showing the time the file was written.
    The timestamp is obtained from the function "dateStamp"
    and formatted according to the string that follows.
-->
<file:outbound-endpoint
    path=".file-output-directory"
    outputPattern=
        "processed-#[header:originalFilename]-
#[function:dateStamp:yyyy-MM-dd_hh:mm:ss]" />

<!--
    Endpoint printing message to the console.
-->
<stdio:outbound-endpoint system="OUT" />

<!--
    Endpoint inserting messages into database using the
    "fileInsert" query in the JDBC connector "jdbcConnector".
-->
<jdbc:outbound-endpoint connector-ref="jdbcConnector"
queryKey="fileInsert"/>
</multicasting-router>
</outbound>
</service>
</model>
</mule>

```

Note that:

- Spring <import> elements inside a Spring <beans> element are used to import the Spring bean configuration file as well as the other two configuration files.
- The <model> element contains one single <service> element; the fileToDBService.
- The <service> element contain one <inbound> and one <outbound> element, both which are optional.  
The <inbound> element specifies how messages are received by the service and the <outbound> element specifies how messages from the service are sent out.  
In older versions of Mule, a bridge service component had to be configured explicitly to connect the inbound part of a service to the outbound part. Starting with Mule 2.0, there is a default pass-through component inserted if nothing else is configured.
- The fileToDBService service contain one single inbound endpoint.  
This is a file inbound endpoint specified in a <file:inbound-endpoint> element.  
The inbound endpoint uses the *connector-ref* attribute to specify that it uses the file connector with the name fileConnector we defined earlier. In addition, the endpoint specifies which directory to monitor using the path attribute of the <file:inbound-endpoint> element.
- The <file:inbound-endpoint> element contains a <file:filename-wildcard-filter> element.  
The *pattern* attribute of this element specify a pattern that names of files in the input directory of the file inbound endpoint directory must match in order to be accepted as input to the endpoint. Multiple patterns can be supplied in a comma-separated list.
- The <service> element contains an <outbound> element after the <inbound> element.

Again, this element specifies how messages from the service are passed on after any processing the service performs.

- The <outbound> element contains a <multicasting-router> element. This element sends messages to all outbound endpoints contained in the <multicasting-router> element. Endpoints in the router can use the *synchronous* attribute to indicate whether they are synchronous or not. If all endpoints in the multicasting router are asynchronous, a message will be sent to all endpoints at the same time. Otherwise a message will be sent to the contained endpoints in the order they are listed.
- The first endpoint in the multicasting router is defined by the <file:outbound-endpoint> element. This endpoint writes messages to the directory specified by the *path* attribute and names files according to the pattern specified by the *outputPattern* attribute.
  - The *outputPattern* attribute of the <file:outbound-endpoint> element contains the following string: processed-#[header:originalFilename]-#[function:dateStamp:yyyy-MM-dd hh:mm:ss]The “#[“ are, as before, placeholder prefixes.  
The “header:originalFilename” indicates that the data to be inserted in the string is to be taken from the message header “originalFilename”.  
The “function:dateStamp:yyyy-MM-dd hh:mm:ss” executes the function “dateStamp” and outputs the date-time information to a string in the specified format.  
Please consult the Expression Evaluator Reference section in the Mule User Guide for additional information.
- The <stdio:outbound-endpoint> element will, as seen in previous examples, log the message to the standard I/O console.
- The <jdbc:outbound-endpoint> element defines the JDBC outbound endpoint that inserts the message into the database.
  - As with other types of endpoints, the *connector-ref* attribute is used to specify which connector the endpoint is to use. In this example, the *jdbcConnector* is the database connector defined earlier in a separate file.  
The *queryKey* attribute is used to decide which database query in the connector that is to be executed when a message arrives to the endpoint.

This concludes the development of this chapter's example program. We are now ready to try it.

## 4.6. Run the Example Program

In this section we will try the example program by starting Mule, dropping an XML-file into the input directory and then look at the console, the database and the output directory to verify that the contents of the file indeed was processed as expected.

When running the example program, we'll use the technique of running the Mule configuration files as described in the [earlier example](#).

It is assumed that the project is configured with the Mule 3.x distribution on the classpath when starting this section.

- Make sure that the database server is running.

Usually the server starts automatically when the computer is started, but if there are problems one must make sure that the server is running and that it is possible to connect to it using the credentials used in the Mule JDBC connector.

- From within Eclipse, connect to the data-source as described in [section 2 of appendix E](#).
- In the Package or Project Explorer, right-click the “mule-config3.xml” file and select Run As -> Mule Server.
- The Mule 3.x server should start up and display console output indicating a successful startup.

- Refresh the “MuleFileToDatabase” project in Eclipse.

Note that there is one new directories that we did not create ourselves. It is named “file-input-directory”. This directory was created by Mule for the inbound file endpoint.

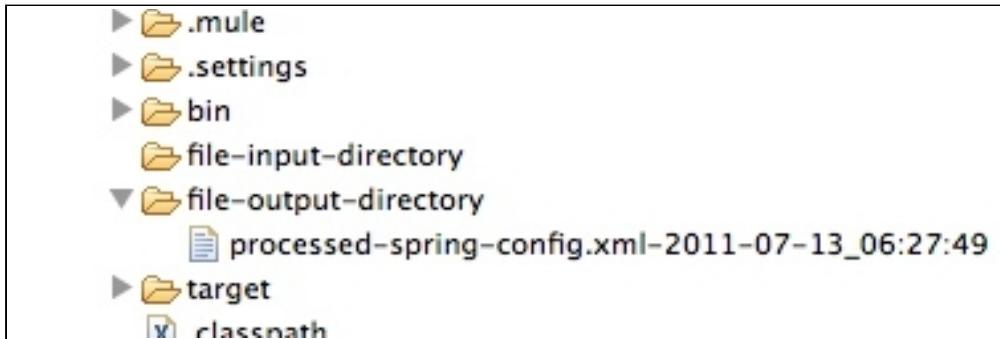
- In Eclipse, copy any of the XML files in the project and paste it into the “file-input-directory”. I used the Spring configuration file.
- The contents of the XML file should be displayed on the console along with a number of log messages from Mule.

The last two log messages tells us that the message, that is the contents of the XML file, was written to a file in the output directory and successfully inserted into the database (the path to the output file has been edited for brevity):

```
...
<bean
    id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="org.postgresql.Driver"
    p:url="jdbc:postgresql://localhost:5432/postgres"
    p:username="postgres"
    p:password="adminadmin"/>
</beans>

INFO 2011-07-13 18:02:22,977 [fileConnector.dispatcher.1]
org.mule.transport.file.FileConnector: Writing file to: ./file-output-
directory/processedspring-config.xml-2011-02-13_06:02:22
INFO 2011-07-13 18:02:23,218 [jdbcConnector.dispatcher.1]
org.mule.transport.jdbc.sqlstrategy.SimpleUpdateSqlStatementStrategy: Executing SQL
statement: 1 row(s) updated
```

- In Eclipse, refresh the project by clicking on it and pressing F5.  
Alternatively, right-click on the project and select Refresh in the menu that appears.  
The input directory should be empty and a new directory named “file-output-directory”, the output directory, should have appeared:



Input and output directories in the project after Mule having processed an XML file.

- View the contents of the “filestore” database table in the “mule” database.  
The process for viewing a database table in Eclipse is described in the [second section of appendix E](#), but you are free to use the tool of your choice.  
There should be one row in the table with the contents of the XML file in the “message” column:

filestore		
id [SERIAL]	message [TEXT(2147483647)]	time_stamp [TIMESTAMP]
1	<?xml version="1.0" encoding="UTF-8"?> <new row>	2011-07-13 18:27:49.590065

Database table “filestore” after Mule having processed a XML file pasted in the input directory.

In this example, we have seen that when we drop a file in the designated input directory, Mule picks it up and sends the contents of the file to the console, an output directory and a database table, all according to our configuration.

Run the Mule 2.x version of the example program:

- Change the Mule distribution to Mule 2.x, as described in [appendix B](#).
- Start the example program by right-clicking the “mule-config2.xml” file and selecting Run As->Mule Server.

The result of running the Mule 2.x version should be similar to what we've already seen.

## 4.7. Mule 3.x Version with Flow

As a sort of appendix to this chapter, we will take a brief look at a Mule 3.x version of this chapter's example program that uses a flow. This configuration file imports the same three configuration files as the Mule 3.x configuration file used [earlier](#).

```
<?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:file="http://www.mulesoft.org/schema/mule/file"
      xmlns:jdbc="http://www.mulesoft.org/schema/mule/jdbc"
      xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/file
          http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

          http://www.mulesoft.org/schema/mule/jdbc
          http://www.mulesoft.org/schema/mule/jdbc/3.2/mule-jdbc.xsd

          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd

          http://www.mulesoft.org/schema/mule/stdio
          http://www.mulesoft.org/schema/mule/stdio/current/mule-stdio.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <spring:beans>
        <spring:import resource="spring-config.xml"/>
        <spring:import resource="mule3-fileconnector.xml"/>
        <spring:import resource="mule3-jdbcconnector.xml"/>
    </spring:beans>

    <!--
        Note that the flow does not contain any routers.
        A flow consists of a message source and a number of
        message processors.
        Outbound endpoints are also message processors.
    -->
    <flow name="fileToDatabase">
        <!--
            Input endpoint receiving files in a directory,
            applying filtering by filename.
        -->
        <file:inbound-endpoint
            connector-ref="fileConnector"
            path=".//file-input-directory">
            <!--
                Only process files with the .xml suffix.
            -->
            <file:filename-wildcard-filter pattern="*.xml"/>
        </file:inbound-endpoint>

        <!--
            Endpoint writing message to file.
            The name of the output file contains the name of
            the input file.
            The name of the file written contains a timestamp
            showing the time the file was written.
            The timestamp is obtained from the function "dateStamp"
            and formatted according to the string that follows.
        -->
        <file:outbound-endpoint
            path=".//file-output-directory"
            outputPattern=
                "processed-#[header:originalFilename]-#[function:dateStamp:yyyy-MM-
dd_hh:mm:ss]"/>

        <!--
            Endpoint printing message to the console.
        -->
        <stdio:outbound-endpoint system="OUT" />
    
```

```

<!--
    Endpoint inserting messages into database using the
    "fileInsert" query in the JDBC connector "jdbcConnector".
-->
</jdbc:outbound-endpoint connector-ref="jdbcConnector" queryKey="fileInsert" />
</flow>
</mule>

```

Some things to note are:

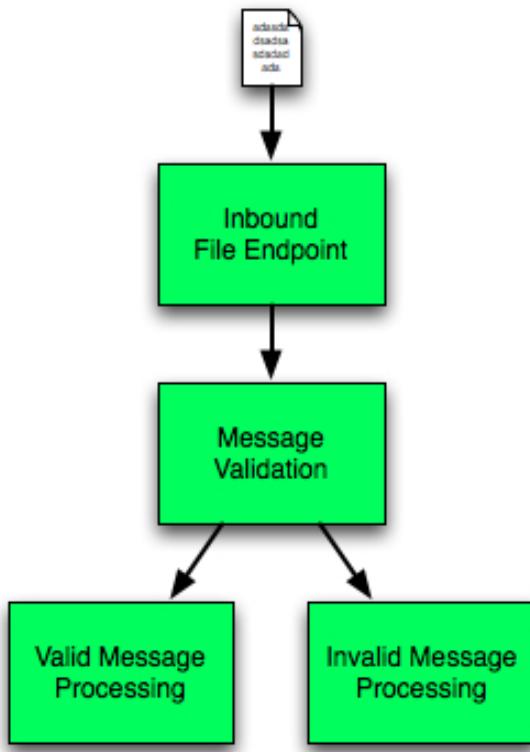
- The components in the flow can be found in the [previous](#) version of the configuration file. Creating the above configuration file was actually accomplished by copying parts of the previous configuration file, with no changes to the configuration of the individual components.
- There are no routers in the flow.  
A flow consists of a message source and a number of message processors. Outbound endpoints are also a kind of message processors.

This concludes this chapter's example program.

## 5. Validate XML Data

In this chapter we will see an example showing how to validate incoming data against one or more XML schemas. We'll see techniques showing how to work around limitations in the Xerces XML parser that is used by Mule for XML validation. We will also see how to route messages differently, depending on whether they have passed validation or not.

The structure of this chapter's Mule configuration looks like this:



Structure of this chapter's example program.

There will be significant differences between the Mule 2.x and 3.x versions of the configuration files so the above structure does not necessarily map very closely to the Mule configurations.

We will use the technique of running the Mule configuration files, described in the previous chapters, so we will not need a starter class.

## 5.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “MuleXMLSchemaValidation”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded. In addition to the basic project setup, we will perform some additional setup for this particular project:

- In the project root, create a directory named “file-input-directory”.  
This is the directory in which files are to be placed when they are to be processed by Mule.
- Again in the project root, create a directory named “xml-data”.  
This directory will contain the XML example files that we are going to send into Mule to be validated.
- In the root of the source code hierarchy, create a package named “schemas”.  
The XML schemas used when validating are to be stored in this package.

## 5.2. Create the XML Schemas

The example will use three XML schemas when validating. Two of these XML schemas, the main schemas, have the same target namespace. This is to show how to work around a problem with the Mule schema validation filter preventing the use of multiple schemas with the same target namespace in one filter.

The third schema is included in the second XML schema but the type defined in the third schema is not used yet. We will look at it at the end of the chapter, to solve the problem of Mule not being able to find the included XML schema.

- In the “schema” package in the source directory, create a file named “schema2.xsd” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ivan="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas"
    attributeFormDefault="unqualified">

    <element name="person" type="ivan:Person"/>
    <complexType name="Person">
        <sequence>
            <element name="firstName" type="string" nillable="false"/>
            <element name="lastName" type="string" nillable="false"/>
            <element name="age" type="int"/>
        </sequence>
    </complexType>
</schema>
```

- In the same package, create a file named “schema3.xsd” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema
    xmlns:xss="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas"
    attributeFormDefault="unqualified">

    <xss:element name="animal" type="Animal"/>
    <xss:complexType name="Animal">
        <xss:sequence>
            <xss:element name="name" type="xs:string" nillable="false"/>
            <xss:element name="birthday" type="xs:date"/>
        </xss:sequence>
    </xss:complexType>
</xss:schema>
```

```
</xs:complexType>
</xs:schema>
```

- Finally, next to the other two schema files, create a file named “schema4.xsd” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    attributeFormDefault="unqualified" elementFormDefault="qualified">

    <xs:simpleType name="ivan_date_type">
        <xs:restriction base="xs:date">
            </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

The fourth XML schema will be used later in the chapter.

### 5.3. Create XML Data Files

The following files are all to be located in the directory “xml-data”.

- The file “animal.xml” with the following contents:

Observant readers notice that this XML fragment should be validated against “schema3.xsd” entered above.

```
<?xml version="1.0" encoding="UTF-8"?>
<ivan:animal
    xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas schema3.xsd ">
    <name>Florpy Skudds</name>
    <birthday>2009-06-01</birthday>
</ivan:animal>
```

- The file “person.xml” with the following contents:

Observant readers notice that this XML fragment should be validated against “schema2.xsd” entered above.

```
<?xml version="1.0" encoding="UTF-8"?>
<ivan:person
    xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas schema2.xsd ">
    <firstName>Stiv</firstName>
    <lastName>Bator</lastName>
    <age>32</age>
</ivan:person>
```

- The file “invalid.xml” with the following contents:

As the name suggests, this XML fragment will not pass validation against any of the above XML schemas.

```
<ivan:person
    xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

</ivan:person>
```

## 5.4. Create the Mule Configuration Files

The example contains two Mule configuration files per Mule version, for a total of four Mule configuration files.

### ***Create the Mule File Connector Configuration Files***

The Mule file connector specifies how files should be read or written from or to a directory in the local file system. These configuration files are identical to those used in the [previous example](#). Both files are to be located in the root of the project's source directory.

The Mule 2.x configuration file is named “mule2-fileconnector.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/file/2.2
          http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd">

    <file:connector name="fileConnector" streaming="false" pollingFrequency="1000">
        <file:expression-filename-parser/>
    </file:connector>
</mule>
```

The Mule 3.x configuration file is named “mule3-fileconnector.xml”:

```
<?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:file="http://www.mulesoft.org/schema/mule/file"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/file
          http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd">

    <file:connector name="fileConnector" streaming="false" pollingFrequency="1000">
        <file:expression-filename-parser/>
    </file:connector>
</mule>
```

These file connectors have the following properties:

- The name is “fileConnector”.  
Used when referring to the connectors from the file inbound endpoints.
- Pass a file, not a file input stream, as the message payload.
- Check the input directory each 1000 mS.
- Store the name of a file in a message property.

## Create the Main Mule Configuration Files

The main Mule configuration files are quite different, although they roughly represent the same structure.

These Mule configuration files are also located in the root of the source directory of the project.

## The Mule 2.x Configuration File

The Mule 2.x configuration file are built-up in a traditional manner of multiple services in a model:

- One input service.  
Responsible for receiving the messages to validate.
- One validation service.  
Attempts to validate messages against the two XML schemas created [earlier](#). If the message passes validation against either of these schemas, it is considered to have passed validation and is passed on to a service that will only receive messages having passed validation. If the message does not pass validation against any of the schemas, it will be passed on to an error service.
- One processing service.  
This service will receive only messages having passed validation. Any processing of valid messages would be implemented in this service. In this example, these messages are just written to a directory named “good-messages” using the same name as the original file.
- One error service.  
This service will receive only messages that do not pass validation.  
Messages having failed validation will be written to a directory named “bad-messages”.

The Mule 2.x configuration file is named “mule2-validation.xml” and looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns:mule-xml="http://www.mulesource.org/schema/mule/xml/2.2"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/file/2.2
          http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

          http://www.mulesource.org/schema/mule/xml/2.2
          http://www.mulesource.org/schema/mule/xml/2.2/mule-xml.xsd

          http://www.mulesource.org/schema/mule/vm/2.2
          http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">

<spring:beans>
    <spring:import resource="mule2-fileconnector.xml"/>
</spring:beans>

<!--
    The <model> in this example contains four different services:
    - An input service responsible for accepting messages in the
-->
```

```

form of files placed in a certain directory.
- A validation service that validates messages.
Depending on whether a message having passed validation or
not, it is either passed on to the processing service or to
the error service.
- A processing service.
This service receives only messages having passed validation.
In this example, these messages are just written to the
good messages directory.
- An error service.
This service receives only messages that do not pass validation.
in this example, these messages are just written to the bad
messages directory.
-->
<model name="xmlValidationModel">
    <service name="xmlInputService">
        <inbound>
            <!--
                Input endpoint receiving files in a directory,
                applying filtering by filename.
            -->
            <file:inbound-endpoint
                connector-ref="fileConnector"
                path=".//file-input-directory">
                <file:filename-wildcard-filter pattern="*.xml"/>
            </file:inbound-endpoint>
        </inbound>
        <outbound>
            <pass-through-router>
                <vm:outbound-endpoint path="message.validation"/>
            </pass-through-router>
        </outbound>
    </service>

    <!--
        Service that validates messages and pass them on depending
        on the result of the validation.
    -->
    <service name="xmlValidationService">
        <inbound>
            <vm:inbound-endpoint path="message.validation"/>
            <!--
                A selective consumer router can apply one filter
                to incoming messages.
                If a message is matched by the filter, in this example
                passes XML validation, the message is passed on to
                the component (if any) and later to any outbound
                endpoint of the service.
                If a message is not matched by the filter, that is
                do not pass XML validation, the catch-all strategy
                is invoked if such is configured. Otherwise the message
                is ignored and a warning message logged.
            -->
            <selective-consumer-router>
                <!--
                    The OR filter contains two or more other filters and
                    accept a message if it matches the criteria of at least
                    one of the contained filters.
                -->
                <or-filter>
                    <!--
                        A message sent to a schema validation filter is
                        matched if it passes validation by the schema(s)
                        of the filter.
                        Due to a bug in the XML parser used for validation,
                        multiple names with the same namespace cannot be
                        used by a single schema validation filter.
                        A work-around is to use an OR filter in combination
                        with a schema validation filter for each schema.
                        The returnResult attribute has been set to false in
                        order to increase the speed of the filter.
                    -->
                    <mule-xml:schema-validation-filter
                        schemaLocations="schemas/schema2.xsd"
                        returnResult="false"/>
                    <mule-xml:schema-validation-filter
                        schemaLocations="schemas/schema3.xsd"/>

```

```

        returnResult="false"/>
    </or-filter>
</selective-consumer-router>
<!--
    Catch-all strategy invoked when a message does not
    pass XML validation.
    Route the message to the error service.
-->
<forwarding-catch-all-strategy>
    <vm:outbound-endpoint path="message.error"/>
</forwarding-catch-all-strategy>
</inbound>
<outbound>
    <!--
        Messages arriving here are valid.
        Pass these messages on to the processing service.
-->
    <pass-through-router>
        <vm:outbound-endpoint path="message.processing"/>
    </pass-through-router>
</outbound>
</service>

<!--
    Service that receives messages having passed validation.
    Writes the message to the good messages directory using the
    original file name.
-->
<service name="xmlProcessingService">
    <inbound>
        <vm:inbound-endpoint path="message.processing"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <file:outbound-endpoint
                path="good-messages/"
                outputPattern="#{header:originalFilename}"/>
        </pass-through-router>
    </outbound>
</service>

<!--
    Service that receives messages having failed validation.
    Writes the message to the bad messages directory using the
    original file name.
-->
<service name="xmlErrorService">
    <inbound>
        <vm:inbound-endpoint path="message.error"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <file:outbound-endpoint
                path="bad-messages/"
                outputPattern="#{header:originalFilename}"/>
        </pass-through-router>
    </outbound>
</service>
</model>
</mule>

```

Note that :

- The file connector configuration file is imported using the Spring mechanism we have seen in earlier examples.
- The first service in the model, the “xmlInputService”, contains an inbound endpoint accepting files with the extension “xml” from a designated directory.
- The “xmlInputService” passes received messages on using the VM transport. VM transport is commonly used as seen in this example, as a means to connect services running in a single Java virtual machine.

- The “xmlValidationService” accepts incoming messages using VM transport.
- The “xmlInputService” contains a <selective-consumer-router> element in its <inbound> element.  
The selective consumer router applies one filter to incoming messages.  
If a message is matched by the filter, it is passed on to the component of the service (if any) and then later to any outbound endpoint of the service.  
If a message is not matched by the filter, it will be passed on to the catch-all strategy of the service. If there is no catch-all strategy configured on the service, the message will be ignored and a warning message written to the log.
- The <selective-consumer-router> contains an <or-filter> element.  
An OR filter enables us to combine multiple filters. A message is matched by the OR filter if it is matched by at least one filter contained in the OR filter.
- The <or-filter> element contains two <schema-validation-filter> elements.  
A schema validation filter validates messages against the schema specified by the *schemaLocation* attribute. A message is matched if it passes validation against the schema. Note that the schema language, which by default is XML, may be specified.
- In a <schema-validation-filter>, the schemaLocations attribute allows us to supply a comma-separated list of schemas to validate against.  
A bug in the Xerces parser supplied by the JavaSE runtime environment prevents us from specifying multiple XML schemas with the same target namespace.
- The construct with the <or-filter> and the two <schema-validation-filter> elements is a work-around due to our XML schemas having the same target namespace.
- The *returnResult* attribute in both the <schema-validation-filter> elements have been set to false.  
The default value of this attribute is true, which causes the result of the validation to become the payload of the message. Setting this attribute to false avoids the conversion of the incoming message payload to a DOM node and also avoids storing the result of the validation.
- The final element in the <inbound> element of the xmlValidationService service is a <forwarding-catch-all-strategy> element.  
As described above when discussing the <selective-consumer-router> element, the forwarding catch-all strategy is used to specify where to send messages that are not accepted by the selective consumer router.
- The <outbound> element of the xmlValidationService service specifies that messages are to be sent to the message processing service.  
Again, messages arriving here are messages that pass validation. We can also see that the VM transport is used to send messages between services in one and the same Java VM.
- The xmlProcessingService service receives messages on the VM transport and writes them to the “good-messages” directory, using the original filename.
- Finally, the xmlErrorService service is almost identical to the processing service:  
Messages are received on the VM transport and written to a directory using the original filename. These messages will be written to the “bad-messages” directory, since they are messages that do not pass validation.

## The Mule 3.x Configuration File

The structure of the Mule 3.x configuration file is quite different from the configuration file in the previous section. Instead of a number of services, the Mule 3.x configuration consists of two flows; the first performs validation of a message against our XML schemas, the second flow is the main flow that routes messages depending on whether they pass validation or not.

The Mule 3.x configuration file is named “mule3-validation.xml” and looks like this:

```
<?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:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:file="http://www.mulesoft.org/schema/mule/file"
    xmlns:mule-xml="http://www.mulesoft.org/schema/mule/xml"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/file
        http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/xml
        http://www.mulesoft.org/schema/mule/xml/3.2/mule-xml.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!--
        This is a private flow, that is a flow without any message sources.
        Such a flow can only be referenced from within the same Mule instance,
        not from outside the JVM in question.
        This flow validates the message payload against two XML schemas.
        If validation succeeds, a message property 'validated' is set to
        'true'.
    -->
    A private flow has its own context and exception strategy.
    This means that exceptions will not propagate out of the private
    flow. If an exception is thrown, the message payload is set to null.

    The processing strategy of the flow must be synchronous,
    otherwise the flag indicating a message having passed validation
    may be checked before it is set.
    <!--
        Any message properties set here will not be present on the
        message if the filter below does not match the message.
    -->
    <flow name="ValidationFlow" processingStrategy="synchronous">
        <!--
            The OR filter contains two or more other filters and
            accept a message if it matches the criteria of at least
            one of the contained filters.
        -->
        <or-filter>
            <!--
                A message sent to a schema validation filter is
                matched if it passes validation by the schema(s)
                of the filter.
                Due to a bug in the XML parser used for validation,
                multiple names with the same namespace cannot be
                used by a single schema validation filter.
                A work-around is to use an OR filter in combination
                with a schema validation filter for each schema.
                The returnResult attribute has been set to false in
                order to increase the speed of the filter.
            -->
```

```

-->
<mule-xml:schema-validation-filter
    schemaLocations="schemas/schema2.xsd"
    returnResult="false"/>
<mule-xml:schema-validation-filter
    schemaLocations="schemas/schema3.xsd"
    returnResult="false"/>
</or-filter>
</message-filter>

<!--
    If message passes validation, set a message property that
    indicates this.
-->
<message-properties-transformer scope="outbound">
    <add-message-property key="validated" value="true"/>
</message-properties-transformer>
</flow>

<!--
    This is the main flow that accepts input from a directory in the
    filesystem, sends messages to the private validation flow.
    Messages having passed validation are written to one directory
    (good messages) while messages causing exceptions, typically that
    does not pass validation, are written to another directory
    (bad messages).
-->
<flow name="SortingFlow">
    <!--
        Input endpoint receiving XML files from a directory.
    -->
    <file:inbound-endpoint connector-ref="fileConnector"
        path=".//file-input-directory">
        <file:filename-wildcard-filter pattern="*.xml"/>
    </file:inbound-endpoint>

    <!-- Delegate message validation to a private flow. -->
    <flow-ref name="ValidationFlow"/>

    <choice>
        <!--
            A message is only considered to have passed validation when
            the 'validated' property has been set to 'true'.
            Note that the property is to be found in the outbound
            property scope.
            The 'header' evaluator evaluates the part of the message
            header specified by the expression attribute.
        -->
        <when expression="OUTBOUND:validated=true" evaluator="header">
            <!--
                Having passed validation, messages are written to
                the good-messages directory.
            -->
            <file:outbound-endpoint path="good-messages/"
                outputPattern="#{header:originalFilename}"/>
        </when>
        <otherwise>
            <!--
                Messages in which the property indicating having passed
                validation is not present or is not set to 'true' are
                written to the bad-messages directory.
            -->
            <file:outbound-endpoint path=".//bad-messages/"
                outputPattern="#{header:originalFilename}"/>
        </otherwise>
    </choice>
</flow>
</mule>

```

Note that :

- The file connector configuration file is imported using the Spring mechanism we have seen in earlier examples.

- The first flow, “ValidationFlow”, only contains a <message-filter> and a <message-properties-transformer>. The “ValidationFlow” is a private flow, a flow that has no message sources. Such a flow can only be used from within the same JVM instance in which the flow resides.
- The “ValidationFlow” flow has the attribute *processingStrategy* set to “synchronous”. This is because the flow must complete before its clients are allowed to examine the result – the value of a message property. If we allowed asynchronous processing, then the “ValidationFlow” flow would not manage to set the message property, in the case of a message passing validation, before the property was examined by the client. Thus it would seem as if all messages failed to pass validation.
- The <message-filter> has the attribute *throwOnUnaccepted* set to false. This indicates that an exception is not to be thrown if the filter contained in the <message-filter> element does not accept a message. Prior to Mule 3.2, exceptions thrown in private flows would propagate out of the flow. This behaviour was considered a bug and has been fixed in Mule 3.2. Also, if an exception is thrown in a private flow, the message payload is set to an instance of the Mule class *NullPayload* and the original payload would be lost. This would mean that the original message payload could not be written to a file, if it failed to pass validation.
- The <message-filter> contains an <or-filter> element. An OR filter enables us to combine multiple filters. A message is matched by the OR filter if it is matched by at least one filter contained in the OR filter.
- The <or-filter> element contains two <schema-validation-filter> elements. A schema validation filter validates messages against the schema specified by the *schemaLocation* attribute. A message is matched if it passes validation against the schema. Note that the schema language, which by default is XML, may be specified.
- In a <schema-validation-filter>, the *schemaLocations* attribute allows us to supply a comma-separated list of schemas to validate against. A bug in the Xerces parser supplied by the JavaSE runtime environment prevents us from specifying multiple XML schemas with the same target namespace.
- The construct with the <or-filter> and the two <schema-validation-filter> elements is a work-around due to our XML schemas having the same target namespace.
- The *returnResult* attribute in both the <schema-validation-filter> elements have been set to false. The default value of this attribute is true, which causes the result of the validation to become the payload of the message. Setting this attribute to false avoids the conversion of the incoming message payload to a DOM node and also avoids storing the result of the validation.
- The “SortingFlow” flow is the main flow of this configuration.
- The “SortingFlow” contains an inbound endpoint accepting files with the extension “xml” from a designated directory.
- A <flow-ref> element is used to refer to the “ValidationFlow” private flow discussed above. Recall that the “ValidationFlow” flow will throw an exception if a message that does not pass validation is encountered.
- The <choice> element is similar to the switch-case construct in Java. It can contain a number of <when> elements, that specify actions to be taken when certain

conditions are met, and an <otherwise> element specifying actions to be taken if none of the conditions in the <when> elements in the <choice> were met.

In this example, messages that has a property with the name “validated” set to the value “true” will be written to the “good-messages” directory. All other messages will be written to the “bad-messages” directory.

- The <when> element has an *expression* attribute with the value “OUTBOUND:validated=true” and an *evaluator* attribute with the value “header”. The message processor inside the <when> element will be invoked when the message header property with the name “validator” in the outbound scope has a value that is equal to “true”.

We will take a closer look at message properties in a [later chapter](#). There is also [a section on message properties](#) in the Recipes-part of this book.

## 5.5. Run the Example Program

We are now ready to try the example program by starting it, dropping the XML example files into the input directory and examine the output directories to see whether the files passed validation or not.

When running the example program, we'll use the technique of running the Mule configuration files as described in the [earlier example](#).

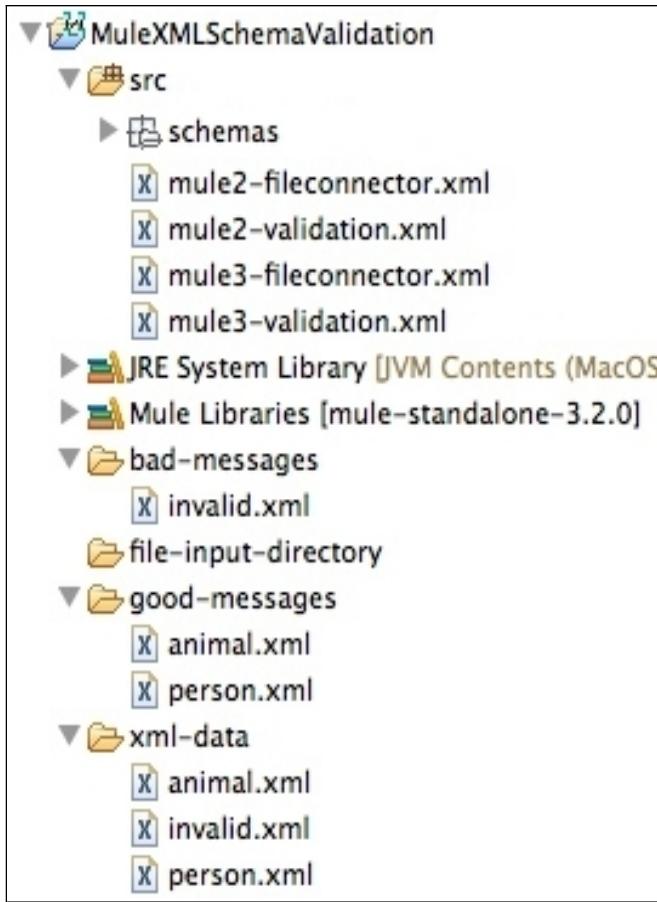
It is assumed that the project is configured with the Mule 3.x distribution on the classpath when starting this section.

- In the Package or Project Explorer in Eclipse, right-click the “mule3-validation.xml” file and select Run As -> Mule Server.
- The Mule 3.x server should start up and display console output indicating a successful startup.
- Refresh the “MuleXMLSchemaValidation” project in Eclipse.  
Note that there is one new directory, “file-input-directory”, created by Mule.
- In the Eclipse package browser, copy the file “animal.xml” and paste it onto the “file-input-directory”.  
Some log output should be generated by Mule.
- Refresh the “MuleXMLSchemaValidation” project in Eclipse.  
Note that there is one new directory, “good-messages”, created by Mule.
- In the Eclipse package explorer, expand the “good-messages” directory.  
The file “animal.xml” that were copied to the input directory has been moved to the “good-messages” output directory. This indicates that the file did pass validation against at least one of our XML schemas.
- Repeat the procedure of copying a file to the “file-input-directory”, but this time use the file “invalid.xml”.
- After having refreshed the “MuleXMLSchemaValidation” project in Eclipse, another directory appears which is named “bad-messages”.
- Examine the contents of the “bad-messages” directory.  
The file “invalid.xml” that we just copied to the input directory has been moved to the “bad-messages” directory. This indicates that the file did not pass validation against any of our XML schemas.
- Finally, repeat the process for the XML-file “person.xml”.  
This file should also pass validation and thus be moved to the “good-messages” directory.

(continued on next page)

- Again, refresh the project in Eclipse.

The files of the project should look like this in the Eclipse Package Explorer:



Project files after having copied the three sample XML-files to the input directory and Mule having performed validation of the files.

Run the Mule 2.x version of the example program:

- Change the Mule distribution to Mule 2.x, as described in [appendix B](#).
- Delete the three directories created by Mule and their contents.  
That is, the “file-input-directory”, the “good-messages” and the “bad-messages” directories.
- Start the example program by right-clicking the “mule2-validation.xml” file and select Run As->Mule Server.

The result of running the Mule 2.x version should be similar to what we've already seen.

We see that Mule is able to route messages with an XML payload depending on whether the XML data passes validation against certain schemas.

## 5.6. Validation and XML Schema Imports

This chapter could have ended here if it were not for a complication that arises if an XML schema used by the Mule schema validator tries to import another XML schema that defines some types that the first schema uses.

### Modify the XML Schema

To make the problem visible, we will modify the XML schema “schema3.xsd” to use the custom date-type defined in the XML schema “schema4.xsd”.

- Modify the XML schema “schema3.xsd” to look like this (modifications highlighted):

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema
    xmlns:xss="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.ivan.com/schemas"
    targetNamespace="http://www.ivan.com/schemas"
    attributeFormDefault="unqualified">

    <xss:include schemaLocation="schema4.xsd"/>

    <xss:element name="animal" type="Animal"/>
    <xss:complexType name="Animal">
        <xss:sequence>
            <xss:element name="name" type="xs:string" nillable="false"/>
            <xss:element name="birthday" type="ivan_date_type"/>
        </xss:sequence>
    </xss:complexType>
</xss:schema>
```

- Change the Mule distribution to Mule 3.x, as described in [appendix B](#).
- In the Package or Project Explorer in Eclipse, right-click the “mule3-validation.xml” file and select Run As -> Mule Server.
- The Mule server is, after some time, terminated due to an error and the console shows, among long stack-traces, the following error message from Mule:

```
...
ERROR 2011-10-14 06:42:44,408 [main] org.mule.MuleServer:
*****
Message : src-resolve: Cannot resolve the name 'ivan_date_type' to a(n) 'type
definition' component.
Type     : org.mule.api.lifecycle.InitialisationException
Code     : MULE_ERROR-71999
JavaDoc :
http://www.mulesoft.org/docs/site/current3/apidocs/org/mule/api/lifecycle/Initialisation
Exception.html
Object   : org.mule.module.xml.filters.SchemaValidationFilter@53d9f80
*****
...
```

Apparently, Mule has not been able to read the XML schema “schema4.xsd” in which the datatype “ivan\_date\_type” is defined.

## **Implement a Resource Resolver**

If we take a look at the API documentation of the class *SchemaValidationFilter* that implements the Mule <schema-validation-filter>, we can see that there is a getter- and setter-method for a resource-resolver: *getResourceResolver* and *setResourceResolver*.

The type of the resource resolver is *LSResourceResolver*. This is an interface that can be found in the JavaSE API documentation. Looking at that API documentation, we can read that a resource-resolver implementing this interface allows an application to intercept the inclusion of external entities – this sounds very appropriate in our case!

- In the source directory of the project, create the package “com.ivan.resolver”.
- Implement a custom resource resolver as in the listing below:

```
package com.ivan.resolver;

import java.net.URL;
import java.util.logging.Logger;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.bootstrap.DOMImplementationRegistry;
import org.w3c.dom.ls.DOMImplementationLS;
import org.w3c.dom.ls.LSInput;
import org.w3c.dom.ls.LSResourceResolver;

/**
 * A custom resource resolver used by the DOM parser to locate
 * external entities included in XML schemas.
 *
 * @author Ivan Krizsan
 */
public class MyResourceResolver implements LSResourceResolver
{
    /* Constant(s): */
    /** Package in which XML schemas are to be located. */
    private final static String SCHEMA_PACKAGE = "schemas/";

    /* Class variable(s): */
    private final static Logger sLogger = Logger
        .getLogger(MyResourceResolver.class.getName());

    /* Instance variable(s): */
    private DOMImplementationLS mDOMImplLS;

    public MyResourceResolver() throws Exception
    {
        /*
         * We need a DOM impl LS, in order to be able to create LSInput
         * objects for the resources we resolve.
         */
        DOMImplementationRegistry theDOMImplRegistry =
            DOMImplementationRegistry.newInstance();
        DOMImplementation theDOMImpl =
            theDOMImplRegistry.getDOMImplementation("XML 3.0");
        mDOMImplLS = (DOMImplementationLS)theDOMImpl.getFeature("LS", "3.0");
    }

    /*
     * (non-Javadoc)
     *
     * @see
     * org.w3c.dom.ls.LSResourceResolver#resolveResource(java.lang
     * .String, java.lang.String, java.lang.String, java.lang.String,
     * java.lang.String)
     */
    @Override
    public LSInput resolveResource(String inType, String inNamespaceURI,
        String inPublicId, String inSystemId, String inBaseURI)
    {
        LSInput theRsrcInput = null;

        /* Log entering this method. */
    }
}
```

```

sLogger.info("Resolve resource with parameters: " + inType + ", "
+ inNamespaceURI + "," + inPublicId + "," + inSystemId + ","
+ inBaseURI);

/*
 * If we can retrieve an URL for the resource, then create a
 * DOM input object for the resource.
 */
URL theURL =
    this.getClass().getClassLoader()
        .getResource(SCHEMA_PACKAGE + inSystemId);
sLogger.info("Resource URL: " + theURL);
if (theURL != null)
{
    theRsrcInput = mDOMImpls.createLSInput();
    theRsrcInput.setBaseURI(inBaseURI);
    theRsrcInput.setSystemId(theURL.toString());
}

return theRsrcInput;
}
}

```

Note that:

- The resource resolver class implements the *LSResourceResolver* interface. This is the type expected by Mule's schema validator.
- The package in which XML schemas are located is hardcoded using the constant *SCHEMA\_PACKAGE*.
- An object implementing the *DOMImplementationLS* interface is created in the resource resolver class' constructor. This object is needed in order to create the appropriate kind of objects, implementing *LSInput*, that specifies how to retrieve a resolved resource. Yes, objects of the type *LSInput* can be created by new-ing a certain class, but this class is a private implementation class and should not be instantiated that way.
- The *resolveResource* method takes a number of parameters. We will see what the different parameters contain when running the example.
- A classloader is used to obtain the URL of the resource and, if successful, an object implementing the *LSInput* interface is created and properties specifying the resource are set on the object.

## Modify the Schema Validators

Looking at the documentation for the Mule 2.x and 3.1 <schema-validation-filter>, we soon discover that there is no attribute allowing us to set a custom resource resolver.

Starting with Mule 3.2, the *resourceResolver-ref* attribute has been added to <schema-validation-filter>.

For the cases in which this attribute does not exist, the solution is to use a custom filter, to which we can supply arbitrary properties.

- Modify the “mule2-validation.xml” Mule configuration file to look like this (modifications highlighted):

```
<?xml version="1.0" encoding="UTF-8"?>
<mule
    xmlns="http://www.mulesource.org/schema/mule/core/2.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
    xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:mule-xml="http://www.mulesource.org/schema/mule/xml/2.2"
    xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
    xsi:schemaLocation="
        http://www.mulesource.org/schema/mule/file/2.2
        http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

        http://www.mulesource.org/schema/mule/core/2.2
        http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

        http://www.mulesource.org/schema/mule/stdio/2.2
        http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

        http://www.mulesource.org/schema/mule/xml/2.2
        http://www.mulesource.org/schema/mule/xml/2.2/mule-xml.xsd

        http://www.mulesource.org/schema/mule/vm/2.2
        http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">

    <spring:beans>
        <spring:import resource="mule2-fileconnector.xml"/>
        <!--
            Declare a resource resolver so we can inject it into the
            schema validation filter.
        -->
        <spring:bean
            id="myResourceResolver"
            class="com.ivan.resolver.MyResourceResolver">
        </spring:bean>
    </spring:beans>

    <!--
        The <model> in this example contains four different services:
        - An input service responsible for accepting messages in the
          form of files placed in a certain directory.
        - A validation service that validates messages.
          Depending on whether a message having passed validation or
          not, it is either passed on to the processing service or to
          the error service.
        - A processing service.
          This service receives only messages having passed validation.
          In this example, these messages are just written to the
          good messages directory.
        - An error service.
          This service receives only messages that do not pass validation.
          in this example, these messages are just written to the bad
          messages directory.
    -->
    <model name="xmlValidationModel">
        <service name="xmlInputService">
            <inbound>
```

```

<!--
    Input endpoint receiving files in a directory,
    applying filtering by filename.
-->
<file:inbound-endpoint connector-ref="fileConnector"
    path=".//file-input-directory">
    <file:filename-wildcard-filter
        pattern="*.xml"/>
</file:inbound-endpoint>
</inbound>
<outbound>
    <pass-through-router>
        <vm:outbound-endpoint path="message.validation"/>
    </pass-through-router>
</outbound>
</service>

<!--
    Service that validates messages and pass them on depending
    on the result of the validation.
-->
<service name="xmlValidationService">
    <inbound>
        <vm:inbound-endpoint path="message.validation"/>
    <!--
        A selective consumer router can apply one filter
        to incoming messages.
        If a message is matched by the filter, in this example
        passes XML validation, the message is passed on to
        the component (if any) and later to any outbound
        endpoint of the service.
        If a message is not matched by the filter, that is
        do not pass XML validation, the catch-all strategy
        is invoked if such is configured. Otherwise the message
        is ignored and a warning message logged.
    -->
    <selective-consumer-router>
        <!--
            The OR filter contains two or more other filters and
            accept a message if it matches the criteria of at least
            one of the contained filters.
        -->
        <or-filter>
            <!--
                A message sent to a schema validation filter is
                matched if it passes validation by the schema(s)
                of the filter.
                Due to a bug in the XML parser used for validation,
                multiple names with the same namespace cannot be
                used by a single schema validation filter.
                A work-around is to use an OR filter in combination
                with a schema validation filter for each schema.
                The returnResult attribute has been set to false in
                order to increase efficiency of the filter.
            -->
            <custom-filter
                class="org.mule.module.xml.filters.SchemaValidationFilter">
                <spring:property name="schemaLocations"
                    value="schemas/schema2.xsd"/>
                <spring:property name="returnResult"
                    value="false"/>
                <spring:property name="resourceResolver"
                    ref="myResourceResolver"/>
            </custom-filter>
            <custom-filter
                class="org.mule.module.xml.filters.SchemaValidationFilter">
                <spring:property name="schemaLocations"
                    value="schemas/schema3.xsd"/>
                <spring:property name="returnResult"
                    value="false"/>
                <spring:property name="resourceResolver"
                    ref="myResourceResolver"/>
            </custom-filter>
        </or-filter>
    </selective-consumer-router>
    <!--
        Catch-all strategy invoked when a message does not
    -->

```

```

        pass XML validation.
        Route the message to the error service.
    -->
    <forwarding-catch-all-strategy>
        <vm:outbound-endpoint path="message.error"/>
    </forwarding-catch-all-strategy>
</inbound>
<outbound>
    <!--
        Messages arriving here are valid.
        Pass these messages on to the processing service.
    -->
    <pass-through-router>
        <vm:outbound-endpoint path="message.processing"/>
    </pass-through-router>
</outbound>
</service>

<!--
    Service that receives messages having passed validation.
    Writes the message to the good messages directory using the
    original file name.
-->
<service name="xmlProcessingService">
    <inbound>
        <vm:inbound-endpoint path="message.processing"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <file:outbound-endpoint path="good-messages/" 
                outputPattern="#{header:originalFilename}"/>
        </pass-through-router>
    </outbound>
</service>

<!--
    Service that receives messages having failed validation.
    Writes the message to the bad messages directory using the
    original file name.
-->
<service name="xmlErrorService">
    <inbound>
        <vm:inbound-endpoint path="message.error"/>
    </inbound>
    <outbound>
        <pass-through-router>
            <file:outbound-endpoint path="bad-messages/" 
                outputPattern="#{header:originalFilename}"/>
        </pass-through-router>
    </outbound>
</service>
</model>
</mule>

```

(continued on next page)

Below will look at the Mule 3.2 version of the Mule 3.x configuration file. A configuration file for earlier versions of Mule 3.x would use the technique used earlier for Mule 2.x.

- Modify the “mule3-validation.xml” Mule configuration file to look like this (modifications highlighted):

```

<?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:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:file="http://www.mulesoft.org/schema/mule/file"
    xmlns:mule-xml="http://www.mulesoft.org/schema/mule/xml"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/file
        http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/xml
        http://www.mulesoft.org/schema/mule/xml/3.2/mule-xml.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <spring:beans>
        <spring:import resource="mule3-fileconnector.xml"/>
        <spring:bean
            id="myResourceResolver"
            class="com.ivan.resolver.MyResourceResolver"/>
    </spring:beans>

    <!--
        This is a private flow, that is a flow without any message sources.
        Such a flow can only be referenced from within the same Mule instance,
        not from outside the JVM in question.
        This flow validates the message payload against two XML schemas.
        If validation succeeds, a message property 'validated' is set to
        'true'.

        A private flow has its own context and exception strategy.
        This means that exceptions will not propagate out of the private
        flow. If an exception is thrown, the message payload is set to null.

        The processing strategy of the flow must be synchronous,
        otherwise the flag indicating a message having passed validation
        may be checked before it is set.
    -->
    <flow name="ValidationFlow" processingStrategy="synchronous">
        <!--
            Any message properties set here will not be present on the
            message if the filter below does not match the message.
        -->

        <message-filter throwOnUnaccepted="false">
            <!--
                The OR filter contains two or more other filters and
                accept a message if it matches the criteria of at least
                one of the contained filters.
            -->
            <or-filter>
                <!--
                    A message sent to a schema validation filter is
                    matched if it passes validation by the schema(s)
                    of the filter.
                    Due to a bug in the XML parser used for validation,
                    multiple names with the same namespace cannot be
                    used by a single schema validation filter.
                    A work-around is to use an OR filter in combination
                    with a schema validation filter for each schema.
                -->
        
```

```

The returnResult attribute has been set to false in
order to increase efficiency of the filter.
Note that in Mule 3.2, the resource resolver can
be set using an attribute on the schema validation
filter element. In earlier versions this had to be
accomplished using a custom filter and injecting the
resource resolver using Spring properties.

-->
<mule-xml:validation-filter
    schemaLocations="schemas/schema2.xsd"
    returnResult="false"
    resourceResolver-ref="myResourceResolver"/>
<mule-xml:validation-filter
    schemaLocations="schemas/schema3.xsd"
    returnResult="false"
    resourceResolver-ref="myResourceResolver"/>
</or-filter>
</message-filter>

<!--
    If message passes validation, set a message property that
    indicates this.
-->
<message-properties-transformer scope="outbound">
    <add-message-property key="validated" value="true"/>
</message-properties-transformer>
</flow>

<!--
    This is the main flow that accepts input from a directory in the
    filesystem, sends messages to the private validation flow.
    Messages having passed validation are written to one directory
    (good messages) while messages causing exceptions, typically that
    does not pass validation, are written to another directory
    (bad messages).
-->
<flow name="SortingFlow">
    <!--
        Input endpoint receiving XML files from a directory.
    -->
    <file:inbound-endpoint connector-ref="fileConnector"
        path=".//file-input-directory">
        <file:filename-wildcard-filter pattern="*.xml"/>
    </file:inbound-endpoint>

    <!-- Delegate message validation to a private flow. -->
    <flow-ref name="ValidationFlow"/>

    <choice>
        <!--
            A message is only considered to have passed validation when
            the 'validated' property has been set to 'true'.
            Note that the property is to be found in the outbound
            property scope.
            The 'header' evaluator evaluates the part of the message
            header specified by the expression attribute.
        -->
        <when expression="OUTBOUND:validated=true" evaluator="header">
            <!--
                Having passed validation, messages are written to
                the good-messages directory.
            -->
            <file:outbound-endpoint path="good-messages/"
                outputPattern="#{header:originalFilename}"/>
        </when>
        <otherwise>
            <!--
                Messages in which the property indicating having passed
                validation is not present or is not set to 'true' are
                written to the bad-messages directory.
            -->
            <file:outbound-endpoint path=".//bad-messages/"
                outputPattern="#{header:originalFilename}"/>
        </otherwise>
    </choice>
</flow>
</mule>

```

Note that:

- A Spring bean implemented by the custom resource resolver we created in the previous section is created in the <spring:beans> element.
- A new attribute, *resourceResolver-ref*, has been added to the <schema-validation-filter> elements.  
The value of the attribute is the name of the resource resolver Spring bean.

### Run the Example Program

Now we are ready to try the example program again!

- In the Package or Project Explorer in Eclipse, right-click the “mule3-validation.xml” file and select Run As -> Mule Server.
- The Mule 3.x server should start up and display console output indicating a successful startup. Note the following console output in red, which tells us that our resource resolver has been invoked and shows us the parameters. In addition, the resolved URL of the XML schema is also shown.

```
...
INFO 2011-10-14 07:01:20,040 [main] org.mule.module.xml.filters.SchemaValidationFilter:
Schema factory implementation:
com.sun.org.apache.xerces.internal.jaxp.validation.XMLSchemaFactory@3c9ff588
INFO 2011-10-14 07:01:20,050 [main] org.mule.module.xml.filters.SchemaValidationFilter:
Schema factory implementation:
com.sun.org.apache.xerces.internal.jaxp.validation.XMLSchemaFactory@2f56f920
Oct 14, 2011 7:01:20 AM com.ivan.resolver.MyResourceResolver resolveResource
INFO: Resolve resource with parameters:
http://www.w3.org/2001/XMLSchema,http://www.ivan.com/schemas,null,schema4.xsd,null
Oct 14, 2011 7:01:20 AM com.ivan.resolver.MyResourceResolver resolveResource
INFO: Resource URL:
file:/Users/ivan/EclipseWorkspace/MULE_EXAMPLES/MuleXMLSchemaValidation/bin/schemas/sche
ma4.xsd
INFO 2011-10-14 07:01:20,131 [main] org.mule.construct.FlowConstructLifecycleManager:
Initialising flow: ValidationFlow
...
...
```

- Drop the example XML files into the “file-input-directory” and verify that they are distributed to the proper directories, as before.

Run the Mule 2.x version of the example program:

- Change the Mule distribution to Mule 2.x, as described in [appendix B](#).
- Delete the three directories created by Mule and their contents.  
That is, the “file-input-directory”, the “good-messages” and the “bad-messages” directories.
- Start the example program by right-clicking the “mule2-validation.xml” file and select Run As->Mule Server.  
The result should be similar to that obtained when running the Mule 3.x version.

This concludes the example in this chapter.

We have seen how to validate XML messages in Mule, how to use multiple XML schemas with the same namespace when validating XML and how to implement and use a custom resource resolver with the Mule schema validator.

## 6. Extract XML Message Payload with XPath

In this chapter we will look at how to use XPath in Mule to extract a selected part of the XML in a message. We will also see how to handle XML namespaces in Mule messages.

We will use the technique of running the Mule configuration files, described in the previous chapters, so we will not need a starter class.

### 6.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “MuleXPathProcessing”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded. In addition to the basic project setup, we will perform an additional setup for this particular project:

- In the project root, create a directory named “xml-data”.  
This directory will contain the XML example files.

### 6.2. Create XML Data Files

The following file contains input data that we will feed into Mule. It is to be located in the directory “xml-data”.

- Create the file “PhoneBook.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ivan:PhoneBook
    xmlns:ivan="http://www.ivan.com/schemas/addressbook"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <ivan:Person lastUpdate="2011-07-21" category="actor">
        <ivan:FirstName>Gösta</ivan:FirstName>
        <ivan:LastName>Ekman</ivan:LastName>
        <ivan:Phone>044-123123</ivan:Phone>
    </ivan:Person>

    <ivan:Person lastUpdate="2011-07-22" category="athlete">
        <ivan:FirstName>J-O</ivan:FirstName>
        <ivan:LastName>Waldner</ivan:LastName>
        <ivan:Phone>08-12312312</ivan:Phone>
    </ivan:Person>

    <ivan:Person lastUpdate="2010-12-11" category="actor">
        <ivan:FirstName>Dolph</ivan:FirstName>
        <ivan:LastName>Lundgren</ivan:LastName>
        <ivan:Phone>0706-32132156</ivan:Phone>
    </ivan:Person>
</ivan:PhoneBook>
```

## 6.3. Create the Mule Configuration Files

This example contains two Mule configuration files per Mule version, for a total of four Mule configuration files.

Two of the configuration files are the file connector configuration files that should be familiar to readers of previous examples.

### ***Create the Mule File Connector Configuration Files***

The Mule file connector specifies how files should be read or written from or to a directory in the local file system. These configuration files are identical to those used in, among other, the [previous example](#).

Both files are to be located in the root of the project's source directory.

The Mule 2.x configuration file is named “mule2-fileconnector.xml”:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/file/2.2
          http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd">

    <file:connector name="fileConnector" streaming="false" pollingFrequency="1000">
        <file:expression-filename-parser/>
    </file:connector>
</mule>
```

The Mule 3.x configuration file is named “mule3-fileconnector.xml”:

```
<?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:file="http://www.mulesoft.org/schema/mule/file"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/file
          http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd">

    <file:connector name="fileConnector" streaming="false" pollingFrequency="1000">
        <file:expression-filename-parser/>
    </file:connector>
</mule>
```

These file connectors have the following properties:

- The name is “fileConnector”.  
Used when referring to the connectors from the file inbound endpoints.
- Pass a file, not a file input stream, as the message payload.
- Check the input directory once every second (1000 mS).
- Store the name of a file in a message property.

## Create the Main Mule Configuration Files

The main Mule configuration files are quite different, although they roughly represent the same structure.

These Mule configuration files are also located in the root of the source directory of the project.

## The Mule 2.x Configuration File

The Mule 2.x configuration file consists of one single service with two endpoints:

- One inbound file endpoint.  
This endpoint accepts messages in the form of files with the “.xml” suffix placed in an input directory with the name “file-input-directory”.
- One outbound file endpoint.  
This endpoint writes received files, using their original name, to a directory with the name “processed-xpath”.

The parts of importance to this example are the transformers in the inbound endpoint.

First, let's take a look at the entire configuration file, which is named “mule2-xpath.xml” and located in the root of the source directory:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:file="http://www.mulesource.org/schema/mule/file/2.2"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xmlns:mule-xml="http://www.mulesource.org/schema/mule/xml/2.2"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/file/2.2
          http://www.mulesource.org/schema/mule/file/2.2/mule-file.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

          http://www.mulesource.org/schema/mule/xml/2.2
          http://www.mulesource.org/schema/mule/xml/2.2/mule-xml.xsd

          http://www.mulesource.org/schema/mule/vm/2.2
          http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">

    <spring:beans>
        <spring:import resource="mule2-fileconnector.xml"/>
    </spring:beans>

    <model name="xpathProcessingModel">
        <service name="xpathProcessingService">
            <inbound>
                <file:inbound-endpoint
                    connector-ref="fileConnector"
                    path=".//file-input-directory">
                    <file:filename-wildcard-filter pattern="*.xml"/>
                <!--
                    Must transform the XML data to DOM in order to receive
                    a DOM node as result of the XPath transform.
                    Otherwise we will receive the contents of the elements
                    in the extracted XML fragment.
                -->
            </inbound>
        </service>
    </model>

```

```

-->
<mule-xml:xml-to-dom-transformer
    returnClass="org.w3c.dom.Document" />

<!--
    Extract a person from the phonebook which is alone
    in his/her category.
-->
<mule-xml>xpath-extractor-transformer
    expression="/ivan:PhoneBook/ivan:Person[not(@category=following-
sibling::ivan:Person/@category)]"
    resultType="NODE">
<!--
    The XML data we are processing uses a namespace
    and we use a namespace prefix in the XPath expression.
    We must thus link the namespace prefix and
    the namespace URI.
-->
<mule-xml:namespace
    prefix="ivan"
    uri="http://www.ivan.com/schemas/addressbook"/>
</mule-xml>xpath-extractor-transformer>

<!--
    Must transform the message from DOM to text to make
    the resulting XML readable.
-->
<mule-xml:dom-to-xml-transformer/>
</file:inbound-endpoint>
</inbound>
<outbound>
    <pass-through-router>
        <file:outbound-endpoint
            path="processed-xpath"
            outputPattern="#{header:originalFilename}">
            <!-- Format the XML so that it is easier to read. -->
            <mule-xml:xml-prettyprinter-transformer/>
        </file:outbound-endpoint>
    </pass-through-router>
</outbound>
</service>
</model>
</mule>

```

Note that:

- The model “xpathProcessingModel” contains one single service: “xpathProcessingService”.
- The <inbound> element of the “xpathProcessingService” contains a file endpoint defined by a <file:inbound-endpoint> element.  
As in earlier examples, it accepts files with the “.xml” suffix that are placed in a directory with the name “file-input-directory” as messages.
- The <file:inbound-endpoint> element contains three transformers.
- The first transformer in the <file:inbound-endpoint> element is a XML-to-DOM transformer (in the <mule-xml:xml-to-dom-transformer> element).  
This transformer transforms XML data to an DOM document contained in the instance of the class *org.w3c.dom.Document*.  
Since we want to extract an XML fragment from the message, we first need to transform the textual representation of the XML data to a DOM object tree on which the XPath expression can operate.
- The second transformer in the <file:inbound-endpoint> element is the XPath extractor transformer (in the <mule-xml>xpath-extractor-transformer> element).  
This transformer applies an XPath expression to the incoming message and returns the

## resulting XML

- The XPath extractor transformer has two attributes: *expression* and *resultType*.  
The *expression* attribute contains the XPath expression. A brief explanation of the XPath expression used in this element will appear in a subsequent section of this chapter.  
The *resultType* attribute specifies what kind of object is to be produced when evaluating the XPath expression. The following types are available, with STRING being the default:  
STRING – java.lang.String  
NODE – org.w3c.dom.Node  
NODESET – org.w3c.dom.NodeList  
BOOLEAN – java.lang.Boolean  
NUMBER – java.lang.Double
- The <mule-xml:xpath-extractor-transformer> element contains a <mule-xml:namespace> element.  
The namespace of the XML document from which we are to extract XML data is “<http://www.ivan.com//schemas/addressbook>”. In the XPath expression, the namespace prefix “ivan” is used. The <mule-xml:namespace> element is used to link a namespace prefix with a namespace URI.
- The Mule documentation claims that namespaces for XPath expressions can be declared globally, using a <mule-xml:namespace-manager> element.  
Regrettably, there is no support for a global namespace manager in the XPath extractor transformer until in Mule 3.2.
- The final transformer in the <file:inbound-endpoint> element is a DOM-to-XML transformer (in the <mule-xml:dom-to-xml-transformer/> element).  
This transformer converts an XML payload in the form of a DOM document to a serialized string representation.  
We do this since we are going to write the XML data to a file that is to be read by a human.
- The <outbound> element of the “xpathProcessingService” contains a <pass-through-router> that in turn contains a <file:outbound-endpoint> element.  
The <file:outbound-endpoint> specifies that messages are to be written to a directory named “processed-xpath” using the original file name from a Mule property.
- The <file:outbound-endpoint> element contains a <mule-xml:xml-prettyprinter-transformer> element.  
The pretty-printer transformer formats the XML string that is written to the file – otherwise the XML will end up on one single row.

## The Mule 3.x Configuration File

The Mule 3.x configuration file is very similar to the Mule 2.x configuration file, but instead of declaring a service, it declares a flow.

The Mule 3.x configuration file is named “mule3-xpath.xml”, is also located in the root of the source directory, and looks like this:

```
<?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:stdio="http://www.mulesoft.org/schema/mule/stdio"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:file="http://www.mulesoft.org/schema/mule/file"
    xmlns:mule-xml="http://www.mulesoft.org/schema/mule/xml"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/file
        http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

        http://www.mulesoft.org/schema/mule/stdio
        http://www.mulesoft.org/schema/mule/stdio/3.2/mule-stdio.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/xml
        http://www.mulesoft.org/schema/mule/xml/3.2/mule-xml.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <spring:beans>
        <spring:import resource="mule3-fileconnector.xml"/>
    </spring:beans>

    <!--
        This element declares a global namespace manager which links
        namespace prefixes and URIs.
        Support for a global namespace manager in the XPath transformer
        was not implemented until in Mule 3.2.
    -->
    <mule-xml:namespace-manager includeConfigNamespaces="false">
        <mule-xml:namespace
            prefix="ivan"
            uri="http://www.ivan.com/schemas/addressbook"/>
    </mule-xml:namespace-manager>

    <flow name="xpathProcessingFlow">
        <!--
            Input endpoint receiving XML files from a directory.
        -->
        <file:inbound-endpoint connector-ref="fileConnector"
            path=".//file-input-directory">
            <file:filename-wildcard-filter pattern="*.xml"/>

        <!--
            Must transform the XML data to DOM in order to receive
            a DOM node as result fo the XPath transform.
            Otherwise we will receive the contents of the elements
            in the extracted XML fragment.
        -->
        <mule-xml:xml-to-dom-transformer returnClass="org.w3c.dom.Document" />

        <!--
            Extract a person from the phonebook which is alone
            in his/her category.
        -->
        <mule-xml>xpath-extractor-transformer
            expression="/ivan:PhoneBook/ivan:Person[not(@category=following-
sibling::ivan:Person/@category)]"
            resultType="NODE">
        <!--
            If using Mule < 3.2 or Mule >= 3.2 without a global
        -->
```

```

namespace manager, namespace prefixes and URIs
must be linked this way.
With Mule 3.2, the <mule-xml:namespace> element can be
commented out since we have a global namespace manager
in this configuration file.
-->
<!--
<mule-xml:namespace
    prefix="ivan"
    uri="http://www.ivan.com/schemas/addressbook" />
-->
</mule-xml:xpath-extractor-transformer>

<!--
    Must transform the message from DOM to text to make
    the resulting XML readable.
-->
<mule-xml:dom-to-xml-transformer/>
</file:inbound-endpoint>

<file:outbound-endpoint path="processed-xpath"
    outputPattern="#{header:originalFilename}">
    <!-- Format the XML so that it is easier to read. -->
    <mule-xml:xml-prettyprinter-transformer/>
</file:outbound-endpoint>
</flow>
</mule>

```

Note that:

- Immediately after the <spring:beans> element including the file connector, there is a <mule-xml:namespace-manager>. This element shows how XML namespace prefixes and URIs can be linked globally. If you are using Mule 3.2 or later, declaring the namespace this way is sufficient.
- The file contains a single flow, “xpathProcessingFlow”.
- The flow contains a file endpoint defined by a <file:inbound-endpoint> element. As in earlier examples, it accepts files with the “.xml” suffix that are placed in a directory with the name “file-input-directory” as messages.
- The <file:inbound-endpoint> element contains three transformers.
- The first transformer in the <file:inbound-endpoint> element is a XML-to-DOM transformer (in the <mule-xml:xml-to-dom-transformer> element). This transformer transforms XML data to an DOM document contained in the instance of the class *org.w3c.dom.Document*. Since we want to extract an XML fragment from the message, we first need to transform the textual representation of the XML data to a DOM object tree on which the XPath expression can operate.
- The second transformer in the <file:inbound-endpoint> element, the <mule-xml>xpath-extractor-transformer> element, is the XPath extractor transformer. This transformer applies an XPath expression to the incoming message and returns the resulting XML
- The XPath extractor transformer has two attributes: *expression* and *resultType*. The *expression* attribute contains the XPath expression. A brief explanation of the XPath expression used in this element will appear in the next section of this chapter. The *resultType* attribute specifies what kind of object is to be produced when evaluating the XPath expression. The following types are available, with STRING being the default: STRING – java.lang.String

NODE – org.w3c.dom.Node  
NODESET – org.w3c.dom.NodeList  
BOOLEAN – java.lang.Boolean  
NUMBER – java.lang.Double

- The <mule-xml>xpath-extractor-transformer> element contains a <mule-xml:namespace> element.  
The namespace of the XML document from which we are to extract XML data is “<http://www.ivan.com//schemas/addressbook>”. In the XPath expression, the namespace prefix “ivan” is used. The <mule-xml:namespace> element is used to link a namespace prefix with a namespace URI.  
Since I was using Mule 3.2 when developing this example and have a global namespace-manager, I have commented out this element.
- The final transformer in the <file:inbound-endpoint> element is a DOM-to-XML transformer (in the <mule-xml:dom-to-xml-transformer/> element).  
This transformer converts an XML payload in the form of a DOM document to a serialized string representation.  
We do this since we are going to write the XML data to a file that is to be read by a human.
- The flow also contains a <file:outbound-endpoint> element.  
This outbound endpoint specifies that messages are to be written to a directory named “processed-xpath” using the original file name from a Mule property.
- The <file:outbound-endpoint> element contains a <mule-xml:xml-prettyprinter-transformer> element.  
The pretty-printer transformer formats the XML string that is written to the file – otherwise the XML will end up on one single row.

### **The XPath Expression**

This section contains a very brief explanation of the XPath expression used in the above Mule configuration files. The expression looks like this:

```
/ivan:PhoneBook/ivan:Person[not(@category=following-sibling::ivan:Person/@category)]
```

This expression selects elements in an XML document that fills the following criteria:

- Has a parent element <PhoneBook> belonging to the namespace with the namespace prefix “ivan”.
- Is named <Person> and belongs to the namespace with the namespace prefix “ivan”.
- Has a *category* attribute which value is not the same as any <Person> elements in the namespace with the namespace prefix “ivan” that has the same <PhoneBook> element as parent and that comes after the <Person> element currently being evaluated.

Expressed in plain words, the aim of the XPath expression is to find a person in a phone-book that belongs to category in which that person is alone (slightly simplified).

## 6.4. Run the Example Program

We are now ready to try the example program out by starting the appropriate Mule configuration file, copying the input file to the input-directory and then observing the result produced in the output directory.

When running the example program, we'll use the technique of running the Mule configuration files as described in an [earlier example](#).

It is assumed that the project is configured with the Mule 3.x distribution on the classpath when starting this section.

- In the Package or Project Explorer in Eclipse, right-click the “mule3-xpath.xml” file and select Run As -> Mule Server.
- The Mule 3.x server should start up and display console output indicating a successful startup.
- Refresh the “MuleXPathProcessing” project in Eclipse.  
Note that there is one new directory, “file-input-directory”, created by Mule.
- In the Eclipse package browser, copy the file “PhoneBook.xml” and paste it onto the “file-input-directory”.  
Some log output should be generated by Mule.
- Refresh the “MuleXPathProcessing” project in Eclipse.  
Note that there is one new directory, “processed-xpath”, created by Mule.
- In the Eclipse package explorer, expand the “processed-xpath” directory and open the file “PhoneBook.xml” located in that directory.  
It should have the following content (some formatting has been applied):

```
<?xml version="1.0" encoding="UTF-8"?>
<ivan:Person xmlns:ivan="http://www.ivan.com/schemas/addressbook"
    category="athlete" lastUpdate="2011-07-22">
    <ivan:FirstName>J-O</ivan:FirstName>
    <ivan:LastName>Waldner</ivan:LastName>
    <ivan:Phone>08-12312312</ivan:Phone>
</ivan:Person>
```

- The root element of the processed XML file is <ivan:Person>. Indeed, this person is the only athlete in our phone-book.
- The namespace has been preserved correctly.
- In the Eclipse package browser select another XML file (a Mule configuration file is fine) and copy it to the “file-input-directory”.
- Refresh the project in Eclipse.
- Open the new file in the “processed-xpath” directory.  
It should have the following contents:

```
# #sr##org.mule.transport.NullPayload1#L5U      # ##xp
```

- Look at the Mule API documentation for the class *NullPayload*. We can see that instances of this class represent a Mule message with empty (null) payload. This is understandable, since the XPath expression did not find any matching nodes in the second file.

Run the Mule 2.x version of the example program:

- Change the Mule distribution to Mule 2.x, as described in [appendix B](#).
- Delete the two directories created by Mule and their contents.  
That is, the “file-input-directory” and the “processed-xpath” directories.
- Start the example program by right-clicking the “mule2-xpath.xml” file and select Run As->Mule Server.

The result of running the Mule 2.x version should be the same as with the Mule 3.x version.

## 6.5. Exercises

The example program is completed, but you are encouraged to some experimentation. Some suggestions for exercises to undertake on your own are:

- Remove the XML pretty-printer transformer.
- Remove the XML-to-DOM transformer.
- Change the resultType of the XPath transformer.  
Recall that the possible values are STRING, NUMBER, BOOLEAN, NODE and NODESET.
- Modify the XPath expression.

This concludes this chapter.

## 7. Monitoring Mule

Instances of Mule 2.x and 3.x can be monitored, and to some extend controlled, using JMX (the Java Management eXtension). In this chapter we will look at how to configure JMX management in a Mule instance, some examples of what can be learned using the JMX management and some examples of how we can control a Mule instance using JMX.

We will also look at two alternatives regarding JMX monitoring of a Mule instance - JConsole and the MX4J JMX web management console.

While this chapter will foremost discuss monitoring and management of Mule instances using JMX, we need a Mule instance to monitor and manage. For this purpose, we will create a small example program that, using either Mule 2.x or Mule 3.x, exposes a SOAP web service.

### 7.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “MonitoringMule”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded. In addition to the basic project setup, we will perform some additional setup for this particular project:

- In the root of the source code hierarchy, create a package named *com.ivan.muleconfig*. This package is to contain the Mule 2.x and Mule 3.x configuration files of this example.
- Again in the root of the source code hierarchy, create a package named *com.ivan.services*. This package will contain the implementation of the SOAP web service.

### 7.2. Create the Web Service Implementation Class

The web service endpoint implementation class implements a JAX-WS service that extends greetings. This class is common for both the Mule 2.x and Mule 3.x versions of the example program.

- In the com.ivan.services package, create a class named HelloService implemented as follows:

```
package com.ivan.services;

import java.util.Date;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

/**
 * The old faithful Hello service implemented as a JAX-WS service.
 *
 * @author Ivan Krizsan
 */
@WebService(serviceName="HelloService")
public class HelloService
{
    @WebResult(name="message", partName="sayHelloResponse")
    public String sayHello(
        @WebParam(name="name", partName="sayHelloRequest") final String inName)
    {
        String theMessage = "Hello, " + inName + ". The time is now: " +
            (new Date());
        System.out.println("**** Message: " + theMessage);
        return theMessage;
    }
}
```

### 7.3. Create the Mule Configuration Files

There are two Mule configuration files, one for each Mule version. Except for one particular element, the configuration files only contain configuration elements that have been discussed in previous examples. The configuration element we haven't seen before is the element used to configure JMX management of a Mule instance.

But first, let's take a look at the two files and then discuss the new element.

- In the package *com.ivan.muleconfig*, create a file named “mule-config2.xml” that contains the following Mule 2.x configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule
    xmlns="http://www.mulesource.org/schema/mule/core/2.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:management="http://www.mulesource.org/schema/mule/management/2.2"
    xmlns:cxf="http://www.mulesource.org/schema/mule/cxf/2.2"
    xsi:schemaLocation="
        http://www.mulesource.org/schema/mule/management/2.2
        http://www.mulesource.org/schema/mule/management/2.2/mule-management.xsd
        http://www.mulesource.org/schema/mule/cxf/2.2
        http://www.mulesource.org/schema/mule/cxf/2.2/mule-cxf.xsd
        http://www.mulesource.org/schema/mule/core/2.2
        http://www.mulesource.org/schema/mule/core/2.2/mule.xsd">

    <!-- Enable JMX management and the MX4J web console. -->
    <management:jmx-default-config registerMx4jAdapter="true" />

    <model name="HelloModel">
        <service name="HelloService">
            <inbound>
                <cxf:inbound-endpoint
                    address="http://localhost:8081/services/HelloService"/>
            </inbound>
            <component>
                <prototype-object class="com.ivan.services.HelloService"/>
            </component>
        </service>
    </model>
</mule>
```

- In the same package, *com.ivan.muleconfig*, create a file named “mule-config3.xml” that contains the following Mule 3.x configuration:

```
<?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:cxf="http://www.mulesoft.org/schema/mule/cxf"
    xmlns:management="http://www.mulesoft.org/schema/mule/management"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/cxf
        http://www.mulesoft.org/schema/mule/cxf/3.2/mule-cxf.xsd

        http://www.mulesoft.org/schema/mule/management
        http://www.mulesoft.org/schema/mule/management/3.2/mule-management.xsd

        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd">

    <!-- Enable JMX management and the MX4J web console. -->
    <management:jmx-default-config registerMx4jAdapter="true" />

    <flow name="HelloFlow">
        <inbound-endpoint address="http://localhost:8081/services/HelloService"/>
        <cxf:jaxws-service serviceClass="com.ivan.services.HelloService"/>
        <component class="com.ivan.services.HelloService"/>
    </flow>
</mule>
```

The way of using the Apache CXF web service stack to expose an endpoint implementation class annotated with JAX-WS annotations should be familiar from the [example in chapter 1](#).

Note that:

- The first child element to the <mule> elements in both Mule configuration files is the <jmx-default-config> element belonging to the management namespace.  
This element provide the most convenient way to configure a JMX support agent in a Mule instance.
- The <jmx-default-config> element contains the *registerMx4jAdapter* attribute that has the value true.  
Setting this attribute to true enables the MX4J web console that enables us to view JMX managed beans etc in a web browser.

## 7.4. Run the Example Program

Running the example configurations, we will need [soapUI](#) or some similar tool to act as client(s) to the web service we will expose. Before proceeding, make sure that you have soapUI installed. With the example program in place and soapUI at our disposal, we are now ready to start to observe and manage Mule instances.

In this section there are two paths; the Mule 2.x path and the Mule 3.x path. If you choose to run the Mule 2.x version of the example program, then you should continue with the Mule 2.x part of the next section. Conversely, if you run the Mule 3.x version of the example program, then proceed with the Mule 3.x part of the next section.

If there are problems starting the project up in Eclipse, try cleaning and rebuilding the project using the appropriate menu options in the Project menu.

### ***Run the Mule 2.x Example Program***

It is assumed that the project is configured with the Mule 2.x distribution on the classpath when starting this section. If not, please refer to [this section in appendix B](#) on how to configure which Mule distribution to use.

- In the Package or Project Explorer, right-click the “mule-config2.xml” file and select Run As -> Mule Server.
- The Mule 2.x server should start up and display something similar to this on the console (some output omitted to conserve space):

```
...
INFO 2011-09-01 17:32:11,859 [main] org.mule.transport.http.HttpMessageReceiver:
Connected: http://localhost:8081/services/HelloService
INFO 2011-09-01 17:32:11,861 [main] org.mule.model.seda.SedaService: Service
_cxfServiceComponent{http://services.ivan.com/}HelloService609112150 has been started
successfully
INFO 2011-09-01 17:32:11,874 [main] org.mule.DefaultMuleContext:
*****
* Mule ESB and Integration Platform
* Version: 2.2.1 Build: 14422
* MuleSource, Inc.
* For more information go to http://mule.mulesource.org
*
* Server started: 9/1/11 5:32 PM
* Server ID: 276bd063-d4ae-11e0-9d8d-bfcdbc45f4d5f
* JDK: 1.6.0_24 (mixed mode)
* OS encoding: UTF-8, Mule encoding: UTF-8
* OS: Mac OS X (10.6.6, x86_64)
* Host: Computer.local (127.0.0.1)
*
* Agents Running:
*   jmx-log4j
*   Rmi Registry: rmi://localhost:1099
*   Jmx Notification Agent (Listener MBean registered)
*   jmx-agent: service:jmx:rmi:///jndi/rmi://localhost:1099/server
*   MX4J Http adaptor: http://localhost:9999
*   Default Jmx Support Agent
*****
```

Note that:

- The HelloService was successfully started.  
The WSDL of the service can be found at <http://localhost:8081/services/HelloService?wsdl>  
Verify that there indeed is a WSDL at this URL in a browser.
- Last in the framed information about the Mule server, there is a listing of running agents

(“Agents Running”).

- Among the running agents, there is a JMX agent.  
In my case, the JMX agent can be found at:  
service:jmx:rmi:///jndi/rmi://localhost:1099/server
- Also among the running agents, there is an entry for the MX4J HTTP adaptor.  
This is the URL at which we can find the MX4J web console we will look at shortly.  
In my case, the URL is: <http://localhost:9999>
- There are other entries in the list of running agents, but in this example we will only use the two mentioned above.

### **Run the Mule 3.x Example Program**

To run the Mule 3.x version of the example program, change the Mule runtime of the project to the Mule 3.x runtime, as described in [this section of appendix B](#).

- In the Package or Project Explorer, right-click the “mule-config3.xml” file and select Run As -> Mule Server.
- The Mule 3.x server should start up and display something similar to this on the console (some output omitted to conserve space):

```
...
INFO 2011-10-18 06:51:52,420 [main] org.mule.module.management.agent.JmxAgent:
Attempting to register service with name: Mule.e148024f-f944-11e0-ae52-
6f64b9e3e0f2:type=Endpoint,service="HelloFlow",connector=connector.http.mule.default,nam
e="endpoint.http.localhost.8081.services.HelloService"
INFO 2011-10-18 06:51:52,420 [main] org.mule.module.management.agent.JmxAgent:
Registered Endpoint Service with name: Mule.e148024f-f944-11e0-ae52-
6f64b9e3e0f2:type=Endpoint,service="HelloFlow",connector=connector.http.mule.default,nam
e="endpoint.http.localhost.8081.services.HelloService"
INFO 2011-10-18 06:51:52,423 [main] org.mule.module.management.agent.JmxAgent:
Registered Connector Service with name Mule.e148024f-f944-11e0-ae52-
6f64b9e3e0f2:type=Connector,name="connector.http.mule.default.1"
INFO 2011-10-18 06:51:52,436 [main] org.mule.DefaultMuleContext:
*****
* Mule ESB and Integration Platform *
* Version: 3.2.0 Build: 22917 *
* MuleSoft, Inc. *
* For more information go to http://www.mulesoft.org *
*
* Server started: 10/18/11 6:51 AM *
* Server ID: e148024f-f944-11e0-ae52-6f64b9e3e0f2 *
* JDK: 1.6.0_24 (mixed mode) *
* OS encoding: UTF-8, Mule encoding: UTF-8 *
* OS: Mac OS X (10.6.6, x86_64) *
* Host: Bo5b.local (127.0.0.1) *
*
* Agents Running: *
* Rmi Registry: rmi://localhost:1099 *
* Jmx Notification Agent (Listener MBean registered) *
* JMX Log4J Agent *
* jmx-agent: service:jmx:rmi:///jndi/rmi://localhost:1099/server *
* MX4J Http adaptor: http://localhost:9999 *
* Default Jmx Support Agent *
*****
```

Note that:

- The HelloFlow with the HelloService was successfully started.  
The WSDL of the service can be found at <http://localhost:8081/services/HelloService?wsdl>  
Verify that there indeed is a WSDL at this URL in a browser.
- Last in the framed information about the Mule server, there is a listing of running agents

(“Agents Running”).

- Among the running agents, there is a JMX agent.  
In my case, the JMX agent can be found at:  
service:jmx:rmi:///jndi/rmi://localhost:1099/server
- Also among the running agents, there is an entry for the MX4J HTTP adaptor.  
This is the URL at which we can find the MX4J web console we will look at shortly.  
In my case, the URL is: <http://localhost:9999>
- There are other entries in the list of running agents, but in this example we will only use the JMX agent and the MX4J agent.

### ***Test the Running Example Program***

This part is common for both the Mule 2.x and Mule 3.x versions of the example program.

- Using soapUI, send at least one request to the running HelloService.  
The details on how to do this is left as an exercise to the reader.
- Do not stop the example program!

We have started a Mule instance that exposes a SOAP web service and it does look like there are some agents that have been started due to our attempt at enabling JMX configuration, but some hard proof would be nice.

## 7.5. Managing a Mule Instance Using JMX

With an instance, be it the Mule 2.x or Mule 3.x version, of this chapter's example program running, we are ready to take a look at the options available when managing the Mule instance. This section include parts specific to Mule 2.x and Mule 3.x.

### Run JConsole

JConsole is a program for JMX management and is part of all recent Java runtimes; it is available in the JavaSE 6 SDK used throughout this book.

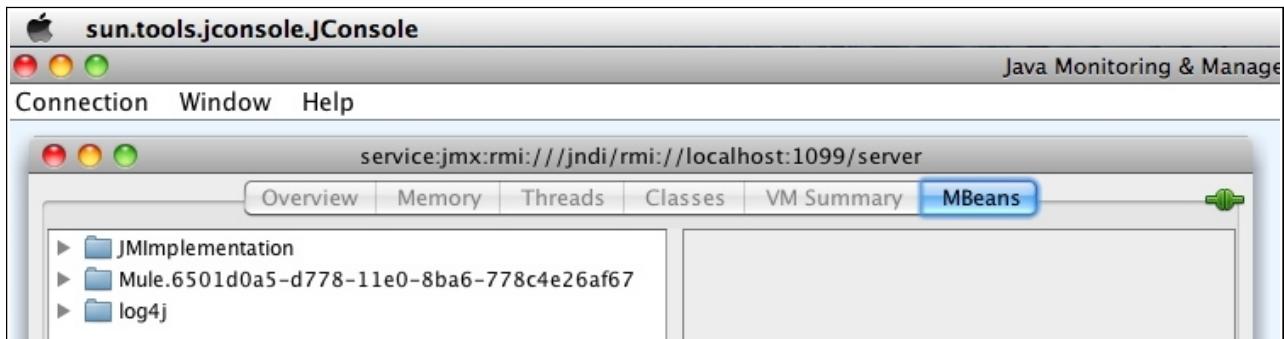
- In a terminal window or equivalent, issue the command `jconsole`.
- When JConsole starts, it will ask for the parameters of a new connection. Select the Remote Process radiobutton and input the JMX agent address “`service:jmx:rmi:///jndi/rmi://localhost:1099/server`”. The exact address may vary, so please check the console output generated when starting the Mule example program.



JConsole new connection dialog with the address to the Mule JMX agent entered.

- Click the Connect button.  
JConsole need some time to establish the connection, so be patient.

- When JConsole has established the connection, a window with the title same as the JMX agent address entered should be opened, like in the picture below.



JConsole after having established a connection to a Mule server.

Note that:

- There is a domain in the left panel which name starts with “Mule”.
- There is a plugged-in connector symbol in the upper right corner of the window. The connector symbol shows the connection-status – in this case JConsole is connected to the JMX service in the Mule instance.

We have now connected to a running Mule server with JConsole.

In the next section we will look at some of the monitoring options available for Mule 2.x and in a subsequent section, monitoring options available for Mule 3.x.

Note that most of the management options available for Mule 2.x are also available for Mule 3.x. Differences will be discussed in a subsequent section on Mule 3.x JMX management.

### **Generate Some Statistics**

To see some numbers in the management and monitoring data we are about to look at, use soapUI to send some requests to the HelloService. Using the Load Test feature if you want to generate a larger number of requests.

You are now ready to look at the JMX management options available in Mule, details on which can be found in the [Mule JMX Management](#) section in the reference part of this book.

## **Starting and Stopping a Mule Instance Using JMX**

To convince ourselves that we indeed are able to manage the Mule instance from JConsole, try the following:

- In a browser, open the URL <http://localhost:8081/services/HelloService>  
The result should be the XML of a SOAP fault message appearing in the browser window.

```
- <soap:Envelope>
  - <soap:Body>
    - <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>No such operation: </faultstring>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

The result of accessing the HelloService endpoint URL from a browser – a SOAP fault.

- In JConsole, in the org.mule.Connector node, go to the “connector.cxf.0.1” node and then to the Operations node and click the stopConnector button.  
A dialog should appear saying that the method was successfully invoked.
- Again, in a browser, access the URL <http://localhost:8081/services/HelloService>  
This time there should be no SOAP fault, just an empty window.
- Back to JConsole, in the org.mule.Connector node, go to the “connector.cxf.0.1” node and then to the Operations node and click the startConnector button.  
A dialog should appear saying that the method was successfully invoked.
- Finally, in a browser, open the URL <http://localhost:8081/services/HelloService>  
The result should again be the XML of a SOAP fault message.

We see that we were able to start and stop a connector in the running Mule instance.

## 7.6. MX4J and Mule

As mentioned earlier in this chapter, there is at least one alternative to using JConsole: MX4J. MX4J is a web-based JMX management console that is embedded in Mule. This is an alternative that can come in handy when it is not possible to use JConsole for one reason or another.

Recall the line in the log output, generated when starting the example program of this chapter, containing “MX4J Http adaptor”. The URL after this text, in my case <http://localhost:9999>, is the URL to the MX4J web console.

If we open this URL in a web browser, we will see the start page. On the Server View page there is a list of the different domains, similar to the left panel in JConsole's MBeans view.

Mule ESB and Integration Platform				Mule					
JMX Management Console									
Server view	MBean view	Timers	Monitors	Relations	MLet	Statistics	About	Filter: ...	Query
<b>MBean By Domain:</b>									
<b>Domain: com.sun.management</b>									
com.sun.management:type=HotSpotDiagnostic									Information on the management interface of the MBean
<b>Domain: java.lang</b>									
java.lang:type=ClassLoading									Information on the management interface of the MBean
java.lang:type=Compilation									Information on the management interface of the MBean
java.lang:type=GarbageCollector,name=ConcurrentMarkSweep									Information on the management interface of the MBean
java.lang:type=GarbageCollector,name=ParNew									Information on the management interface of the MBean
java.lang:type=Memory									Information on the management interface of the MBean
java.lang:type=MemoryManager,name=CodeCacheManager									Information on the management interface of the MBean
java.lang:type=MemoryPool,name=CMS Old Gen									Information on the management interface of the MBean
java.lang:type=MemoryPool,name=CMS Perm Gen									Information on the management interface of the MBean
java.lang:type=MemoryPool,name=G1 Eden Space									Information on the management interface of the MBean
java.lang:type=MemoryPool,name=Par Survivor Space									Information on the management interface of the MBean
java.lang:type=OperatingSystem									Information on the management interface of the MBean
java.lang:type=Runtime									Information on the management interface of the MBean
java.lang:type=Thread									Information on the management interface of the MBean
<b>Domain: java.util.logging</b>									
java.util.logging:type=Logging									Information on the management interface of the MBean
<b>Domain: JMImplementation</b>									
JMImplementation:type=MBeanServerDelegate									Represents the MBean server from the management point of view.
<b>Domain: [log4j]</b>									
log4j:type=appender-console									This MBean acts as a management facade for log4j appenders.
log4j:type=appender-console,layout=org.apache.log4j.PatternLayout									This MBean acts as a management facade for log4j layouts.
log4j:type=logger									This MBean acts as a management facade for a org.apache.log4j.Logger instance.
log4j:type=Hierarchy									This MBean acts as a management facade for org.apache.log4j.Hierarchy.
<b>Domain: Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2</b>									
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=Configuration									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=ManagementAgent									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=Connector, name=connector.http.0.1									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=EndpointService									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=FlowConstructService									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=ModService									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=NotificationBroadcaster									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=NotificationListener									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=Statistics, Flow=HelloFlow									Information on the management interface of the MBean
Mule_efaea4c5-e4d2-11e0-8696-57fc1fa2f0a2:type=Statistics, name=AllStatistics									Information on the management interface of the MBean

MX4J Mule JMX web-based management console, Server View.

We can see that there is a domain which name starts with “Mule”, similar to what we saw in JConsole. If we click an MBean in this domain, for example the HelloFlow MBean under the Flow node, we are taken to the MBean View:

Mule ESB and Integration Platform						mule
JMX Management Console						
<a href="#">Server view</a> <a href="#">MBean view</a> <a href="#">Timers</a> <a href="#">Monitors</a> <a href="#">Relations</a> <a href="#">Metrics</a> <a href="#">Statistics</a> <a href="#">About</a>						
<b>MBean: Mule.efaee4c5-e4d2-11e0-8696-57fc1fc2fa2e2:type=Endpoint,service="HelloFlow",connector=connector.http.0,name="endpoint.http.localhost.8081.services.HelloService"</b>						
<b>Attributes</b>						
Name	Description	Type		Value	New Value	
Address	Attribute exposed for management	java.lang.String		http://localhost:8081/services>HelloService	Read-only attribute	
ComponentName	Attribute exposed for management	java.lang.String		Helloflow	Read-only attribute	
Connected	Attribute exposed for management	boolean		true	Read-only attribute	
Inbound	Attribute exposed for management	boolean		true	Read-only attribute	
MessageExchangePattern	Attribute exposed for management	org.mule.MessageExchangePattern		REQUEST_RESPONSE	Read-only attribute	
Name	Attribute exposed for management	java.lang.String		endpoint.http.localhost.8081.services>HelloService	Read-only attribute	
Outbound	Attribute exposed for management	boolean		false	Read-only attribute	
						Set all
<b>Operations</b>						
Name	Return type	Description				
connect	void	Operation exposed for management				Invoke
disconnect	void	Operation exposed for management				Invoke
<b>Constructors</b>						

MX4J Mule JMX web-based management console, MBean View.

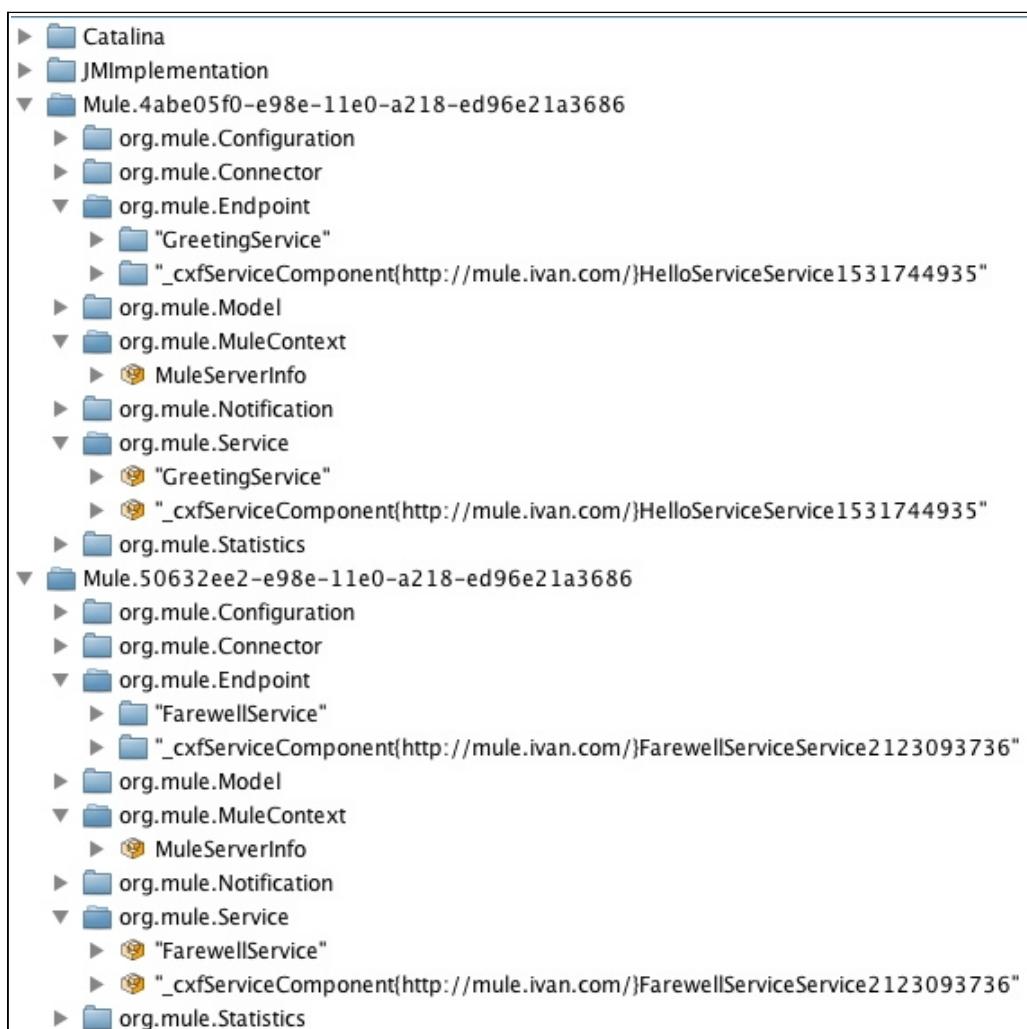
In the MBean view, we can view attributes, change writeable attributes and invoke any operations available on the MBean.

These are the basic options available in the MX4J Mule JMX management console. Exploration of additional features are left as an exercise for the reader.

## 7.7. Monitoring Mule in Web Applications

It is possible to monitor Mule web applications running in a web or application container as we saw in the chapter on [Mule in Web Applications](#). To do this we use the <management:jmx-default-config> element in the same way as we have seen in this chapter.

If we connect JConsole to a Tomcat instance that contains two web applications that both contain Mule configuration files, we can see the following:



Examining a Tomcat server with two web applications that use Mule in JConsole.

Note that:

- Each web application has its own set of Mule-related MBeans.
- For each web application, there is a Mule context.

This concludes the chapter on Mule monitoring and management.

## 8. Mule Notifications

Mule supports an implementation of the [Observer design pattern](#), enabling notifications to be sent to listeners when certain events occur. Examples of events are an endpoint receiving or sending a message, changes in the state of a Mule model, an exception was thrown etc. It is also possible to implement custom listeners which receives custom events.

Notifications can be configured either in the Mule configuration file or programmatically. The notification configuration can be static, meaning that it does not change after the Mule context has been started, or even dynamic, allowing for listeners to be registered after the Mule context has been started.

In this chapter we will create a Mule application that contains a notification listener that receives notifications when a web service endpoint receives requests and sends responses.

We will also look at the different types of notification listeners available in Mule.

### 8.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “MuleNotifications”. The Mule 3 hot deployment can be switched off from the start, as this feature will not be used.

### 8.2. Create the Service Implementation Class

Since we are going to expose a web service endpoint, we need a service implementation class. The implementation is similar to what we saw in [chapter one](#), so no further comments will be made.

- In the package *com.ivan.mule.service*, create the *HelloService* class implemented as follows:

```
package com.ivan.mule.services;

import java.util.Date;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

/**
 * SOAP web service endpoint implementation class that implements
 * a service that extends greetings.
 */
@WebService
public class HelloService
{
    @WebResult(name="greeting")
    public String greet(@WebParam(name="name") String inName)
    {
        return "Hello " + inName + ", the time is now " + new Date();
    }
}
```

## 8.3. Create the Notification Listeners

Notification listeners are similar to observers in the Observer design pattern. Notification listeners are, however, more strongly coupled to the kind of event that they listen to.

The notification listeners of this example are implemented in three different classes; one abstract base class implementing common behaviour, the Mule 2.x and the Mule 3.x notification listener. Both the Mule 2.x and Mule 3.x notification listeners expect to receive the same kind of notifications; notifications created when an endpoint sends or receives a message.

Note that the Mule 2.x notification listener will not compile properly when the Mule 3.x libraries are on the classpath and vice versa. Please ignore these errors when developing the example.

### Create the Common Notification Listener Base Class

The common notification listener base class implements the following, common to both the notification listeners in this example:

- Logging of notification listener instance creation.
- Storing an unique instance identifier; an integer number.  
This identifier is used when logging to the console, in order for us to be able to tell the console output from the listeners apart.
- Logging of notification properties.
- Logging of message payload.

We are now ready to create the notification listener base class.

- In the package *com.ivan.mule.notificationlisteners*, create the *MyAbstractMsgNotificationListener* class implemented like this:

```
package com.ivan.mule.notificationlisteners;

import org.mule.DefaultMuleMessage;
import org.mule.api.MuleMessage;
import org.mule.api.context.notification.ServerNotification;
import org.mule.api.endpoint.ImmutableEndpoint;
import org.mule.context.notification.EndpointMessageNotification;

/**
 * Abstract message notification listener implementing common
 * properties of such listeners.
 *
 * @author Ivan Krizsan
 */
public abstract class MyAbstractMsgNotificationListener
{
    protected static int sCurrentInstanceNumber = 1;
    protected int mInstanceNumber = sCurrentInstanceNumber++;

    /**
     * Default constructor.
     * Logs creation of instances of the listener.
     */
    public MyAbstractMsgNotificationListener()
    {
        System.out.println("\n*** MyMsgNotificationListener " + mInstanceNumber
                           + " instance created.");
    }

    /**
     * Logs the Mule message payload from the supplied message. If the
     * supplied object is not a Mule message, a message stating this
     * is logged.
     */
}
```

```

 * @param inNotificationSource Notification source, which is the
 * message received/sent by the endpoint.
 */
protected void logMessagePayload(final Object inNotificationSource)
{
    if (inNotificationSource instanceof DefaultMuleMessage)
    {
        MuleMessage theMessage = (DefaultMuleMessage)inNotificationSource;
        try
        {
            System.out.println("    Message payload object: "
                + theMessage.getPayload());
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
    } else
    {
        System.out.println("    Notification source is not a Mule message!");
        System.out.println("    Notification source type: "
            + inNotificationSource.getClass());
    }
}

/**
 * Logs an action description based on the supplied action code.
 * The action name is available in the notification, the following
 * code shows how to use the action code.
 *
 * @param inActionCode Action code used to find action
 * description.
 */
protected void logActionDescription(final int inActionCode)
{
    String theActionName;

    switch (inActionCode)
    {
        case EndpointMessageNotification.MESSAGE_RECEIVED:
            theActionName = "Message received";
            break;
        case EndpointMessageNotification.MESSAGE_DISPATCHED:
            theActionName = "Message dispatched";
            break;
        case EndpointMessageNotification.MESSAGE_SENT:
            theActionName = "Message sent";
            break;
        /*
         * EndpointMessageNotification.MESSAGE_RESPONSE constant is
         * only defined in Mule 3.x.
         */
        case 805:
            theActionName = "Responded to message";
            break;
        default:
            theActionName = "Unknown action: " + inActionCode;
            break;
    }
    System.out.println("    Action description: " + theActionName);
}

/**
 * Retrieves the immutable endpoint from the supplied endpoint
 * message notification. This is the only difference between Mule
 * 2.x and Mule 3.x, so it has been extracted to this method.
 *
 * @param inNotification Notification from which to retrieve
 * immutable endpoint.
 * @return Immutable endpoint from notification.
 */
abstract protected ImmutableEndpoint retrieveNotificationMutableEndpoint(
    final ServerNotification inNotification);

/**
 * Processes notifications.
 *
 * @param inNotification Notification to process.

```

```

/*
protected void myOnNotification(final ServerNotification inNotification)
{
    System.out.println("\n*** MyMsgNotificationListener " + mInstanceNumber
        + " received a notification:");

    /*
     * Extract properties from the notification and the associated message.
     */
    String theActionName = inNotification.getActionName();
    int theActionCode = inNotification.getAction();
    String theNotificationType = inNotification.getClass().getName();
    Object theNotificationSource = inNotification.getSource();
    ImmutableEndpoint theEndpoint =
        retrieveNotificationMutableEndpoint(inNotification);
    String theEndpointConnector = theEndpoint.getConnector().getName();
    String theResourceIdentifier = inNotification.getResourceIdentifier();

    System.out.println("    Notification type: " + theNotificationType);
    System.out.println("    Action name: " + theActionName);
    System.out.println("    Action code: " + theActionCode);
    System.out.println("    Notification source type: "
        + theNotificationSource.getClass());
    System.out.println("    Endpoint connector: " + theEndpointConnector);
    System.out.println("    Resource identifier: " + theResourceIdentifier);

    logActionDescription(theActionCode);
    logMessagePayload(theNotificationSource);
}
}

```

Note that:

- The class does not implement any particular interface or inherit from any particular class. As we will see later, Mule 2.x and Mule 3.x notification listeners differ in that Mule 3.x notification listener interfaces uses generics. Thus it is not possible to have the superclass implement one single interface covering both cases.
- The *logMessagePayload* method takes a Java object as parameter and, if the object is a Mule message, attempts to retrieve and log the message payload.
- The *logActionDescription* method finds a string describing the action of the endpoint from an integer containing an action code.  
The notification object already contains a method, *getActionName*, from which the name of the action may be obtained. The *logActionDescription* method is here to show how to work with action codes from notifications using constants.
- In the *logActionDescription* method, there is a case for the integer 805. In Mule 3.x there is a constant, *EndpointMessageNotification.MESSAGE\_RESPONSE*, for this number. This constant is not available in Mule 2.x.
- There is a method named *retrieveNotificationMutableEndpoint* that returns an *ImmutableEndpoint* object.  
The names of the methods from which to retrieve an *ImmutableEndpoint* from notification objects are different in Mule 2.x and Mule 3.x, as we will see later. This call was refactored into a separate method that subclasses must implement, in order to avoid unnecessary code duplication.
- The method *myOnNotification* retrieves and logs a number of properties from the notification object.  
Again, the fact that the Mule 3.x notification listener interface uses generics prevent us from implementing a single *onNotification* method that can be used by both child classes.

## Create the Mule 2.x Notification Listener

With the common notification listener superclass in place, implementing the version-specific child classes becomes almost trivial. First up is the Mule 2.x notification listener:

- In the package `com.ivan.mule.notificationlisteners`, create the `MyMsgNotificationListener2` class implemented in the following manner:

```
package com.ivan.mule.notificationlisteners;

import org.mule.api.context.notification.EndpointMessageNotificationListener;
import org.mule.api.context.notification.ServerNotification;
import org.mule.api.endpoint.ImmutableEndpoint;
import org.mule.context.notification.EndpointMessageNotification;

/**
 * A notification listener that is notified when an endpoint sends or
 * receives a message. Mule 2.x version.
 * The onNotification method specified by the notification listener
 * interface this class implements is implemented in the superclass.
 *
 * @author Ivan A Krizsan
 */
public class MyMsgNotificationListener2 extends
    MyAbstractMsgNotificationListener implements
    EndpointMessageNotificationListener
{
    @Override
    protected ImmutableEndpoint retrieveNotificationMutableEndpoint(
        final ServerNotification inNotification)
    {
        /*
         * Mule 2.x way of retrieving the endpoint object from
         * the notification.
         */
        return ((EndpointMessageNotification)inNotification).getEndpoint();
    }

    /**
     * Processes notifications.
     * Need an onNotification method specific to the version of Mule used,
     * since Mule 3.x uses generics in the interface declaration while
     * Mule 2.x does not.
     *
     * @param inNotification Notification to process.
     */
    @Override
    public void onNotification(final ServerNotification inNotification)
    {
        myOnNotification(inNotification);
    }
}
```

Note that:

- The notification listener class extends our abstract notification listener class, implemented earlier, and implements the `EndpointMessageNotificationListener` interface. The interface is chosen based on the kind of notifications we want to receive. The different notification listener interfaces available in Mule are described in the [Notification reference section](#).
- The method `retrieveNotificationMutableEndpoint` uses the `getEndpoint` method on the notification object to retrieve an `ImmutableEndpoint` object. This is the Mule 2.x way of retrieving an endpoint object from the notification.
- The `onNotification` method takes a parameter of the type `ServerNotification`. `ServerNotification` is a common superclass for a number of notifications that represent something having happened in the Mule server. For details, please refer to the Mule API

Javadoc documentation.

- The *onNotification* method invokes the *myOnNotification* method in the superclass. With Mule 2.x, there is no need to cast the notification object, as the *onNotification* method receives a *ServerNotification* object as parameter.

## Create the Mule 3.x Notification Listener

The Mule 3.x notification listener class only differs slightly from the Mule 2.x notification listener.

- In the package *com.ivan.mule.notificationlisteners*, create the *MyMsgNotificationListener3* class. Implement it like this:

```
package com.ivan.mule.notificationlisteners;

import org.mule.api.context.notification.EndpointMessageNotificationListener;
import org.mule.api.context.notification.ServerNotification;
import org.mule.api.endpoint.ImmutableEndpoint;
import org.mule.context.notification.EndpointMessageNotification;

/**
 * A notification listener that is notified when an endpoint sends or
 * receives a message. Mule 3.x version.
 * The onNotification method specified by the notification listener
 * interface this class implements is implemented in the superclass.
 *
 * @author Ivan A Krizsan
 */
public class MyMsgNotificationListener3 extends
    MyAbstractMsgNotificationListener implements
    EndpointMessageNotificationListener<EndpointMessageNotification>
{
    @Override
    protected ImmutableEndpoint retrieveNotificationMutableEndpoint(
        final ServerNotification inNotification)
    {
        /*
         * Mule 3.x way of retrieving the endpoint object from
         * the notification.
         */
        return ((EndpointMessageNotification)inNotification).getImmutableEndpoint();
    }

    /**
     * Processes notifications.
     * Need an onNotification method specific to the version of Mule used,
     * since Mule 3.x uses generics in the interface declaration while
     * Mule 2.x does not.
     *
     * @param inNotification Notification to process.
     */
    @Override
    public void onNotification(final EndpointMessageNotification inNotification)
    {
        myOnNotification(inNotification);
    }
}
```

Note that:

- The notification listener class extends our abstract notification listener class, implemented earlier, and implements the *EndpointMessageNotificationListener<EndpointMessageNotification>* interface. The use of generics allows us create an endpoint notification listener that only receives a particular type, a subclass of *EndpointMessageNotification*, of notifications. Creating custom notification listeners is beyond the scope of this tutorial – please refer to the Mule documentation for details.

The interface is chosen based on the kind of notifications we want to receive. The different notification listener interfaces available in Mule are described in the [Notification reference section](#).

- The method *retrieveNotificationMutableEndpoint* uses the *getImmutableEndpoint* method on the notification object to retrieve an *ImmutableEndpoint* object. This is the Mule 3.x way of retrieving an endpoint object from the notification.
- The *onNotification* method takes a parameter of the type *EndpointMessageNotification*. *EndpointMessageNotification* is a specific type of notification sent in connection to an endpoint receiving or sending a message.
- The *onNotification* method invokes the *myOnNotification* method in the superclass. There is no need to cast the *EndpointMessageNotification* object, since it is also a *ServerNotification* object due to inheritance.

## 8.4. Create the Mule Configuration Files

This example uses one Mule configuration file per Mule version. The configuration of the notification listeners, which is what is of interest in this example, only differs slightly between the two Mule versions. Both versions of the configuration files share the following structure:

- Definition of notification listener beans.  
These are plain Spring beans, implemented by the classes we developed above.
- Specification of notifications to receive and notification listeners.
- Definition of a web service endpoint.  
This is the endpoint that, for the cause of the example, receives requests and sends replies.

### Create the Mule 2.x Configuration File

The Mule 2.x configuration file is to be located in the root of the source directory and have the name “mule-config2.xml”. It has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://www.mulesource.org/schema/mule/cxf/2.2"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"

      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/cxf/2.2
          http://www.mulesource.org/schema/mule/cxf/2.2/mule-cxf.xsd

          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--
        Notification listeners are Spring beans.
        The class implementing the bean implements an interface which
        type depends on the kind of notification we wish to receive.
    -->
    <spring:bean
        name="notificationListener1"
        class="com.ivan.mule.notificationlisteners.MyMsgNotificationListener2"/>
    <spring:bean
        name="notificationListener2"
        class="com.ivan.mule.notificationlisteners.MyMsgNotificationListener2"/>

    <!--
        In this global block, not particular to any model or service,
        we declare which kind of notifications we have listeners for
        and which the listeners are.
        The dynamic attribute enables, if set to true, registering for
        notifications after the Mule context has been started.
        Default value of the dynamic attribute is false.
    -->
    <notifications dynamic="true">
        <!--
            Enable notifications for the specified event type.
            The standard event types are:
            CONTEXT           - Mule context started, stopped etc.
            MODEL             - Model lifecycle state changed or
                                components in the model registered or
                                unregistered.
            SERVICE           - Service started, stopped, etc.
            SECURITY          - Request for authorization occurred.
            ENDPOINT-MESSAGE - Message received or sent by endpoint.
        -->
```

COMPONENT-MESSAGE	- Message processed by component.
MANAGEMENT	- State of Mule instance or its resources changed.
CONNECTION	- Connector connected to, released connection or failed connection attempt to resource.
REGISTRY	- Event occurred on the Mule registry.
CUSTOM	- Custom notification.
EXCEPTION	- An exception was thrown.
TRANSACTION	- Transaction begun, committed or rolled back.
ROUTING	- A routing event occurred.

There are additional attributes available on the <notification> element, allowing for fine-grained control when declaring custom notifications.

```

-->
<notification event="ENDPOINT-MESSAGE" />
<!--
    Disable notifications for the specified event type.
    Disabling of notifications has precedence over notification enabling.
    Note that it is possible to disable notifications of one particular event type for a certain interface. This means the event will be sent to listeners with other interfaces.
-->
<disable-notification interface="COMPONENT-MESSAGE" />
<!--
    If we add subscription="GreetingService" to the notification listener declaration below, the listener will only listen to notifications from the GreetingService.
    This will result in the listener receiving only notifications from the CXF connector and not the HTTP connector.
-->
<!--
    The first notification listener (notificationListener1) will receive events from all source, while the second (notificationListener2) only will receive events from the resource with the name "GreetingService".
-->
<notification-listener
    ref="notificationListener1" />
<notification-listener
    ref="notificationListener2"
    subscription="GreetingService" />
</notifications>

<!-- Model with an inbound webservice endpoint. -->
<model name="GreetingModel" >
    <service name="GreetingService" >
        <inbound>
            <cxft:inbound-endpoint
                address="http://localhost:8182/services/GreetingService" />
        </inbound>

        <component>
            <prototype-object class="com.ivan.mule.services.HelloService" />
        </component>

        <outbound>
            <pass-through-router>
                <stdio:outbound-endpoint system="OUT" />
            </pass-through-router>
        </outbound>
    </service>
</model>
</mule>
```

Note that:

- There are two Spring beans, “notificationListener1” and “notificationListener2”, defined using the same implementation class.  
We define two notification listeners in order to apply filtering to one of them and compare the notifications received by the listeners.
- There is a <notifications> element defined outside of any models or services.

This is where we define which kinds of notifications to listen for, which notifications not to listen for and the notification listeners of the application.

- The <notifications> element has an attribute named *dynamic*.

Notification listeners can be registered in the Mule configuration file, but may also be registered programmatically. As per default, it is not possible to register notification listeners after the Mule context has started. Setting the *dynamic* attribute on the <notifications> element to true allows for programmatic registration of notification listeners after the Mule context has been started.

In this particular example, we do not register any notification listeners programmatically. The dynamic attribute has only been included to have an excuse for describing this facility.

- There is a <notification> child element in the <notifications> element, which looks like this:

```
<notification event="ENDPOINT-MESSAGE" />
```

This element tells Mule that we wish to receive notifications on endpoint messages; when they are sent or received. The different kinds of events available in Mule are listed in the [Notification section of the reference](#).

- The next element is a <disable-notification> element, which looks like this:

```
<disable-notification interface="COMPONENT-MESSAGE" />
```

This element disables the sending of notifications for component messages. Note that we have to use the *interface* attribute to specify the kind of event and cannot use the *event* attribute, which also is available on the element. Again, the different kinds of events available in Mule are listed in the [Notification section of the reference](#).

Since there are no component message notification listeners declared in this example, this line is superfluous and is only here to show that it is possible to disable notifications as well.

- Next, there are two <notification-listener> elements.

This kind of element is used to register a notification listener. Using the *ref* attribute, we specify which Spring bean is to be the listener.

- The second <notification-listener> element has a *subscription* attribute.

If no *subscription* attribute is present, the notification listener will receive notifications of the type specified by the notification interface(s) the listener implement from all components. Using the *subscription* attribute, the listener will only receive notifications from the component which name is given as the value of the attribute.

We will see this behaviour when we run the example program in a while.

- The remaining part of the Mule 2.x configuration file contains a model which declares a SOAP web service endpoint. For details on this part, please refer to [chapter 1](#).

Note, however, the name of the inbound service in the model - “GreetingService”, which matches the name we used in the declaration of one of the notification listeners.

## Create the Mule 3.x Configuration File

The Mule 3.x configuration file is to be located at the same location as the Mule 2.x configuration file, that is in the root of the source directory. It is to be named “mule-config3.xml” and is, to a large extent, identical with the Mule 2.x configuration file. It has the following contents:

```
<?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:cxf="http://www.mulesoft.org/schema/mule/cxf"
      xmlns:stdio="http://www.mulesoft.org/schema/mule/stdio"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/cxf
          http://www.mulesoft.org/schema/cxf/current/mule-cxf.xsd

          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd

          http://www.mulesoft.org/schema/mule/stdio
          http://www.mulesoft.org/schema/mule/stdio/current/mule-stdio.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!--
        Notification listeners are Spring beans.
        The class implementing the bean implements an interface which
        type depends on the kind of notification we wish to receive.
    -->
    <spring:bean
        name="notificationListener1"
        class="com.ivan.mule.notificationlisteners.MyMsgNotificationListener3"/>
    <spring:bean
        name="notificationListener2"
        class="com.ivan.mule.notificationlisteners.MyMsgNotificationListener3"/>

    <!--
        In this global block, not particular to any model or service,
        we declare which kind of notifications we have listeners for
        and which the listeners are.
        The dynamic attribute enables, if set to true, registering for
        notifications after the Mule context has been started.
        Default value of the dynamic attribute is false.
    -->
    <notifications dynamic="false">
        <!--
            Enable notifications for the specified event type.
            The standard event types are:
            CONTEXT           - Mule context started, stopped etc.
            MODEL             - Model lifecycle state changed or
                                components in the model registered or
                                unregistered.
            SERVICE           - Service started, stopped, etc.
            SECURITY          - Request for authorization occurred.
            ENDPOINT-MESSAGE - Message received or sent by endpoint.
            COMPONENT-MESSAGE - Message processed by component.
            MANAGEMENT         - State of Mule instance or its resources changed.
            CONNECTION         - Connector connected to, released connection
                                or failed connection attempt to resource.
            REGISTRY           - Event occurred on the Mule registry.
            CUSTOM             - Custom notification.
            EXCEPTION          - An exception was thrown.
            TRANSACTION         - Transaction begun, committed or rolled back.
            ROUTING            - A routing event occurred.

            There are additional attributes available on the <notification>
            element, allowing for fine-grained control when declaring
            custom notifications.
        -->
        <notification event="ENDPOINT-MESSAGE"/>
        <!--
            Disable notifications for the specified event type.
            Disabling of notifications has precedence over notification
        -->
    
```

```

enabling.
Note that it is possible to disable notifications of one
particular event type for a certain interface. This means
the event will be sent to listeners with other interfaces.
-->
<disable-notification interface="COMPONENT-MESSAGE" />
<!--
    If we add subscription="GreetingService" to the notification
    listener declaration below, the listener will only listen
    to notifications from the GreetingService.
    This will result in the listener receiving only notifications
    from the CXF connector and not the HTTP connector.
-->
<!--
    The first notification listener (notificationListener1)
    will receive events from all source, while the second
    (notificationListener2) only will receive events from the
    resource with the name "GreetingFlow".
-->
<notification-listener
    ref="notificationListener1" />
<notification-listener
    ref="notificationListener2"
    subscription="GreetingFlow" />
</notifications>

<!-- Flow with an inbound webservice endpoint. -->
<flow name="GreetingFlow">
    <inbound-endpoint
        address="http://localhost:8182/services/GreetingService"
        exchange-pattern="request-response" />
    <cxf:jaxws-service serviceClass="com.ivan.mule.services.HelloService" />
    <component>
        <prototype-object class="com.ivan.mule.services.HelloService" />
    </component>
    <log-component/>
</flow>
</mule>

```

Note that:

- There are two Spring beans, “notificationListener1” and “notificationListener2”, defined using the same implementation class.  
We define two notification listeners in order to apply filtering to one of them and compare the notifications received by the listeners.
- There is a <notifications> element defined outside of any models, flows or services.  
This is where we define which kinds of notifications to listen for, which notifications not to listen for and the notification listeners of the application.
- The <notifications> element has an attribute named *dynamic*.  
Notification listeners can be registered in the Mule configuration file, but may also be registered programmatically. As per default, it is not possible to register notification listeners after the Mule context has started. Setting the *dynamic* attribute on the <notifications> element to true allows for programmatic registration of notification listeners after the Mule context has been started.  
In this particular example, we do not register any notification listeners programmatically. The dynamic attribute has only been included to have an excuse for describing this facility.
- There is a <notification> child element in the <notifications> element, which looks like this:  
`<notification event="ENDPOINT-MESSAGE" />`  
This element tells Mule that we wish to receive notifications on endpoint messages; when they are sent or received. The different kinds of events available in Mule are listed in the [Notification section of the reference](#).

- The next element is a <disable-notification> element, which looks like this:  
`<disable-notification interface="COMPONENT-MESSAGE" />`  
 This element disables the sending of notifications for component messages. Note that we have to use the *interface* attribute to specify the kind of event and cannot use the *event* attribute, which also is available on the element. Again, the different kinds of events available in Mule are listed in the [Notification section of the reference](#).  
 Since there are no component message notification listeners declared in this example, this line has no effect and is only here to show that it is possible to disable notifications as well.
- Next, there are two <notification-listener> elements.  
 Notification listener elements are used to register notification listeners. The *ref* attribute is used to specify which Spring bean is to be the listener.
- The second <notification-listener> element has a *subscription* attribute.  
 If no *subscription* attribute is present, the notification listener will receive notifications of the type specified by the notification interface(s) the listener implement from all components. Using the *subscription* attribute, the listener will only receive notifications from the component whose name is given as the value of the attribute.  
 We will see this behaviour when we run the example program shortly.
- The remaining part of the Mule 3.x configuration file contains a flow which declares a SOAP web service endpoint. For details on this part, please refer to [chapter 1](#).  
 Note, however, the name of the flow - “GreetingFlow”, which matches the name we used in the declaration of one of the notification listeners.

## 8.5. Run the Example Program

When running the example configurations, we will need [soapUI](#) to act as client to the web service we will expose. Before proceeding, make sure that you have soapUI installed.

With the example program in place and soapUI ready, we are now set to run the example program.

### ***Run the Mule 2.x Example Program***

If the project is not configured with the Mule 2.x distribution on the classpath when starting this section, please refer to [this section in appendix B](#) on how to configure which Mule distribution to use.

- In the Package or Project Explorer, right-click the “mule-config2.xml” file and select Run As -> Mule Server.
- The Mule 2.x server should start up and display something similar to this on the console (output omitted to conserve space):

```
...
*** MyMsgNotificationListener 1 instance created.
*** MyMsgNotificationListener 2 instance created.
...
INFO: Setting the server's publish address to be
http://localhost:8182/services/GreetingService
...
INFO 2012-01-19 16:13:40,681 [main] org.mule.transport.cxf.CxfMessageReceiver:
Connected: http://localhost:8182/services/GreetingService
...
INFO 2012-01-19 16:13:40,755 [main] org.mule.DefaultMuleContext:
*****
* Mule ESB and Integration Platform
* Version: 2.2.1 Build: 14422
*
```

Note that:

- Two notification listener instances were created.
- The web service URL is <http://localhost:8182/services/GreetingService>.  
Thus the WSDL can be found at <http://localhost:8182/services/GreetingService?wsdl>.

Nothing more happens, since the endpoint is neither receiving any requests, nor sending any responses.

## Create a soapUI Client

In order for notifications to be posted in our example program, we need to send requests to the web service endpoint exposed by the example program. We do this using soapUI:

- Start soapUI.
- Create a new project in soapUI.  
Name the project “NotificationClient” and use the following URL as the Initial WSDL/WADL: <http://localhost:8182/services/GreetingService?wsdl>
- Go to Eclipse and clear the console.  
We want to reduce the amount of output that we will analyze later.
- Expand the node named “greet” under “HelloServiceServiceSoapBinding”.
- Double-click the “Request 1” under the “greet” node.
- Enter a name in the <name> element of the request.  
The result should look like this:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ser="http://services.mule.ivan.com/">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:greet>
      <!--Optional:-->
      <name>Ivan</name>
    </ser:greet>
  </soapenv:Body>
</soapenv:Envelope>
```

- Send the request by clicking the little green arrow in the upper left corner of the request window.
- The resulting response should look like this:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:greetResponse xmlns:ns2="http://services.mule.ivan.com/">
      <greeting>Hello Ivan, the time is now Thu Jan 19 16:29:29 CET 2012</greeting>
    </ns2:greetResponse>
  </soap:Body>
</soap:Envelope>
```

## Examine the Mule 2.x Example Program Result

We are now ready to examine the console output from the Mule 2.x version of the example program.

- Go to the Eclipse console and look at the output, which should read something like this:

```
*** MyMsgNotificationListener 1 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: received  
Action code: 801  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.http.0  
Resource identifier:  
_cxfServiceComponent{http://services.mule.ivan.com/}HelloServiceService506786885  
Action description: Message received  
Message payload object:  
org.apache.commons.httpclient.ContentLengthInputStream@29565e9d  
  
*** MyMsgNotificationListener 2 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: received  
Action code: 801  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.cxf.0  
Resource identifier: GreetingService  
Action description: Message received  
Message payload object: Ivan  
  
*** MyMsgNotificationListener 1 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: received  
Action code: 801  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.cxf.0  
Resource identifier: GreetingService  
Action description: Message received  
Message payload object: Ivan  
INFO 2012-01-19 16:35:12,286 [connector.stdio.0.dispatcher.2]  
org.mule.transport.stdio.StdoutMessageDispatcher: Connected:  
endpoint.outbound.stdio://system.out  
Hello Ivan, the time is now Thu Jan 19 16:35:12 CET 2012  
*** MyMsgNotificationListener 2 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: dispatched  
Action code: 802  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.stdio.0  
Resource identifier: GreetingService  
Action description: Message dispatched  
Message payload object: Hello Ivan, the time is now Thu Jan 19 16:35:12 CET 2012  
  
*** MyMsgNotificationListener 1 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: dispatched  
Action code: 802  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.stdio.0  
Resource identifier: GreetingService  
Action description: Message dispatched  
Message payload object: Hello Ivan, the time is now Thu Jan 19 16:35:12 CET 2012
```

Note that:

- There is a total of five notifications received by our notification listeners.
- Two of the notifications were received by notification listener 2 and three were received by notification listener 1.

The reason for this is that we used the *subscription* attribute on the <notification-listener>

element used to declare notification listener 2, limiting the notifications to only those from the component “GreetingService”.

- Examining the first notification listener output shown below, we can draw some conclusions:

```
...
*** MyMsgNotificationListener 1 received a notification:
Notification type: org.mule.context.notification.EndpointMessageNotification
Action name: received
Action code: 801
Notification source type: class org.mule.DefaultMuleMessage
Endpoint connector: connector.http.0
Resource identifier:
_cxfServiceComponent{http://services.mule.ivan.com/}HelloServiceService506786885
Action description: Message received
Message payload object:
org.apache.commons.httpclient.ContentLengthInputStream@29565e9d
...
...
```

- The endpoint received a request (see Action name and Action code).
- The source of the notification was a Mule message (see Notification source type).
- The connector that received the message was a HTTP connector (see Endpoint connector).
- The origin of the notification is a CXF component with a long name (see Resource identifier).
- The payload of the received Mule message is a stream (see Message payload object).
- There is no corresponding notification for notification listener 2.
- Examining the second and third notification listener outputs we can draw the following conclusions:
  - The endpoint received a request (see Action name and Action code).  
Actually, this is the same request that caused the first notification output, but received by another component.
  - The source of the notifications was a Mule message (see Notification source type).
  - The connector that received the notifications was a CXF connector (see Endpoint connector).
  - The payload of the received Mule message is the string “Ivan” (see Message payload object).
  - The two notification messages are almost identical.  
Both the notification listeners have received a notification of one and the same event.
- After the second and third notification listener output, there is a log message:

```
INFO 2012-01-19 16:35:12,286 [connector.stdio.0.dispatcher.2]
org.mule.transport.stdio.StdioMessageDispatcher: Connected:
endpoint.outbound.stdio://system.out
Hello Ivan, the time is now Thu Jan 19 16:35:12 CET 2012
```

Due to our Mule configuration, the greeting string generated by our Hello service is output to the console.

- Finally, the fourth and fifth notification listener logs the dispatch of the response send to the client of the web service.

We can see that the message payload is the greeting string produced by our Hello service.

## ***Run the Mule 3.x Example Program***

We will now run the Mule 3.x version of the example program:

- Configure the Eclipse project with the Mule 3.x distribution on the classpath. Please refer to [this section in appendix B](#) for detailed instructions.
- In the Package or Project Explorer, right-click the “mule-config3.xml” file and select Run As -> Mule Server.  
We can see that Mule 3.x starts up and two instances of our notification listener is created, as was the case with the Mule 2.x version.
- Clear the console in Eclipse.
- From soapUI, send a request to the web service.  
There is no need to re-create or even change the soapUI project – we can continue to use the very same project created [earlier](#).

In Eclipse, there should be some output in the console which we'll take a closer look at next.

## Examine the Mule 3.x Example Program Result

Running the Mule 3.x version of the example program produces a result that is very similar to that of the Mule 2.x version. In this section we'll mainly focus on the differences.

- Examine the console output in Eclipse, which should look something like this:

```
*** MyMsgNotificationListener 1 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: received  
Action code: 801  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.http.mule.default  
Resource identifier: GreetingFlow  
Action description: Message received  
Message payload object:  
org.apache.commons.httpclient.ContentLengthInputStream@26514577  
  
*** MyMsgNotificationListener 2 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: received  
Action code: 801  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.http.mule.default  
Resource identifier: GreetingFlow  
Action description: Message received  
Message payload object:  
org.apache.commons.httpclient.ContentLengthInputStream@26514577  
INFO 2012-01-20 06:48:43,469 [connector.http.mule.default.receiver.02]  
org.mule.component.simple.LogComponent:  
*****  
* Message received in service: GreetingFlow. Content is: 'Hello Ivan, the *  
* time is now Fri Jan 20 06:48:43 CET 2012'  
*****  
  
*** MyMsgNotificationListener 1 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: response  
Action code: 805  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.http.mule.default  
Resource identifier: GreetingFlow  
Action description: Responded to message  
Message payload object: org.mule.transport.http.HttpResponse@3b5789a5  
  
*** MyMsgNotificationListener 2 received a notification:  
Notification type: org.mule.context.notification.EndpointMessageNotification  
Action name: response  
Action code: 805  
Notification source type: class org.mule.DefaultMuleMessage  
Endpoint connector: connector.http.mule.default  
Resource identifier: GreetingFlow  
Action description: Responded to message  
Message payload object: org.mule.transport.http.HttpResponse@3b5789a5
```

Note that:

- There are only four notification listener output sections.  
This is due to the Mule 3.x flow differing slightly from the Mule 2.x module.  
Mule 2.x uses a construct where messages first arrives at a HTTP endpoint and then are forwarded to a CXF endpoint. Mule 3.x, on the other hand, uses only a HTTP endpoint to receive messages, which then are forwarded to a CXF service component. The CXF component is not an endpoint and will thus not give rise to notifications in our example program.
- The resource identifier is always “GreetingFlow”.  
Even if we give the endpoint in the flow a name, the value of the resource identifier will still

be the name of the flow.

- We cannot see the actual payload of the Mule messages in the output from the notification listeners.

The endpoint in the Mule 3.x version of the example program deals with the Mule message very early/late in the processing chain; that is, immediately when receiving the request or immediately prior to sending the response. At these stages, the payload is accessed either through a stream or in a HTTP response object.

This concludes this chapter, in which we took a look at Mule notifications. Additional information about Mule notification can be found in the [Notifications](#) section in part two of this book.

## 8.6. Additional Exercises

For the curious, here are some suggestions for additional exercises for which this chapter's example program can act as a starting point.

- Remove the subscription attribute found in one of the <notification-listener> elements in the Mule configuration files.  
Re-run the example program and examine the output.
- Create another notification listener that listens to COMPONENT-MESSAGE notifications.  
Don't forget to remove the <disable-notification> element from the Mule configuration files before running the example program, otherwise your new notification listener will not receive any notifications!
- Implement the firing of a custom notification in the Hello service and a custom notification listener that listens for the custom notification.

## **9. Exception Handling in Mule**

In this chapter we will see how Mule behaves when something goes wrong and an exception is thrown. This will be but an introduction to the following features related to error handling:

- Exception-strategies.

We'll try out both default and custom exception strategies.

Exception strategies can be defined to be model-global as well as service-local. Both options will be examined.

- Exception-based routing.

Messages can be routed depending on whether attempting to contact a service results in an exception or not.

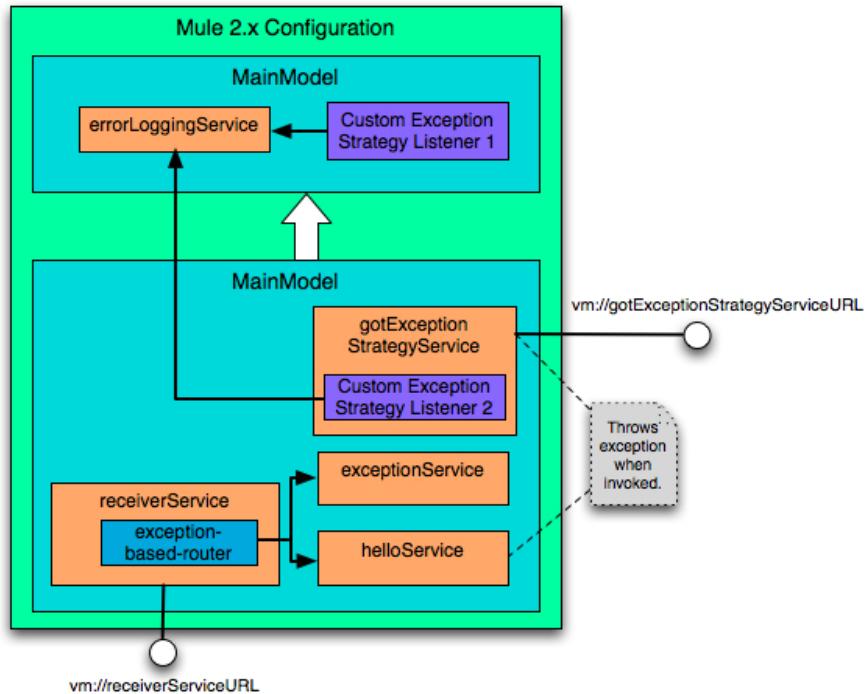
- Custom exception listeners.

Custom exception listeners were part of the example in [chapter three](#) and are used in this chapter as well.

The configurations for Mule 2.x and Mule 3.x will differ quite significantly. It would have been possible to use models and services in Mule 3.x too, but I wanted to show how to achieve exception handling with Mule 3.x-specific features.

## 9.1. Mule 2.x Configuration Structure

The structure of this chapter's Mule 2.x example will look like this:



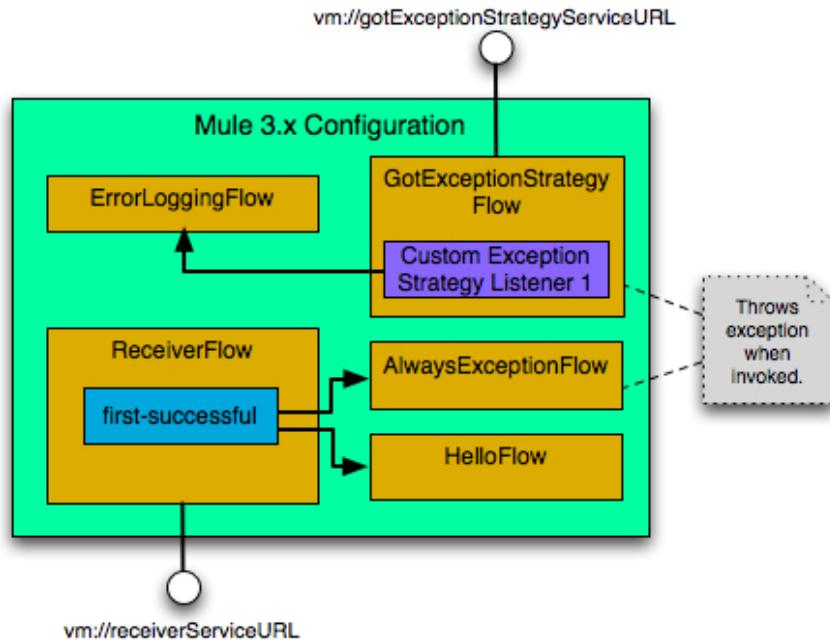
Structure of the Mule 2.x configuration of this chapter's example.

Note the following about the above figure:

- There are two models named “`MainModel`”.  
The reason for this is that the model-global exception strategy as well as the error logging service are defined in a parent-model which is then inherited by a child model.  
Recall that, as shown in [chapter three](#), in order to support model-inheritance, the parent and child models must have the same name.
- The model exposes two endpoints that we'll send messages to.
- The service “`receiverService`” passes incoming messages on to two other services; “`exceptionService`” and “`helloService`”.  
The “`receiverService`” does not have an exception strategy of its own and no exceptions will occur in this service, rather in one of the services it invokes.
- The service “`gotExceptionStrategyService`” will have a service-local exception strategy and an exception will occur in this very service each time it is invoked.

## 9.2. Mule 3.x Configuration Structure

The structure of the Mule 3.x configuration in this chapter will look like this:



Structure of the Mule 3.x configuration of this chapter's example.

Note the following about the Mule 3.x configuration structure:

- There are no parent-child relations.  
It is not possible to inherit properties between flows.  
We will see that one or more services can inherit one and the same exception strategy defined in a common, abstract, parent simple-service in the chapter [Mule Configuration Patterns](#).
- There are no global exception strategy listeners.  
Both exception strategy listeners used are flow-local and will only receive notifications when an exception occurred in the same flow as in which the listener is defined.
- The configuration exposes two endpoints that we'll send messages to.
- The flow "ReceiverFlow" passes incoming messages on to two other flows; "AlwaysExceptionFlow" and "HelloFlow".  
The flow "ReceiverFlow" does not have an exception strategy of its own and no exceptions will occur in this flow, rather in one of the flows it invokes.
- The flow "GotExceptionStrategyFlow" has an exception strategy and an exception will occur in this flow each time it is invoked.

## 9.3. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it "MuleExceptionHandling". The Mule 3 hot deployment can be switched off from the start, as this feature will not be used.

## 9.4. Create the Service Implementation Classes

The example makes use of three services, which implementations all are identical regardless of whether used with Mule 2.x or Mule 3.x.

- Exception Service
- Hello Service
- Logging Service

### Create the Exception Service Implementation Class

The Exception service always throws an exception when receiving a message, in order for us to obtain exceptions to handle in this example.

- In the package `com.ivan.services`, implement the `ExceptionService` class as this:

```
package com.ivan.services;

import org.mule.api.DefaultMuleException;
import org.mule.api.MuleEventContext;
import org.mule.api.lifecycle.Callable;

/**
 * A service that always throws an exception when being invoked.
 */
public class ExceptionService implements Callable
{
    private static int sExceptionId = 1;

    @Override
    public Object onCall(MuleEventContext inEventContext) throws Exception
    {
        int theExceptionId = sExceptionId++;
        System.out.println("**** In ExceptionService.onCall(): " + theExceptionId);
        Exception theNestedException = new Exception(
            "I am a nested exception with id " + theExceptionId);
        throw new DefaultMuleException(
            "I am an outer exception with id " + theExceptionId, theNestedException);
    }
}
```

Note that:

- The service class implements the `Callable` interface.  
This interface allows a service to receive Mule events, but also introduces a dependency on Mule. Compare this to the service implementation, a POJO, that we used in [chapter three](#). See the section [Implementing the Callable Interface](#) below for more information.
- An identifying number is appended to the message of exceptions.  
This number is also logged to the console when the service is invoked.  
This will enable us to see which service invocations cause an error to be reported.
- The `DefaultMuleException` thrown by the `onCall` method wraps an exception of the type `Exception`.
- If we only wanted a component that throws an exception every time it is invoked, then the following configuration line would be enough:  
`<test:component throwException="true"/>`  
The above class is implemented to have nested exceptions and to allow for debugging.

## Create the Hello Service Implementation Class

The Hello service replies incoming messages with a greeting. This service always succeeds in processing incoming requests.

- In the `com.ivan.services` package, implement the `HelloService` class.

```
package com.ivan.services;

import java.util.Date;
import org.mule.api.MuleEventContext;
import org.mule.api.lifecycle.Callable;

/**
 * The old faithful Hello service.
 *
 * @author Ivan Krizsan
 */
public class HelloService implements Callable
{
    @Override
    public Object onCall(final MuleEventContext inEventContext) throws Exception
    {
        String theMessage =
            "Hello. The time is now: " + (new Date());
        System.out.println("**** In HelloService.onCall(): " + theMessage);
        return theMessage;
    }
}
```

Note that:

- As with the previous service implementation class, the service class implements the `Callable` interface.  
See the section [Implementing the Callable Interface](#) below for more information.
- A plain string object is returned from the `onCall` method.  
The string will thus become the payload of a Mule message that in turn becomes the result of the service invocation.

## Create the Logging Service Implementation Class

The Logging service logs messages received. This service also contains static methods to log messages. These methods are used by other parts of the example program.

- In the same package as the other services, `com.ivan.services`, implement the `LoggingService` class:

```
package com.ivan.services;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import org.mule.api.ExceptionPayload;
import org.mule.api.MuleEventContext;
import org.mule.api.MuleMessage;
import org.mule.api.lifecycle.Callable;
import org.mule.api.transport.PropertyScope;

/**
 * Service that logs message payload and properties.
 *
 * @author Ivan Krizsan
 */
public class LoggingService implements Callable
{

    /* (non-Javadoc)
     * @see org.mule.api.lifecycle.Callable#onCall(org.mule.api.MuleEventContext)
     */
    @Override
    public Object onCall(MuleEventContext inEventContext) throws Exception
    {
        MuleMessage theReceivedMsg = inEventContext.getMessage();

        System.out.println("***** Logging service received a message:");
        logMessage(theReceivedMsg);

        return theReceivedMsg;
    }

    /**
     * Logs information, including payload and properties in the
     * INBOUND and OUTBOUND scopes, of the supplied Mule message.
     * Also logs any exception payload associated with the supplied
     * Mule message.
     *
     * @param inMuleMessage Message to log information about.
     */
    public static void logMessage(final MuleMessage inMuleMessage)
    {
        Map<String, Object> theReceivedMsgProperties;
        long theCurrentTime = System.currentTimeMillis();

        System.out.println("    Current time: " + theCurrentTime);
        try
        {
            System.out.println("    Message payload: " +
                inMuleMessage.getPayloadAsString());
        } catch (final Exception theException)
        {
            // Ignore exceptions
        }

        /* Log message properties in the inbound and outbound scopes. */
        System.out.println("    Message properties: ");
        theReceivedMsgProperties =
            LoggingService.retrieveMessageProperties(inMuleMessage,
                PropertyScope.INBOUND);
        System.out.println("    INBOUND: " + theReceivedMsgProperties);
        theReceivedMsgProperties =
            LoggingService.retrieveMessageProperties(inMuleMessage,
                PropertyScope.OUTBOUND);
        System.out.println("    OUTBOUND: " + theReceivedMsgProperties);
    }
}
```

```

ExceptionPayload theExceptionPayload =
    inMuleMessage.getExceptionPayload();

System.out.println("    Exception payload: " + theExceptionPayload);
/*
 * Print the exception stack-trace to the console using
 * System.out to avoid interleaving of output from System.out and
 * System.err, to which the parameter-less printStackTrace()
 * sends output to.
 */
if (theExceptionPayload != null)
{
    System.out.println("*** EXCEPTION STACKTRACE START:");
    theExceptionPayload.getException().printStackTrace(System.out);
    System.out.println("*** EXCEPTION STACKTRACE END.");
}
else
{
    System.out.println("    NO EXCEPTION PAYLOAD AVAILABLE");
}

/**
 * Retrieves message properties in supplied scope from supplied
 * Mule message.
 *
 * @param inMuleMessage Mule message which properties to retrieve.
 * @param inPropertyScope Scope which properties to retrieve.
 * @return Map containing all message properties.
 */
public static Map<String, Object> retrieveMessageProperties(
    final MuleMessage inMuleMessage, final PropertyScope inPropertyScope)
{
    Map<String, Object> theReceivedMsgProperties =
        new HashMap<String, Object>();
    Set<String> thePropertyName = inMuleMessage.getPropertyNames(
        inPropertyScope);

    for (String thePropertyName : thePropertyName)
    {
        Object thePropertyValue = inMuleMessage.getProperty(thePropertyName,
            inPropertyScope);
        theReceivedMsgProperties.put(thePropertyName, thePropertyValue);
    }
    return theReceivedMsgProperties;
}
}

```

Note that:

- As with the previous service implementation classes, the service class implements the *Callable* interface.  
See the section [Implementing the Callable Interface](#) below for more information.
- The *logMessage* method logs the following data about a message:
  - The string-representation of the message's payload.
  - Properties in the message's inbound and outbound scopes.
  - The exception payload of the message, or null if the message has no exception payload.
  - If the message has an exception payload, log the stack trace for the exception of the exception payload.
- In the *retrieveMessageProperties* method, note how message properties are retrieved supplying a scope.  
This is to assure identical behaviour of the method regardless of whether running on Mule 2.x or Mule 3.x. Methods performing property retrieval without taking a property scope as parameter have been deprecated in Mule 3.x.

## 9.5. The Callable Interface

All three service implementation classes implemented in the previous section implement the `org.mule.api.lifecycle.Callable` interface. While implementing this interface introduces a dependency on Mule, it can be useful for the following reasons:

- Send events synchronously and asynchronously from the service.  
The `MuleEventContext` interface contains `sendEvent` and `sendEventAsync` methods.
- Request events synchronously.  
Using the `requestEvent` method in the `MuleEventContext` interface.
- Stop further processing of the message.  
Using `MuleEventContext.setStopFurtherProcessing`.
- Manipulate the current transaction, if any.  
Use the `getTransaction` method to retrieve the transaction for the current event.  
Use the `markTransactionForRollback` to mark the current transaction, if any, for rollback.  
Both these methods can be found in the `MuleEventContext` interface.

The `onCall` method of a service implementing the `Callable` interface is to return an object. Such an object can be one of the following types:

- An object implementing `MuleMessage`.  
The returned object will be the result of the service invocation.
- An instance of `VoidResult`.  
The original message, as received by the service, will be the result of the service invocation.
- A non-null reference to any other object.  
The object will become the payload of a Mule message, which will be the result of the service invocation.

## 9.6. Create the Starter Classes

In this chapter's example we will use starter classes, one for Mule 2.x and another for Mule 3.x, as to be able to choose the endpoint which to invoke, construct messages with the desired payload and message properties and, finally, to examine the results of the service invocations.

Details on the implementation of these starter classes will be discussed in the subsequent chapter on [Mule Programmatic Use](#).

### Create the Mule 2.x Starter Class

The Mule 2.x starter class is, not entirely surprising, to be used with the Mule 2.x runtime.

- In the package *com.ivan.starter*, implement the class *Mule2ExceptionHandlingExampleStarter*:

```
package com.ivan.starter;

import org.mule.DefaultMuleMessage;
import org.mule.api.MuleContext;
import org.mule.api.MuleException;
import org.mule.api.MuleMessage;
import org.mule.config.spring.SpringXmlConfigurationBuilder;
import org.mule.context.DefaultMuleContextFactory;
import org.mule.module.client.MuleClient;
import com.ivan.services.LoggingService;

/**
 * Starter program for the exception handling example.
 * Starts Mule, sends a message to the Mule configuration and finally
 * examines the result.
 * Version for Mule 2.x.
 *
 * @author Ivan A Krizsan
 */
public class Mule2ExceptionHandlingExampleStarter
{
    private final static String MULE_CONFIG_FILE =
        "com/ivan/muleconfig/mule2-config.xml";
    private final static String MULE_SERVICE1_URL = "vm://receiverServiceURL";
    private final static String MULE_SERVICE2_URL =
        "vm://gotExceptionStrategyServiceURL";

    public static void main(String[] args) throws Exception
    {
        (new Mule2ExceptionHandlingExampleStarter()).doExample();
    }

    private void doExample()
    {
        MuleContext theContext = null;
        try
        {
            MuleClient theMuleClient;
            /* Create a new message, setting its payload. */
            MuleMessage theMuleMessage = new DefaultMuleMessage("I am a Mule message!");

            /* Start Mule context with the first configuration file. */
            theContext = startMule(MULE_CONFIG_FILE);

            /* Create a Mule client and send the message to it. */
            theMuleClient = createMuleClient(theContext);
            theMuleMessage = sendMessageToMule(theMuleMessage, theMuleClient);

            System.out.println("**** Finished invoking Mule configuration!");
            LoggingService.logMessage(theMuleMessage);
        } catch (final Exception theException)
        {
            theException.printStackTrace();
        } finally
```

```

    {
        /*
         * Dispose of the Mule contexts. Disposing the context
         * automatically stops it and we do not have to bother
         * with the exception declared by the stop method.
         * Must dispose if we wish the application to terminate,
         * otherwise there will be unreleased resources.
        */
        if (theContext != null)
        {
            theContext.dispose();
        }
    }

private MuleContext startMule(final String inMuleConfigFile) throws MuleException
{
    String[] theConfigFiles = { inMuleConfigFile };

    DefaultMuleContextFactory theContextFactory =
        new DefaultMuleContextFactory();
    SpringXmlConfigurationBuilder theConfigBuilder =
        new SpringXmlConfigurationBuilder(theConfigFiles);
    MuleContext theMuleContext =
        theContextFactory.createMuleContext(theConfigBuilder);
    theMuleContext.start();

    return theMuleContext;
}

private MuleClient createMuleClient(final MuleContext inMuleContext)
throws MuleException
{
    MuleClient theMuleClient = new MuleClient(inMuleContext);
    return theMuleClient;
}

private MuleMessage sendMessageToMule(final MuleMessage inMessage,
final MuleClient inMuleClient) throws Exception
{
    MuleMessage theReceivedMsg;

    /* Modify the URL of the service which to send the message here. */
    theReceivedMsg = inMuleClient.send(MULE_SERVICE1_URL, inMessage);
    return theReceivedMsg;
}
}

```

Note that:

- The flow of the starter class is quite simple:
  - Start Mule with a configuration file.
  - Create a Mule message.
  - Send the Mule message to a service in the configuration.
  - Examine the result.
- There are two different Mule endpoint URLs defined as constants in the class. One is MULE\_SERVICE1, the other is MULE\_SERVICE2. The first endpoint does not directly cause an exception to be thrown, but invokes a service that causes an exception to be thrown. The second endpoint will cause an exception to be thrown in the service containing the endpoint. We will see what this means to the client invoking the different services.
- The URL of the service which to send the message to can be changed in the *sendMessageToMule* method.
- Again, further details on the starter classes are available in the [next chapter](#).

## Create the Mule 3.x Starter Class

The Mule 3.x starter class is to be used with the Mule 3.x runtime.

- In the package `com.ivan.starter`, implement the class `Mule3ExceptionHandlingExampleStarter`:

```
package com.ivan.starter;

import org.mule.DefaultMuleMessage;
import org.mule.api.MuleContext;
import org.mule.api.MuleException;
import org.mule.api.MuleMessage;
import org.mule.api.config.ConfigurationBuilder;
import org.mule.api.context.MuleContextHolder;
import org.mule.api.context.MuleContextFactory;
import org.mule.config.spring.SpringXmlConfigurationBuilder;
import org.mule.context.DefaultMuleContextBuilder;
import org.mule.context.DefaultMuleContextFactory;
import org.mule.module.client.MuleClient;
import com.ivan.services.LoggingService;

/**
 * Starter program for the exception handling example.
 * Starts Mule, sends a message to the Mule instance and finally examines the result.
 * Version for Mule 3.x.
 *
 * @author Ivan Krizsan
 */
public class Mule3ExceptionHandlingExampleStarter
{
    private final static String MULE_CONFIG_FILE =
        "com/ivan/muleconfig/mule3-config.xml";
    private final static String MULE_SERVICE1_URL = "vm://receiverServiceURL";
    private final static String MULE_SERVICE2_URL =
        "vm://gotExceptionStrategyServiceURL";

    public static void main(String[] args) throws Exception
    {
        (new Mule3ExceptionHandlingExampleStarter()).doExample();
    }

    private void doExample()
    {
        MuleContext theContext = null;
        try
        {
            MuleClient theMuleClient;
            /*
             * Start the first Mule context with the configuration file.
             * Need to do this, since the context will be used when
             * creating a new message.
             */
            theContext = startMule(MULE_CONFIG_FILE);

            /* Create the message to send. */
            MuleMessage theMuleMessage = new DefaultMuleMessage(
                "I am a Mule message!", theContext);

            /* Create a Mule client and send the message to it. */
            theMuleClient = createMuleClient(theContext);
            theMuleMessage = sendMessageToMule(theMuleMessage, theMuleClient);

            System.out.println("**** Finished invoking Mule configuration!");
            LoggingService.logMessage(theMuleMessage);
        } catch (final Exception theException)
        {
            theException.printStackTrace();
        } finally
        {
            /*
             * Dispose of the Mule contexts. Disposing the context
             * automatically stops it and we do not have to bother
             * with the exception declared by the stop method.
             * Must dispose if we wish the application to terminate,
             * otherwise there will be unreleased resources.
        }
    }
}
```

```

        */
        if (theContext != null)
        {
            theContext.dispose();
        }
    }

private MuleContext startMule(final String inMuleConfigFile) throws MuleException
{
    /* Starts an instance of Mule using the supplied configuration file. */
    String[] theConfigFiles = { inMuleConfigFile };

    MuleContextFactory theContextFactory = new DefaultMuleContextFactory();
    ConfigurationBuilder theConfigBuilder =
        new SpringXmlConfigurationBuilder(theConfigFiles);
    MuleContextBuilder theContextBuilder = new DefaultMuleContextBuilder();
    MuleContext theMuleContext = theContextFactory.createMuleContext(
        theConfigBuilder, theContextBuilder);
    theMuleContext.start();

    return theMuleContext;
}

private MuleClient createMuleClient(final MuleContext inMuleContext)
throws MuleException
{
    MuleClient theMuleClient = new MuleClient(inMuleContext);
    return theMuleClient;
}

private MuleMessage sendMessageToMule(final MuleMessage inMessage,
final MuleClient inMuleClient) throws Exception
{
    /* Modify the URL of the service which to send the message here. */
    MuleMessage theReceivedMsg = inMuleClient.send(MULE_SERVICE2_URL, inMessage);
    return theReceivedMsg;
}
}

```

Note that:

- The flow of the starter class is identical to the Mule 2.x version:
  - Start Mule with a configuration file.
  - Create a Mule message.
  - Send the Mule message to a service in the configuration.
  - Examine the result.
- There are two different Mule endpoint URLs defined as constants in the class. One is MULE\_SERVICE1, the other is MULE\_SERVICE2. The first endpoint does not directly cause an exception to be thrown, but invokes a flow that causes an exception to be thrown. The second endpoint will cause an exception to be thrown in the flow containing the endpoint. We will see what this means to the client invoking the different services.
- The URL of the service which to send the message to can be changed in the *sendMessageToMule* method.
- In the *doExample* method, we attempt to dispose the Mule context in order to shut down Mule. This works as expected in versions of Mule 3.x prior to version 3.2.0, but there seem to be some problem in more recent versions. You may need to shut down instances of the example program manually.
- Again, further details on the starter classes are available in the [next chapter](#).

## 9.7. Create the Exception Listeners

For the custom exception strategies used in this chapter's example, we will need an exception listener class. Since Mule 2.x and Mule 3.x use different interfaces for exception listeners, we must implement one listener class for Mule 2.x and another for Mule 3.x.

### Create the Mule 2.x Exception Listener

When implementing a custom exception strategy, we need to implement an exception listener class. There are a few choices on how to implement such a class:

- Directly implement the *java.beans.ExceptionListener* interface.  
Having your exception listener implement this interface is the most basic approach to implementing an exception listener.
- Inherit from the *org.mule.AbstractExceptionListener* class.  
The *AbstractExceptionListener* class implements logging, transaction handling and message routing methods. Abstract methods exist for handling different types of exceptions, such as messaging-, routing- and standard-exceptions.
- Inherit from the *org.mule.DefaultExceptionStrategy* class.  
The *DefaultExceptionStrategy* class provides default implementations of the abstract methods in the *AbstractExceptionListener* class.
- Inherit from the *org.mule.tck.functional.QuietExceptionStrategy* class.  
Only produce DEBUG-level log output for the different types of exceptions.
- Inherit from the *org.mule.service.DefaultServiceExceptionStrategy* class.  
Subclass of the *DefaultExceptionStrategy* class, this class also maintains per-service statistics about errors and routed messages.

In this example, we will use the second alternative; extending the *AbstractExceptionListener* class.

- In the package *com.ivan.mule*, implement the class *MyMule2ExceptionListener*:

```
package com.ivan.mule;

import org.mule.AbstractExceptionListener;
import org.mule.api.MuleMessage;
import org.mule.api.endpoint.ImmutableEndpoint;

import com.ivan.services.LoggingService;

/**
 * Custom Mule 2 exception listener.
 * To alter the logging-behaviour of the default exception strategy,
 * override the logException(Throwable) method.
 *
 * @author Ivan A Krizsan
 */
public class MyMule2ExceptionListener extends AbstractExceptionListener
{
    private String mListenerName;

    @Override
    protected void logException(final Throwable inException)
    {
        /* Never log exceptions here. */
    }

    @Override
    public void handleMessagingException(final MuleMessage inMessage,
```

```

        final Throwable inException)
    {
        System.out.println("**** MyMule2ExceptionListener.handleMessagingException: "
            + mListenerName);
        LoggingService.logMessage(inMessage);

        /*
         * Route the exception. Also handles transactions.
         */
        routeException(inMessage, null, inException);
    }

@Override
public void handleRoutingException(final MuleMessage inMessage,
    final ImmutableEndpoint inEndpoint, final Throwable inException)
{
    System.out.println("**** MyMule2ExceptionListener.handleRoutingException: "
        + mListenerName);
    System.out.println("    Message id: " + inMessage.getUniqueId());
    System.out.println("    Endpoint: " + inEndpoint.getName());
    System.out.println("    Exception: " + inException.getMessage());

    /*
     * Route the exception. Also handles transactions.
     */
    routeException(inMessage, inEndpoint, inException);
}

@Override
public void handleLifecycleException(final Object inComponent,
    final Throwable inException)
{
    System.out.println("**** MyMule2ExceptionListener.handleLifecycleException: "
        + mListenerName);
    System.out.println("    Component: " + inComponent);
    System.out.println("    Exception: " + inException.getMessage());

    /*
     * Route the exception. Also handles transactions.
     */
    routeException(null, null, inException);
}

@Override
public void handleStandardException(final Throwable inException)
{
    System.out.println("**** MyMule2ExceptionListener.handleStandardException: "
        + mListenerName);
    System.out.println("    Exception: " + inException.getMessage());

    /*
     * Route the exception. Also handles transactions.
     */
    routeException(null, null, inException);
}

public String getListenerName()
{
    return mListenerName;
}

public void setListenerName(final String inListenerName)
{
    mListenerName = inListenerName;
}
}

```

Note that:

- The *MyMule2ExceptionListener* class extends the *AbstractExceptionListener* class. One consequence is that there are four exception-handling methods that must be implemented, since they are declared as abstract in the parent class.

- The *logException* method is overridden.  
Since we'll implement our own, more detailed, logging of exceptions, the original logging of the exception handler is completely disabled by replacing the *logException* method with an empty method.
- Each of the exception handling methods logs information to the console.  
The information logged depends on the type of exception handled.
- Each of the exception handling methods calls the *routeException* method.  
The *routeException* method routes the message causing the exception to one or more endpoints declared in the `<custom-exception-strategy>` element.  
If there are no endpoints to route the message to, any current transaction will be marked for rollback by the *routeException* method.
- The class has an instance variable, *mListenerName*, with associated getter and setter methods.  
This exposes a property named “*listenerName*”, which is used to identify an instance of the exception listener in log messages.

### **Create the Mule 3.x Exception Listener**

When using Mule 3.x, there are more options available when implementing a custom exception listener class. The separation between messaging exceptions and system exceptions are made more clear and there are more fine-grained options available.

The following options are available implementing an exception listener when using Mule 3.2.0.

- Directly inherit from *org.mule.exception.AbstractExceptionStrategy*.  
Common parent for both messaging and system exception handlers that implements common behaviour of exception handlers.  
It is recommended to consider the type of exception handler one wants to develop and inherit from either *AbstractSystemExceptionStrategy* or *AbstractMessagingExceptionStrategy*.
- Directly implement the *org.mule.api.exception.SystemExceptionHandler* interface.  
System exceptions are considered to be exceptions that are not connected to the processing of a Mule message. Note that the *handleException* method only takes one single parameter – an exception.
- Directly inherit from *org.mule.exception.AbstractSystemExceptionStrategy*.  
Child class of *AbstractExceptionStrategy*.  
This exception strategy class implements basic handling of system exceptions.
- Directly inherit from the *org.mule.exception.DefaultSystemExceptionStrategy*.  
This class implements the default exception strategy for system exceptions.  
Child class of *AbstractSystemExceptionStrategy*.
- Directly implement the *org.mule.api.exception.MessagingExceptionHandler* interface.  
A messaging exception is an exception that occurred in connection to the processing of a Mule message.  
The event object returned by the *handleException* method is the event that is to continue to be routed through the remaining part of the flow. This gives the exception handler an opportunity to modify, or even replace, the event that is passed on through the remaining part of the flow after the exception occurred.
- Directly inherit from *org.mule.exception.AbstractMessagingExceptionStrategy*.

Child class of *AbstractExceptionStrategy*.

Implements basic handling of messaging exceptions. Sets the message payload to the null payload and sets the exception payload to the thrown exception.

- Directly inherit from *org.mule.exception.DefaultMessagingExceptionStrategy*.  
Child class of *AbstractMessagingExceptionStrategy*.  
Implements the default exception strategy for system exceptions.
- Inherit from the *org.mule.tck.functional.QuietExceptionStrategy* class.  
Only produce DEBUG-level log output for the different types of exceptions.

In the example program, we will implement a messaging exception strategy by inheriting from *AbstractMessagingExceptionStrategy*.

- In the package *com.ivan.mule*, implement the class *MyMule3ExceptionListener*:

```
package com.ivan.mule;

import org.mule.api.MuleContext;
import org.mule.api.MuleEvent;
import org.mule.api.MuleMessage;
import org.mule.api.exception.RollbackSourceCallback;
import org.mule.exception.AbstractMessagingExceptionStrategy;

import com.ivan.services.LoggingService;

/**
 * Custom Mule 3 exception listener.
 *
 * @author Ivan Krizsan
 */
public class MyMule3ExceptionListener extends AbstractMessagingExceptionStrategy
{
    private String mListenerName;

    public MyMule3ExceptionListener(final MuleContext inMuleContext)
    {
        super(inMuleContext);
    }

    @Override
    protected void logException(final Throwable inException)
    {
        /* Never log exceptions here. */
    }

    /* (non-Javadoc)
     * @see
     */
    org.mule.exception.AbstractMessagingExceptionStrategy#doHandleException(java.lang.Exception,
        org.mule.api.MuleEvent, org.mule.api.exception.RollbackSourceCallback)
    */
    @Override
    protected void doHandleException(final Exception inException,
        final MuleEvent inEvent,
        final RollbackSourceCallback inRollbackMethod)
    {
        MuleMessage theMessage = inEvent.getMessage();

        System.out.println("**** MyMule3ExceptionListener.doHandleException: "
            + mListenerName);
        LoggingService.logMessage(theMessage);

        /*
         * Let the superclass handle transactions, routing etc of the
         * exception.
         */
        super.doHandleException(inException, inEvent, inRollbackMethod);
    }

    public String getListenerName()
    {
        return mListenerName;
    }
}
```

```

    }

    public void setListenerName(String inListenerName)
    {
        mListenerName = inListenerName;
    }
}

```

Note that:

- The exception listener class *MyMule3ExceptionListener* extends the class *AbstractMessagingExceptionStrategy*.  
Contrary to the Mule 2.x exception listener superclass, the Mule 3.x class *AbstractMessagingExceptionStrategy* does not have any abstract methods that we are required to implement so we just override the methods which behaviour we want to modify.
- The *logException* method is overridden.  
Since we'll implement our own, more detailed, logging of exceptions, the original logging of the exception handler is completely disabled by replacing the *logException* method with an empty method.
- The *doHandleException* method is overridden.  
The message outputs information to the console and invokes the superclass *doHandleException* method. The superclass method handles updating of any statistics, rollback of an active transaction etc.  
We could also have overridden the *handleException* method with three parameters.
- The class has an instance variable, *mListenerName*, with associated getter and setter methods.  
This exposes a property named “*listenerName*”, which is used to identify an instance of the exception listener in log messages.

## 9.8. Create the Mule Configuration Files

This chapter's example program uses a total of three Mule configuration files; two for the Mule 2.x version and one for the Mule 3.x version.

The Mule 2.x version uses, not surprisingly, the Mule 2.x configuration style, with a model that contains a number of services. Since model inheritance is possible, a parent model has been defined in a separate file. The parent model define a model-global exception listener and the error logging service.

In the other configuration file, among other things, there is a service with a service-local exception strategy.

The Mule 3.x version uses a flow, instead of a model and services. Since flows do not support inheritance, we cannot declare a common exception listener. In addition, when using flows, the only place where an exception strategy can be declared is inside a flow. While this can be seen as a limitation, it does reduce ambiguity.

### Create the Mule 2.x Configuration Files

The two Mule 2.x configuration files are to be located in the package *com.ivan.muleconfig*.

- Create a file named “mule2-exceptionstrategymodel.xml” in the above mentioned package. It has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/vm/2.2
          http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <!--
        This model declares the model-global exception strategy and
        the error-logging service used by the exception strategy to
        log messages causing exceptions.
    -->
    <model name="MainModel">
        <!--
            An exception strategy, custom or default, is declared first
            in a model. Such an exception strategy is applied to all the
            services and components in the model.
            If no customization of the exception strategy is required,
            use the <default-service-exception-strategy> element.
        -->
        <custom-exception-strategy class="com.ivan.mule.MyMule2ExceptionListener">
            <!--
                Messages caught by the exception strategy are sent to
                the endpoint(s) listed below.
                With Mule 2.x, multiple endpoints may be specified.
            -->
            <outbound-endpoint address="vm://errorLoggingServiceInbound">
                <message-properties-transformer>
                    <add-message-property key="exceptionListener" value="model-global"/>
                </message-properties-transformer>
            </outbound-endpoint>
            <!--
                Name of the exception listener instance.
                Used when writing log messages to tell the different
                listeners apart.
            -->
        
```

```

-->
<spring:property name="listenerName" value="Listener 1"/>
</custom-exception-strategy>

<!--
    Service that logs errors by printing the message payload and
    properties to the console.
-->
<service name="errorLoggingService">
    <inbound>
        <vm:inbound-endpoint path="errorLoggingServiceInbound"
synchronous="true"/>
    </inbound>
    <component class="com.ivan.services.LoggingService"/>
</service>
</model>
</mule>

```

Note that:

- The configuration file define a model named “MainModel”.
- The model contains a custom exception strategy.  
A custom exception strategy allows us to supply a custom implementation class, using the *class* attribute of the `<custom-exception-strategy>` element. The exception strategy applies to all services and components in the model.
- The custom exception strategy is implemented by the class *com.ivan.mule.MyMule2ExceptionListener*.
- The custom exception strategy is declared first in the model.
- The `<custom-exception-strategy>` element contains a `<outbound-endpoint>` element.  
This causes messages caught by the exception strategy to be sent to the specified endpoint. When using Mule 2.x, multiple endpoints may be specified.
- The `<outbound-endpoint>` element contains a `<message-properties-transformer>` element.  
The message properties transformer adds a message property with the name “exceptionListener” and the value “model-global” to the Mule message before it is passed on to the outbound endpoint. See the section on [Message Properties](#) in the reference part of this book for additional details on message properties.
- The `<custom-exception-strategy>` element also contains a `<spring:property>` element.  
The custom exception strategy is a specialized type of Spring bean and we can, in regular Spring manner, inject property values into such a bean.
- A service named “errorLoggingService” is declared in the model.  
This service is a synchronous service that receives messages over the VM transport, implemented by the *LoggingService* class we implemented [earlier](#).

We are now ready to create the second Mule 2.x configuration file, which contains two receiving services, the Hello service and the service that always causes an exception to be thrown.

- Create a file named “mule2-config.xml” in the *com.ivan.muleconfig* package.  
It has the following contents:

```

<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:stdio="http://www.mulesource.org/schema/mule/stdio/2.2"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
      xmlns:spring="http://www.springframework.org/schema/beans"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/stdio/2.2
          http://www.mulesource.org/schema/mule/stdio/2.2/mule-stdio.xsd

          http://www.mulesource.org/schema/mule/vm/2.2
          http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<spring:beans>
    <spring:import resource="mule2-exceptionstrategymodel.xml"/>
</spring:beans>

<!--
    The model-global exception strategy is declared in the parent
    model with the same name, thus the inherit attribute being set to true.
-->
<model name="MainModel" inherit="true">
    <!--
        Service that receives a message and tries to send the message
        to multiple endpoints until finding one that does not throw an exception.
    -->
    <service name="receiverService">
        <inbound>
            <inbound-endpoint address="vm://receiverServiceURL"/>
        </inbound>
        <outbound>
            <!--
                The exception based router attempts to send the message
                to the contained endpoints in the order listed
                until having encountered an endpoint that does not
                throw an exception.
            -->
            <exception-based-router>
                <!--
                    All endpoints but the last in the exception-based router
                    will be forced to be synchronous. The last outbound endpoint
                    will be asynchronous unless the synchronous attribute is
                    explicitly set to true, as in this case. The synchronous
                    attribute on the inbound endpoint to which the message is
                    sent will not be considered by the router.
                -->
                <vm:outbound-endpoint path="exceptionServiceInbound"/>
                <vm:outbound-endpoint
                    path="helloServiceInbound" synchronous="true"/>
            </exception-based-router>
        </outbound>
    </service>

    <!--
        Service that will always throw an exception when invoked.
    -->
    <service name="exceptionService">
        <inbound>
            <inbound-endpoint address="vm://exceptionServiceInbound"/>
        </inbound>
        <component>
            <singleton-object class="com.ivan.services.ExceptionService"/>
        </component>
    </service>

    <!--
        Greeting service that prints a greeting and the time when invoked.
    -->

```

```

-->
<service name="helloService">
    <inbound>
        <vm:inbound-endpoint path="helloServiceInbound"/>
    </inbound>
    <component>
        <singleton-object class="com.ivan.services.HelloService"/>
    </component>
</service>

<!--
    Service with its own exception strategy.
    The service will throw an exception when being invoked.
-->
<service name="gotExceptionStrategyService">
    <inbound>
        <inbound-endpoint address="vm://gotExceptionStrategyServiceURL"/>
    </inbound>

    <component>
        <singleton-object class="com.ivan.services.ExceptionService"/>
    </component>

    <!--
        An exception strategy, custom or default, is declared last
        in a service and applies to all components in the service.
        This strategy will handle the exceptions that occur in the
        service, but not in other services to which messages are
        passed on.
        If no customization of the exception strategy is required,
        use the <default-service-exception-strategy> element.
        The exception strategy local to the service will be given
        precedence over the exception strategy defined on the
        model level.
    -->
    <custom-exception-strategy class="com.ivan.mule.MyMule2ExceptionListener">
        <!--
            Messages caught by the exception strategy are sent to
            the endpoint listed below.
            With Mule 2.x, multiple endpoints may be specified.
        -->
        <outbound-endpoint address="vm://errorLoggingServiceInbound">
            <!--
                Upon passing the message to the error logging service,
                set a message property.
            -->
            <message-properties-transformer>
                <add-message-property
                    key="exceptionListener" value="service-local"/>
            </message-properties-transformer>
        </outbound-endpoint>
        <!--
            Name of the exception listener instance.
            Used when writing log, to tell the different listeners apart.
        -->
        <spring:property name="listenerName" value="Listener 2"/>
    </custom-exception-strategy>
</service>
</model>
</mule>

```

Note that:

- The above Mule configuration imports the Mule configuration that we previously defined.
- The configuration file defines a model named “MainModel”.  
The model also has the *inherit* attribute set to true. In order to be able to inherit from a parent model, the parent and child models must have the same name.
- The model contains a service named “receiverService”.  
This service exposes an inbound endpoint that uses the VM transport.  
This is one of the two endpoints that can be seen in the figure describing the [Mule 2.x](#)

[configuration structure](#) depicted in the beginning of this chapter.

- The <outbound> element of the “receiverService” uses an exception based router. An exception based router attempts to send a message to the outbound endpoints listed in a <exception-based-router> element, trying one endpoint at a time until it finds one that succeeds. For details on the exception-based router, please refer to the section on [Exception-Dependent Message Routing](#) in the reference section of this book.  
It should be noted that all outbound endpoints in an <exception-based-router> element, except the last, will be forced to be synchronous.
- The last <outbound-endpoint> element in the <exception-based-router> element has the *synchronous* attribute set to true.  
The client of a service, in this case the <outbound-endpoint> element of the <exception-based-router>, determine whether the service is invoked synchronously or asynchronously. The synchronous attribute on the service's inbound endpoint is disregarded.  
This is also true for the services that are exposed to programmatic clients.
- The model contains a service named “exceptionService”.  
This is an ordinary service backed by a single instance of the class implementing the service. As the “receiverService”, it also exposes an inbound endpoint that uses the VM transport. We recall from [when we created the service implementation class](#) that this service will always throw an exception.
- The next service in the model is the “helloService”.  
This is the service that will extend a greeting, without a name, when invoked.  
It also exposes an endpoint that uses the VM transport.
- The next service, “gotExceptionStrategyService”, is similar to the “exceptionService”. It exposes an inbound endpoint that uses the VM transport, albeit with a different address. The two services uses the same [service implementation class](#); that which will always throw an exception when invoked.
- There is a <custom-exception-strategy> in the “gotExceptionStrategyService”.  
The “gotExceptionStrategyService” service defines a service-local custom exception strategy. Such an exception strategy is declared last in the service and applies to all components in the service. Any service-local exception strategy takes precedence over any exception strategy defined on the model level.
- The custom exception strategy is implemented by the class *com.ivan.mule.MyMule2ExceptionListener*.
- The <custom-exception-strategy> element contains a <outbound-endpoint> element.  
This causes messages caught by the exception strategy to be sent to the specified endpoint. When using Mule 2.x, multiple endpoints may be specified.
- The <outbound-endpoint> element contains a <message-properties-transformer> element.  
The message properties transformer adds a message property with the name “exceptionListener” and the value “service-local” to the Mule message before it is passed on to the outbound endpoint. See the section on [Message Properties](#) in the reference part of this book for additional details on message properties.
- The <custom-exception-strategy> element also contains a <spring:property> element.  
The custom exception strategy is a specialized type of Spring bean and we can, in regular Spring manner, inject property values into such a bean.

## Create the Mule 3.x Configuration File

The Mule 3.x configuration uses a number of flows, exposing services for external and internal use. All endpoints in the configuration file uses the request-response message exchange pattern. The reason for this is mainly to increase the readability of the output generated by the example program.

- Create a file named “mule3-config.xml” in the package *com.ivan.muleconfig*. It has the following contents:

```
<?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:vm="http://www.mulesoft.org/schema/mule/vm"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
        http://www.mulesoft.org/schema/mule/core
        http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

        http://www.mulesoft.org/schema/mule/vm
        http://www.mulesoft.org/schema/mule/vm/3.2/mule-vm.xsd

        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!--
        Flow that receives messages from an external source and
        passes them on for further processing.
        In Mule 3.x the exchange-pattern used by an endpoint is
        one-way as per default, except for in HTTP(S), TCP, SSL and
        servlet endpoints.
        As opposed to Mule 2.x, the exchange-pattern declared on
        the inbound endpoint decides which message exchange pattern
        will be used.
    -->
    <flow name="ReceiverFlow">
        <inbound-endpoint address="vm://receiverServiceURL"
            exchange-pattern="request-response"/>
        <!--
            The first-successful message processor attempts to send the
            message to the the contained endpoints in the order listed
            until having encountered an endpoint that succeeds.
            The exchange-pattern attribute of the outbound endpoints
            in the first-successful message processor can be set to
            "request-response" for better reliability, but may be
            "one-way" as well.
        -->
        <first-successful>
            <vm:outbound-endpoint path="exceptionServiceInbound"
                exchange-pattern="request-response"/>
            <vm:outbound-endpoint path="helloServiceInbound"
                exchange-pattern="request-response"/>
        </first-successful>
    </flow>
    <!--
        Flow that exposes a service that, each time a message is
        sent to it, will cause an exception to be thrown.
    -->
    <flow name="AlwaysExceptionFlow">
        <inbound-endpoint address="vm://exceptionServiceInbound"
            exchange-pattern="request-response"/>
        <component>
            <singleton-object class="com.ivan.services.ExceptionService"/>
        </component>
    </flow>
    <!--
        Flow that exposes the Hello service.
    -->
    <flow name="HelloFlow">
        <vm:inbound-endpoint path="helloServiceInbound"
            exchange-pattern="request-response"/>
```

```

<component>
    <singleton-object class="com.ivan.services.HelloService" />
</component>
</flow>

<!--
    Flow that exposes the error logging service.
-->
<flow name="ErrorLoggingFlow">
    <vm:inbound-endpoint path="errorLoggingServiceInbound"
        exchange-pattern="request-response" />

    <component class="com.ivan.services.LoggingService" />
</flow>

<!--
    Flow that exposes a service that will always throw an
    exception when being invoked.
    This flow has a custom exception listener.
-->
<flow name="GotExceptionStrategyFlow">
    <inbound-endpoint
        address="vm://gotExceptionStrategyServiceURL"
        exchange-pattern="request-response" />

    <component>
        <singleton-object class="com.ivan.services.ExceptionService" />
    </component>

    <!--
        This flow has a custom exception listener that produces
        more detailed information in the log about the message
        that was processed when the exception occurred.
        Exception strategies are declared in the end of a flow
        declaration.
-->
    <custom-exception-strategy class="com.ivan.mule.MyMule3ExceptionListener">
        <!--
            Messages caught by the exception strategy are sent to
            the endpoint listed below.
            With Mule 3.x, only one single endpoint may be specified.
-->
        <outbound-endpoint address="vm://errorLoggingServiceInbound"
            exchange-pattern="request-response" />
        <!--
            Add a message property to the message before sending it
            to the log service.
-->
        <message-properties-transformer scope="outbound">
            <add-message-property
                key="exceptionListener"
                value="In GotExceptionStrategyFlow" />
        </message-properties-transformer>
    </outbound-endpoint>
        <!--
            Name of the exception listener instance.
            Used when writing log messages to tell the different
            listeners apart.
-->
        <spring:property name="listenerName" value="Listener 1" />
    </custom-exception-strategy>
</flow>
</mule>

```

Note that:

- The Mule configuration file contains five `<flow>` elements.  
We could refactor the configuration to have fewer flows, but this solution was deemed to be appropriate for this example.
- The first flow is named “ReceiverFlow”.
- The “ReceiverFlow” `<flow>` element contains an `<inbound-endpoint>` element.

This element exposes an inbound endpoint using the VM transport with the address “vm://receiverServiceURL”.

This is one of the two endpoints that can be seen in the figure describing the [Mule 3.x configuration structure](#) depicted in the beginning of this chapter.

- The <inbound-endpoint> element discussed in above has a *exchange-pattern* attribute with the value “request-response”.

In Mule 3.x, the exchange pattern declared on the service and the exchange pattern used by the client must match, otherwise there may be unexpected results.

Also, we want the exposed service to be synchronous, in order for the client to wait until there is a result from the request available.

- The “ReceiverFlow” <flow> element contains a <first-successful> element.

The <first-successful> element defines a message processor which is similar to the exception-based router used in the Mule 2.x configuration in this chapter.

For details on the first-successful message processor, please refer to the section on [Exception-Dependent Message Routing](#) in the reference section of this book.

Outbound in the first-successful message processor may be asynchronous or synchronous, though the latter will provide a higher degree of reliability.

- The <first-successful> element contains two outbound endpoints.

Mule 3.x does not impose synchronicity on the endpoints contained in the <first-successful> element. The endpoints in this example have the *exchange-pattern* attribute set to “request-response” to increase reliability.

- The next flow is named “AlwaysExceptionFlow”.

As the name indicates, it exposes a service that will throw an exception each time a message is sent to it. The address of the service is “vm://exceptionServiceInbound”.

- The next flow is named “HelloFlow”.

This flow provides a service that will extend a greeting, without a name, when invoked. It exposes an endpoint that uses the VM transport.

- The next flow, “ErrorLoggingFlow”, provides a service that logs errors.

Again, the service exposed uses the VM transport.

- The last flow is named “GotExceptionStrategyFlow”.

- The “GotExceptionStrategyFlow” <flow> element contains an <inbound-endpoint> element.

This element exposes am inbound endpoint using the VM transport with the address “vm://gotExceptionStrategyServiceURL”.

This is one of the two endpoints that can be seen in the figure describing the [Mule 3.x configuration structure](#) depicted in the beginning of this chapter.

- The service in the “GotExceptionStrategyFlow” will cause an exception to be thrown each time a message is sent to the service.

- There is a <custom-exception-strategy> element in the “ GotExceptionStrategyFlow” flow. The exception strategy is declared at the end of the flow and applies to components and services of the flow.

- The custom exception strategy is implemented by the class *com.ivan.mule.MyMule3ExceptionListener*.

- The <custom-exception-strategy> element contains an <outbound-endpoint> element.

This causes messages caught by the exception strategy to be sent to the specified endpoint. When using Mule 3.x, only one single endpoint may be specified.

- The <outbound-endpoint> element contains a <message-properties-transformer> element. The message properties transformer adds a message property with the name “exceptionListener” and the value “In GotExceptionStrategyFlow” to the Mule message before it is passed on to the outbound endpoint. See the section on [Message Properties](#) in the reference part of this book for additional details on message properties.
- The <custom-exception-strategy> element also contains a <spring:property> element. The custom exception strategy is a specialized type of Spring bean and we can inject property values into such a bean.

## 9.9. Run the Example Program

With all the different parts of this chapter's example program, we are now ready to run it and examine its behaviour. Due to the differences between Mule 2.x and Mule 3.x, both versions of the example program will be examined in detail. In addition, each version of the program will be run twice, sending messages to different endpoints.

Recall that we are using starter classes to start the Mule 2.x and Mule 3.x versions of the example program.

### ***Run the Mule 2.x Version of the Example Program***

In this section, we'll run the Mule 2.x version of the example program.

Ensure that your example program project is configured with the Mule 2.x distribution on the classpath as described in [appendix B](#), before proceeding.

### **Send a Message to the First Endpoint**

Before starting the Mule 2.x version of the example program for the first time, open the class *Mule2ExceptionHandlingExampleStarter* and examine the *sendMessageToMule* method. The URL passed as the first parameter to the *send* method called is to be MULE\_SERVICE1\_URL, as in this code snippet:

```
...
    private MuleMessage sendMessageToMule(final MuleMessage inMessage,
        final MuleClient inMuleClient) throws Exception
    {
        MuleMessage theReceivedMsg;

        /* Modify the URL of the service which to send the message here. */
        theReceivedMsg = inMuleClient.send(MULE_SERVICE1_URL, inMessage);
        return theReceivedMsg;
    }
...
}
```

This URL is *vm://receiverServiceURL*, which, if you remember [the figure at the start of this chapter](#), is the endpoint exposed by the “receiverService” service.

- Right-click on the *Mule2ExceptionHandlingExampleStarter* class and select Run As -> Java Application.  
There will be a significant amount of output on the console, we will look at the relevant parts one-by-one.

### **Examine the Output**

This first line indicates that the exception service was called and that the exception thrown from the service will contain the id 1 in its message.

```
*** In ExceptionService.onCall(): 1
```

This section indicate that the exception listener implemented in the *MyMule2ExceptionListener* class received a notification. The notification was received by the *handleMessagingException*, which means that an exception occurred during the processing of a message. The message payload, message properties in the inbound and outbound scopes and exception payload are listed. The exception listener that received the notification has the name “Listener 1” - this is the exception listener defined on the model-level.

```
*** MyMule2ExceptionListener.handleMessagingException: Listener 1
Current time: 1328803879326
Message payload: I am a Mule message!
Message properties:
    INBOUND: {}
    OUTBOUND: {MULE_CORRELATION_GROUP_SIZE=2, MULE_ENCODING=UTF-8,
MULE_CORRELATION_ID=b14c5888-5338-11e1-ac1-c97990807652,
MULE_ENDPOINT=vm://exceptionServiceInbound,
MULE_ORIGINATING_ENDPOINT=endpoint.vm.exceptionServiceInbound}
Exception payload: null
NO EXCEPTION PAYLOAD AVAILABLE
```

The following log output is produced by Mule when routing the exception.

We can note the following about the message logged:

- There is no property with the name “exceptionListener” in the message's outbound scope. Apparently the message properties transformer has not been applied yet.
- The message payload is a string (highlighted in yellow).
- The last endpoint that received the message has the address “vm://exceptionServiceInbound”. This is indicated by the MULE\_ENDPOINT message property in the outbound scope. The MULE\_ORIGINATING\_ENDPOINT message property also supply the same information, but in a different form.

```
[02-09 17:11:19] ERROR MyMule2ExceptionListener [main]: Message being processed is:
org.mule.transport.DefaultMessageAdapter/org.mule.transport.DefaultMessageAdapter@15364e
e5{id=b14c5888-5338-11e1-ac1-c97990807652, payload=java.lang.String,
properties=Properties{invocation:{}, inbound:{}, outbound:{MULE_ENCODING=UTF-8,
MULE_CORRELATION_GROUP_SIZE=2, MULE_CORRELATION_ID=b14c5888-5338-11e1-ac1-c97990807652,
MULE_ENDPOINT=vm://exceptionServiceInbound,
MULE_ORIGINATING_ENDPOINT=endpoint.vm.exceptionServiceInbound}, session:{}, },
correlationId=b14c5888-5338-11e1-ac1-c97990807652, correlationGroup=2, correlationSeq=-1,
encoding=UTF-8, exceptionPayload=null}
```

Next, the logging service receives the message and logs its contents. Note that:

- There now is a message property with the name “exceptionListener” in the message's outbound scope (highlighted in blue). The property has the value “model-global”. This tells us that the exception notification was received by the custom exception strategy defined in the model.
- The message payload is now an instance of *ExceptionMessage* (highlighted in yellow). This type of message wraps the message that caused the exception and holds information on the component in which the exception occurred and the endpoint that received the message immediately prior to the exception was thrown.

```
***** Logging service received a message:
Current time: 1328803879336
Message payload: ExceptionMessage{message=I am a Mule message!,
context={MULE_ENCODING=UTF-8, MULE_CORRELATION_GROUP_SIZE=2,
MULE_CORRELATION_ID=b14c5888-5338-11e1-ac1-c97990807652,
MULE_ENDPOINT=vm://exceptionServiceInbound,
MULE_ORIGINATING_ENDPOINT=endpoint.vm.exceptionServiceInbound}exception=org.mule.api.ser
vice.ServiceException: Component that caused exception is:
SedaService{exceptionService}. Message payload is of type: String,
componentName='exceptionService', endpointUri=vm://exceptionServiceInbound,
timeStamp=Thu Feb 09 17:11:19 CET 2012}
Message properties:
    INBOUND: {}
    OUTBOUND: {MULE_ENCODING=UTF-8, MULE_CORRELATION_GROUP_SIZE=2,
MULE_CORRELATION_ID=b14c5888-5338-11e1-ac1-c97990807652,
MULE_ENDPOINT=vm://errorLoggingServiceInbound, MULE_REMOTE_SYNC=true,
MULE_ORIGINATING_ENDPOINT=endpoint.vm.errorLoggingServiceInbound,
exceptionListener=model-global}
Exception payload: null
NO EXCEPTION PAYLOAD AVAILABLE
```

The following line tells us that the Hello service has received and processed a message. We can thus

conclude that the exception-based router did its job; first it attempted to send the message to the exception service. When that failed, it sent the message to the Hello service.

```
*** In HelloService.onCall(): Hello. The time is now: Thu Feb 09 17:11:19 CET 2012
```

This output indicates that our starter-program received the response message from Mule and displays the contents of the response message.

We can see that:

- The message has a payload (highlighted in yellow).  
The payload is the string that was produced by the Hello service.
- There is no exception payload.
- The message property with the name “exceptionListener” is not present in the message.

```
*** Finished invoking Mule configuration!
Current time: 1328803879352
Message payload: Hello. The time is now: Thu Feb 09 17:11:19 CET 2012
Message properties:
    INBOUND: {}
    OUTBOUND: {MULE_CORRELATION_GROUP_SIZE=2, MULE_ENCODING=UTF-8,
    MULE_CORRELATION_ID=b14c5888-5338-11e1-acc1-c97990807652,
    MULE_ENDPOINT=vm://helloServiceInbound,
    MULE_ORIGINATING_ENDPOINT=endpoint.vm.helloServiceInbound}
Exception payload: null
NO EXCEPTION PAYLOAD AVAILABLE
```

## Sending Message to the Second Endpoint

Again, open the class *Mule2ExceptionHandlingExampleStarter* and examine the *sendMessageToMule* method. Change the URL passed as the first parameter to the *send* method called is to be *MULE\_SERVICE2\_URL*, as in this code snippet:

```
...
private MuleMessage sendMessageToMule(final MuleMessage inMessage,
    final MuleClient inMuleClient) throws Exception
{
    MuleMessage theReceivedMsg;

    /* Modify the URL of the service which to send the message here. */
    theReceivedMsg = inMuleClient.send(MULE_SERVICE2_URL, inMessage);
    return theReceivedMsg;
}
...
```

This URL is *vm://gotExceptionStrategyServiceURL*, which, if you remember [the figure at the start of this chapter](#), is the endpoint exposed by the “*getExceptionStrategyService*” service.

- Right-click on the *Mule2ExceptionHandlingExampleStarter* class and select Run As -> Java Application.  
We will look at the relevant parts of the console output one-by-one.

## Examine the Output

This first line indicates that the exception service was called and that the exception thrown from the service will contain the id 1 in its message.

```
*** In ExceptionService.onCall(): 1
```

This section indicate that the method *handleMessagingException* in the exception listener implemented in the *MyMule2ExceptionListener* class received a notification. The method name tells us that an exception occurred during the processing of a message. The message payload, message properties in the inbound and outbound scopes and exception payload are listed.

Instead of the model-global exception listener “Listener 1”, the exception listener “Listener 2” defined in the service “*gotExceptionStrategyService*”.

```
*** MyMule2ExceptionListener.handleMessagingException: Listener 2
Current time: 1328851964543
Message payload: I am a Mule message!
Message properties:
    INBOUND: {}
    OUTBOUND: {MULE_ENCODING=UTF-8,
    MULE_ENDPOINT=vm://gotExceptionStrategyServiceURL,
    MULE_ORIGINATING_ENDPOINT=endpoint.vm.gotExceptionStrategyServiceURL}
Exception payload: null
NO EXCEPTION PAYLOAD AVAILABLE
```

The following log output is produced by Mule when routing the exception.

We can note the following about the message logged:

- There is no property with the name “exceptionListener” in the message's outbound scope.  
The message properties transformer has not been applied yet.
- The message payload is a string (highlighted in yellow).
- The last endpoint that received the message has the address “*vm://gotExceptionServiceURL*”.

This is indicated by the *MULE\_ENDPOINT* message property in the outbound scope.  
The *MULE\_ORIGINATING\_ENDPOINT* message property also supply the same

information, but in a different form.

```
[02-10 06:32:44] ERROR MyMule2ExceptionListener [main]: Message being processed is:  
org.mule.transport.DefaultMessageAdapter/org.mule.transport.DefaultMessageAdapter@34f340  
71{id=a5ae9aee-53a8-11e1-a213-9d6be1359575, payload=java.lang.String,  
properties=Properties{invocation:{}, inbound:{}, outbound:{MULE_ENCODING=UTF-8,  
MULE_ENDPOINT=vm://gotExceptionStrategyServiceURL,  
MULE_ORIGINATING_ENDPOINT=endpoint.vm.gotExceptionStrategyServiceURL}, session:{}, },  
correlationId=null, correlationGroup=-1, correlationSeq=-1, encoding=UTF-8,  
exceptionPayload=null}
```

Here, the logging service has received the message and logs its contents. Note that:

- The message payload is now an instance of *ExceptionMessage* (highlighted in yellow). This type of message wraps the message that caused the exception and holds information on the component in which the exception occurred (highlighted in green) and the endpoint that received the message immediately prior to the exception was thrown.
- There now is a message property with the name “exceptionListener” in the message's outbound scope (highlighted in blue). The property has the value “service-local”. This tells us that the exception notification was received by the custom exception strategy defined in the service.

```
***** Logging service received a message:  
Current time: 1328851964596  
Message payload: ExceptionMessage{message=I am a Mule message!,  
context={MULE_ENCODING=UTF-8, MULE_ENDPOINT=vm://gotExceptionStrategyServiceURL,  
MULE_ORIGINATING_ENDPOINT=endpoint.vm.gotExceptionStrategyServiceURL}  
exception=org.mule.api.service.ServiceException: Component that caused exception is:  
SedaService{gotExceptionStrategyService}.  
Message payload is of type: String, componentName='gotExceptionStrategyService',  
endpointUri=vm://gotExceptionStrategyServiceURL, timeStamp=Fri Feb 10 06:32:44 CET 2012}  
Message properties:  
    INBOUND: {}  
    OUTBOUND: {MULE_ENCODING=UTF-8, MULE_ENDPOINT=vm://errorLoggingServiceInbound,  
    MULE_REMOTE_SYNC=true,  
    MULE_ORIGINATING_ENDPOINT=endpoint.vm.errorLoggingServiceInbound,  
    exceptionListener=service-local}  
Exception payload: null  
NO EXCEPTION PAYLOAD AVAILABLE
```

The final part of the output indicates that our starter-program received a response message from Mule. We can see that:

- The message has a null payload (highlighted in yellow).
- There is an exception payload from which we also can obtain an exception stacktrace. The exception payload type is highlighted in orange.
- The message property with the name “exceptionListener” is still in the message's outbound scope (highlighted in blue).
- There are messages, highlighted in green in the output below, in the exceptions. This indicate that the exception with id 1 caused the exception payload of the message received by the starter program.

We can also see that the outer and inner exceptions thrown from the exception service has retained their relation, with the outer exception wrapping the inner exception.

```
*** Finished invoking Mule configuration!  
Current time: 1328851964623  
Message payload: {NullPayload}  
Message properties:  
    INBOUND: {}  
    OUTBOUND: {MULE_ENCODING=UTF-8, MULE_ENDPOINT=vm://errorLoggingServiceInbound,  
    MULE_REMOTE_SYNC=true,  
    MULE_ORIGINATING_ENDPOINT=endpoint.vm.errorLoggingServiceInbound,  
    exceptionListener=service-local}  
Exception payload: org.mule.message.DefaultExceptionPayload@976484e  
*** EXCEPTION STACKTRACE START:  
org.mule.api.service.ServiceException: Component that caused exception is:  
SedaService{gotExceptionStrategyService}. Message payload is of type: String
```

```

        at
org.mule.component.DefaultLifecycleAdapter.invoke(DefaultLifecycleAdapter.java:216)
        ...
        at org.mule.module.client.MuleClient.send(MuleClient.java:595)
        at
com.ivan.starter.Mule2ExceptionHandlingExampleStarter.sendMessageToMule(Mule2ExceptionHa
ndlingExampleStarter.java:99)
        at
com.ivan.starter.Mule2ExceptionHandlingExampleStarter.doExample(Mule2ExceptionHandlingEx
ampleStarter.java:47)
        at
com.ivan.starter.Mule2ExceptionHandlingExampleStarter.main(Mule2ExceptionHandlingExample
Starter.java:28)
Caused by: org.mule.api.DefaultMuleException: I am an outer exception with id 1
        at com.ivan.services.ExceptionService.onCall(ExceptionService.java:26)
        ... many more
Caused by: java.lang.Exception: I am a nested exception with id 1
        at com.ivan.services.ExceptionService.onCall(ExceptionService.java:24)
        ... many more
*** EXCEPTION STACKTRACE END.

```

We have seen that the exceptions from a service can be handled at two different levels when using <model> and <service> configurations; either on the model-level or on the service-level.

We have also seen the different stages of the processing of an exception, with any exception listener receiving the unaltered message, as it was at the time when the exception occurred, and enrichment of the message before it was sent to a logging service by the exception handler.

Not directly related to exception handling, but nevertheless of importance is the observation that the sender of a message decides whether the interaction with the endpoint is to be synchronous or asynchronous. This, of course, under the assumption that the transport used supports both modes of communication.

## **Run the Mule 3.x Version of the Example Program**

In this section, we'll run the Mule 3.x version of the example program.

Before proceeding, ensure that your example program project is configured with the Mule 3.x distribution on the classpath, as described in [appendix B](#), before proceeding.

### **Send a Message to the First Endpoint**

Before starting the Mule 3.x version of the example program for the first time, open the class *Mule3ExceptionHandlingExampleStarter* and examine the *sendMessageToMule* method. The URL passed as the first parameter to the *send* method called is to be MULE\_SERVICE1\_URL, as shown in this code snippet:

```
...
private MuleMessage sendMessageToMule(final MuleMessage inMessage,
    final MuleClient inMuleClient) throws Exception
{
    MuleMessage theReceivedMsg;

    /* Modify the URL of the service which to send the message here. */
    theReceivedMsg = inMuleClient.send(MULE_SERVICE1_URL, inMessage);
    return theReceivedMsg;
}
...
```

This URL is vm://receiverServiceURL, which, if you recall [the figure at the start of this chapter](#), is the endpoint exposed by the “receiverService” service.

- Right-click on the *Mule3ExceptionHandlingExampleStarter* class and select Run As -> Java Application.  
There will be some output on the console - we will look at the relevant parts one-by-one.

## Examine the Output

This first line indicates that the exception service was called and that the exception thrown from the service will contain the id 1 in its message.

```
*** In ExceptionService.onCall(): 1
```

Since the ReceiverFlow does not declare an exception strategy, the default exception strategy is used when an exception occurs trying to send a message to the service that always throws an exception. Note that:

- The exception stack is displayed.  
This is the hierarchy retrieved when calling *getCause* first on the exception and then on every subsequent object retrieved by the *getCause* method.
- The root exception stack trace is printed.  
The root exception is the exception that return null from the *getCause* method.

We see that this is the exception that was wrapped when thrown from the Exception service.

```
[02-09 16:32:16] ERROR DefaultMessagingExceptionStrategy [main]:*****
Message           : I am an outer exception with id 1
Code              : MULE_ERROR-10999
-----
Exception stack is:
1. I am a nested exception with id 1 (java.lang.Exception)
   com.ivan.services.ExceptionService:24 (null)
2. I am an outer exception with id 1 (org.mule.api.DefaultMuleException)
   com.ivan.services.ExceptionService:26
(HTTP://WWW.MULESOFT.ORG/DOCS/SITE/CURRENT3/APIDOCS/ORG/MULE/API/DefaultMuleException.html)
-----
Root Exception stack trace:
java.lang.Exception: I am a nested exception with id 1
  at com.ivan.services.ExceptionService.onCall(ExceptionService.java:24)
  at
org.mule.model.resolvers.CallableEntryPointResolver.invoke(CallableEntryPointResolver.java:50)
  at
org.mule.model.resolvers.DefaultEntryPointResolverSet.invoke(DefaultEntryPointResolverSet.java:39)
  + 3 more (set debug level logging or '-Dmule.verbose.exceptions=true' for everything)
*****
```

The following line tells us that the Hello service has received and processed a message. We can thus conclude that the first-successful message processor did its job; first it attempted to send the message to the exception service. When that failed, it sent the message to the Hello service.

```
*** In HelloService.onCall(): Hello. The time is now: Thu Feb 09 16:32:16 CET 2012
```

This output indicates that our starter-program received the response message from Mule and displays the contents of the response message. The Mule session identifier has been replaced with “...” to conserve space.

We can see that:

- The message has a payload (highlighted in yellow).  
The payload is the string that was produced by the Hello service.
- There is no exception payload.

```
*** Finished invoking Mule configuration!
Current time: 1328801536926
Message payload: Hello. The time is now: Thu Feb 09 16:32:16 CET 2012
Message properties:
  INBOUND: {MULE_SESSION=..., MULE_CORRELATION_SEQUENCE=-1,
            MULE_CORRELATION_GROUP_SIZE=-1, MULE_ENCODING=UTF-8}
  OUTBOUND: {MULE_SESSION=..., MULE_CORRELATION_SEQUENCE=-1,
```

```
MULE_CORRELATION_GROUP_SIZE=-1, MULE_ENCODING=UTF-8 }
Exception payload: null
NO EXCEPTION PAYLOAD AVAILABLE
```

## Sending Message to the Second Endpoint

Again, open the class *Mule3ExceptionHandlingExampleStarter* and examine the `sendMessageToMule` method. Change the URL passed as the first parameter to the `send` method called is to be `MULE_SERVICE2_URL`, as in this code snippet:

```
...
private MuleMessage sendMessageToMule(final MuleMessage inMessage,
    final MuleClient inMuleClient) throws Exception
{
    MuleMessage theReceivedMsg;

    /* Modify the URL of the service which to send the message here. */
    theReceivedMsg = inMuleClient.send(MULE_SERVICE2_URL, inMessage);
    return theReceivedMsg;
}
...
```

This URL is `vm://gotExceptionStrategyServiceURL`, which, if you remember [the figure at the start of this chapter](#), is the endpoint exposed by the “`getExceptionStrategyService`” service.

- Right-click the *Mule3ExceptionHandlingExampleStarter* class and select Run As -> Java Application.  
We will look at the relevant parts of the console output part-by-part.

## Examine the Output

The Mule session identifier has been replaced with “...” in all the subsequent output to conserve space.

This first line indicates that the exception service was called and that the exception thrown from the service will contain the id 1 in its message.

```
*** In ExceptionService.onCall(): 1
```

This section indicate that the method `doHandleException` in the custom exception listener implemented in the *MyMule3ExceptionListener* class received a notification. The message payload, message properties in the inbound and outbound scopes and exception payload are listed.

- The custom exception listener has the name “Listener 1”.  
Since there is only one single flow with a custom exception strategy, this is of less significance.
- The Mule message payload with its properties are exactly as at the time of the exception.  
We see that the message property with the name “exceptionListener” is not present in the message, which means that the message properties transformer has not yet been applied.
- The last endpoint that received the message is the `vm://gotExceptionStrategyServiceURL` endpoint, which can be seen in the `MULE_ENDPOINT` and `MULE_ORIGINATING_ENDPOINT` message properties.

```
*** MyMule3ExceptionListener.doHandleException: Listener 1
Current time: 1329151241026
Message payload: I am a Mule message!
Message properties:
    INBOUND: {MULE_SESSION=..., MULE_ENDPOINT=vm://gotExceptionStrategyServiceURL,
              MULE_ORIGINATING_ENDPOINT=endpoint.vm.gotExceptionStrategyServiceURL}
    OUTBOUND: {MULE_CORRELATION_SEQUENCE=-1, MULE_CORRELATION_GROUP_SIZE=-1,
               MULE_ENCODING=UTF-8}
Exception payload: null
```

```
NO EXCEPTION PAYLOAD AVAILABLE
```

The following log output is produced by Mule when routing the exception.

```
[02-13 17:40:41] ERROR MyMule3ExceptionListener [main]: Message being processed is: I am a Mule message!
```

This output shows that the logging service has received the message. Note that:

- The message payload is now an instance of *ExceptionMessage* (highlighted in yellow). This type of message wraps the message that caused the exception and holds information on the component in which the exception occurred (highlighted in green) and the endpoint that received the message immediately prior to the exception was thrown (highlighted in grey).
- There now is a message property with the name “exceptionListener” in the message's outbound scope (highlighted in blue). The property has the value “In GotExceptionStrategyFlow”. This tells us that the exception notification was received by the custom exception strategy defined in the flow “GotExceptionStrategyFlow”.

```
***** Logging service received a message:  
Current time: 1329151241050  
Message payload: ExceptionMessage{payload=I am a Mule message!,  
context={MULE_CORRELATION_SEQUENCE=-1, MULE_CORRELATION_GROUP_SIZE=-1,  
MULE_ENCODING=UTF-8}exception=org.mule.component.ComponentException: Component that  
caused exception is:  
DefaultJavaComponent{GotExceptionStrategyFlow.component.486001617}.  
Message payload is of type: String, componentName='GotExceptionStrategyFlow',  
endpointUri=vm://gotExceptionStrategyServiceURL, timeStamp=Mon Feb 13 17:40:41 CET 2012}  
Message properties:  
    INBOUND: {MULE_SESSION=..., MULE_CORRELATION_SEQUENCE=-1,  
              MULE_CORRELATION_GROUP_SIZE=1, MULE_ENCODING=UTF-8,  
              MULE_ENDPOINT=vm://errorLoggingServiceInbound,  
              MULE_ORIGINATING_ENDPOINT=endpoint.vm.errorLoggingServiceInbound,  
              exceptionListener=In GotExceptionStrategyFlow}  
    OUTBOUND: {MULE_CORRELATION_SEQUENCE=-1, MULE_CORRELATION_GROUP_SIZE=1,  
              MULE_ENCODING=UTF-8}  
Exception payload: null  
NO EXCEPTION PAYLOAD AVAILABLE
```

As with the console output we have seen previously, the final part of the output indicates that the starter-program received a response message from Mule. We can see that:

- The message has a null payload (highlighted in yellow).
- There is an exception payload from which we also can obtain an exception stacktrace. The exception payload type is highlighted in orange.
- The message property with the name “exceptionListener” is no longer in the message's outbound scope.

The original message has been replaced with a message with a null payload and an exception payload.

- There are messages, highlighted in green in the output below, in the exceptions. This indicate that the exception with id 1 caused the exception payload of the message received by the starter program.

We can also see that the outer and inner exceptions thrown from the exception service has retained their relation, with the outer exception wrapping the inner exception.

```
*** Finished invoking Mule configuration!  
Current time: 1329151241112  
Message payload: {NullPayload}  
Message properties:  
    INBOUND: {MULE_CORRELATION_SEQUENCE=-1, MULE_CORRELATION_GROUP_SIZE=-1,  
              MULE_ENCODING=UTF-8}  
    OUTBOUND: {MULE_SESSION=..., MULE_CORRELATION_SEQUENCE=-1,  
              MULE_CORRELATION_GROUP_SIZE=-1, MULE_ENCODING=UTF-8}  
Exception payload: org.mule.message.DefaultExceptionPayload@6c267f18  
*** EXCEPTION STACKTRACE START:
```

```

org.mule.component.ComponentException: Component that caused exception is:
DefaultJavaComponent{GotExceptionStrategyFlow.commponent.486001617}. Message payload is
of type: String
    at
org.mule.component.DefaultComponentLifecycleAdapter.invoke(DefaultComponentLifecycleAdapter.java:359)
    ...
    at
com.ivan.starter.Mule3ExceptionHandlingExampleStarter.sendMessageToMule(Mule3ExceptionHandlingExampleStarter.java:112)
    at
com.ivan.starter.Mule3ExceptionHandlingExampleStarter.doExample(Mule3ExceptionHandlingExampleStarter.java:57)
    at
com.ivan.starter.Mule3ExceptionHandlingExampleStarter.main(Mule3ExceptionHandlingExampleStarter.java:35)
Caused by: org.mule.api.DefaultMuleException: I am an outer exception with id 1
    at com.ivan.services.ExceptionService.onCall(ExceptionService.java:26)
    ... many more
Caused by: java.lang.Exception: I am a nested exception with id 1
    at com.ivan.services.ExceptionService.onCall(ExceptionService.java:24)
    ... many more
*** EXCEPTION STACKTRACE END.

```

We have observed that a flow will always have an exception strategy – if it is not explicitly defined, then it will be a default exception strategy.

We have also seen the different stages of the processing of an exception, with any exception listener receiving the unaltered message, as it was at the time when the exception occurred, and enrichment of the message before it was sent to a logging service by the exception handler.

In Mule 3.x, neither sender nor receiver of a message single-handedly decide the message exchange pattern.

## 9.10. Exercises

There are some differences regarding how Mule 2.x and Mule 3.x interpret message exchange patterns for endpoints. A suggested exercise is to experiment with different message exchange patterns on inbound- and outbound-endpoints and note the order in which the console output appears. Note whether the exchange-pattern in the sender and receiver must match in order for the service to be invoked.

Mule 2.x uses the *synchronous* attribute, while Mule 3.x has the *exchange-pattern* attribute.

## **10. Mule Programmatic Use and Message Properties**

Not only can we use Mule as an ESB that, after having routed, transformed and filtered a message, invokes a service we have developed. Mule can also be used as a building-block of applications where the application invokes different Mule configurations at different stages.

The following program shows how two different Mule configurations are programmatically combined:

- A message is sent to the first Mule configuration.
- The resulting message is used as input to the second Mule configuration.
- The message received as a result of invoking the second Mule configuration is programmatically examined and the result printed to the console.

In addition, we will also look at message properties and how their behaviour differ between Mule 2.x and Mule 3.x.

### **10.1. Introduction to Message Properties**

Message properties are additional information consisting of a key and a value that can be enclosed with Mule messages, like HTTP headers with HTTP requests. It is even the case that Mule message properties will be transformed into HTTP headers when a Mule message is send over the HTTP transport protocol.

In a Mule message there are five different scopes, as defined in the *PropertyScope* class:

- APPLICATION  
Read-only scope providing access to properties in the Mule registry.  
Not enabled by default. For more information, please refer to the Mule API documentation of the *MuleRegistry* interface.
- INBOUND  
Properties from client requests.
- INVOCATION  
Lasts during the processing of a service invocation. Typically only used internally by Mule.
- OUTBOUND  
Properties returned from client requests.
- SESSION  
Lasts during a session spanning multiple requests. If a transport protocol with session semantics is used, Mule will use this mechanism for session management; for example HTTP sessions. Otherwise an internal session mechanism will be used.

In Mule 2.x, Mule will not move properties between the different scopes. The opposite is true for Mule 3.x; properties will be moved between scopes by Mule. We will see examples of both kinds of behaviour in this chapter's example program.

## 10.2. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it “MuleProgrammaticUse”. The Mule 3 hot deployment can be switched off from the start, as this feature is not of use when running Mule embedded.

## 10.3. Create the Mule Configuration Files

This example contains two Mule configuration files per Mule version, for a total of four Mule configuration files.

The two configuration files for a Mule version are quite similar – the only differences are the values stored in the message property and the data appended to the message payload. Thus only one of the configuration files will be commented upon.

All the Mule configuration files reside in one and the same package in the source directory:

- Create the package *com.ivan.muleconfig* in the root of the project's source directory.

### Mule 2.x Configuration Files

The first Mule 2.x configuration file is named “mule2-config.xml” and is located in the package *com.ivan.muleconfig* created earlier.

```
<?xml version="1.0" encoding="UTF-8"?>
<mule
    xmlns="http://www.mulesource.org/schema/mule/core/2.2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
    xsi:schemaLocation="
        http://www.mulesource.org/schema/mule/core/2.2
        http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

        http://www.mulesource.org/schema/mule/vm/2.2
        http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">

    <model name="myMuleModel">
        <service name="myMuleService">
            <inbound>
                <inbound-endpoint address="vm://myMuleServiceURL">
                    <message-properties-transformer>
                        <!--
                            The expression in the value attribute is used to
                            append the "[set by config1]" string to the current
                            value of the message property with the name
                            "MuleProperty2".
                        -->
                    <add-message-property key="MuleProperty2"
                        value="#[string:#[header:MuleProperty2]#[string:[set by
config1]]]"/>
                    </message-properties-transformer>
                    <!-- Append a string to the payload of the message. -->
                    <append-string-transformer
                        message="[string appended by first config]"/>
                </inbound-endpoint>
            </inbound>
        </service>
    </model>
</mule>
```

Note that:

- The configuration file contains one module with a single service.
- The *address* attribute of the service's inbound endpoint is “vm://myMuleServiceURL”. This is the URL we will later use in our program to send messages that are to be processed by the service defined in this configuration file.
- The <vm:inbound-endpoint> element may be used instead of the <inbound-endpoint> element with the same functionality.
- The <inbound-endpoint> element contains a <message-properties-transformer> element. As seen in an earlier example, this transformer is used to manipulate message properties.
- The <message-properties-transformer> does not specify a scope on which to operate. The Mule 2.2.1 message properties transformer only operates on the OUTBOUND scope, regardless of how it is configured.
- In the <message-properties-transformer> element, there is a <add-message-property> element that adds a property with the name “MuleProperty2” to the properties of a message.
- The *value* attribute of the <add-message-property> element contains a long expression that should be interpreted as follows:  
Cast the following as a string: the value of the message header property named “MuleProperty2” followed by cast the following as a string: “[set by config1]”.
- Finally, the <inbound-endpoint> element contains a <append-string-transformer> element. As the name of the element indicate, this is a message transformer that appends a string. The string is appended to the payload of messages passing through the transformer.

The second Mule 2.x configuration file is named “mule2-config2.xml” and is also located in the *com.ivan.muleconfig* package. The configuration is identical to the first Mule 2.x configuration file except for some values. In addition, comments have been left out.

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:vm="http://www.mulesource.org/schema/mule/vm/2.2"
      xsi:schemaLocation="
          http://www.mulesource.org/schema/mule/core/2.2
          http://www.mulesource.org/schema/mule/core/2.2/mule.xsd

          http://www.mulesource.org/schema/mule/vm/2.2
          http://www.mulesource.org/schema/mule/vm/2.2/mule-vm.xsd">

    <model name="myMuleModel">
        <service name="myMuleService">
            <inbound>
                <inbound-endpoint address="vm://myMuleServiceURL">
                    <message-properties-transformer>
                        <add-message-property
                            key="MuleProperty2"
                            value="#[string:#[header:MuleProperty2]#[string:[set by
config2]]]" />
                    </message-properties-transformer>
                    <append-string-transformer
                        message="[string appended by second config]" />
                </inbound-endpoint>
            </inbound>
        </service>
    </model>
</mule>
```

## Mule 3.x Configuration Files

In the Mule 3.x configuration files a flow is used instead of the model-service construct seen in the Mule 2.x configuration files.

The first Mule 3.x configuration file is named “mule3-config.xml” and is located in the package `com.ivan.muleconfig`:

```
<?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:vm="http://www.mulesoft.org/schema/mule/vm"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

          http://www.mulesoft.org/schema/mule/vm
          http://www.mulesoft.org/schema/mule/vm/3.2/mule-vm.xsd">

<flow name="myMuleFlow">
    <!--
        In Mule 3.2 the exchange-pattern must explicitly be set to
        request-response, since the default is one-way.
        If the one-way exchange pattern is used, null messages will
        be received as "response".
    -->
    <inbound-endpoint
        address="vm://myMuleServiceURL"
        exchange-pattern="request-response">
        <!--
            Note how a scope, in which message properties are
            manipulated, can be specified on the message
            properties transformer.
            The default scope is the outbound scope.
        -->
        <message-properties-transformer scope="outbound">
            <!--
                The expression in the value attribute is used to
                append the "[set by config]" string to the current
                value of the message property with the name
                "MuleProperty2".
                Note how a message property scope now can be specified
                in the expression retrieving the property value.
                The default scope, if none is supplied, is OUTBOUND.
            -->
            <add-message-property
                key="MuleProperty2"
                value="#[string:#[header:INBOUND:MuleProperty2]#[string:[set by
config]]]" />
            </message-properties-transformer>
            <!-- Append a string to the payload of the message. -->
            <append-string-transformer
                message="[string appended by first config]" />
        </inbound-endpoint>
        <echo-component/>
    </flow>
</mule>
```

Note that:

- The `<flow>` element contains an `<inbound-endpoint>` element which `address` attribute has the value “`vm://myMuleServiceURL`”.  
This is the URL we will later use in our program to send messages that are to be processed by the flow defined in this configuration file.
- The `<inbound-endpoint>` element has a property named `exchange-pattern` with the value “`request-response`”.  
Per default, inbound endpoints in Mule 3.2 are one-way, that is, will not return any response. If we do not modify the exchange pattern, we will receive a null message as response. The default value may vary between different versions of Mule, so when in doubt,

specify a value.

- The <inbound-endpoint> element contains a <message-properties-transformer> element. As seen in an earlier example, this transformer is used to manipulate message properties.
- The <vm:inbound-endpoint> element may be used instead of the <inbound-endpoint> element with the same functionality.
- The <message-properties-transformer> has a *scope* attribute with the value “outbound”. As opposed to the Mule 2.x message properties transformer, the Mule 3.x version is scope-aware. Using the *scope* attribute, we can specify the message properties scope on which the elements contained in the transformer operates on.  
The default scope is OUTBOUND scope.
- In the <message-properties-transformer> element, there is a <add-message-property> element that adds a property with the name “MuleProperty2” to the properties of a message.
- The *value* attribute of the <add-message-property> element contains a long expression that should be interpreted as follows:  
Cast the following as a string: the value of the message header property named “MuleProperty2” from the INBOUND scope followed by cast the following as a string: “[set by config1]”.
- The expression in the *value* attribute of the <add-message-property> specifies from which message property scope the value should be read. This is new for Mule 3.x.  
The default scope is OUTBOUND scope.
- Finally, the <inbound-endpoint> element contains a <append-string-transformer> element. As the name of the element indicate, this is a message transformer that appends a string. The string is appended to the payload of messages passing through the transformer.

The second Mule 3.x configuration file is named “mule3-config2.xml” and is also located in the *com.ivan.muleconfig* package. The configuration is identical to the first Mule 3.x configuration file except for some values appended to the message property and the payload. In addition, comments have been left out.

```
<?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:vm="http://www.mulesoft.org/schema/mule/vm"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

          http://www.mulesoft.org/schema/mule/vm
          http://www.mulesoft.org/schema/mule/vm/3.2/mule-vm.xsd">

    <flow name="myMuleFlow">
        <inbound-endpoint
            address="vm://myMuleServiceURL" exchange-pattern="request-response">
            <message-properties-transformer>
                <add-message-property
                    key="MuleProperty2"
                    value="#[string:#[header:INBOUND:MuleProperty2]#[string:[set by
config2]]]" />
                </message-properties-transformer>
                <append-string-transformer
                    message="[string appended by second config]" />
            </inbound-endpoint>
            <echo-component/>
        </flow>
    </mule>
```

## 10.4. Create the Starter Classes

This chapter's example program will use two classes, one for each version of Mule, that starts the two configurations up and sends a message to the first one and then to the second one.

As we will see, there are some significant differences between the two starter classes related to how message properties are handled in Mule 2.x and Mule 3.x.

First we'll create a package for the starter classes:

- Create the package `com.ivan.starter` in the root of the project's source directory.

### Create the Mule 2.x Starter Class

The implementation of the Mule 2.x starter class is quite straightforward.

- Create a class named `MuleProgrammaticUseStarter2` implemented as this:

```
package com.ivan.starter;

import org.mule.DefaultMuleMessage;
import org.mule.api.MuleContext;
import org.mule.api.MuleException;
import org.mule.api.MuleMessage;
import org.mule.api.transport.PropertyScope;
import org.mule.config.spring.SpringXmlConfigurationBuilder;
import org.mule.context.DefaultMuleContextFactory;
import org.mule.module.client.MuleClient;

/**
 * This class invokes two Mule configurations in sequence,
 * passing the result of the first invocation as input to the
 * second.
 * Version for Mule 2.x.
 *
 * @author Ivan Krizsan
 */
public class MuleProgrammaticUseStarter2
{
    /* Constant(s): */
    private final static String MULE_CONFIG_FILE1 =
        "com/ivan/muleconfig/mule2-config.xml";
    private final static String MULE_CONFIG_FILE2 =
        "com/ivan/muleconfig/mule2-config2.xml";
    private final static String MULE_SERVICE_URL = "vm://myMuleServiceURL";
    private final static String MESSAGE_PROPERTY_NAME = "MuleProperty2";

    /* Instance variable(s): */

    public static void main(String[] args) throws Exception
    {
        (new MuleProgrammaticUseStarter2()).doExample();
    }

    private void doExample()
    {
        MuleContext theContext1 = null;
        MuleContext theContext2 = null;
        try
        {
            MuleClient theMuleClient;
            /*
             * Create a new message, setting its payload and one
             * message property.
             * The message property is placed in the OUTBOUND scope.
             */
            MuleMessage theMuleMessage = new DefaultMuleMessage(
                "[original message payload]");
            theMuleMessage.setProperty(MESSAGE_PROPERTY_NAME,
                "[Set by program]", PropertyScope.OUTBOUND);

            /* Start Mule context with the first configuration file. */
        }
    }
}
```

```

theContext1 = startMule(MULE_CONFIG_FILE1);

/*
 * Examine the message before the message has been processed
 * by the first Mule configuration.
 * This is done after having started the Mule server, in order
 * for a Mule server context to be present.
 */
System.out.println("***** Before first invocation:");
logMessageContents(theMuleMessage);

/* Create a Mule client and send the message to it. */
theMuleClient = createMuleClient(theContext1);
theMuleMessage = sendMessageToMule(theMuleMessage, theMuleClient);

/*
 * Examine the message after the message has been processed
 * by the first Mule configuration.
 */
System.out.println("***** After first invocation:");
logMessageContents(theMuleMessage);

/*
 * Start a new Mule context with the second configuration file,
 * create a client and send the message received from the
 * above invocation to it.
 * No need to move message properties between scopes with
 * Mule 2.x.
 */
theContext2 = startMule(MULE_CONFIG_FILE2);
theMuleClient = createMuleClient(theContext2);
theMuleMessage = sendMessageToMule(theMuleMessage, theMuleClient);

/*
 * Examine the message after the message has been processed
 * by the second Mule configuration.
 */
System.out.println("***** After second invocation:");
logMessageContents(theMuleMessage);
} catch (final Exception theException)
{
    theException.printStackTrace();
} finally
{
    /*
     * Dispose of the Mule contexts. Disposing the context
     * automatically stops it and we do not have to bother
     * with the exception declared by the stop method.
     * Must dispose if we wish the application to terminate,
     * otherwise there will be unreleased resources.
     */
    if (theContext1 != null)
    {
        theContext1.dispose();
    }
    if (theContext2 != null)
    {
        theContext2.dispose();
    }
}
}

private MuleContext startMule(final String inMuleConfigFile) throws MuleException
{
    /*
     * Starts an instance of Mule configured according to the supplied
     * configuration file.
     */
    String[] theConfigFiles = { inMuleConfigFile };

    DefaultMuleContextFactory theContextFactory =
        new DefaultMuleContextFactory();
    SpringXmlConfigurationBuilder theConfigBuilder =
        new SpringXmlConfigurationBuilder(theConfigFiles);
    MuleContext theMuleContext =
        theContextFactory.createMuleContext(theConfigBuilder);
}

```

```

        theMuleContext.start();

        return theMuleContext;
    }

    private MuleClient createMuleClient(final MuleContext inMuleContext)
        throws MuleException
    {
        MuleClient theMuleClient = new MuleClient(inMuleContext);
        return theMuleClient;
    }

    private MuleMessage sendMessageToMule(final MuleMessage inMessage,
        final MuleClient inMuleClient) throws Exception
    {
        MuleMessage theReceivedMsg;

        theReceivedMsg = inMuleClient.send(MULE_SERVICE_URL, inMessage);
        return theReceivedMsg;
    }

    private void logMessageContents(final MuleMessage inMessage)
        throws Exception {
        /*
         * The regular approach to programmatically retrieving message
         * properties with Mule 2.x is:
         * theMsgPropertyValue = theMuleMessage.getStringProperty(
         *     MESSAGE_PROPERTY_NAME, "default value");
         * This would search for the particular property in all the
         * Mule scopes.
         * However, to stress the difference with Mule 3.x, the code
         * below is used which includes specifying a property scope.
         */
        String theMsgPropertyValue = (String)inMessage.getProperty(
            MESSAGE_PROPERTY_NAME, PropertyScope.INBOUND);
        System.out.println("    INBOUND property value: " +
            theMsgPropertyValue);
        theMsgPropertyValue = (String)inMessage.getProperty(
            MESSAGE_PROPERTY_NAME, PropertyScope.OUTBOUND);
        System.out.println("    OUTBOUND property value: " +
            theMsgPropertyValue);

        String theMsgPayload = inMessage.getPayloadAsString();
        System.out.println("    Message payload: " + theMsgPayload);
    }
}

```

Note that:

- The constant string `MULE_SERVICE_URL` that holds the address of the Mule service to which we send messages has the prefix “`vm://`”.  
This prefix tells Mule that it should use the VM transport, which is a transport used within one and the same JVM. Omitting this transport prefix will cause an error.
- In the `doExample` method, there are two variables of the type `MuleContext`.  
For each Mule 2.x configuration we create one Mule instance, represented by a Mule context.
- In the `doExample` method, a Mule message is created with only a message payload, a string, as parameter.  
As we will see later, a `MuleContext` object must be supplied when creating messages when using Mule 3.x. Using Mule 2.x, no Mule context need to be supplied when creating messages.
- The message property with the name “`MuleProperty2`” is set to the value “[Set by program]”.
- A Mule instance is started using the first Mule configuration file.

Given the name of a configuration file, the *startMule* method is responsible for starting a Mule instance configured according to the configuration file.

- The contents of the Mule message is logged to the console before it is being sent to Mule for processing.

- A *MuleClient* client object is created for the first Mule context.

Mule clients are used to send and receive messages from a Mule instance. Multiple clients may be created for one context, allowing multiple threads to interact with one Mule instance concurrently.

Given a Mule context, the *createMuleClient* method is responsible for creating a Mule client for the supplied context.

- The Mule message is sent to the first Mule instance using the Mule client just created.

When sending a message, an URL must be supplied. This URL specifies the transport and the destination of the message.

- Each message sent to a Mule client is processed in a session of its own.

This allows a client to send more messages to a Mule instance while previously sent messages are being processed.

This is handled internally by Mule and is not made apparent by the code of this example. Please refer to the *MuleSession* interface in the Mule API documentation for information.

- The URL to which the message is sent, “vm://myMuleServiceURL”, matches the value of the address attribute in the <inbound-endpoint> element in the Mule configuration file.

- The contents of the Mule message received as result of sending the first Mule message to the first Mule instance is logged to the console.

- A second Mule instance is started configured according to the second configuration file. Again, this is accomplished using the *startMule* method.

- Another Mule client is created, in order to send and receive messages to/from the second Mule instance.

- In the same manner as the message was sent to the first Mule instance, the message is sent to the second Mule instance.

The URL is the same as when sending the message to the first instance. This is because the inbound endpoint address in the two Mule configuration files is the same.

- The Mule message is logged to the console.

- The Mule contexts representing the first and second Mule instance are disposed.

Stopping a Mule context is not enough, if we want the application to terminate properly when our starter class has finished executing.

Disposing a Mule context first stops it and then releases the resources held by the Mule instance.

## Create the Mule 3.x Starter Class

The implementation of the Mule 3.x starter class is not as straightforward as the Mule 2.x starter class; message properties need to be moved to the proper scope since Mule 3.x will move message properties behind scopes. Details of this behaviour will be observed when we run the different versions of the example program.

- Create a class named *MuleProgrammaticUseStarter3* implemented as this:

```
package com.ivan.starter;

import org.mule.DefaultMuleMessage;
import org.mule.api.MuleContext;
import org.mule.api.MuleException;
import org.mule.api.MuleMessage;
import org.mule.api.config.ConfigurationBuilder;
import org.mule.api.context.MuleContextBuilder;
import org.mule.api.context.MuleContextFactory;
import org.mule.api.transport.PropertyScope;
import org.mule.config.spring.SpringXmlConfigurationBuilder;
import org.mule.context.DefaultMuleContextBuilder;
import org.mule.context.DefaultMuleContextFactory;
import org.mule.module.client.MuleClient;

/**
 * This class invokes two Mule configurations in sequence,
 * passing the result of the first invocation as input to the
 * second.
 * Version for Mule 3.x.
 *
 * @author Ivan Krizsan
 */
public class MuleProgrammaticUseStarter3
{
    /* Constant(s): */
    private final static String MULE_CONFIG_FILE1 =
        "com/ivan/muleconfig/mule3-config.xml";
    private final static String MULE_CONFIG_FILE2 =
        "com/ivan/muleconfig/mule3-config2.xml";
    private final static String MULE_SERVICE_URL = "vm://myMuleServiceURL";
    private final static String MESSAGE_PROPERTY_NAME = "MuleProperty2";

    /* Instance variable(s): */

    public static void main(String[] args) throws Exception
    {
        (new MuleProgrammaticUseStarter3()).doExample();
    }

    private void doExample()
    {
        MuleContext theContext1 = null;
        MuleContext theContext2 = null;
        try
        {
            /*
             * Start the first Mule context with the first configuration
             * file. Need to do this, since the context will be used
             * when creating a new message.
             */
            theContext1 = startMule(MULE_CONFIG_FILE1);

            /*
             * Create the message to send to the two different Mule
             * configurations.
             * Set both message payload and a message property.
             */
            MuleMessage theMuleMessage = new DefaultMuleMessage(
                "[original message payload]", theContext1);
            theMuleMessage.setProperty(MESSAGE_PROPERTY_NAME,
                "[Set by program]", PropertyScope.OUTBOUND);

            /*

```

```

        * Examine the message before the message has been processed
        * by the first Mule configuration.
        */
System.out.println("***** Before first invocation:");
logMessageContents(theMuleMessage);

/* Create a client and send the message to it. */
MuleClient theMuleClient = createMuleClient(theContext1);
theMuleMessage = sendMessageToMule(theMuleMessage, theMuleClient);

/*
 * Examine the message after the message has been processed
 * by the first Mule configuration.
 */
System.out.println("***** After first invocation:");
logMessageContents(theMuleMessage);

/*
 * Copy the message property to the outbound scope, or else
 * it will be cleared.
 * Clear the INBOUND scope to make presentation of message
 * look more clear.
 */
String theMsgPropertyValue = theMuleMessage.getProperty(
    MESSAGE_PROPERTY_NAME, PropertyScope.INBOUND);
theMuleMessage.setProperty(MESSAGE_PROPERTY_NAME,
    theMsgPropertyValue, PropertyScope.OUTBOUND);
theMuleMessage.clearProperties(PropertyScope.INBOUND);

/*
 * Examine the message before the message has been processed
 * by the second Mule configuration.
 */
System.out.println("***** Before second invocation:");
logMessageContents(theMuleMessage);

/*
 * Start a new Mule context with the second configuration file,
 * create a client and send the message received from the
 * above invocation to it.
 */
theContext2 = startMule(MULE_CONFIG_FILE2);
theMuleClient = createMuleClient(theContext2);
theMuleMessage = sendMessageToMule(theMuleMessage, theMuleClient);

/*
 * Examine the message after the message has been processed
 * by the second Mule configuration.
 */
System.out.println("***** After second invocation:");
logMessageContents(theMuleMessage);
} catch (final Exception theException)
{
    theException.printStackTrace();
} finally
{
    /*
     * Dispose of the Mule contexts. Disposing the context
     * automatically stops it and we do not have to bother
     * with the exception declared by the stop method.
     * Must dispose if we wish the application to terminate,
     * otherwise there will be unreleased resources.
     */
    if (theContext1 != null)
    {
        theContext1.dispose();
    }
    if (theContext2 != null)
    {
        theContext2.dispose();
    }
}
}

private MuleContext startMule(final String inMuleConfigFile) throws MuleException
{

```

```

/*
 * Starts an instance of Mule configured according to the supplied
 * configuration file.
 */
String[] theConfigFiles = { inMuleConfigFile };

MuleContextFactory theContextFactory = new DefaultMuleContextFactory();
ConfigurationBuilder theConfigBuilder =
    new SpringXmlConfigurationBuilder(theConfigFiles);
MuleContextBuilder theContextBuilder = new DefaultMuleContextBuilder();
MuleContext theMuleContext = theContextFactory.createMuleContext(
    theConfigBuilder, theContextBuilder);
theMuleContext.start();

return theMuleContext;
}

private MuleClient createMuleClient(final MuleContext inMuleContext)
throws MuleException
{
    MuleClient theMuleClient = new MuleClient(inMuleContext);
    return theMuleClient;
}

private MuleMessage sendMessageToMule(final MuleMessage inMessage,
    final MuleClient inMuleClient) throws Exception
{
    MuleMessage theReceivedMsg;
    theReceivedMsg = inMuleClient.send(MULE_SERVICE_URL, inMessage);
    return theReceivedMsg;
}

private void logMessageContents(final MuleMessage inMessage)
throws Exception {
/*
 * Note how a message scope is supplied when retrieving
 * message properties.
 * Property retrieval methods which do not require a scope
 * to be specified have been deprecated in Mule 3.x.
 */
String theMsgPropertyValue = (String)inMessage.getProperty(
    MESSAGE_PROPERTY_NAME, PropertyScope.INBOUND);
System.out.println("    INBOUND property value: " +
    theMsgPropertyValue);
theMsgPropertyValue = (String)inMessage.getProperty(
    MESSAGE_PROPERTY_NAME, PropertyScope.OUTBOUND);
System.out.println("    OUTBOUND property value: " +
    theMsgPropertyValue);

String theMsgPayload = inMessage.getPayloadAsString();
System.out.println("    Message payload: " + theMsgPayload);
}
}

```

Note that:

- The constant string `MULE_SERVICE_URL` that holds the address of the Mule service to which we send messages has the prefix “`vm://`”. This prefix tells Mule that it should use the VM transport, which is a transport used within one and the same JVM. Omitting this transport prefix will cause an error.
- In the `doExample` method, there are two variables of the type `MuleContext`. For each Mule 3.x configuration we create one Mule instance, represented by a Mule context.
- In the `doExample` method, a Mule instance is started using the first Mule configuration file. Given the name of a configuration file, the `startMule` method is responsible for starting a Mule instance configured according to the configuration file.
- Next, a Mule message is created with a message payload, a string, and a Mule context as

parameters.

In Mule 3.x, a *MuleContext* object must always be supplied when creating messages.

- The message property with the name “MuleProperty2” is set to the value “[Set by program]”.
- The contents of the Mule message is logged to the console before it is being sent to Mule for processing.
- A *MuleClient* client object is created for the first Mule context.

Mule clients are used to send and receive messages from a Mule instance. Multiple clients may be created for one context, allowing multiple threads to interact with one Mule instance concurrently.

Given a Mule context, the *createMuleClient* method is responsible for creating a Mule client for the supplied context.

- The Mule message is sent to the first Mule instance using the Mule client just created. When sending a message, an URL must be supplied. This URL specifies the transport and the destination of the message.
- The URL to which the message is sent, “vm://myMuleServiceURL”, matches the value of the address attribute in the <inbound-endpoint> element in the Mule configuration file.
- As with Mule 2.x, each message sent to a Mule client is processed in a session of its own. This allows a client to send more messages to a Mule instance while previously sent messages are being processed.

This is handled internally by Mule and is not made apparent by the code of this example. Please refer to the *MuleSession* interface in the Mule API documentation for information.

- Immediately after having received the response message after having invoked the first Mule instance, the message contents is logged to the console.

As before, Mule 3.x moves message properties so more logging was implemented, in order to make this visible.

- The “MuleProperty2” message property of the response message is copied from the INBOUND message scope to the OUTBOUND message scope.
- The INBOUND message scope in the response message is cleared. The INBOUND message scope will be cleared when the message is sent to a Mule 3.x instance, so we clear it now to make the program output more clear.
- The contents of the Mule message received as result of sending the first Mule message to the first Mule instance is logged to the console.

- A second Mule instance is started configured according to the second configuration file. Again, this is accomplished using the *startMule* method.

- Another Mule client is created, in order to send and receive messages to/from the second Mule instance.

- In the same manner as the message was sent to the first Mule instance, the message is sent to the second Mule instance.

The URL is the same as when sending the message to the first instance. This is because the inbound endpoint address in the two Mule configuration files is the same.

- The Mule message is logged to the console.
- The Mule contexts representing the first and second Mule instance are disposed.

Stopping a Mule context is not enough, if we want the application to terminate properly when our starter class has finished executing.

Disposing a Mule context first stops it and then releases the resources held by the Mule instance.

Developing the two starter classes has, besides showing how to use Mule programmatically, given us a feeling that Mule 2.x and Mule 3.x handles message properties differently.

Using Mule 2.x and 3.x programmatically is quite similar – the only difference this chapter's example program makes obvious is that with Mule 3.x, a Mule context must be supplied when creating a Mule message.

## 10.5. Run the Example Program

Now it is time to run the Mule 2.x and Mule 3.x versions of the example program and compare the output. The focus is on Mule message properties and message scopes.

Note that our example program only logs one particular message property, not all the message properties in a certain scope.

Remember that we are to run the starter classes of the example program, NOT the Mule configuration files!

If there are any problems running the starter classes, clean and rebuild the project in Eclipse as a first attempt at a remedy.

### ***Run the Mule 2.x Version of the Example Program***

Before running the Mule 2.x version of the starter class, please assure that the project is indeed configured with the Mule 2.x libraries on the classpath, as described in [appendix B](#).

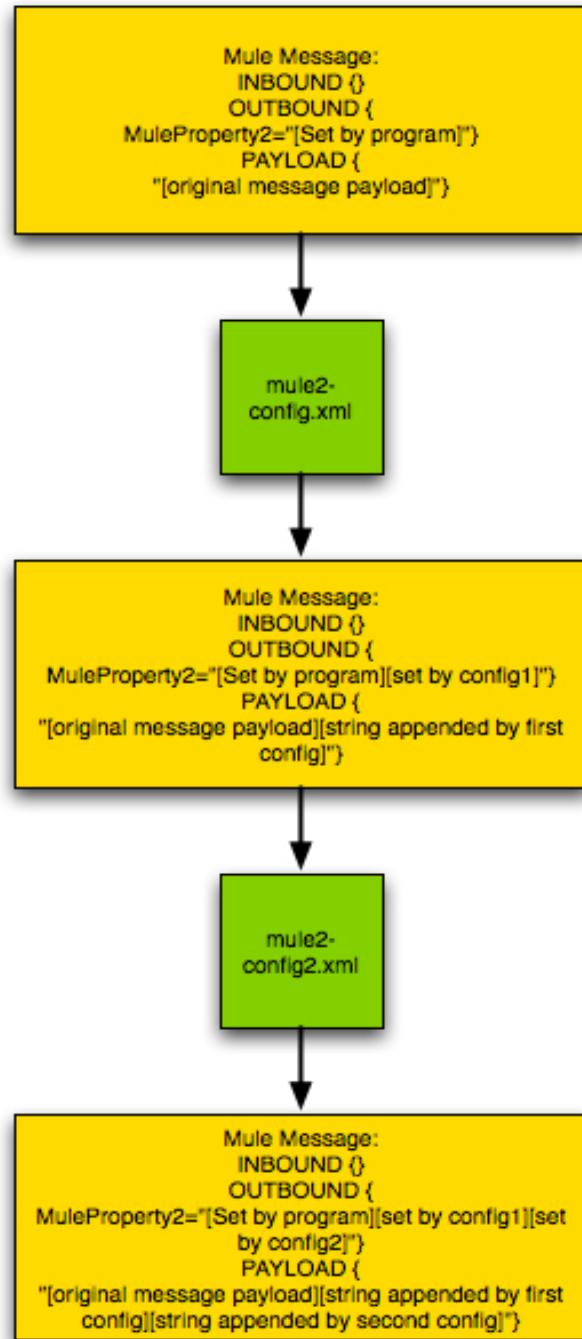
- In the Package Explorer in Eclipse, right-click the “MuleProgrammaticUseStarter2” class and select Run As -> Java Application.
- The application should produce the following console output and then terminate:

```
***** Before first invocation:  
INBOUND property value: null  
OUTBOUND property value: [Set by program]  
Message payload: [original message payload]  
***** After first invocation:  
INBOUND property value: null  
OUTBOUND property value: [Set by program][set by config1]  
Message payload: [original message payload][string appended by first config]  
***** After second invocation:  
INBOUND property value: null  
OUTBOUND property value: [Set by program][set by config1][set by config2]  
Message payload: [original message payload][string appended by first config][string appended by second config]
```

Note that:

- The property value in the OUTBOUND message properties scope is appended by each Mule configuration.  
When trying to retrieve the value of the “MuleProperty2” message property from the INBOUND scope, the result is always null.  
Recall the Mule 2.x configuration files we developed [earlier](#). When appending the string to the value of our message property, no message scope was specified, neither in the expression (not supported in Mule 2.x) nor in the <message-properties-transformer>. Thus the scope used is always the OUTBOUND message property scope.
- The message payload is appended by each Mule configuration.

The following picture visualizes the contents of the Mule message passing through the two Mule 2.x instances:



Content of a Mule message passing through the two Mule 2.x instances of the example program.

We can conclude that message properties are not moved by Mule 2.x and that our message property is placed in the OUTBOUND scope and remains there for the duration of the entire execution of the example program.

## **Running the Mule 3.x Version of the Example Program**

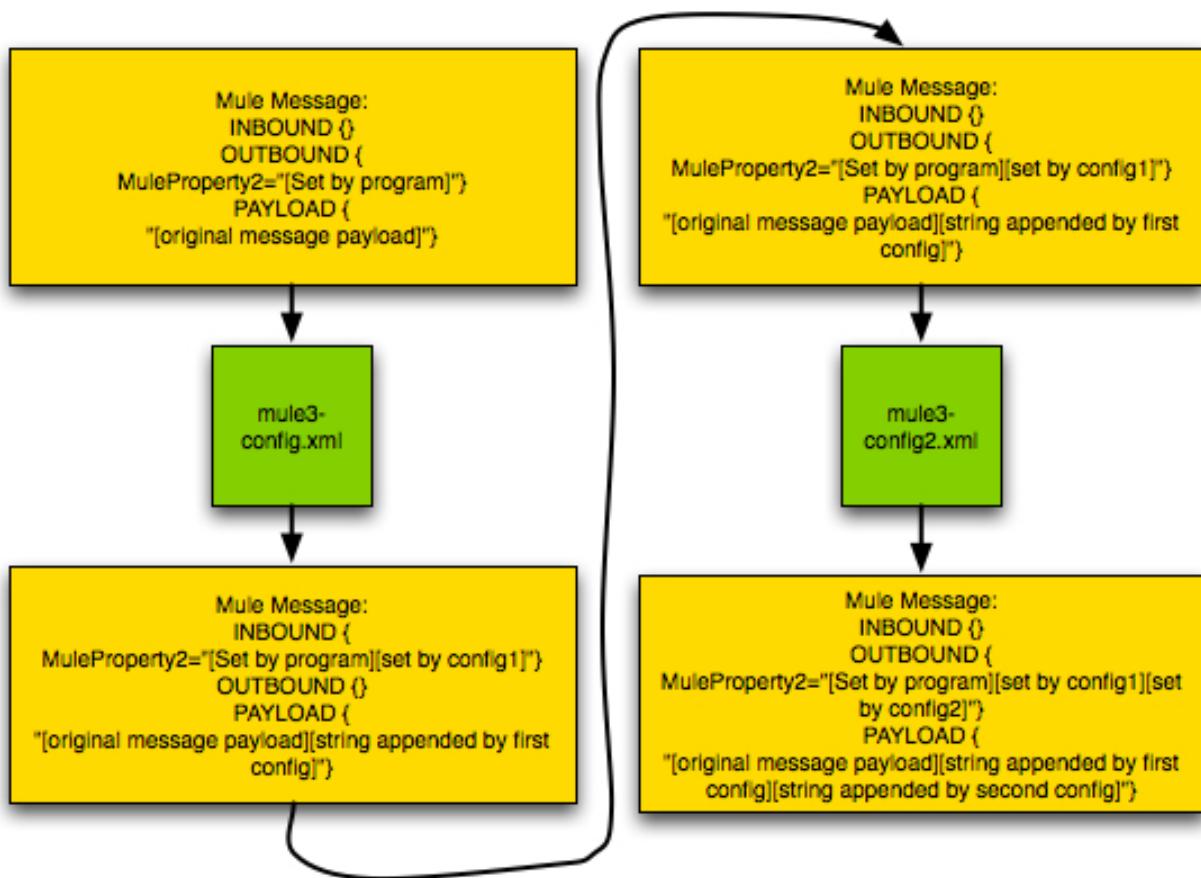
- Configure the “MuleProgrammaticUse” Eclipse project so that it has the Mule 3.x libraries on the classpath, according to the procedure described in [appendix B](#).
- In the Package Explorer in Eclipse, right-click the “MuleProgrammaticUseStarter3” class and select Run As -> Java Application.
- The application should produce the following console output and then terminate (log produced by Mule has been removed for sake of clarity):

```
***** Before first invocation:  
INBOUND property value: null  
OUTBOUND property value: [Set by program]  
Message payload: [original message payload]  
***** After first invocation:  
INBOUND property value: [Set by program][set by config1]  
OUTBOUND property value: null  
Message payload: [original message payload][string appended by first config]  
***** Before second invocation:  
INBOUND property value: null  
OUTBOUND property value: [Set by program][set by config1]  
Message payload: [original message payload][string appended by first config]  
***** After second invocation:  
INBOUND property value: [Set by program][set by config1][set by config2]  
OUTBOUND property value: null  
Message payload: [original message payload][string appended by first config][string  
appended by second config]
```

Note that:

- Before sending the message to the first Mule instance, our message property was found in the OUTBOUND message scope.
- In the response message from the first Mule instance, our message property was found in the INBOUND message scope.  
Thus, Mule 3.x has moved the message property from the OUTBOUND scope to the INBOUND scope.  
So when is the message property moved? If we look at the Mule 3.x configuration file developed [earlier](#), we can see that the message property is retrieved from the INBOUND scope, processed by appending a string and then placed in the OUTBOUND scope.  
However, when the example program prints the message contents after having invoked the first Mule instance, the message property is found in the INBOUND scope.  
Thus Mule 3.x moves the message property twice; once before the message is processed by the configuration file and once after the message has been processed by the configuration file.
- After the first invocation, the message property is found in the INBOUND message scope, but before sending the message to the second Mule 3.x instance, the message property is once again in the OUTBOUND scope.  
Recall that the Mule 3.x starter class developed [earlier](#) contained code that moved the message property from the INBOUND to the OUTBOUND message scope between the first and second invocation.
- The message payload is appended by each Mule configuration in the same way as seen in the Mule 2.x version of the example program.

The following picture visualizes the contents of the Mule message passing through the two Mule 3.x instances:



Content of a Mule message passing through the two Mule 3.x instances of the example program.

We can conclude that Mule 3.x behaves in the following way concerning moving message properties between the different scopes:

- When sending a message to a Mule configuration, the message properties in the OUTBOUND scope are moved to the INBOUND scope.
- Message properties that are to be made available to the client sending the message are placed in the OUTBOUND scope by the Mule configuration.
- Having finished executing the Mule configuration, the message properties of the response message are moved from the OUTBOUND scope to the INBOUND scope.
- Before moving message properties from one scope to another, the target scope is cleared and all information in it lost.

## 10.6. Exercises

You are suggested to experiment with Mule message properties and message property scopes. See what is available at the different stages during the lifetime of a Mule message.

For the advanced, tracing into the Mule code and examining how Mule message properties are processed can be an interesting exercise.

## 11. Testing Mule Configurations

In this chapter we will look at how to test a scenario which involves a single Mule configuration file and one service component implemented in Java. The Mule literature calls this kind of tests functional tests.

It is possible to test individual service components, transformers, filters etc using unit tests. Such testing is considered to be regular test-driven development and will not be discussed in this book.

The configuration we are to test in both the Mule 2.x and Mule 3.x versions of this chapter's example program consists of a single HTTP endpoint that receives requests. If a request contains the parameter with the name "data" and the value "123" in the URL, it is considered to be a valid request and an ACK response message will be given. Otherwise the request is considered invalid and will yield a NACK response message.

### 11.1. Create the Project

Create the project as described in the appendix [Create a Mule Project](#), naming it "MuleUnitTesting". The Mule 3 hot deployment can be switched off from the start, as this feature will not be used.

### 11.2. Create the Tests

In the spirit of test-driven development, we create the tests first. The tests implementations consists of one class implementing common test code for both the Mule 2.x and 3.x versions of the example, as well as one Mule 2.x specific and one Mule 3.x specific test case.

#### Create the Common Test Class

The common test class consists of the two tests that we apply to the validation service; one test that issues a request containing the sought-after parameter with the specific value and another test that issue a request that does contain the sought-after parameter, but with another value.

The reason in this example for extracting the tests to this common class is just to avoid blatant code duplication.

- In the package *com.ivan.tests*, implement the *CommonValidatorTestCase* class like shown in the following listing:

```
package com.ivan.tests;

import java.util.HashMap;
import java.util.Map;

import junit.framework.Assert;

import org.mule.api.MuleException;
import org.mule.api.MuleMessage;
import org.mule.module.client.MuleClient;
import org.mule.tck.FunctionalTestCase;

/**
 * Implements common code used to test the validation service
 * for both Mule 2.x and Mule 3.x.
 *
 * @author Ivan Krizsan
 */
public abstract class CommonValidatorTestCase extends FunctionalTestCase
{
    /**
     * Tests sending a good request to the validator service and ensure
     * that we get the proper response.
     */
}
```

```

    * @throws Exception If error occurs during test.
    * Indicates test failure.
    */
    public void testGoodRequest() throws Exception
    {
        MuleMessage theResponseMessage = sendHttpRequest(
            "http://localhost:8080/validator?data=123");

        /* Check the response message and its payload. */
        Assert.assertNotNull(theResponseMessage);
        Assert.assertNotNull(theResponseMessage.getPayload());
        Assert.assertNull(theResponseMessage.getExceptionPayload());
        Assert.assertTrue(theResponseMessage.getPayloadAsString().contains(
            "good message"));
    }

    /**
     * Tests sending a bad request to the validator service and ensure
     * that we get the proper response.
     *
     * @throws Exception If error occurs during test.
     * Indicates test failure.
     */
    public void testBadRequest() throws Exception
    {
        MuleMessage theResponseMessage = sendHttpRequest(
            "http://localhost:8080/validator?data=1");

        /* Check the response message and its payload. */
        Assert.assertNotNull(theResponseMessage);
        Assert.assertNotNull(theResponseMessage.getPayload());
        Assert.assertNull(theResponseMessage.getExceptionPayload());
        Assert.assertTrue(theResponseMessage.getPayloadAsString().contains(
            "NACK"));
    }

    private MuleMessage sendHttpRequest(final String inUrl) throws MuleException
    {
        /* Configure Mule message properties. */
        Map<String, String> theMessageProperties =
            new HashMap<String, String>();
        theMessageProperties.put("http.method", "GET");

        /*
         * Create the Mule client and use it to send the message,
         * which in this example is null.
         * Note that, using the Mule client, we can send messages
         * to endpoints using different transports by altering the
         * URL of the endpoint.
         * Example of transports are http and vm.
         */
        MuleClient theClient = new MuleClient(muleContext);
        MuleMessage theResponseMessage = theClient.send(inUrl, null,
            theMessageProperties);
        return theResponseMessage;
    }
}

```

Note that:

- The *CommonValidatorTestCase* class extends the Mule class *FunctionalTestCase*.  
The class *FunctionalTestCase* is a JUnit test case and serves as a base class for tests that initializes Mule using a Mule configuration file.  
The class is found in both Mule 2.x and Mule 3.x, so the above class is used by both versions of the example program we are developing.
- The *testGoodRequest* method sends a request that is expected to produce an ACK result to a HTTP endpoint and verifies the result.  
Note the presence of the string “data=123” in the URL.
- The *testBadRequest* method sends a request that is expected to produce a NACK result to

the same HTTP endpoint and verifies the result.  
Note that the URL now contains the string “data=1”.

- The *sendHttpRequest* method creates a *MuleClient* object and uses this object to send a request.  
The request can be sent to endpoints using different kinds of transports, for instance the Mule VM transport or HTTP transport, by altering the prefix of the URL.
- When sending a request using a *MuleClient* object, we could have enclosed a Mule message.  
In order to keep the example simple and focus on the testing part, a simple Mule configuration that does not receive Mule messages is used. We can thus pass null instead of a Mule message when sending the requests.
- The *sendHttpRequest* method sets up a map containing a single key-value pair.  
These are Mule message properties that we can, despite sending a null Mule message, enclose with the request.  
In this example, using the HTTP transport, these properties will be enclosed as HTTP request headers.

With the above class in place, the Mule 2.x and Mule 3.x classes become trivial – just a matter of specifying the appropriate Mule configuration file to use when initializing Mule.

### Create the Mule 2.x Test

The Mule 2.x test class inherits from the common test class devised above and the only thing we need to do is to specify which Mule configuration file the test is to use.

- In the package *com.ivan.tests*, implement the *Validator2xTestCase* like this:

```
package com.ivan.tests;

/**
 * Tests the Mule 2.x validator configuration.
 *
 * @author Ivan Krizsan
 */
public class Validator2xTestCase extends CommonValidatorTestCase
{
    /**
     * Retrieves the name of the configuration file to be used
     * with the test case.
     *
     * @return Mule configuration file name.
     */
    @Override
    protected String getConfigResources()
    {
        return "com/ivan/muleconfig/mule-config221.xml";
    }
}
```

Note that:

- The *Validator2xTestCase* class implements the *getConfigResources* method.  
This is the method which result tell the test case which Mule configuration file to use when setting up Mule for the test. The method is abstract in the *FunctionTestCase* parent class, which is the reason the *CommonValidatorTestCase* class is abstract.

## Create the Mule 3.x Test

The only difference between the Mule 3.x test class and the Mule 2.x test class we just developed is the use of a different Mule configuration file.

- In the package `com.ivan.tests`, implement the `Validator3xTestCase` as follows:

```
package com.ivan.tests;

/**
 * Tests the Mule 3.x validator configuration.
 *
 * @author Ivan Krizsan
 */
public class Validator3xTestCase extends CommonValidatorTestCase
{

    /**
     * Retrieves the name of the configuration file to be used
     * with the test case.
     *
     * @return Mule configuration file name.
     */
    @Override
    protected String getConfigResources()
    {
        return "com/ivan/muleconfig/mule-config321.xml";
    }
}
```

Note that:

- The `Validator3xTestCase` class implements the `getConfigResources` method. This is the method which result tell the test case which Mule configuration file to use when setting up Mule for the test. The method is abstract in the `FunctionTestCase` parent class, which is the reason the `CommonValidatorTestCase` class is abstract.

## 11.3. Create the Mule Configuration Files

In an attempt to create identical configuration files, except for the namespace declarations, I created a Mule 3.x configuration file that uses a model and a service. Regretfully, Mule 2.x lacks the transformer that transforms HTTP request parameters in the URL to a map in the message payload, so the Mule 2.x configuration does differ slightly from the Mule 3.x dito.

### Create the Mule 2.x Configuration File

The Mule 2.x configuration file is quite straightforward and should by now be familiar to readers of this book.

- In the package `com.ivan.muleconfig`, create a Mule configuration file named “`mule-config221.xml`” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesource.org/schema/mule/core/2.2"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:http="http://www.mulesource.org/schema/mule/http/2.2"
      xmlns:https="http://www.mulesource.org/schema/mule/https/2.2"
      xsi:schemaLocation="http://www.mulesource.org/schema/mule/http/2.2
                           http://www.mulesource.org/schema/mule/http/2.2/mule-http.xsd
                           http://www.mulesource.org/schema/mule/https/2.2
                           http://www.mulesource.org/schema/mule/https/2.2/mule-https.xsd
                           http://www.mulesource.org/schema/mule/core/2.2
                           http://www.mulesource.org/schema/mule/core/2.2/mule.xsd">

<model name="validationModel">
    <service name="validationService">
        <inbound>
            <!--
                Endpoint receiving requests that are to be validated.
                This particular endpoint receives HTTP requests and extracts
                HTTP request parameters from the URL to a map.
                Note that this endpoint is synchronous, in order to be
                able to deliver a ACK or NACK as the result of the request.
            -->
            <http:inbound-endpoint
                address="http://localhost:8080/validator"
                synchronous="true">
                </http:inbound-endpoint>
        </inbound>
        <component>
            <singleton-object class="com.ivan.services.ValidationService2x" />
        </component>
    </service>
</model>
</mule>
```

Note that:

- There is one single model that contains a single service.
- The service has an inbound endpoint.  
The endpoint uses the HTTP transport, is synchronous and has the address  
<http://localhost:8080/validator>.
- The service has a component that is served by a single instance of the `ValidationService2x` class that we will implement in the next section of this chapter.
- There are no transformations applied to incoming messages.  
As we will see when implementing the `ValidationService2x` class, the HTTP parameters can, without any special effort on our side, be found in the message properties of incoming Mule messages.

## Create the Mule 3.x Configuration File

The Mule 3.x configuration file looks a lot like the Mule 2.x configuration file because of reasons mentioned in the introduction to this section.

- In the package `com.ivan.muleconfig`, create a Mule configuration file named “`mule-config321.xml`” with the following contents:

```
<?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:http="http://www.mulesoft.org/schema/mule/http"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd

          http://www.mulesoft.org/schema/mule/http
          http://www.mulesoft.org/schema/mule/http/3.2/mule-http.xsd">

    <model name="validationModel">
        <service name="validationService">
            <inbound>
                <!--
                    Endpoint receiving requests that are to be validated.
                    This particular endpoint receives HTTP requests and extracts
                    HTTP request parameters from the URL to a map.
                    Note that this endpoint uses the request-response message
                    exchange pattern, in order to be able to deliver a ACK or
                    NACK as the result of the request.
                -->
                <http:inbound-endpoint
                    address="http://localhost:8080/validator"
                    exchange-pattern="request-response">
                    <!--
                        The body-to-parameter-map transformer transforms a HTTP
                        request message to a map containing all the request
                        parameters and their values.
                    -->
                    <http:body-to-parameter-map-transformer/>
                </http:inbound-endpoint>
            </inbound>
            <component>
                <singleton-object class="com.ivan.services.ValidationService3x" />
            </component>
        </service>
    </model>
</mule>
```

Note that:

- There is one single model that contains a single service.
- The service has an inbound endpoint.  
The endpoint uses the HTTP transport, is synchronous and has the address  
<http://localhost:8080/validator>.
- The service has a component that is served by a single instance of the `ValidationService3x` class that we will implement in the next section of this chapter.
- There is a body-to-parameter-map transformer in the inbound endpoint.  
This transformer inserts the HTTP request parameters from the URL into a map, which then becomes the payload of the Mule message that is passed on to the service's component.

## 11.4. Create the Service Implementation Classes

The example makes use of two service implementation classes, one for Mule 2.x and one for Mule 3.x, which implement the validation service. Due to the differences in the configuration files seen above, the service implementations differ slightly in how parameter information is retrieved.

### Create the Mule 2.x Service Implementation Class

The Mule 2.x service implementation class checks the message properties for the presence of the HTTP parameter and returns an ACK or NACK message depending on the presence of the expected parameter with the expected value.

- In the package *com.ivan.services*, implement the class *ValidationService2x* as this:

```
package com.ivan.services;

import java.util.Date;
import java.util.Map;
import org.mule.api.MuleEventContext;
import org.mule.api.MuleMessage;
import org.mule.api.lifecycle.Callable;

/**
 * Service that validates received messages and produces a ACK or NACK
 * message depending on the contents of the messages.
 *
 * @author Ivan Krizsan
 */
public class ValidationService2x implements Callable
{
    private final static String ACK_MESSAGE =
        "ACK good message ";
    private final static String NACK_MESSAGE =
        "NACK bad message ";
    private final static String DATA_PARAM = "data";
    private final static String EXPECTED_DATA_VALUE = "123";

    @Override
    public Object onCall(final MuleEventContext inEventContext) throws Exception
    {
        Date theCurrentDate = new Date();
        String theResponseString = null;
        MuleMessage theRequestMessage = inEventContext.getMessage();

        /*
         * Check if the "data" parameter is present in the message properties
         * and has the correct value.
         */
        String theDataValue = (String)theRequestMessage.getProperty(DATA_PARAM);
        if (EXPECTED_DATA_VALUE.equals(theDataValue))
        {
            theResponseString = ACK_MESSAGE;
        } else
        {
            theResponseString = NACK_MESSAGE;
        }

        return theResponseString + theCurrentDate;
    }
}
```

Note that:

- No special processing of the parameters in the URL is done.  
Mule message properties are automatically created for the URL parameters.
- The value of the “data” parameter is retrieved from the message properties of the incoming request message.

## Create the Mule 3.x Service Implementation Class

The Mule 3.x version of the service implementation class expects the payload of messages arriving to the service to be a map that contains the HTTP parameters from the URL of the original request.

- In the package `com.ivan.services`, implement the class `ValidationService3x` like this:

```
package com.ivan.services;

import java.util.Date;
import java.util.Map;

import org.mule.api.MuleEventContext;
import org.mule.api.MuleMessage;
import org.mule.api.lifecycle.Callable;

/**
 * Service that validates received messages and produces a ACK or NACK
 * message depending on the contents of the messages.
 *
 * @author Ivan Krizsan
 */
public class ValidationService3x implements Callable
{
    private final static String ACK_MESSAGE =
        "ACK good message ";
    private final static String NACK_MESSAGE =
        "NACK bad message ";
    private final static String DATA_PARAM = "data";
    private final static String EXPECTED_DATA_VALUE = "123";

    @Override
    public Object onCall(final MuleEventContext inEventContext) throws Exception
    {
        Date theCurrentDate = new Date();
        String theResponseString = null;
        MuleMessage theRequestMessage = inEventContext.getMessage();

        /*
         * Check if the "data" parameter is present in the payload and
         * has the correct value.
         */
        Map<String, String> thePayloadMap =
            (Map<String, String>)theRequestMessage.getPayload();
        String theDataValue = thePayloadMap.get(DATA_PARAM);
        if (EXPECTED_DATA_VALUE.equals(theDataValue))
        {
            theResponseString = ACK_MESSAGE;
        } else
        {
            theResponseString = NACK_MESSAGE;
        }

        return theResponseString + theCurrentDate;
    }
}
```

Note that:

- The payload of request messages reaching the service is a map in which both keys and values are strings.  
As we will see later when developing the Mule 3.x configuration file, this is due to a transformation having been applied to the incoming request.
- The value of the “data” parameter is retrieved from the message payload map.

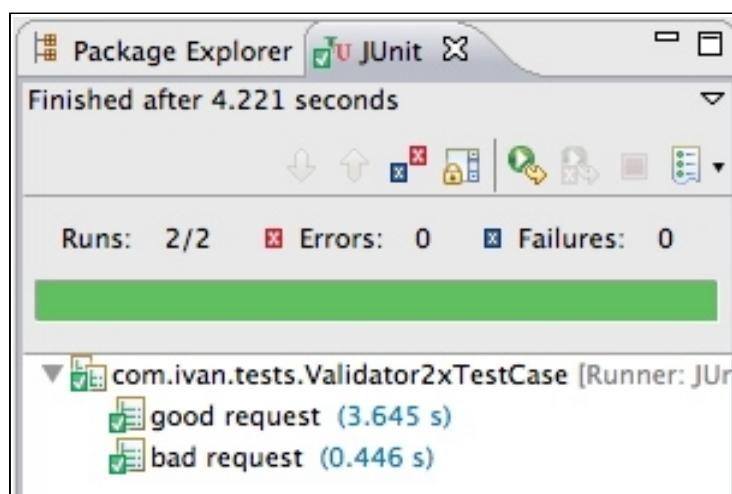
## 11.5. Run the Example Program

With the tests, the service implementations and the Mule configuration files in place, we are now ready to run this chapter's example program, which consists of the two tests.

### Run the Mule 2.x Example Program

If the project is not configured with the Mule 2.x distribution on the classpath, please refer to [this section in appendix B](#) on how to configure which Mule distribution to use.

- In the Package or Project Explorer, right-click the *Validator2xTestCase* class and select Run As -> JUnit Test.
- After some time, we should see the green bar in the JUnit view in Eclipse. This indicates that all the tests in the test class passed.



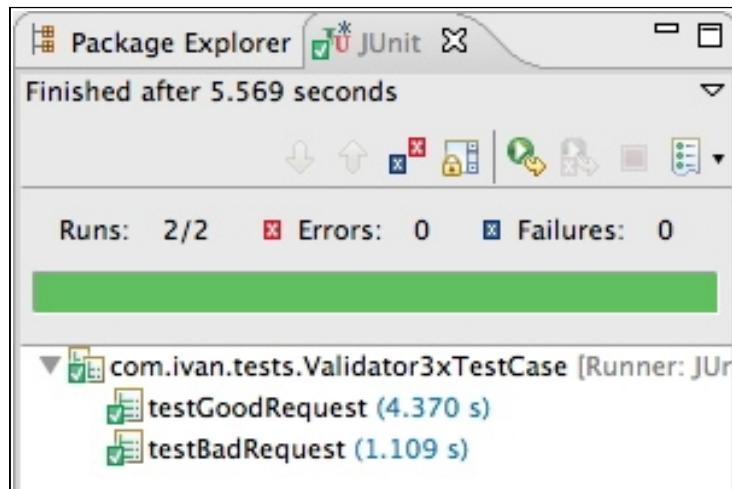
The tests of the *Validator2xTestCase* having passed.

- If we look in the console, there is quite some output, which will not be reproduced here. Notice that the Mule “splash screen” that is printed when Mule has successfully started is printed two times. This means that Mule is started and shut down once for each test-method in the test class.

## **Run the Mule 3.x Example Program**

The Mule 3.x version of the example program is similar, if not identical, to the Mule 2.x version.

- Configure the project to use Mule 3.x, as described in [this section in appendix B](#).
- In the Package or Project Explorer, right-click the *Validator3xTestCase* class and select Run As -> JUnit Test.
- After some time, we should see the green bar in the JUnit view in Eclipse. This indicates that all the tests in the test class passed.



The tests of the *Validator3xTestCase* having passed.

- In the console we can see that, as in the Mule 2.x version of the example program, Mule is started and stopped for each test-method in the test class.

## **11.6. Additional Exercises**

Since we have a test in place that is able to determine if a service behaves as expected, one suggested additional exercise is to develop a version of the Mule 3.x configuration file that uses a flow.

## **12. Create Mule Projects with Maven**

Apache Maven is a widely used software project management tool that aids in structuring software projects and managing dependencies, among other things. Maven archetypes are project templates that can be used to create different kinds of software projects that use Maven.

When developing applications that use Mule, you usually target one particular version of Mule. Under such circumstances, Maven can aid in creating a project with the relevant dependencies.

When developing with Mule, the following Maven archetypes are available:

- Project  
Project template for creating standalone Mule applications.
- Module  
Project template for creating a new, or update an existing, Mule module.
- Transport  
Project template for creating a new Mule transport.
- Example  
Project template for creating a Mule example project.
- Catalog  
Project template for creating new configuration patterns and catalogs of patterns.

In this chapter, we will use Maven to create the basis for a standalone Mule project using the Project template and learn how to build and deploy the project from within Eclipse using Maven.

The chapter's example will only show creation of a Mule 3.x project. The procedure for creating a Mule 2.x project is identical, apart from the version number of the Mule server the application is targeted at.

If you are new to Maven, this chapter may serve as an introduction, describing each procedure in detail. You are however advised to consult additional Maven documentation.

If you are an experienced Maven user, mainly the two first sections may be of interest and the subsequent sections can be skipped.

For more information about Maven, please refer to the [Apache Maven project webpage](#) and the [free books from Sonatype](#).

## 12.1. Prerequisites

We will work with Maven from a terminal, or command-line, window so the mvn command must be installed. For the example in this chapter I have used Maven 3.0.3, but other versions may work equally well.

To determine if the Maven command is available, follow these steps:

- Open a Terminal/Command-line window.
- Enter “mvn -version” (without quotes) and press return.
- If the command is not found, you need to install Maven as described on the [Maven download webpage](#).
- On my system, the output from the command looks like this:

```
Apache Maven 3.0.3 (r1075438; 2011-03-01 01:31:09+0800)
Maven home: /usr/share/maven
...
```

Of most interest is the first row, in which the Maven version is shown. The other rows will vary depending on your operating system, the Java runtime installed etc.

In addition to the Maven command, you also need to be connected to the internet, in order for Maven to be able to download libraries and other artifacts needed when creating the project, building it etc.

## 12.2. Create the Project

With Maven installed, we are now ready to create a Maven project for a Mule standalone application.

- Open a Terminal/Command-line window.
- In the terminal window, go to the location in the file system where you want to create the project.  
This can be, for instance, in your Eclipse workspace.
- Execute the following command in the terminal window:

```
mvn mule-project-archetype:create -DgroupId=com.ivan.mule -DartifactId=MuleMavenProject  
-Dversion=1.0.0-SNAPSHOT -DmuleVersion=3.2.1
```

This creates a Maven project with the following properties:

- Uses the Mule project archetype.
- Maven artifact group id “com.ivan.mule”.
- Maven artifact id, which also will become the Eclipse project name, “MuleMavenProject”.
- Maven artifact version “1.0.0-SNAPSHOT”.
- The project will use Mule version 3.2.1.
- Answer the following questions asked by Maven in the terminal/command-line window (suggested answers in parentheses):
  - Project description.
  - Mule version project is targeted at (use default value).
  - Whether the project will be hosted in the MuleForge Maven repository (no).
  - Project base Java package (com/ivan/mule).  
Note slashes between package names!
  - Mule transports used by the project (use default transports).
  - Mule modules used by the project (use default modules).
- Confirm that project was successfully created:

```
...  
[INFO] Archetype created in dir: /Users/ivan/Desktop/MuleMavenProject  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 12:36.813s  
[INFO] Finished at: Tue Jan 03 22:42:21 CST 2012  
[INFO] Final Memory: 6M/81M  
...
```

If there are problems when creating the project, try adding “-U” (without quotes) to the above Maven command. This forces Maven to check for resource updates in remote repositories and updates any local copy of the Maven Mule project archetype.

- In the terminal window, go into the newly created project directory.  
In my case the name of the directory is “MuleMavenProject”.
- Examine the contents of the project directory.  
There should be two files, “MULE-README.txt” and “pom.xml”, and one directory named “src”.

The project has now been created, but it is just a skeleton-project. Keep the terminal window open – we will use it in the next section when preparing the project to be imported into Eclipse.

## 12.3. Import Project Into Eclipse

Our next step is to import the new Mule project into Eclipse, so that we can develop the Mule application and any associated tests etc.

- In the terminal window, execute the following command:

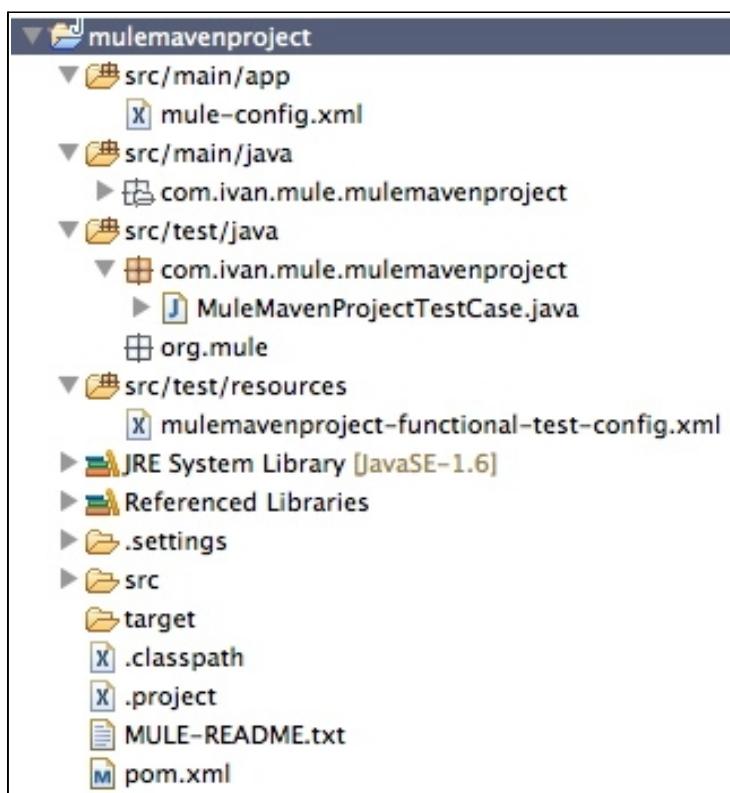
```
mvn -DdownloadSources=true eclipse:eclipse
```

This creates an Eclipse project from the Maven pom.xml file and downloads any available source code for the libraries used by the project.

- Import the project into Eclipse.

In the File menu, select Import... and then Existing Projects into Workspace. Finally browse to the root directory of the project to import, which is the MuleMavenProject directory.

There should now be a new project in Eclipse with one Mule configuration file (mule-config.xml), one Mule test-case (MuleMavenProjectTestCase.java) and one Mule configuration file for the test (mulemavenproject-functional-test-config.xml):



The new Mule project created with Maven and imported into Eclipse.

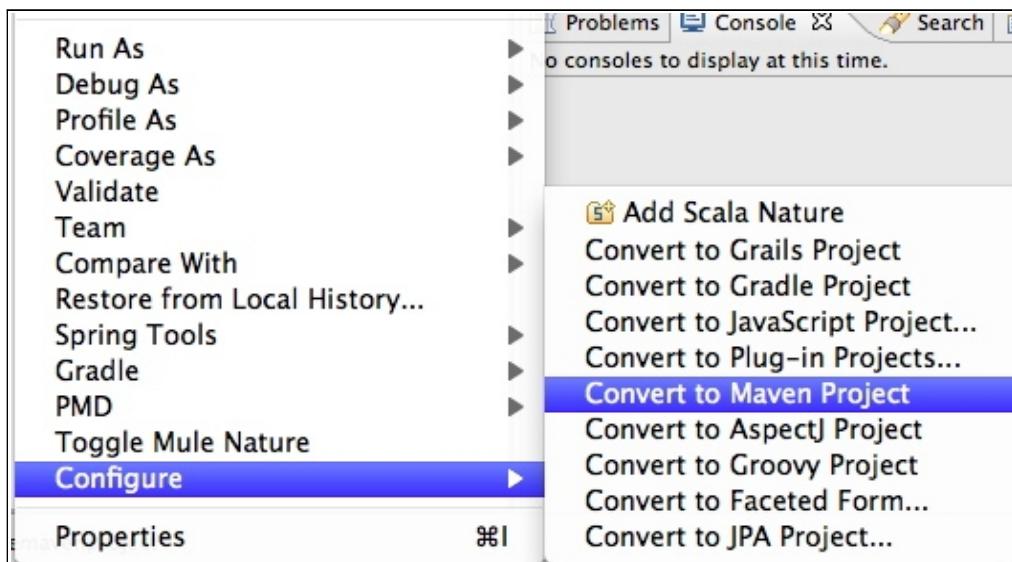
## 12.4. Configure Project in Eclipse

It should be noted that when the new Mule project just has been created and imported into Eclipse, it does not use Maven dependency management, nor is it possible to use Maven to build and deploy the project.

We will now configure the Eclipse project to use Maven dependency management and to be able to use Maven from within Eclipse to build and deploy the project.

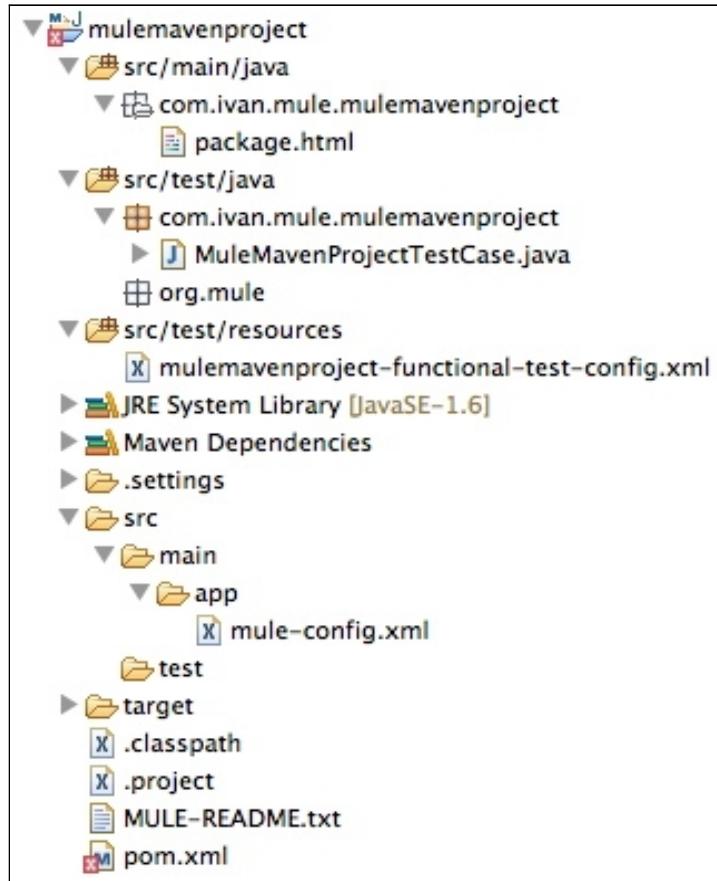
A prerequisite for being able to use Maven from within Eclipse is some kind of Maven plugin. If you use SpringSource Tool Suite, as I have done in this section, there will already be a Maven plugin installed in your IDE.

- In the Eclipse Package Explorer (or in the Project Explorer), right-click on the project node, select Configure and, in the sub-menu that appears, select Convert to Maven Project.



Configuring the Eclipse project to use Maven in Eclipse.

In the Package Explorer, you should now be able to see a new node named Maven Dependencies:

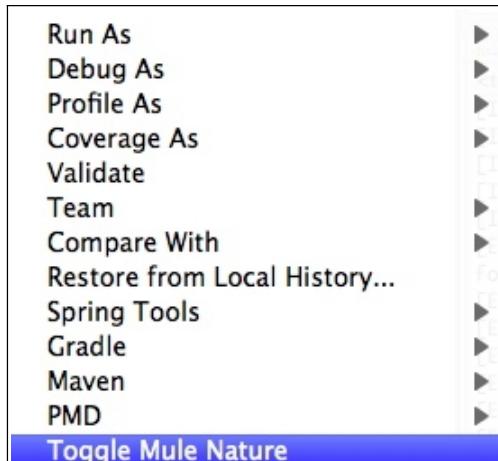


Eclipse project after Maven dependency management has been enabled.

There may be an error in the Maven pom.xml file stating that “Plugin execution not covered by lifecycle configuration”. This will not affect the project and the message may be ignored.

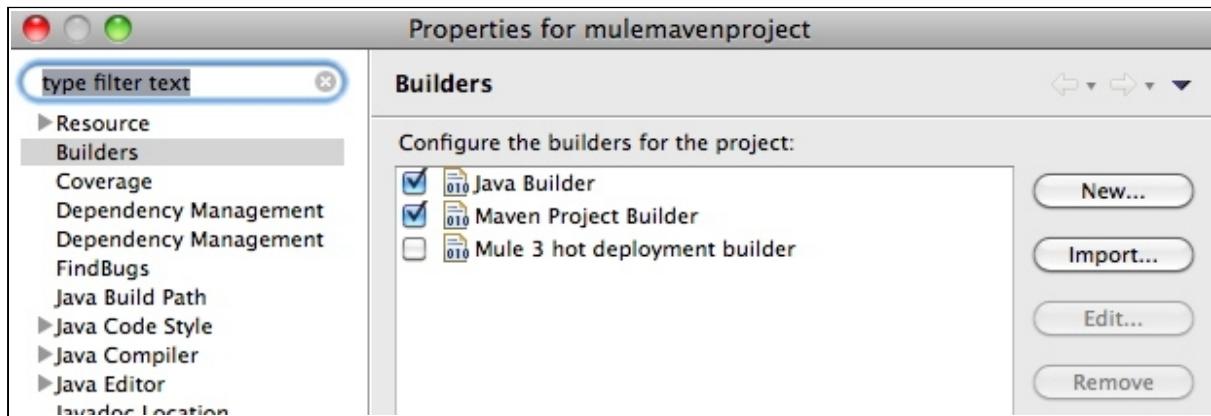
Optionally, we can also add the Mule nature to the Eclipse project and, as part of this, disable the Mule 3 hot deployment builder that tends to cause build failures.

- In the Eclipse Package Explorer (or in the Project Explorer), right-click on the project node and select Toggle Mule Nature.



Toggling the Mule nature of an Eclipse project.

- In the Eclipse project properties, navigate to the Builders node, deselect the Mule 3 hot deployment builder and click the OK button to save the project properties.



Disabling the Mule 3 hot deployment builder in an Eclipse project.

The Eclipse project is now configured to use Maven dependency management. In the next section we will look at how to build and deploy the project using Maven.

## 12.5. Use Maven in Eclipse

Not only can Maven aid us in handling the project dependencies, but Maven can also help us to build our project and deploy it to a running Mule instance. Using the “mvn” command, we can do this from a terminal/command-line window. In this section we will also learn how to use Maven from within Eclipse.

### **Maven Goals**

Maven goals are similar to commands that can be issued to Maven. When working with a Mule application, the following Maven goals are commonly used:

<b>Maven Goal</b>	<b>Description</b>
package	Create a zip-archive that contains the Mule application.
install	Create a zip-archive containing the Mule application, store it in the local Maven repository and deploy it to a Mule server.
deploy	Create a zip-archive that contains the Mule application and store it in the local Maven repository.
clean	Delete the zip-archive that contains the Mule application along with any other artifacts generated by Maven when building the Mule application.
test	Runs all the tests present in the project.

In addition, all the Maven goals listed above, except for “clean”, causes the tests present in the project to be run. If any test fails, the Maven command will fail.

Tests can be skipped by setting the “maven.test.skip” parameter with the value true when invoking Maven.

## **Create an Eclipse Maven Run Configuration**

In order to be able to tell Maven to execute a goal from within Eclipse, we need to create a Maven Build Run Configuration in Eclipse.

The example in this section will build the Mule application and deploy it to a running standalone Mule server.

The standalone Mule server is started using the following procedure:

- Locate the directory containing the Mule server that you want to run.  
For Mule 3.2.0, this directory can have the name “mule-standalone-3.2.0”. It contains subdirectories with the names “apps”, “bin”, “conf”, “docs” etc.
- Open a terminal/command-line window and go to the Mule server directory.
- Set the environment variable MULE\_HOME to the path locating the Mule server.  
\*nix: env MULE\_HOME="[insert path to Mule server here]"  
Windows: set MULE\_HOME=[insert path to Mule server here]
- Go into the “bin” directory.
- Start the Mule server.  
\*nix: ./mule  
Windows: mule.bat

After some console output, you should see the following message indicating successful startup of the Mule server:

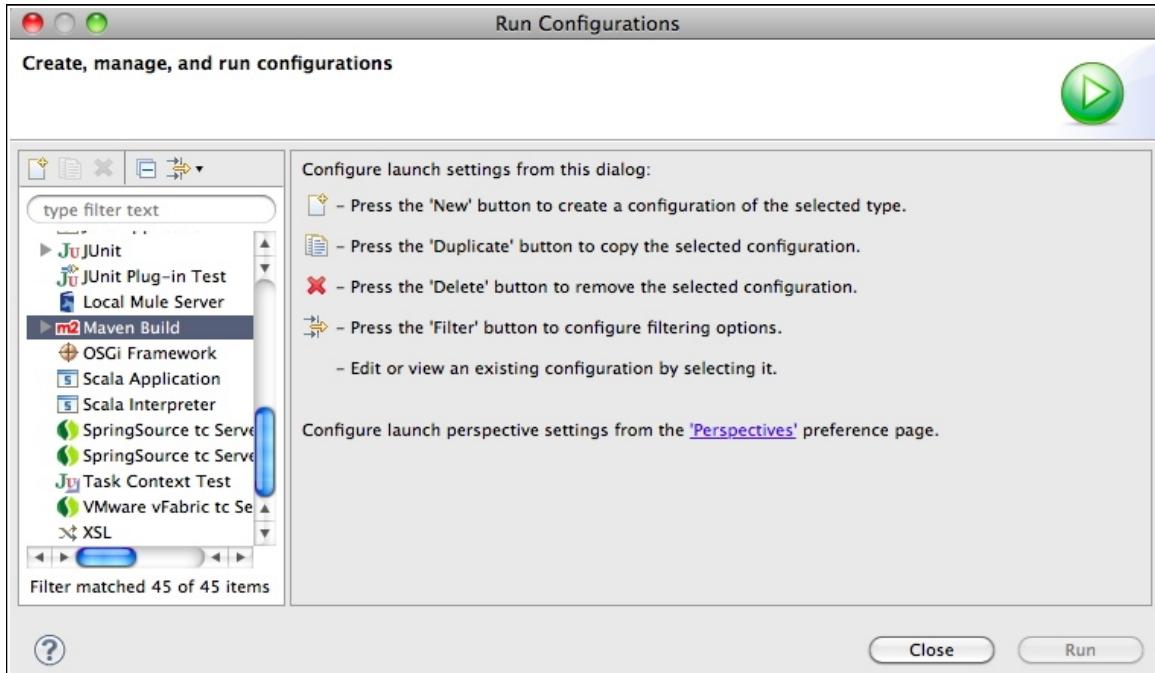
```
...
INFO 2012-01-10 06:47:08,957 [WrapperListener_start_runner]
org.mule.module.launcher.DeploymentService:
+++++
+ Mule is up and kicking (every 5000ms)
+++++
...
```

With the Mule server up and running, we are now ready to create the Eclipse run configuration:

- In Eclipse, select Run Configurations... in the Run menu.  
You can also use the small, round, run button in the toolbar and select Run Configurations... in the menu that appears when clicking the small triangle immediately to the right of the run button.

( continued on next page)

- On the left side of the Run Configurations dialog, locate the Maven Build group.



Locating the Maven Build group among the run configurations in Eclipse.

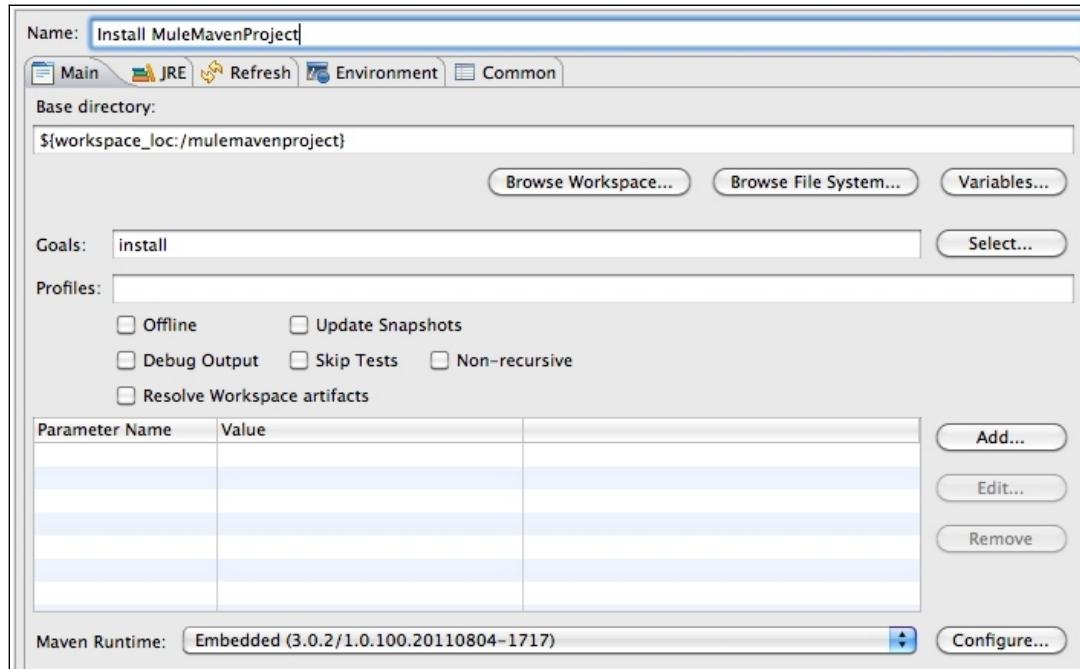
- Right-click the Maven Build node (selected in the above picture) and select New. A new run configuration should appear in the Maven Build node and its settings appears on the right side in the dialog.
- In the run configuration, enter “Install MuleMavenProject” as the name of the run configuration.  
This is an arbitrary name that helps us to identify the run configuration.
- Click the Browse Workspace button and select the “mulemavenproject” project as base directory of the run configuration.  
This is the directory that contains the project's Maven pom.xml file.

(continued on next page)

- In the Goals field, enter “install”.

This is the Maven goal that is to be executed when this run configuration is used and it should be changed according to the purpose of the run configuration.

The Main tab of the run configuration settings should now look like in the picture below:

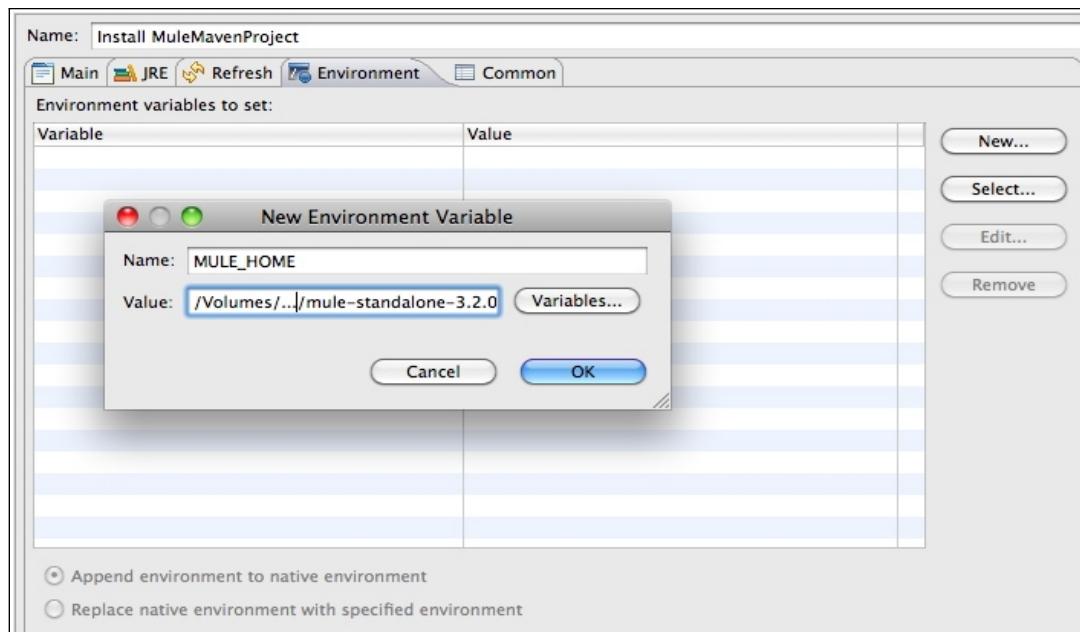


Creating a Maven run configuration; main settings tab.

- Switch to the Environment tab in the run configuration dialog.

- Click the New button to create an environment variable.

Enter the variable-name “MULE\_HOME” and the path to the server directory of the Mule server we started earlier. Environment variables listed in this tab will be set prior to executing the Maven goal of the run configuration.



Creating a Maven run configuration; setting the environment variable specifying the location of the standalone Mule server to which the Mule application will be deployed.

- Click the Apply button in the lower right part of the dialog.

- Click the Run button below the Apply button.

The project should now be built and deployed to the Mule server.

The last part of the console output from Maven should look similar to this:

```
...
[INFO] Executing tasks
[copy] Copying 1 file to /Volumes/.../mule-standalone-3.2.0/apps
[INFO] Executed tasks
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 19.528s
[INFO] Finished at: Tue Jan 10 18:05:21 CET 2012
[INFO] Final Memory: 12M/81M
[INFO] -----
```

Note the row indicating that one file was copied to the “apps” directory in the standalone Mule server directory – this is the Mule application being deployed to the server.

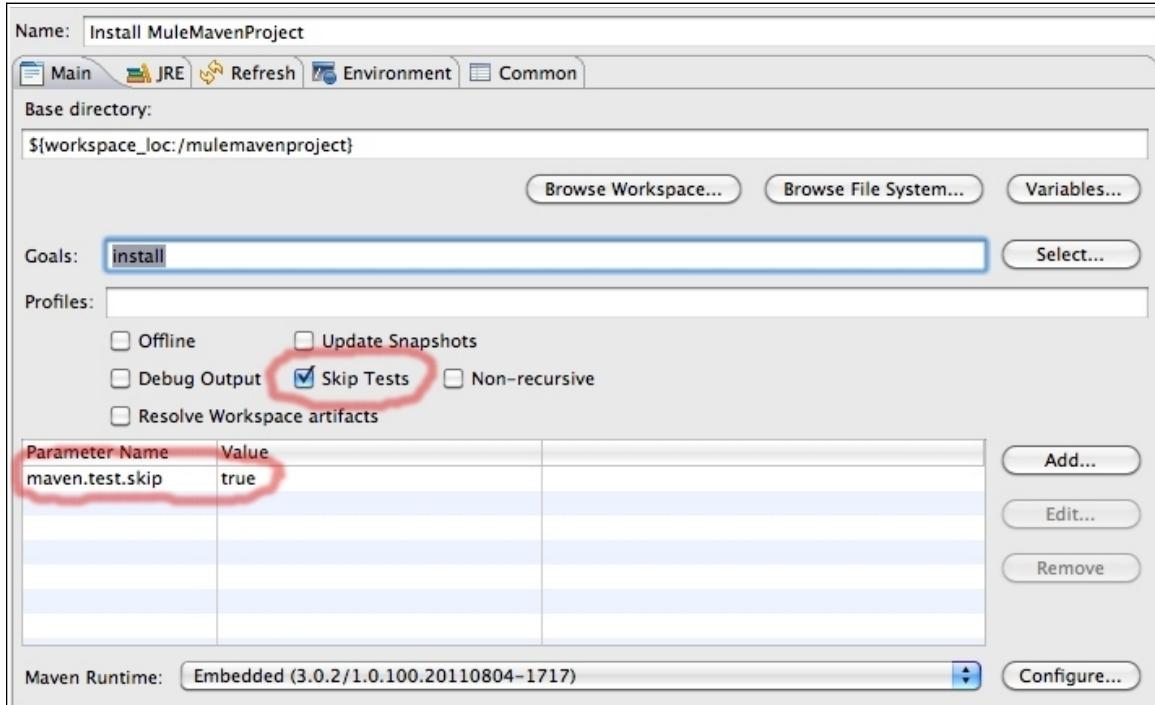
- Examine the console output form the Mule standalone server, which should output similar to the following:

```
...
INFO 2012-01-10 18:05:22,848 [Mule.app.deployer.monitor.1.thread.1]
org.mule.module.launcher.DefaultMuleDeployer: Exploding a Mule application archive:
file:/Volumes/.../mule-standalone-3.2.0/apps/mulemavenproject-1.0.0-SNAPSHOT.zip
INFO 2012-01-10 18:05:22,852 [Mule.app.deployer.monitor.1.thread.1]
org.mule.module.launcher.application.DefaultMuleApplication:
+++++
+ New app 'mulemavenproject-1.0.0-SNAPSHOT' +
+++++
INFO 2012-01-10 18:05:23,536 [Mule.app.deployer.monitor.1.thread.1]
org.mule.module.launcher.DeploymentService:
+++++
+ Started app 'mulemavenproject-1.0.0-SNAPSHOT' +
+++++
```

The above output indicates that our Mule application has successfully been deployed to the Mule server and started.

If you, for some reason, do not want the tests of the project to be executed as part of a Maven build, you can configure the Eclipse run configuration to skip tests.

- In the Main tab of the Eclipse run configuration, either check the “Skip Tests” checkbox or add the parameter “maven.test.skip” with the value true:



Configuring a Maven run configuration in Eclipse to skip test execution.  
Use one of the options circled in red.

If you are executing Maven from the console, then add “-Dmaven.test.skip” to the parameters of the “mvn” command to skip execution of tests.

This concludes this chapter, in which we created a Mule project, built it and deployed it to a Mule server, all using Maven.

## **13. Mule Configuration Patterns**

In Mule 3.x there are four configuration patterns that represent common scenarios. The patterns are:

- Bridge
- Simple Service
- Validator
- Webservice Proxy

In this chapter we will look at each of these configuration patterns.

Note that these configuration patterns can only be used with Mule 3.x and the minimum version is 3.1.1, in order for the examples in this chapter to work as expected.

### **13.1. Create the Project**

All the examples for the different configuration patterns are developed in one and the same project. Since the examples will only run on Mule 3.x, the project is to be configured for Mule 3.x from the start.

Create the project as described in the appendix [Create a Mule Project](#), naming it “MulePatterns”. The Mule 3 hot deployment can be switched off from the start, as this feature will not be used.

In addition, make the following preparations in the Eclipse project:

- Create a source package named *com.ivan.exceptionhandlers*.
- Create a source package named *com.ivan.jaxb\_generated*.
- Create a source package named *com.ivan.muleconfig*.
- Create a source package named *com.ivan.schemas*.
- Create a source package named *com.ivan.services*.
- Create a directory named “example-data” in the root of the Eclipse project.

## 13.2. The Bridge Pattern

The bridge Mule pattern allows for creation of bridges, or adapters, that receive messages on one endpoint and send them to another endpoint. This can be useful when receiving messages on an endpoint that uses one protocol, for instance JMS, and passing them on to another endpoint that uses another protocol, for instance JDBC.

The bridge pattern also allows for transformation of the message being sent out from the bridge as well as transformation of any response message received as a result of the message sent out from the bridge.

- As a preparation for the simple-service examples, create a Mule configuration file named “mule-bridge-config.xml” in the *com.ivan.muleconfig* package with the following contents:

```
<?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:vm="http://www.mulesoft.org/schema/mule/vm"
      xmlns:pattern="http://www.mulesoft.org/schema/mule/pattern"
      xmlns:file="http://www.mulesoft.org/schema/mule/file"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

          http://www.mulesoft.org/schema/mule/vm
          http://www.mulesoft.org/schema/mule/vm/current/mule-vm.xsd

          http://www.mulesoft.org/schema/mule/file
          http://www.mulesoft.org/schema/mule/file/3.2/mule-file.xsd

          http://www.mulesoft.org/schema/mule/pattern
          http://www.mulesoft.org/schema/mule/pattern/3.2/mule-pattern.xsd">

    <!-- File connector. -->
    <file:connector
        name="fileConnector"
        streaming="false"
        pollingFrequency="1000">
    </file:connector>

    <!-- Transforms a byte array to an object. -->
    <byte-array-to-object-transformer name="bytesToObjectXform"/>

    <!-- INSERT STUFF HERE. -->
</mule>
```

Note that:

- The Mule configuration file contains a file connector.  
We'll use this connector for the file inbound endpoints in the example.
- The Mule configuration file contains a global transformer named “bytesToObjectXform”.  
This transformer transforms a byte array to an object – a string, to be more specific.

We'll look at two bridge examples, one synchronous bridge and one asynchronous bridge. Both examples uses bridges receiving and sending messages over the Mule VM transport. Trying out other transports in connection to the bridge pattern is left as an exercise for the reader. Both the examples consist of Mule configuration only.

## Synchronous Bridge

The first example shows a synchronous bridge that, when having received a message, passes it on and waits for a response. Apart from synchronous communication, this example also shows the following facilities of the bridge pattern:

- Bridge inheritance.  
An abstract bridge can be defined and inherited from by bridges that are to share the common properties of the parent.
- Transformation of response messages.
- Configuration of in- and outbound endpoints using attributes of the <bridge> element.

## Create Inbound and Outbound Services

This bridge example make use of two flows/services:

A receiver flow, which receives a message when a file is placed in a certain directory and passes it on. A separate receiver flow is used in order to be able to log response messages.

A writer flow that receives messages synchronously and logs them to the console. Response messages from this flow are identical to the corresponding request message.

The writer flow assumes that the payload of incoming messages are contained in a byte array.

- Add the following flow configurations to the “mule-bridge-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
First receiver flow.
Receives contents of files placed in a certain directory,
synchronously sending messages to the first bridge and,
when a response is received, logs the message payload to
the console.
-->
<flow name="receiverFlow1">
    <file:inbound-endpoint
        connector-ref="fileConnector"
        path=".//receiver1-input-directory">
    </file:inbound-endpoint>

    <outbound-endpoint
        address="vm://bridgeInbound"
        exchange-pattern="request-response"/>

    <logger level="WARN" message="Receiver 1: #[payload]"/>
</flow>

<!--
First writer flow.
Synchronously receives messages and logs their payload to the
console.
Assumes that message payloads are byte arrays and need
to be converted to a string in order to produce a
human-readable log.
-->
<flow name="writerFlow1">
    <inbound-endpoint
        address="vm://writerService"
        exchange-pattern="request-response"/>

    <logger level="ERROR" message="Writer 1: #[groovy:new String(payload)]"/>
</flow>
...
```

Note that:

- The first flow, “receiverFlow1”, receives messages from a file inbound endpoint and synchronously pass them on to a VM outbound endpoint.  
The name of the directory in which the inbound endpoint will look for files is “receiver1-input-directory”.
- The outbound endpoint of the “receiverFlow1” has the message exchange pattern “request-response”.
- The “receiverFlow1” logs response messages returned from the VM outbound endpoint.  
The expression in the log that retrieves the message payload is #[payload]. This means that the message payload must be a string, in order for the payload to be displayed in a human-readable form.
- If it weren't for the logging of response messages, the “receiverFlow1” could have been replaced by a bridge.
- The second flow, the “writerFlow1”, receives messages from a VM inbound endpoint and logs the messages.
- The inbound endpoint of the “writerFlow1” has the message exchange pattern “request-response”.
- The log message in the “writerFlow1” flow's logger contains the expression #[groovy:new String(payload)].  
The “writerFlow1” flow makes the transformation from byte array payload to a string in connection to writing the log.
- The “writerFlow1” flow logs the message payload assuming it is a byte array, while the “receiverFlow1” flow logs the message payload assuming it is a string.  
The bridge that we insert between these two flows, or services, will have to ensure that the message payload reaching the “writerFlow1” flow is a byte array and that the message payload reaching the logger of the “receiverFlow1” flow is a string.
- The outbound endpoint of the “receiverFlow1” flow uses the message exchange pattern “request-response”. The inbound endpoint of the “writerFlow1” flow uses the same message exchange pattern, so there is no mismatch concerning this regard.

We now have some requirements for the bridge that is to connect the two flows discussed above.

## Bridge the Services

When bridging the two flows, or services, constructed in the previous section, we will, for the sake of the example, use not one, but two bridges. This is to show that bridges support inheritance. We can define an abstract bridge that extracts properties common to multiple bridges and thus reduce repetition and simplify maintenance.

In this example, we define an abstract parent bridge that has the request-response message exchange pattern and that transforms response messages going through the bridge from a byte array to an object (string). The child bridge then only need to specify, apart from its parent, the inbound and outbound endpoints that it is to connect.

- Add the following flow configurations to the “mule-bridge-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
    Parent, abstract, bridge.
    Bridges can be declared as abstract in order to serve as a
    common parent to one or more bridges.
    The Spring bean inheritance mechanism facilitates bridge inheritance.
    This bridge specifies the use of the request-response
    message exchange pattern and the use of a byte-to-object
    transformer on response messages.
-->
<pattern:bridge
    name="syncResponseXformBridge"
    abstract="true"
    exchange-pattern="request-response"
    responseTransformer-refs="bytesToObjectXform" />

<!--
    First bridge.
    This bridge transfers messages from the first bridge input
    endpoint to the first writer service.
    This bridge inherits some properties from the above, abstract,
    bridge. This cause this bridge to use the request-response
    message exchange pattern and to apply a byte-to-object
    transform on response messages received from the first writer
    service.
-->
<pattern:bridge
    name="receiverToWriterBridge"
    parent="syncResponseXformBridge"
    inboundAddress="vm://bridgeInbound"
    outboundAddress="vm://writerService" />
...
...
```

Note that:

- The parent bridge is named “synchResponseXformBridge”.
- The parent bridge is abstract.

This is accomplished by setting value of the *abstract* attribute of the <bridge> element to true.

- The parent bridge uses the message exchange pattern “request-response”.

- The parent bridge specifies that the transformer “bytesToObjectXform” is to be applied to response messages passing through the bridge.

The value of the *responseTransformer-refs* attribute can contain a list of transformers to be applied to response messages, which are response messages received from the service which address is the outbound address of the bridge.

If multiple names occur in the transformer list, they are to be separated by spaces.

- The first, concrete, bridge is named “receiverToWriterBridge”.
- The first bridge has the parent bridge “synchResponseXformBridge”. This relation is declared using the *parent* attribute of the <bridge> element.
- The first bridge declare an inbound and an outbound address. The inbound address is the address to which the receiver flow sends messages it receives. The outbound address is the address of the inbound endpoint of the writer flow declared earlier. It is also possible to use references to global endpoints, using the *inboundEndpoint-ref* and *outboundEndpoint-ref* attributes of the <bridge> element, or, as we will see an example of later, declare in- and outbound endpoints as child elements of the <bridge> element.

## Run the Synchronous Bridge

We are now ready to test the synchronous bridge.

- Right-click the “mule-bridge-config.xml” Mule configuration file and select Run As -> Mule Server.
- Refresh the project.  
A new directory, “receiver1-input-directory”, should appear in the root of the project. This is the directory in which we place files which contents is to be sent to the receiver flow of the example.
- In Eclipse, copy the file “person.xml” in the “example-data” directory and paste it onto the “receiver1-input-directory”.
- Examine the console.  
The following should have been logged to the console, among other things:

```
...
ERROR 2012-03-07 16:39:07,488 [receiverFlow1.stage1.02]
org.mule.api.processor.LoggerMessageProcessor: Writer 1: <?xml version="1.0"
encoding="UTF-8"?>
<ivan:person
    xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas schema2.xsd ">
    <firstName>Stiv</firstName>
    <lastName>Bator</lastName>
    <age>32</age>
</ivan:person>

WARN 2012-03-07 16:39:07,509 [receiverFlow1.stage1.02]
org.mule.api.processor.LoggerMessageProcessor: Receiver 1: <?xml version="1.0"
encoding="UTF-8"?>
<ivan:person
    xmlns:ivan="http://www.ivan.com/schemas"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ivan.com/schemas schema2.xsd ">
    <firstName>Stiv</firstName>
    <lastName>Bator</lastName>
    <age>32</age>
</ivan:person>
```

Note that:

- Since the entire message flow is synchronous, the messages are displayed in the order that they are processed by components participating in the message flow.
- The first log entry was written by the writer (green highlight).  
In addition, it was written using the ERROR log level, which is what the logger in the writer

is configured to use.

- The XML in the first log entry is human-readable.

The conversion in the logger from byte array to string worked as expected. This also means that the message payload at this stage in the message flow indeed is a byte array.

- The second log entry was written by the receiver (blue highlight).

The log level was WARN, which is what the receiver's log is configured to use.

- The XML in the second log entry is also human-readable.

The logger in the receiver writes the payload directory to the log. We can thus conclude that the bridge has successfully transformed the response message from the writer from a byte array to a string.

We conclude that the bridge has fulfilled the requirements noted earlier and conveyed messages between the two flows/services as well as made the necessary transformations of the message format.

## **Asynchronous Bridge**

The second bridge example shows an asynchronous bridge that passes messages from a file inbound endpoint to a writer flow/service without waiting for any response. Due to using asynchronous communication, there is no reason for having a receiver flow/service – there will be no response message to log.

In addition to asynchronous message exchange, the example also shows the following:

- Transformation of messages received by the bridge prior to passing them on.
- Configuration of the inbound and outbound endpoints of the bridge as child elements of the <bridge> element.
- Configuration of an exception strategy on the bridge.
- Noting that a bridge can be transactional.

## **Create the Outbound Service**

The asynchronous bridge example uses one single flow/service; a writer flow/service that receives messages asynchronously and logs them to the console. No response messages are sent from the writer flow/service.

This writer flow assumes that the payload of incoming messages are strings.

- Add the following flow configurations to the “mule-bridge-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
    Second writer flow.
    Asynchronously receives messages and logs their payload to the
    console.
    Assumes that message payloads are strings which require no
    further transformation to produce a human-readable log.
-->
<flow name="writerFlow2">
    <inbound-endpoint
        address="vm://writerService2"
        exchange-pattern="one-way"/>

        <logger level="ERROR" message="Writer 2: #[payload]" />
</flow>
...
```

Note that:

- The flow “writerFlow2” receives messages from a VM inbound endpoint and logs the messages.
- The inbound endpoint of the “writerFlow2” has the message exchange pattern “one-way”.
- The log message in the “writerFlow2” flow’s logger directly logs the message payload, as retrieved by the expression #[payload].  
The assumption is made that the message payload is a string.

Since the inbound endpoint is to be declared in the bridge, we are now ready to configure the bridge.

## Bridge the Services

As noted earlier, there is no receiver service in this example. Instead, the bridge we are about to configure conveys data received by an inbound file endpoint to the writer flow/service we created in the previous section.

In this second bridge example, we will use one single bridge which has the inbound and outbound endpoints configured using child elements of the `<bridge>` element, as opposed to using attributes as in the previous example.

For the sake of showing that it is possible, we will also configure an exception strategy on the bridge as well as making the bridge transactional.

- Add the following flow configurations to the “mule-bridge-config.xml” file, immediately before the closing `<mule>` tag:

```
...
<!--
    Second bridge.
    This bridge transfers messages directory from a file inbound
    endpoint to the second writer output in an asynchronous
    fashion.
    There is a byte-to-object transformer applied on incoming
    messages.
    Note also how the transacted attribute on the <bridge> element is
    set to true. This enables transaction support for the bridge,
    provided that the transports used by the inbound and outbound
    endpoints supports transactions.
    Both the endpoints are configured using child elements of the
    <bridge> element.
    The bridge also has a exception strategy configured, which will
    log exceptions.

-->
<pattern:bridge
    name="receiverToWriterBridge2"
    exchange-pattern="one-way"
    transformer-refs="bytesToObjectXform"
    transacted="true">
    <file:inbound-endpoint
        connector-ref="fileConnector"
        path=".//receiver2-input-directory">
    </file:inbound-endpoint>
    <vm:outbound-endpoint path="writerService2"/>

    <default-exception-strategy>
        <logger level="ERROR" message="An exception occurred! "/>
    </default-exception-strategy>
</pattern:bridge>
...
```

Note that:

- The bridge is named “`receiverToWriterBridge2`”.
- The bridge is configured to use the one-way message exchange pattern.  
Using this message exchange pattern, the bridge will not respond to incoming messages nor await responses from outgoing messages.
- The `<bridge>` element contains a `transformer-refs` attribute referencing the global byte-array-to-object transformer contained in the configuration file.  
This transformer is applied to messages received by the bridge, before they are passed on to the outbound endpoint.
- The `<bridge>` element contains the `transacted` attribute that has the value “`true`”.  
This is actually not relevant to this example, since the neither the inbound nor the outbound

endpoint of the bridge connects to a transactional resource.

If they did, setting this attribute to true will cause the bridge to push back a message received on the inbound endpoint, for instance from a JMS queue, if sending the message to the outbound endpoint, for instance inserting it into a database table, fails.

- The bridge does not declare an inbound nor an outbound address.
- The `<bridge>` element contains an `<inbound-endpoint>` and an `<outbound-endpoint>` child elements.

Instead of using inbound and outbound addresses, as in the previous example, we can declare the endpoints of the bridge as child elements of the `<bridge>` element.

As mentioned in the previous example, but not shown in this chapter, is the possibility to use references to global endpoints, with the `inboundEndpoint-ref` and `outboundEndpoint-ref` attributes of the `<bridge>` element.

- The `<bridge>` element contains a `<default-exception-strategy>` child element.

It is possible to configure an exception strategy for the bridge itself, which will come into effect if an error occurs when a message passes through the bridge.

For details on exception handling strategies, please refer to the chapter on [Exception Handling in Mule](#).

## Run the Asynchronous Bridge

We are now ready to test the asynchronous bridge.

- Right-click the “mule-bridge-config.xml” Mule configuration file and select Run As -> Mule Server.
- Refresh the project.  
A new directory, “receiver2-input-directory”, should appear in the root of the project. This is the directory in which we place files which contents is to be sent to the receiver flow of the example.
- In Eclipse, copy the file “person.xml” in the “example-data” directory and paste it onto the “receiver2-input-directory”.

The XML contents of the “person.xml” file should have been written to the console by the second writer. There will be only one single occurrence of the XML message, since there is only one single logger in the message flow.

### 13.3. The Simple Service Pattern

The simple service pattern enables swift development of a service with one inbound endpoint and one component. This section will show the different possibilities available when using this pattern. In addition, we will see how to develop services that uses different ways to process the payload of messages received - for instance JAXB, JAX-WS and JAX-RS.

- As a preparation for the simple-service examples, create a Mule configuration file named “mule-simpleservices-config.xml” in the *com.ivan.muleconfig* package with the following contents:

```
<?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:mule-xml="http://www.mulesoft.org/schema/mule/xml"
      xmlns:test="http://www.mulesoft.org/schema/mule/test"
      xmlns:pattern="http://www.mulesoft.org/schema/mule/pattern"
      xmlns:ivan="http://www.ivan.com/person"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.2/mule.xsd

          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

          http://www.mulesoft.org/schema/mule/xml
          http://www.mulesoft.org/schema/mule/xml/current/mule-xml.xsd

          http://www.mulesoft.org/schema/mule/test
          http://www.mulesoft.org/schema/mule/test/3.2/mule-test.xsd

          http://www.mulesoft.org/schema/mule/pattern
          http://www.mulesoft.org/schema/mule/pattern/3.2/mule-pattern.xsd">

    <!--
        Global transformers used by all the example simple-services.
    -->
    <append-string-transformer name="firstAppender"
        message="[first appender msg]"/>
    <append-string-transformer name="secondAppender"
        message="[second appender msg]"/>

</mule>
```

Note that:

- There are a number of namespace prefixes defined in the Mule configuration file. These are prefixes for different Mule configuration namespaces which we will need in the example, except for the “ivan” prefix, which is for a XML schema namespace.
- The Mule configuration file contains two transformers. Both the transformers append a string to the message to which they are applied.
- Both transformers have a name. This in combination with their location in the configuration file makes them global transformers.

## JAX-RS Simple Service

In this part of the example, we will develop a JAX-RS web service using the simple service pattern. We will also see how to apply transformers on incoming messages before they are sent to the component of the service and on the response message from the component of the service.

### Create the Component Implementation Class

The component/service implementation class used with the JAX-RS service will also be used with the JAX-WS service developed in the next section, so it contains not only JAX-RS annotations, but also a JAX-WS annotation.

- In the `com.ivan.services` package, implement the `HelloService` class as this:

```
package com.ivan.services;

import java.util.Date;
import javax.jws.WebService;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

/**
 * Web service endpoint implementation class that implements
 * a service that extends greetings.
 *
 * @author Ivan A Krizsan
 */
@Path("greeting")
@WebService
public class HelloService
{
    /**
     * Greets the person with the supplied name.
     *
     * @param inName Name of person to greet.
     * @return Greeting.
     */
    @GET
    @Path("{name}")
    @Produces("text/html")
    public String greet(@PathParam("name") final String inName)
    {
        return "Hello " + inName + ", the time is now " + new Date();
    }
}
```

Note that:

- The class is annotated with the JAX-WS `@WebService` annotation.  
This will cause JAX-WS to expose all public methods in a SOAP web service.
- The class is annotated with the JAX-RS `@Path` annotation.  
The `@Path` annotation specifies that the greeting-resource managed by this class is available at the “greeting” path. This path is relative to the service endpoint URL as will be specified in the Mule configuration file later.
- The `greet` method is annotated with the JAX-RS `@GET` annotation.  
When issuing a HTTP GET to the greeting resource, the `greet` method is the method that will become invoked.
- The `greet` method is also annotated with the JAX-RS `@Path` annotation.  
In the case of the `greet` method, the `@Path` annotation is used to tell JAX-RS that a part of

the URL will be used as a parameter to the annotated method.

- Finally, the greet method is also annotated with the JAX-RS @Produces annotation. The @Produces annotation determines what type of result will produce. In this case, it will be HTML text. The method does not actually produce HTML text, but using this type of result, we will be able to view the string returned from the *greet* method in a browser.

## Modify the Mule Configuration File

To complete the JAX-RS simple service example, we append a simple service configuration to our Mule configuration file:

- Add the following to the “mule-simpleservices-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
    Example of simple service exposing a JAX-RS web service.
    The transformers listed in the transformer-refs attribute are
    applied to the message before it is sent to the service's
    component.
    The transformers listed in the responseTransformer-refs attribute
    are applied to the message returned by the service's component.

    Note that the appenders does not affect the parameter nor
    the result of the JAX-RS service. They may affect the payload.

    Simple services only work with request-response inbound endpoints,
    so we cannot use, for instance, file inbound endpoints, since they
    are one-way.
-->
<pattern:simple-service
    name="MySimpleJAXRSService"
    address="http://localhost:8182/Services/JAXRS/"
    transformer-refs="firstAppender"
    responseTransformer-refs="secondAppender"
    type="jax-rs">
    <component class="com.ivan.services.HelloService"/>
</pattern:simple-service>
...
...
```

Note that:

- The <simple-service> element belongs to the pattern namespace.
- There is an *address* attribute in the <simple-service> element.  
This attribute is used to specify the root URL of the JAX-RS resource. Thus, the greeting resource will have the URL <http://localhost:8182/Services/JAXRS/greeting/>.
- There is a *transformer-refs* attribute in the <simple-service> element.  
Using this attribute, we can specify the transformer(s) that are to be applied to messages received by the service, prior to messages being passed to the component of the service. The value of the attribute contains the names of the global transformers, separated by spaces.
- The <simple-service> element contains a *responseTransformer-refs* attribute.  
Similar to the transformer-refs attribute, this attribute also allows us to specify one or more transformer(s). These transformers are, however, to be applied to message produced by the component, that is the response message of the simple service.
- The <simple-service> element contains a *type* attribute.  
The value of the *type* attribute tells Mule whether to expose the simple service as a JAX-WS or JAX-RS web service. If omitted, the simple service will expose a plain endpoint using the

transport specified in the *address* attribute. We will see examples of this later.

- The <simple-service> has a <component> child element.  
This element is used to specify the class implementing the component that are to process messages received by the service.

## Run the JAX-RS Simple Service

We are now ready to try the JAX-RS simple service.

- Right-click the “mule-simpleservices-config.xml” Mule configuration file and select Run As -> Mule Server.
- In a web browser, enter the URL <http://localhost:8182/Services/JAXRS/greeting/James>.  
The name at the end of the URL may be replaced with arbitrary name. A greeting string similar to the following should be displayed in the browser:

```
Hello James, the time is now Wed Feb 29 07:01:14 CET 2012
```

Note that:

- Neither of the transformers configured on the simple service have appended their message to the greeting string we see in the browser.  
The transformers affect the payload of the message, but when the JAX-RS web service stack used by Mule analyzes the request, it uses values from Mule message headers. When the response is created, it is written to an output stream maintained in parallel with the regular message payload and the transformer does not affect the data of the stream.

## Simple Services and Inheritance

Using the inheritance mechanism available to Spring beans, we can declare a simple parent service which can be inherited from by other simple services.

In this part of the example, a simple parent service is configured with a custom exception strategy. A child simple service is also configured, to show that the exception strategy applies to children of the parent service.

In addition, we will use the test component, which is described in the [Testing](#) reference in the second part of this book, and see the transformers that we defined when creating the Mule configuration file in action.

## Modify the Mule Configuration File

We continue to build on the same Mule configuration file, “mule-simpleservices-config.xml”:

- Add the following to the contents of the Mule configuration file, immediately before the closing `<mule>` tag:

```
...
<!--
    Using the inheritance mechanism available for Spring beans,
    we can define an abstract simple-service which defines an
    exception strategy.
    This simple-service can then be inherited by concrete
    simple-service instances, which will all have the same
    exception strategy.
-->
<pattern:simple-service name="exceptionStrategyService" abstract="true">
    <default-exception-strategy>
        <!--
            The custom behaviour of the exception strategy consists
            of producing an extra line of log output.
        -->
        <logger level="ERROR" message="An exception occurred! "/>
    </default-exception-strategy>
</pattern:simple-service>

<!--
    This simple-service inherits from the above simple-service,
    adding properties necessary for a concrete service.
-->
<pattern:simple-service
    name="MySimpleRegularService"
    parent="exceptionStrategyService"
    address="http://localhost:8182/Services/Regular/"
    transformer-refs="firstAppender"
    responseTransformer-refs="secondAppender">
    <!--
        This test component will transform the message to its
        string representation and append a string to the message.
        Mule expressions may be used in the string to be appended.
    -->
    <test:component
        appendString="( processed by test component at #[function:system] )">
    </test:component>
</pattern:simple-service>
...
...
```

Note that:

- The simple service named “exceptionStrategyService” is abstract.  
The `abstract` attribute of the `<simple-service>` element has the value true.  
This mechanism is implemented by the Spring framework and is referred to as “Bean Definition Inheritance”. Please consult the Spring framework documentation for additional details.

- The simple service named “exceptionStrategyService” contains a <default-exception-strategy> child element. This element is used to define the exception strategy of the simple service. in this example, we use a default exception strategy which sends any messages it receives to a <logger> component.
- The next <simple-service> element, named “MySimpleRegularService”, uses the *parent* attribute with the value “exceptionStrategyService”. This indicates that the simple service “MySimpleRegularService” is to inherit from the “exceptionStrategyService”. Again, this mechanism is implemented by the Spring framework.
- The *address*, *transformer-refs* and *responseTransformer-refs* attributes of the simple service “MySimpleRegularService” have the same functions as described in the [previous part](#) of this example; specifying the address of the service's endpoint and adding one or more transformations before and after the simple service component.
- The simple service “MySimpleRegularService” contains a <test:component> element. This test component is described in detail in the [Testing](#) section of the reference part of this book. In this particular configuration, the test component appends the string specified by the *appendString* attribute to the message payload. We will soon modify the test component as to throw an exception each time it receives a message.

## Run the Simple Service with Inheritance

We are now ready to try the simple service with inheritance.

Note that all the examples related to simple services are located in one and the same Mule configuration file and will thus be started at the same time. The URL used to access the service endpoints is what differentiates between the individual examples.

- Right-click the “mule-simpleservices-config.xml” Mule configuration file and select Run As -> Mule Server.
  - In a web browser, enter the URL <http://localhost:8182/Services/Regular/test>.
- A string similar to the following should be displayed in the browser:

```
/Services/Regular/test[first appender msg]( processed by test component at
1330579939908 )[second appender msg]
```

In the console log, we can also see the following output from the test component:

```
INFO 2012-03-01 06:32:19,908 [connector.http.mule.default.receiver.02]
org.mule.tck.functional.FunctionalTestComponent:
*****
* Message Received in service: MySimpleRegularService. Content is: *
* /Services/Regular/test[first appender msg]*
*****
```

Note that:

- The first appender, specified in the *transformer-refs* attribute of the simple service, has appended its string prior to the message was processed by the service's component. The console output also confirms this.
- The test component of the service has appended a string to the message.
- Finally, the second appender, specified in the *responseTransformer-refs* attribute of the

simple service, has appended its string after the message was processed by the test component.

We see that the transformers are applied in the order expected and the test component also seem to do its job. This is all good and well, but so far, we have not been able to verify that inheritance between simple services does indeed work.

In order to do this, we need to reconfigure the test component to throw an exception every time it receives a message.

- Stop the Mule server running the example program.
- Modify the test component in the simple service to look like this (changes highlighted):

```
... <test:component  
    throwException="true"  
    appendString="( processed by test component at #[function:systime] )">  
</test:component>  
...
```

- Start the Mule server by right-clicking the “mule-simpleservices-config.xml” configuration file and select Run As -> Mule Server.
- Once again, in a web browser, enter the URL <http://localhost:8182/Services/Regular/test>. This time, the following string should be displayed in the browser:

```
Component that caused exception is:  
DefaultJavaComponent{MySimpleRegularService.commponent}. Message payload is of type:  
String
```

- Examining the console output, we can see (parts omitted for clarity):

```
ERROR 2012-03-01 06:51:23,704 [connector.http.mule.default.receiver.02]  
org.mule.exception.DefaultMessagingExceptionStrategy:  
*****  
Message : Functional Test Service Exception  
Code : MULE_ERROR--2  
-----  
Exception stack is:  
1. Functional Test Service Exception (org.mule.tck.exceptions.FunctionalTestException)  
   org.mule.tck.functional.FunctionalTestComponent:182  
(http://www.mulesoft.org/docs/site/current3/apidocs/org/mule/tck/exceptions/FunctionalTestException.html)  
-----  
Root Exception stack trace:  
...  
*****  
  
ERROR 2012-03-01 06:51:23,705 [connector.http.mule.default.receiver.02]  
org.mule.exception.DefaultMessagingExceptionStrategy: Message being processed is:  
/Services/Regular/test\[first appender msg\]  
ERROR 2012-03-01 06:51:23,759 [connector.http.mule.default.receiver.02]  
org.mule.api.processor.LoggerMessageProcessor: An exception occurred!
```

Note that:

- A *FunctionalTestException* has occurred in an instance of *FunctionalTestComponent* (highlighted in green).
- We can see the payload of the message that was being processed when the exception occurred (highlighted in yellow).
- The logger defined in the default exception strategy of the parent simple service has written its message to the console (highlighted in blue).

## JAX-WS Simple Service

In this next part of the simple service example, we will expose the greeting service implemented [earlier](#) as a JAX-WS SOAP web service.

The differences between exposing a JAX-WS and a JAX-RS web service using the Mule simple service pattern are:

- The *type* attribute in the <simple-service> element will have the value “jax-ws”. Recall that the *type* attribute had the value “jax-rs” in our earlier example.
- The component implementation class must be annotated with at least the @WebService JAX-WS annotation.  
The component implementation class created for the [JAX-RS simple service example](#) is already annotated with the @WebService annotation, so we can use it without changes for this example too.

## Modify the Mule Configuration File

With the component implementation class in place, we are ready to add yet another simple service configuration to our Mule configuration file.

- Add the following to the “mule-simpleservices-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
This simple-service exposes a SOAP web service endpoint
using JAX-WS.
Transformers does not affect SOAP messages.
It will, however, garble the WSDL by appending a text
string to it, rendering it invalid.
-->
<pattern:simple-service
    name="MySimpleSOAPService"
    parent="exceptionStrategyService"
    address="http://localhost:8182/Services/SOAP"
    transformer-refs="firstAppender"
    responseTransformer-refs="secondAppender"
    type="jax-ws">
    <component class="com.ivan.services.HelloService"/>
</pattern:simple-service>
...
```

Note that:

- The astute reader may notice that the comments in the above XML fragment state that the transformers configured on the simple service does not affect SOAP messages going in and out of the service, but cause the WSDL to become invalid. Nevertheless, these are included in order for us to see the consequences first-hand.
- The *type* attribute in the <simple-service> element has the value “jax-ws”. This is according to the introductory note of this part of the simple service example.

## Run the JAX-WS Simple Service

We can now try the JAX-WS simple service.

- Right-click the “mule-simpleservices-config.xml” Mule configuration file and select Run As -> Mule Server.
- In a web browser, enter the URL <http://localhost:8182/Services/SOAP?wsdl>.  
The result should be an error that is conveyed differently, depending on the browser you use.  
In Firefox, it looks like this:

```
XML Parsing Error: junk after document element
Location: http://localhost:8182/Services/SOAP?wsdl
Line Number 51, Column 20:

</wsdl:definitions>[second appender msg]
-----^
```

Error message trying to access the WSDL of the JAX-WS simple service in Firefox.

Note that:

- The string after the <wsdl:definitions> element end tag is the message from the second transformer configured on the simple service.

In order to avoid corrupting the WSDL, we remove the response transformer on the simple service. The resulting <simple-service> element looks like this:

```
...
<pattern:simple-service
    name="MySimpleSOAPService"
    parent="exceptionStrategyService"
    address="http://localhost:8182/Services/SOAP"
    transformer-refs="firstAppender"
    type="jax-ws">
    <component class="com.ivan.services.HelloService"/>
</pattern:simple-service>
...
```

Save the modified Mule configuration file, stop the Mule server and start it again. This time the WSDL should appear in the browser.

If we send a request to the service using, for instance, soapUI, we will be able to see that the transformer we have configured on the simple service does not affect the message.

## JAXB Simple Service

The next simple service example will show how to use JAXB to unmarshal XML data in a request sent to a simple service.

### Create JAXB Classes

The JAXB classes are to be generated using the Dali Java Persistence Tools, bundled with Eclipse starting with the Indigo release.

Before being able to generate the JAXB classes, we need an XML schema:

- In the package *com.ivan.schemas*, create a file named “PersonSchema.xsd” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ivan="http://www.ivan.com/schemas"
  targetNamespace="http://www.ivan.com/schemas"
  attributeFormDefault="unqualified">

  <element name="person">
    <complexType>
      <sequence>
        <element name="firstName" type="string" nillable="false"/>
        <element name="lastName" type="string" nillable="false"/>
        <element name="age" type="int"/>
      </sequence>
    </complexType>
  </element>
</schema>
```

The Dali Persistence Tools generator will, using the above XML schema, generate the JAXB bean classes used in this example.

- Right-click the file PersonSchema.xsd and select Generate → JAXB Classes....
- In the first dialog, enter *com.ivan.jaxb\_generated* in the Package field.
- Click the Next button.
- In the second dialog, check the “Treat input as XML schema” checkbox.
- Click the Finish button.
- Refresh the project in Eclipse.

If we now look in the *com.ivan.jaxb\_generated* package, there should be three files; ObjectFactory.java, package-info.java and Person.java.

### Create Example Data

To be able to test the JAXB simple service, we need some example data to send to it. The following XML fragment adheres to the PersonSchema XML schema created above.

- In the “example-data” directory in the root of the project, create a file named “person.xml” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<ivan:person
  xmlns:ivan="http://www.ivan.com/schemas"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ivan.com/schemas schema2.xsd ">
  <firstName>Stiv</firstName>
  <lastName>Bator</lastName>
  <age>32</age>
```

```
</ivan:person>
```

The contents of this file is to serve as the payload data when a request is sent to the JAXB simple service using, for instance, soapUI.

## Create the Component Implementation Class

If we are to use JAXB to unmarshal XML data received with requests, a couple of Mule-specific annotations are required. We will also use the generated *Person* class.

- In the package com.ivan.services, create the *JAXBConsumerService* class with the following implementation:

```
package com.ivan.services;

import java.util.Map;
import org.mule.api.annotations.param.InboundHeaders;
import org.mule.api.annotations.param.Payload;
import com.ivan.jaxb_generated.Person;

/**
 * Service that consumes XML data serialized to JAXB beans.
 * The JAXB generated classes need to be on the classpath and
 * any root elements must be annotated with @XmlElement.
 *
 * @author Ivan A Krizsan
 */
public class JAXBConsumerService
{
    public String greet(@Payload final Person inPerson,
                        @InboundHeaders("*") Map inRequestHeaders)
    {
        String theGreeting = "Hello, ";

        theGreeting += inPerson.getFirstName();
        theGreeting += "!";
        System.out.println("**** JAXBConsumerService: " + inRequestHeaders);
        System.out.println("**** JAXBConsumerService: " +
                           inPerson.getFirstName() + " " + inPerson.getLastName());

        return theGreeting;
    }
}
```

Note that:

- The *inPerson* parameter of the *greet* method is of the type *Person*. *Person* is one of the classes generated from the XML schema using the JAXB schema compiler.
- The *inPerson* parameter of the *greet* method is annotated with the *@Payload* annotation. This annotation is a Mule-specific annotation that tells Mule to try to transform request message payload to the annotated type using internal transformers, of which JAXB transform is one.
- The *inRequestHeaders* parameter of the *greet* method is annotated with the *@InboundHeaders* annotation. This annotation is also Mule-specific, telling Mule to inject the headers received with the request message into the parameter. The type of the parameter must be *Map*. The value in the annotation may be a comma-separated list of names of headers to inject into the parameter map or, as in the above example, “\*” which will result in all headers being injected into the parameter map. In this example, the *inRequestHeaders* map will receive the HTTP headers, since we will use HTTP to send requests to the simple service.

## Modify the Mule Configuration File

As we will soon see, nothing special is needed in the Mule configuration file in connection to the JAXB simple service. We just configure a simple service to use the component implementation class created earlier.

- Add the following simple service configuration to the “mule-simpleservices-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
    This simple-service uses JAXB to unmarshal the XML data
    being the payload incoming messages.
-->
<pattern:simple-service
    name="MySimpleJAXBService"
    address="http://localhost:8182/Services/JAXB">
    <component class="com.ivan.services.JAXBConsumerService"/>
</pattern:simple-service>
...
```

Note that:

- There is nothing special about this simple service, except for names, that could lead us to believe that it uses JAXB to unmarshal request payload data.

## Run the JAXB Simple Service

We can now start the JAXB simple service.

- Right-click the “mule-simpleservices-config.xml” Mule configuration file and select Run As -> Mule Server.

In order to be able to send a request to the JAXB simple service, I have chosen to use soapUI. Any tool that is able to send a HTTP POST request with a XML payload will do just fine.

The following steps describe how to send a request from soapUI to the JAXB simple service:

- Start soapUI.
- In soapUI, select New soapUI Project from the File Menu.
- In the New soapUI Project dialog, enter any name for the project.
- In the New soapUI Project dialog, check the checkbox Add REST Service (Opens dialog to create REST service).
- Click the OK button.
- In the New REST Service dialog that appears, enter the URL <http://localhost:8182/Services> in the Service Endpoint input field, ensure that the Create Resource (Opens dialog to create a REST resource) checkbox is checked and click the OK button.
- In the New REST Resource dialog that appears, enter “JAXB” in the Resource Name and Resource Path/Endpoint input fields and click the OK button.
- In the New REST Method dialog that appears ensure that the HTTP Method POST is selected and click the OK button.
- Paste the contents of the person.xml example data file in the lower left text box in the request window that is opened.

- Click the small green arrow that points right in the upper left corner of the request window to send the request to the service.
- Examine the text box on the right in the request window.  
The greeting string “Hello, Stiv!” should appear.
- Examining the console, the following output should have appeared:

```
...
INFO 2012-03-02 16:09:05,790 [connector.http.mule.default.receiver.02]
org.mule.module.xml.transformer.jaxb.JAXBContextResolver: No common Object of type
'class javax.xml.bind.JAXBContext' configured, creating a local one for:
SimpleDataType{type=org.apache.commons.httpclient.ContentLengthInputStream,
mimeType='application/xml'}, SimpleDataType{type=com.ivan.jaxb_generated.Person,
mimeType='*/*'}
*** JAXBConsumerService: {http.request=/Services/JAXB, Host=localhost:8182, Content-
Length=318, http.method=POST, User-Agent=Jakarta Commons-HttpClient/3.1,
http.request.path=/Services/JAXB, MULE_ORIGINATING_ENDPOINT=endpoint.http.localhost.8182.Services.JAXB, Connection=false,
http.context.path=/Services/JAXB, http.version=HTTP/1.1, Accept-Encoding=gzip,deflate, MULE_REMOTE_CLIENT_ADDRESS=/127.0.0.1:53350, Keep-Alive=false, Content-
Type=application/xml}
*** JAXBConsumerService: Stiv Bator
```

Note that:

- The HTTP request headers have been printed to the console (highlighted in green).
- The first and last name from the request payload has been printed to the console (highlighted in red).

We can draw the conclusion that the request payload has indeed been unmarshalled using JAXB and the generated JAXB bean class.

## XPath Simple Service

The final simple service example shows how one or more XPath expressions can be used to extract data from a XML payload in a request. The example also shows how to use a global component, a Spring bean, as the component of a simple service.

Note that the use of a global namespace manager in this example require the use of Mule 3.2 or later.

### Create the Component Implementation Class

The XPath simple service also require a custom component implementation class since, as with the JAXB simple service, we will make use of custom annotations to insert the XPath expressions to use in the Java code.

- In the package `com.ivan.services`, create the `XPathConsumerService` class with the following implementation:

```
package com.ivan.services;

import org.mule.api.annotations.expression.XPath;

/**
 * Service that consumes XML data using XPath expressions to
 * extract data from a request.
 *
 * @author Ivan A Krizsan
 */
public class XPathConsumerService
{
    public String greet(
        @XPath(value="/ivan:person/firstName") final String inFirstName,
        @XPath(value="/ivan:person/lastName") final String inLastName)
    {
        String theGreeting = "Hello, " + inFirstName + "!";
        System.out.println("**** XPathConsumerService: " +
                           inFirstName + " " + inLastName);
        return theGreeting;
    }
}
```

Note that:

- Both the parameters of the `greet` method are annotated with the `@XPath` annotation. This annotation is a Mule-specific annotation that enables XPath expressions to be executed on an XML payload and the result to be injected into the annotated variable.
- Both the `@XPath` annotations contain a `value` element that contains an XPath expression. The first XPath expression retrieves the contents of the `<firstName>` element in the `<person>` element and the second XPath expression retrieves the contents of the `<lastName>` element, also in the `<person>` element.

## Modify the Mule Configuration File

As with the JAXB simple service, there is nothing in the Mule configuration file that indicates that we use XPath to extract data from the XML data of incoming messages.

In addition to the simple service configuration, there is also the definition of a global component, which is a plain Spring bean, and a global namespace manager.

The use of this kind of namespace manager require Mule 3.2 or later.

- Add the following simple service configuration to the “mule-simpleservices-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
    Global namespace manager.
    In this example, the namespace manager is needed to define
    the namespace prefix used in XPath expressions in
    the simple service that has method parameters annotated
    with the @XPath annotation.
-->
<mule-xml:namespace-manager includeConfigNamespaces="false">
    <mule-xml:namespace
        prefix="ivan" uri="http://www.ivan.com/schemas" />
</mule-xml:namespace-manager>

<!-- Global component. -->
<spring:bean
    name="xpathConsumer"
    class="com.ivan.services.XPathConsumerService"/>

<!-- XPath simple service. -->
<pattern:simple-service
    name="MySimpleXPathService"
    address="http://localhost:8182/Services/XPath"
    component-ref="xpathConsumer">
</pattern:simple-service>
...
...
```

Note that:

- There is nothing special about this simple service, except for names, that could lead us to believe that it uses XPath to extract information from the request payload data.
- There is a <mule-xml:namespace-manager> element.  
As we saw in [chapter 6](#), this element is used to declare a global namespace manager. In this example, the namespace manager associates the namespace prefix “ivan” with the XML namespace “<http://www.ivan.com/schemas>”.
- There is a <spring:bean> element.  
This element contains a Spring bean definition. This is the global component that we will use as the component of the XPath simple service.
- Finally, there is a simple service definition.  
The simple service definition is similar to earlier simple service examples, except for the fact that it contains a *component-ref* attribute.  
This attribute is used to refer to the Spring bean that is to serve as the component of the simple service.

## Run the XPath Simple Service

The XPath simple service example is started in the same manner as the earlier simple service examples:

- Right-click the “mule-simpleservices-config.xml” Mule configuration file and select Run As -> Mule Server.

To use soapUI to send a request with an XML payload to the XPath simple service, use the steps described in the JAXB Simple Service example [earlier](#), but replace the “JAXB” in the Resource Name and Resource Path/Endpoint with “XPath” in the New REST Resource dialog.

## 13.4. The Validator Pattern

The validator pattern is very useful when developing a service proxy or facade that only allows messages meeting certain criteria to pass through.

An example is validation of messages received by a service using XML over HTTP (not SOAP). A validator can ensure that the messages that are allowed to pass further into the system for processing can be validated against some XML schema.

A validator receives messages synchronously and will, depending on the message, respond with a positive or negative acknowledge message to the client. Messages meeting the filtering criteria of the validator are passed on asynchronously. This enables the client to be informed of whether the request passed validation and the underlying processing logic to handle the request at some later time.

The example in this section will showcase the different options of the validator pattern in an example that receives a HTTP request with a number of parameters, only accepting requests that contain a certain parameter with a certain value.

- As a preparation for the validator examples, create a Mule configuration file named “mule-validator-config.xml” in the *com.ivan.muleconfig* package with the following contents:

```
<?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:pattern="http://www.mulesoft.org/schema/mule/pattern"
      xmlns:http="http://www.mulesoft.org/schema/mule/http"
      xmlns:test="http://www.mulesoft.org/schema/mule/test"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd

          http://www.mulesoft.org/schema/mule/pattern
          http://www.mulesoft.org/schema/mule/pattern/3.2/mule-pattern.xsd

          http://www.mulesoft.org/schema/mule/http
          http://www.mulesoft.org/schema/mule/http/3.2/mule-http.xsd

          http://www.mulesoft.org/schema/mule/test
          http://www.mulesoft.org/schema/mule/test/3.2/mule-test.xsd">

    <!--
        Endpoint receiving requests that are to be validated.
        This particular endpoint receives HTTP requests and extracts
        HTTP request parameters from the URL to a map.
        Note that this endpoint uses the request-response message
        exchange pattern, in order to be able to deliver a ACK or
        NACK as the result of the request.
        The endpoint is a global endpoint.
    -->
    <http:endpoint
        name="httpReceiver1"
        address="http://localhost:8080/validator1"
        exchange-pattern="request-response">
        <!--
            The body-to-parameter-map transformer transforms a HTTP
            request message to a map containing all the request
            parameters and their values.
        -->
        <http:body-to-parameter-map-transformer/>
    </http:endpoint>

    <!--
        Global filter that can be used by validators to determine
        whether a message will be accepted by the validator or not.
        An OGNL expression tests for the presence of the key-value
        pair with the key "data" and the value "123" in the map
        being the message payload.
    -->
```

```

<expression-filter
    name="globalData123Filter"
    evaluator="ognl"
    expression="data=='123' />

<!--
    This flow contains the endpoint that is to receive messages that
    have passed validation.
-->
<flow name="goodMessageFlow">
    <inbound-endpoint
        address="vm://goodMessageService"
        exchange-pattern="one-way"/>
    <test:component logMessageDetails="true"/>
</flow>

</mule>

```

Note that:

- The Mule configuration contains a global HTTP endpoint.  
This is the first of the two HTTP endpoints that are to receive requests. The endpoint address can be found in the *address* attribute of the `<http:endpoint>` element. The exchange pattern is request-response, as specified by the *exchange-pattern* attribute of the `<http:endpoint>` element.
- The global HTTP endpoint contains a transformer.  
The body-to-parameter-map-transformer transforms HTTP requests to a map that contains the parameters from the HTTP URL and their corresponding values.
- The Mule configuration contains a global filter declaration.  
The expression-filter, named “globalData123Filter”, applies an OGNL expression to the message payload and accepts messages which cause the result of the evaluation to be true.
- The Mule configuration contains a flow named “goodMessageFlow”.  
This flow exposes an inbound-endpoint that will receive messages that have passed validation. Incoming messages are logged using the test component. For details on the test component, please consult the reference section on [Testing](#).

Both the validator examples in this section require only Mule configuration and no additional resources.

## First Validator Example

The first validator example will show basic validator configuration, using mostly attributes of the <pattern:validator> element. The filter that determines whether the validator will accept a message or not will, however, be configured using a child element of the <pattern:validator> element, since we will use an attribute to refer to a global filter in the second example.

### Modify the Mule Configuration File

The first validator example consists of a single validator definition in the Mule configuration file:

- Add the following to the “mule-validator-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
A validator receives synchronous requests and apply a filter
to the request message payloads.
If a message matches the filter, then it is forwarded in an
asynchronous manner to the outbound address/endpoint of the
validator and an acknowledge message is returned to the client.
If a message does not match the filter, a negative acknowledge
message will be returned to the client and the message will not
be forwarded.
-->
<pattern:validator name="receiverValidator"
    inboundEndpoint-ref="httpReceiver1"
    ackExpression="#[string:ACK good message: #[message:payload]]"
    nackExpression="#[string:NACK bad message: #[message:payload]]"
    outboundAddress="vm://goodMessageService">
<!--
    Validator-local filter that determines whether a message
    will be accepted by the validator or not.
    Note that, due to a bug in Mule, a filter declared using
    a child element cannot be used in combination with
    an outbound endpoint declared as a child element.
-->
<expression-filter evaluator="ognl" expression="data=='123'"/>
</pattern:validator>
...
```

Note that:

- The <pattern:validator> element contains an *inboundEndpoint-ref* attribute. Using this attribute, we specify the inbound endpoint of the validator to be the global HTTP endpoint defined earlier in the Mule configuration file. This is the endpoint over which this validator receives incoming messages. Alternatives to the *inboundEndpoint-ref* attribute are the *inboundAddress* attribute and an <inbound-endpoint> child element. The former can be used when a simple endpoint address is sufficient, while the latter also allows for more detailed configuration of the validator's inbound endpoint.
- The <pattern:validator> element contains an *ackExpression* and a *nackExpression* attribute. Using these attributes, we specify the expressions that creates positive and negative acknowledge messages that are returned to a client depending on whether a request is accepted by the validator or not.
- The <pattern:validator> element contains an *outboundAddress* attribute. Using this attribute, the endpoint to which messages that are accepted by the validator are sent may be specified. The endpoint which address is given in this attribute must use the one-way message exchange pattern.

- The <pattern:validator> element contains an <expression-filter> child element. This filter is what determines whether a message received by the validator will be accepted, and forwarded, or rejected and discarded. In this example we use an OGNL expression that checks whether a property of the message with the name “data” has the value 123.

## Run the First Validator Example

With the above Mule configuration in place, we can now test the first validator example.

- Right-click the “mule-validator-config.xml” Mule configuration file and select Run As -> Mule Server.
- In a local web browser, enter the following URL: <http://localhost:8080/validator1?data=0>  
In the browser, you should see the following response:  
NACK bad message: {data=0}  
No additional console output should have been written.
- Now enter the following URL in the browser: <http://localhost:8080/validator1?data=123>  
In the browser, the following should appear:  
ACK good message: {data=123}  
In the console, output similar to the following should have appeared (MULE\_SESSION property data shortened):

```

INFO 2012-03-22 06:47:24,057 [goodMessageFlow.stage1.03]
org.mule.tck.functional.FunctionalTestComponent:
*****
* Message Received in service: goodMessageFlow. Content is: {data=123} *
*****
INFO 2012-03-22 06:47:24,058 [goodMessageFlow.stage1.03]
org.mule.tck.functional.FunctionalTestComponent: Full Message payload:
{data=123}

Message properties:
INVOCATION scoped properties:
INBOUND scoped properties:
MULE_ENCODING=UTF-8
MULE_ENDPOINT=vm://goodMessageService
MULE_ORIGINATING_ENDPOINT=endpoint.vm.goodMessageService
MULE_SESSION=r00AB...
OUTBOUND scoped properties:
MULE_CORRELATION_GROUP_SIZE=-1
MULE_CORRELATION_SEQUENCE=-1
MULE_ENCODING=UTF-8
SESSION scoped properties:

```

Note that:

- We received the positive acknowledge message only when the parameter “data”, with the value 123, was appended to the URL.
- The message was only logged when the parameter “data”, with the value 123, was appended to the URL.  
This means that only messages containing this entry in the map holding the message were forwarded to the goodMessageFlow.

## **Second Validator Example**

The second validator example will show validator inheritance, similar to what has been shown for the previous configuration patterns and the possibility to configure an exception strategy for a validator. We will also see how an inbound and outbound endpoint of a validator can be configured as child elements of the <pattern:validator> element.

Theoretically, it should be possible to configure both the inbound and outbound endpoints as well as the filter of a validator as child elements of the <pattern:validator> element. Regretfully, a bug in Mule 3.2.1 prevents this. This is the reason for configuring the filter as a child element in the previous validator example and using an attribute to refer to a global filter in this example.

### **Modify the Mule Configuration File**

The second validator example consists of two validator definitions in the Mule configuration file; the first being an abstract validator and the second being a concrete validator that inherits from the first validator:

- Add the following to the “mule-validator-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
    Validators can be abstract, specifying properties that can
    be inherited by multiple child validators.
-->
<pattern:validator name="exceptionStrategyValidator" abstract="true">
    <!--
        An exception strategy may be declared in the validator,
        handling exceptions occurring while messages are passing
        through the validator.
    -->
    <default-exception-strategy>
        <logger level="ERROR" message="An exception occurred! "/>
    </default-exception-strategy>
</pattern:validator>

<!--
    This validator shows the following features of validators:
    - This validator inherits from an abstract validator and,
      in this particular case, inherits an exception strategy.
    - The inbound endpoint of a validator can be specified
      using a child element of the <validator> element.
    - The outbound endpoint of a validator can be specified
      using a child element of the <validator>.
-->
<pattern:validator name="receiverValidator2"
    parent="exceptionStrategyValidator"
    validationFilter-ref="globalData123Filter"
    ackExpression="#[string:ACK good message: #[message:payload]]"
    nackExpression="#[string:NACK bad message: #[message:payload]]">
    <!--
        Validator-local declaration of the endpoint on which the
        validator receives messages.
    -->
    <http:inbound-endpoint
        host="localhost"
        port="8080"
        path="validator2"
        exchange-pattern="request-response">
        <http:body-to-parameter-map-transformer/>
    </http:inbound-endpoint>
    <!--
        The outbound endpoint on which the validator forwards
        accepted messages can also be declared as a child element.
        Note that, due to a bug in Mule, an outbound endpoint
        declared using a child element cannot be used in combination
        with a filter declared as a child element.
    -->
```

```
<outbound-endpoint address="vm://goodMessageService" />
</pattern:validator>
...
```

Note that:

- The first validator, named “exceptionStrategyValidator”, is abstract.  
As with earlier configuration pattern examples, this uses the Spring bean inheritance mechanism, which allows us to define an abstract validator that specifies common properties inherited by one or more validators inheriting from the abstract validator.
- The abstract validator contains a <default-exception-strategy> element.  
An exception strategy can be configured for a validator, specifying how exceptions that occur during message processing in the validator are handled.
- The second validator, named “receiverValidator2”, contains the *parent* attribute.  
This attribute is used to specify any parent validator from which to inherit properties. In this example, the second validator inherits the exception strategy specified in the first validator.
- The second validator contains the *validationFilter-ref* attribute.  
This attribute is used to refer to a global filter definition, which will determine whether a message received by the validator will be accepted, and forwarded, or rejected and discarded.
- As seen in the first validator example, the second validator's <pattern:validator> element contain an *ackExpression* and a *nackExpression* attribute.  
Using these attributes, we specify the expressions that creates positive and negative acknowledge messages that are returned to a client depending on whether a request is accepted by the validator or not.
- The second validator's <pattern:validator> element contains a <http:inbound-endpoint> element.  
This shows an alternative way to declare the inbound endpoint of the validator, which gives the opportunity to configure an endpoint that is specific for the validator in a more detailed way.
- The <http:inbound-endpoint> element contains a <http:body-to-parameter-map-transformer> element.  
The body-to-parameter-map-transformer transforms HTTP requests to a map that contains the parameters from the HTTP URL and their corresponding values.
- The second validator's <pattern:validator> element contains an <outbound-endpoint> element.  
This enable us to configure a validator-specific outbound endpoint and, for instance, apply one or more transformations to the messages that are sent out from the validator.

## Run the Second Validator Example

With the above Mule configuration in place, we can now test the second validator example.

- Right-click the “mule-validator-config.xml” Mule configuration file and select Run As -> Mule Server.
- In a local web browser, enter the following URL: <http://localhost:8080/validator2?data=2>  
In the browser, you should see the following response:  
NACK bad message: {data=2}  
No additional console output should have been written.
- Now enter the following URL in the browser: <http://localhost:8080/validator2?data=123>  
In the browser, the following should appear:  
ACK good message: {data=123}  
In the console, output similar to the following should have appeared (MULE\_SESSION property data shortened):

```
INFO 2012-03-22 16:49:09,584 [connector.http.mule.default.receiver.03] org.mule.lifecycle.AbstractLifecycleManager: Initialising: 'connector.VM.mule.default.dispatcher.761170597'. Obj is: VMMessagesDispatcher
INFO 2012-03-22 16:49:09,584 [connector.http.mule.default.receiver.03] org.mule.lifecycle.AbstractLifecycleManager: Starting: 'connector.VM.mule.default.dispatcher.761170597'. Obj is: VMMessagesDispatcher
INFO 2012-03-22 16:49:09,611 [goodMessageFlow.stage1.02] org.mule.tck.functional.FunctionalTestComponent:
*****
* Message Received in service: goodMessageFlow. Content is: {data=123} *
*****
INFO 2012-03-22 16:49:09,612 [goodMessageFlow.stage1.02] org.mule.tck.functional.FunctionalTestComponent: Full Message payload:
{data=123}

Message properties:
INVOCATION scoped properties:
INBOUND scoped properties:
    MULE_ENCODING=UTF-8
    MULE_ENDPOINT=vm://goodMessageService
    MULE_ORIGINATING_ENDPOINT=endpoint.vm.goodMessageService
    MULE_SESSION=r00A...
OUTBOUND scoped properties:
    MULE_CORRELATION_GROUP_SIZE=-1
    MULE_CORRELATION_SEQUENCE=-1
    MULE_ENCODING=UTF-8
SESSION scoped properties:
```

Note that:

- The second validator behaves in way identical to that of the first validator.
- We received the positive acknowledge message only when the parameter “data”, with the value 123, was appended to the URL.
- The message was only logged when the parameter “data”, with the value 123, was appended to the URL.  
This means that only messages containing this entry in the map holding the message were forwarded to the goodMessageFlow.

### Third Validator Example

In the third validator example, we will see how a validator can be configured to return error messages to client if the validator fails to send a valid message to the outbound endpoint of the validator.

### Modify the Mule Configuration File

The third validator example consists of one validator definition in the Mule configuration file.

- Add the following to the “mule-validator-config.xml” file, immediately before the closing `<mule>` tag:

```
...
<!--
    A validator can also be configured to return an error message if
    there was a problem sending a valid message on the outbound endpoint.
    If such an error message is configured, using the errorExpression
    attribute, the message exchange pattern between the validator and
    the outbound endpoint automatically becomes request-response.
-->
<pattern:validator name="receiverValidator3"
    validationFilter-ref="globalData123Filter"
    ackExpression="#[string:ACK good message: #[message:payload]]"
    nackExpression="#[string:NACK bad message: #[message:payload]]"
    errorExpression="#[string:An error occurred passing the message on. Message
payload: #[message:payload]]">
    <http:inbound-endpoint
        host="localhost"
        port="8080"
        path="validator3"
        exchange-pattern="request-response">
        <http:body-to-parameter-map-transformer/>
    </http:inbound-endpoint>
<!--
    A bad URL is intentionally used here to show what happens when
    the validator fails to send a valid message to the outbound
    endpoint.
-->
    <outbound-endpoint address="http://localhost:8123/badurl"/>
</pattern:validator>
...

```

Note that:

- The third validator is named “`receiverValidator3`”.
- The validator contains the `validationFilter-ref` attribute.  
This attribute is used to refer to a global filter definition, which will determine whether a message received by the validator will be accepted, and forwarded, or rejected and discarded.
- The validator contains an `ackExpression` and a `nackExpression` attribute.  
Using these attributes, we specify the expressions that creates positive and negative acknowledge messages that are returned to a client depending on whether a request is accepted by the validator or not.
- The validator contains an attribute named `errorExpression`.  
Using this attribute we specify the expressions that create error messages that are returned to a client if the validator fails to send a valid message on the outbound endpoint.
- The validator contains a `<http:inbound-endpoint>` child element.  
This element specifies the inbound endpoint on which the validator receives messages.
- The `<http:inbound-endpoint>` element contains a `<http:body-to-parameter-map-transformer/>` element.

The body-to-parameter-map-transformer transforms HTTP requests to a map that contains the parameters from the HTTP URL and their corresponding values.

- The validator contains an <outbound-endpoint> element.

As in the previous example, this is the outbound endpoint to which the validator forwards messages that have been accepted by the validator's filter.

The outbound endpoint in this example has deliberately been configured with an invalid address, so as to generate errors when trying to forward messages.

## Run the Third Validator Example

With the above Mule configuration in place, we can now test the third validator example.

- Right-click the “mule-validator-config.xml” Mule configuration file and select Run As -> Mule Server.
- In a local web browser, enter the following URL: <http://localhost:8080/validator3?data=3>  
In the browser, you should see the following response:  
NACK bad message: {data=3}  
No additional console output should have been written.
- Now enter the following URL in the browser: <http://localhost:8080/validator3?data=123>  
In the browser, the following should appear:  
An error occurred passing the message on. Message payload: {data=123}  
In the console, output similar to the following should have appeared:

```
ERROR 2012-03-26 16:35:27,386 [connector.http.mule.default.receiver.04]
org.mule.construct.Validator$ErrorAwareEventReturningMessageProcessor:
org.mule.api.transport.DispatchException: Failed to route event via endpoint:
DefaultOutboundEndpoint{endpointUri=http://localhost:8123/badurl,
connector=HttpConnector
{
    name=connector.http.mule.default
    lifecycle=start
    this=fa5e4e4
    numberOfConcurrentTransactedReceivers=4
    createMultipleTransactedReceivers=true
    connected=true
    supportedProtocols=[http]
    serviceOverrides=<none>
}
, name='endpoint.http.localhost.8123.badurl', mep=REQUEST_RESPONSE, properties={},
transactionConfig=Transaction{factory=null, action=INDIFFERENT, timeout=0},
deleteUnacceptedMessages=false, initialState=started, responseTimeout=10000,
endpointEncoding=UTF-8, disableTransportTransformer=false}. Message payload is of type:
HashMap
```

Note that:

- The validator did not forward the message that were not accepted by the validator's filter.  
This behaviour is the same as we have seen in the previous examples and should thus come as no surprise.
- The validator returned an error message when trying to forward a message accepted by the validator's filter.  
This is the expected behaviour, since we configured the outbound endpoint of the validator with an illegal address. We see that the mechanism indeed works as described in the introduction of this section.
- An error was logged when the validator tried to forward a message accepted by the validator's filter.  
This is the default exception strategy of the validator handing the exception that occurred when the validator tried to forward the message.

## 13.5. The Web Service Proxy Pattern

The final Mule configuration pattern is the web service proxy pattern. This pattern enables us to quickly set up a proxy for a web service. The proxy can apply one or more transformations, implement handling of errors that occur when invoking the web service behind the proxy as well as redirect requests for the WSDL of the web service behind the proxy.

The web service proxy example requires more elaborate preparations, since we will use soapUI to set up a SOAP web service that we are to create a proxy for.

### ***Example Preparations***

In preparing for the web service proxy example, we will:

- Create a skeleton Mule configuration file.
- Implement a custom transformer.
- Set up a WSDL for the web service we are to proxy.
- Create a soapUI mock service.

### **Create the Mule Configuration File**

First, let's create a Mule configuration file for the web service proxy examples.

- Create a Mule configuration file named “mule-wsproxy-config.xml” in the *com.ivan.muleconfig* package with the following contents:

```
<?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:mule-xml="http://www.mulesoft.org/schema/mule/xml"
      xmlns:pattern="http://www.mulesoft.org/schema/mule/pattern"
      xsi:schemaLocation="
          http://www.mulesoft.org/schema/mule/core
          http://www.mulesoft.org/schema/mule/core/3.1/mule.xsd

          http://www.mulesoft.org/schema/mule/xml
          http://www.mulesoft.org/schema/mule/xml/3.2/mule-xml.xsd

          http://www.mulesoft.org/schema/mule/pattern
          http://www.mulesoft.org/schema/mule/pattern/3.2/mule-pattern.xsd">

    <!-- Transformer that unpacks zipped data. -->
    <gzip-uncompress-transformer name="unzipTransformer"/>

    <!-- Transformer that logs messages and message payloads. -->
    <custom-transformer
        name="loggingTransformer"
        class="com.ivan.transformers.MessageLoggingTransformer"/>

</mule>
```

Note that:

- The Mule configuration file contains a transformer that unpacks zipped data.  
When working with this example, I noticed that soapUI compresses response data. This transformer unpacks such data as to make it human-readable.
- The Mule configuration file contains a custom transformer.  
We will shortly implement this transformer and see that it indeed does what the comments say; logs Mule messages and message payloads.

## Implement a Custom Logging Transformer

The custom transformer in this example is used to log Mule messages and message payloads to the console without altering them. The reason for implementing such a transformer is that the web service proxy allows for insertion of one or more transformers both on outbound and inbound messages. Using a logging transformer, we can very conveniently add logging wherever we want to.

- In the Eclipse project, create the source package *com.ivan.transformers*.
- In the package created above, implement a custom transformer in a class named *MessageLoggingTransformer* as this:

```
package com.ivan.transformers;

import org.mule.api.MuleMessage;
import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractMessageTransformer;

/**
 * Mule 3.x transformer that logs message that passes through
 * the transformer to the console. Messages are not altered in any way.
 *
 * @author Ivan Krizsan
 */
public class MessageLoggingTransformer extends AbstractMessageTransformer
{
    @Override
    public Object transformMessage(final MuleMessage inMessage,
        final String inOutputEncoding) throws TransformerException
    {
        try
        {
            System.out.println("***** Message: " + inMessage.toString());
            System.out.println("***** Message payload: " +
                inMessage.getPayloadAsString());
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
        return inMessage;
    }
}
```

Note that:

- The *MessageLoggingTransformer* class extends the *AbstractMessageTransformer*. *AbstractMessageTransformer* provides basic implementation of a transformer that has access to the current message. Please refer to the Mule 3.x API documentation for additional details.
- The *transformMessage* method logs message properties and the message payload.
- The message received by the *transformMessage* method is returned by the method. The message is unaltered.

## Create the Service WSDL

The WSDL created in this step will serve both as a specification for the soapUI mock service as well as a custom WSDL that we later will configure the web service proxy to respond with whenever the WSDL of the actual web service is requested.

- In the root of the project's source hierarchy (the “src” folder), create a file named “HelloService.wsdl” with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:helloWorld/sample/ivan/com"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="urn:helloWorld/sample/ivan/com"
    xmlns:types="urn:helloWorld/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    name="HelloWorld">

    <wsdl:types>
        <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:helloWorld/types" xmlns="urn:helloWorld/types">

            <xsd:element name="hello">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="name" nillable="true" type="xsd:string"/>
                        <xsd:element name="first_number" type="xsd:int"/>
                        <xsd:element name="second_number" type="xsd:int"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>

            <xsd:element name="helloResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="message" nillable="true" type="xsd:string"/>
                        <xsd:element name="sum" type="xsd:long"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>

    <wsdl:message name="helloRequestMsg">
        <wsdl:part element="types:hello" name="helloParameters"/>
    </wsdl:message>
    <wsdl:message name="helloResponseMsg">
        <wsdl:part element="types:helloResponse" name="helloResult"/>
    </wsdl:message>

    <wsdl:portType name="HelloWorld">
        <wsdl:operation name="hello">
            <wsdl:input message="tns:helloRequestMsg" name="helloRequest"/>
            <wsdl:output message="tns:helloResponseMsg" name="helloResponse"/>
        </wsdl:operation>
    </wsdl:portType>

    <wsdl:binding name="HelloWorldBinding" type="tns:HelloWorld">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <wsdl:operation name="hello">
            <soap:operation/>
            <wsdl:input name="helloRequest">
                <soap:body use="literal"/>
            </wsdl:input>
            <wsdl:output name="helloResponse">
                <soap:body use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="HelloWorldService">
        <wsdl:port name="port" binding="tns:HelloWorldBinding">
```

```
<soap:address location="http://www.someserver.com"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

This is an example of a request message that will be accepted by a service adhering to the above WSDL:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:typ="urn:helloWorld/types">
  <soapenv:Header/>
  <soapenv:Body>
    <typ:hello>
      <name>Ivan</name>
      <first_number>2</first_number>
      <second_number>3</second_number>
    </typ:hello>
  </soapenv:Body>
</soapenv:Envelope>
```

...and this is an example of a response message that a service adhering to the above WSDL may use to respond to a request:

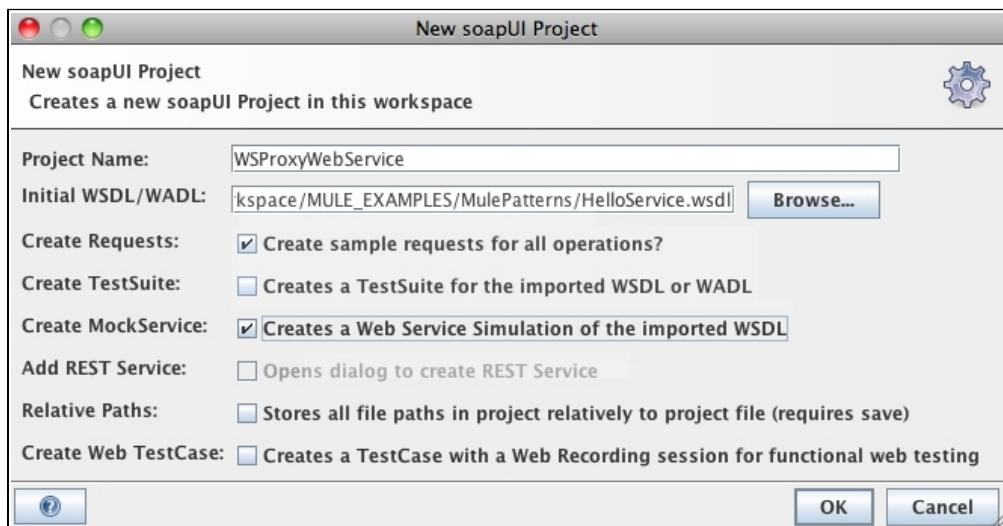
```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:typ="urn:helloWorld/types">
  <soapenv:Header/>
  <soapenv:Body>
    <typ:helloResponse>
      <message>Hello there!</message>
      <sum>5</sum>
    </typ:helloResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

## Create the Mock Service in soapUI

In this step we will create a mock service in soapUI and test it from within soapUI. Before proceeding, make sure you have downloaded, installed and started [soapUI](#).

- In soapUI, go to the File menu and select New soapUI Project.
- Fill in the new project dialog.

I chose the project name “WSProxyWebService”, which is not significant. Make sure that the checkboxes “Create requests” and “Create MockService” are checked. For the “Initial WSDL/WADL”, locate and select the “HelloService.wsdl” WSDL we created earlier.



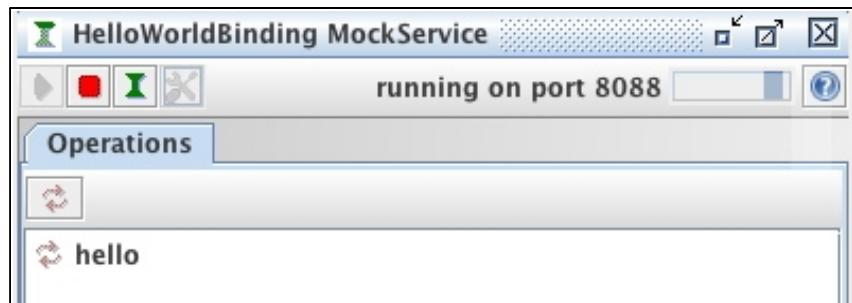
Creating the mock web service project in soapUI.

- Immediately after having created the new project, a Generate MockService dialog appears. In this dialog and the following Generate MockService dialog, just accept the default values and click OK.
- In the “HelloWorldBinding MockService” window that appears to the right, double-click the “hello” operation.
- In the “hello” window that appears, double-click the “Response 1” mock response.
- Modify the mock response.  
I chose to modify my mock response like this, changing the contents of the <message> and <sum> elements:

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:typ="urn:helloWorld/types">
    <soapenv:Header/>
    <soapenv:Body>
        <typ:helloResponse>
            <message>Hello there!</message>
            <sum>42</sum>
        </typ:helloResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

- Find the “HelloWorldBinding MockService” window and start the mock service by clicking the small green arrow in the upper left corner of the window.

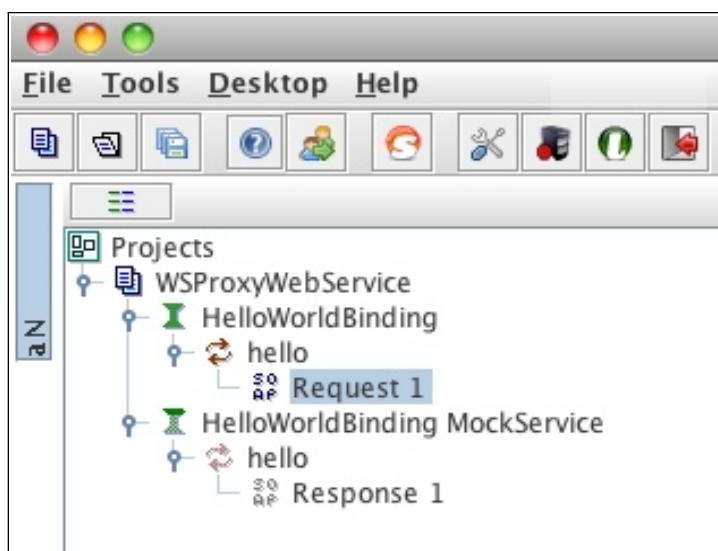
The green arrow should be disabled and there should be a message saying that the mock service is running.



Having started the mock service in soapUI.

The mock service should now be up and running. To confirm this, we'll test it from within soapUI:

- In the projects tree in soapUI, find and double-click the “Request 1” of the “hello” operation in the “HelloWorldBinding”.



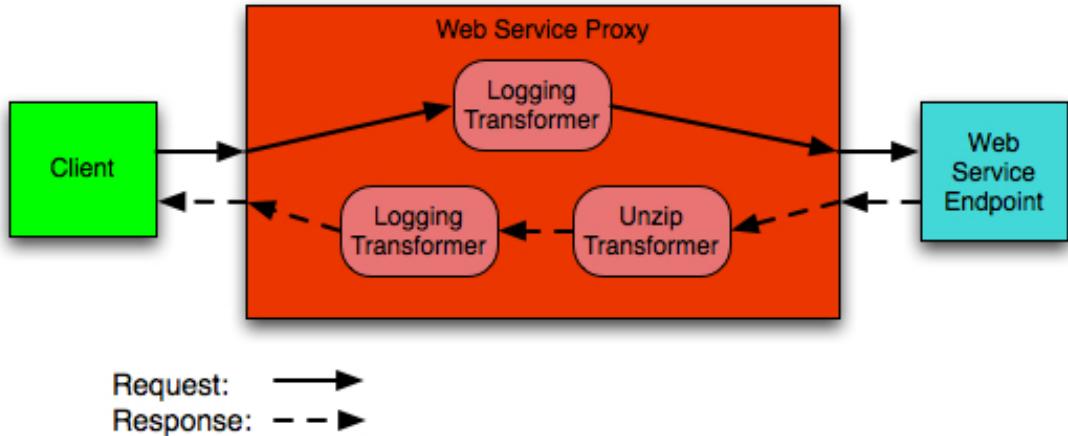
Opening the test-request for the “hello” operation in soapUI.

- Make sure that the URL at the top of the “Request 1” window contains the string “mockHelloWorldBinding”.  
If it hasn't, select the mock service's URL from the drop-down list. It should be something like “<http://localhost:8088/mockHelloWorldBinding>”, where localhost may be replaced with the name of your computer.
- In the “Request 1” window that appears, click the right triangle in the upper left corner of the window.  
This sends the test-request to the mock service.
- Also in the “Request 1” window, but on the right-hand side, examine the response of the request we just issued.  
It should be identical with the mock response we configured earlier.
- In a web browser, issue a request for the mock web service's WSDL.  
The URL should look something like <http://localhost:8088/mockHelloWorldBinding?wsdl>.  
You should see a WSDL that is almost identical to the one we created earlier.

We have now successfully set up the mock web service and managed to send requests directly to the mock service. After having developed the web service proxy, we will redirect requests sent from soapUI to the proxy, instead of sending them directly to the mock service.

### **First Web Service Proxy Example**

With the mock web service in place, we are now ready to develop our first web service proxy example. Its structure can be illustrated by following figure:



Flow of requests and responses in the first web service proxy example.

The client, green in the above figure, and the web service endpoint, blue in the above figure, are both realized using soapUI. The web service proxy, red in the above figure, is to be implemented using Mule.

## Modify the Mule Configuration File

The first web service proxy example consists of a single proxy definition in the Mule configuration file:

- Add the following to the “mule-wsproxxyx-config.xml” file, immediately before the closing `<mule>` tag:

```
...
<!--
    SOAP web service proxy that logs request and response messages
    that pass through the proxy.
    In addition, response messages are unpacked before being logged.
    The version of Mule 3.x used when developing this example adds
    the Accept-Encoding HTTP header with the value gzip, which causes
    the responses from soapUI to be zipped.
    If the response messages are garbled or if exceptions occur in the
    unzipTransformer, try removing this transformer from the list in the
    responseTransformer-refs attribute of the <pattern:web-service-proxy>
    element.
-->
<pattern:web-service-proxy
    name="helloWorldProxy1"
    transformer-refs="loggingTransformer"
    responseTransformer-refs="unzipTransformer loggingTransformer"
    inboundAddress="http://localhost:8090/helloWorld1"
    outboundAddress="http://localhost:8088/mockHelloWorldBinding" />
...
```

Note that:

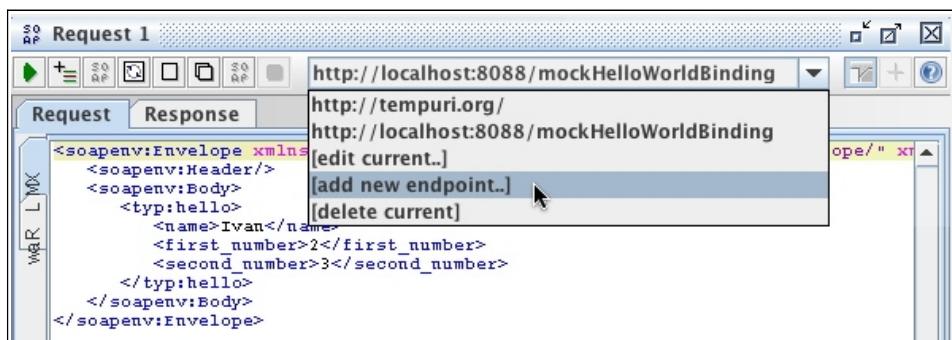
- The web service proxy is configured using one single `<pattern:web-service-proxy>` element in the Mule configuration file.
- The `<pattern:web-service-proxy>` element contains a *transformer-refs* attribute. The *transformer-refs* attribute is used to configure the transformer(s) that are applied to Mule messages before they are sent out from the proxy to the web service that is being proxied. If more than one transformer is to be applied, a space is to be inserted between the names of the transformers.
- The `<pattern:web-service-proxy>` element contains a *responseTransformer-refs* attribute. Similar to the *transformer-refs* attribute, the value of this attribute specify one or more transformers that are to be applied to Mule messages. However, the *responseTransformer-refs* attribute specify the transformer(s) that are to be applied to response messages from the proxied web service, before they are returned to the client that issued the corresponding request to the proxy.
- The *responseTransformer-refs* attribute specify two transformers. The first transformer is named “unzipTransformer” and the second is named “loggingTransformer”. These transformers are applied in the order in which they are listed, that is the “unzipTransfomrer” is applied first and then the “loggingTransformer” is applied. The reason for applying the “unzipTransformer” to response messages is that soapUI compresses response messages and they must be decompressed before being human-readable.
- The `<pattern:web-service-proxy>` element contains a *inboundAddress* attribute. This attribute specify the endpoint address at which the web service proxy will listen for requests.
- The `<pattern:web-service-proxy>` element contains a *outboundAddress* attribute.

The *outboundAddress* attribute specifies the endpoint address to which the web service proxy will forward incoming requests.

## Run the Web Service Proxy Example

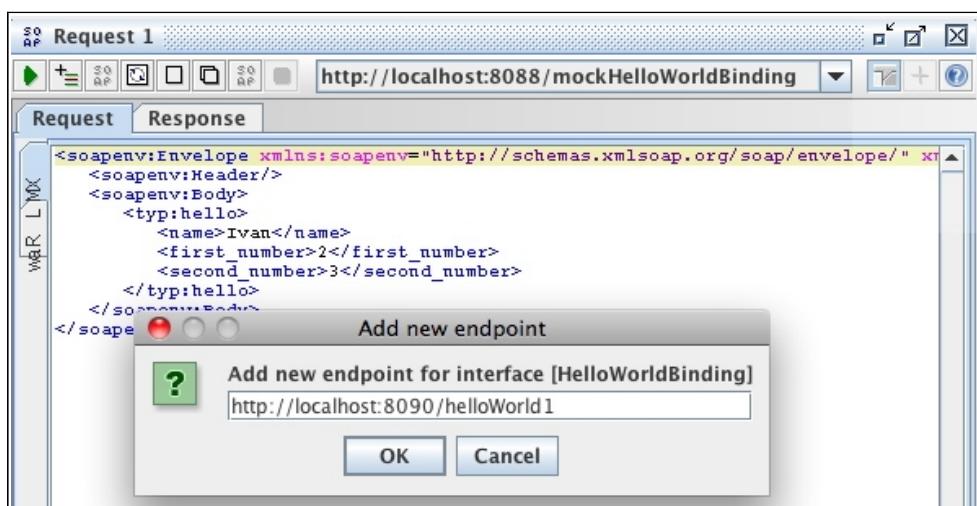
With the above Mule configuration in place, we can now test the first web service proxy example.

- Right-click the “mule-wsproxy-config.xml” Mule configuration file and select Run As -> Mule Server.
- Make sure that the mock service in soapUI is started.  
For details on setting up and starting the mock service in soapUI, please refer to the [Example Preparations](#) earlier.
- Open the “Request 1” that we used to test the mock service in the [Example Preparations](#) earlier.  
In the next few steps we are going to change the address to which requests are sent, so that they are sent to the web service proxy we just created, instead of directly to the mock service.
- Click the area which contains the URL in the Request 1 window and select “[add new endpoint..]”.



Adding a new endpoint which to send a request to in soapUI.

- In the Add New Endpoint dialog that appears, enter the URL <http://localhost:8090/helloWorld1> and click OK.



Specifying the URL of the new endpoint which to send requests in soapUI.

- Click the little green triangle in the upper-left corner of the Request 1 window. This will send the request to the web service proxy, which in turn will forward the request to the mock web service.
- You should see output similar to the following in the console:

```
...
***** Message:
org.mule.DefaultMuleMessage
{
    id=ecafed59-8010-11e1-869b-45fdb5b2e92c
    payload=org.apache.commons.httpclient.ContentLengthInputStream
    correlationId=<not set>
    correlationGroup=-1
    correlationSeq=-1
    encoding=utf-8
    exceptionPayload=<not set>

Message properties:
INVOCATION scoped properties:
INBOUND scoped properties:
    Accept-Encoding=gzip,deflate
    Connection=false
    Content-Type=text/xml; charset=utf-8
    Host=localhost:8090
    Keep-Alive=false
    MULE_ORIGINATING_ENDPOINT=endpoint.http.localhost.8090.helloWorld1
    MULE_REMOTE_CLIENT_ADDRESS=/127.0.0.1:49887
    User-Agent=Jakarta Commons-HttpClient/3.1
    http.context.path=/helloWorld1
    http.method=POST
    http.request=/helloWorld1
    http.request.path=/helloWorld1
    http.version=HTTP/1.1
OUTBOUND scoped properties:
    MULE_ENCODING=utf-8
SESSION scoped properties:
}
***** Message payload: <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:typ="urn:helloWorld/types">
<soapenv:Header/>
<soapenv:Body>
    <typ:hello>
        <name>Ivan</name>
        <first_number>2</first_number>
        <second_number>3</second_number>
    </typ:hello>
</soapenv:Body>
</soapenv:Envelope>
INFO 2012-04-06 19:49:58,013 [connector.http.mule.default.receiver.02]
org.mule.transport.service.DefaultTransportServiceDescriptor: Loading default outbound
transformer: org.mule
...
***** Message:
org.mule.DefaultMuleMessage
{
    id=ed0475fd-8010-11e1-869b-45fdb5b2e92c
    payload=[B
    correlationId=<not set>
    correlationGroup=-1
    correlationSeq=-1
    encoding=utf-8
    exceptionPayload=<not set>

Message properties:
INVOCATION scoped properties:
INBOUND scoped properties:
    Connection=false
    Content-Encoding=gzip
    Content-Type=text/xml; charset=utf-8
    Keep-Alive=false
    Server=Jetty(6.1.x)
    Transfer-Encoding=chunked
    http.method=POST
    http.request=http://localhost:8088/mockHelloWorldBinding
```

```

http.status=200
http.version=HTTP/1.1
OUTBOUND scoped properties:
MULE_ENCODING=utf-8
MULE_SESSION=r00A...
SESSION scoped properties:
}
***** Message payload: <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:typ="urn:helloWorld/types">
<soapenv:Header/>
<soapenv:Body>
<typ:helloResponse>
<message>Hello there!

```

Note that:

- The part of the log highlighted in blue is the request message.  
We can see that the SOAP request is identical to that we have seen in soapUI.
- The part of the log highlighted in green is log messages from Mule initializing its components.  
This part has been edited, to increase readability.
- The part of the log highlighted in orange is the response message.  
We can see that the SOAP response is identical to that we have configured the mock service in soapUI to respond with.
- We can read the XML of the SOAP response.  
If you are up to some experimentation, try removing the “unzipTransformer” from the *responseTransformer-ref*s attribute of the <pattern:web-service-proxy> element and examine the response.  
Conversely, if you cannot read the response message XML, then you should try to remove the “unzipTransformer”, since it may be that your response is not compressed.
- Mule session identifier data has been shortened, to avoid cluttering the output.

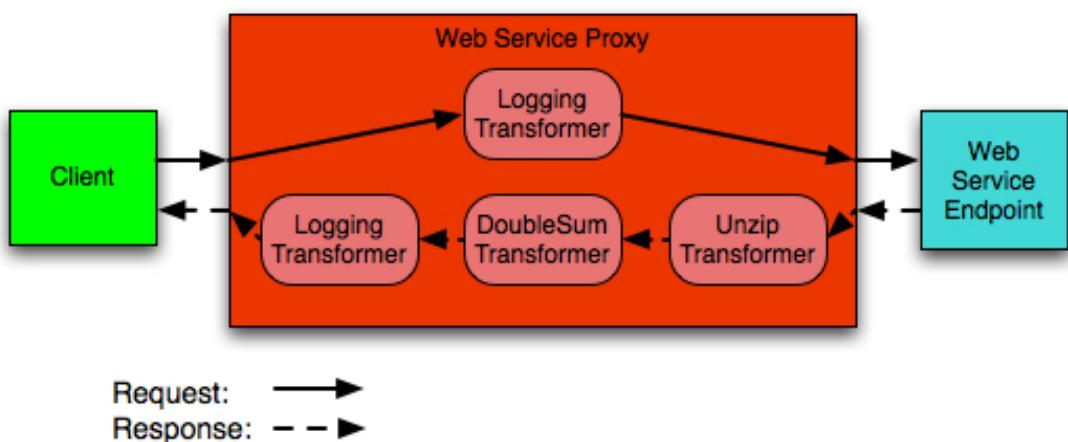
## Second Web Service Proxy Example

The second web service proxy example will showcase more features of the web service proxy pattern:

- Configuration of an exception strategy in a web service proxy.
- Inheritance and abstract web service proxies.
- In- and outbound web service proxy endpoints configured using child elements.
- XSL transform applied in a web service proxy.

While not specific to web service proxies, this is a general example on how to apply an XSL transformation.

The structure of the web service proxy is similar to that of the first example, with a new response message transformer added:



Flow of requests and responses in the second web service proxy example.

As in the previous example, the client and the web service endpoint, green and blue respectively in the above figure, are both realized using soapUI. The web service proxy, red in the above figure, is to be implemented using Mule.

The client and web service endpoint are reused from the previous example so if you haven't set up soapUI for the previous web service proxy example, please refer to the [Example Preparation](#) section earlier.

We will also reuse the implementation of the logging transformer that also was created as part of the preparations for the first web service proxy example.

## Create the Double Sum XSL Transform

Before being able to develop the Mule configuration for this example, we need to create a XSL transform that multiplies the contents of a <sum> element in an XML fragment with two. This transformation will be applied to response messages from the mock web service. We recall that such a message may look like this:

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:typ="urn:helloWorld/types">
    <soapenv:Header/>
    <soapenv:Body>
        <typ:helloResponse>
            <message>Hello there!</message>
            <sum>42</sum>
        </typ:helloResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

The part the XSL transform is to affect is highlighted in yellow.

- In the root of the source-tree in Eclipse, create a file named “sumDoubleTransform.xsl” with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xss="http://www.w3.org/2001/XMLSchema"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:typ="urn:helloWorld/types"
    exclude-result-prefixes="xs" version="2.0">

    <!--
        Template that multiplies the contents of a <sum> element
        with two.
    -->
    <xsl:template match="sum">
        <sum><xsl:value-of select="text() * 2"/></sum>
    </xsl:template>

    <!--
        Template that copies elements and attributes, without modifications.
    -->
    <xsl:template match="node( ) | @* ">
        <xsl:copy>
            <xsl:apply-templates select="@* | node( )" />
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

While teaching XSL transform development is outside the scope of this book, a few words may be appropriate.

- The XSL transform consists of two templates.
- The first template replaces a <sum> element and its contents with a <sum> element that contains the contents of the original <sum> element multiplied by two.
- The second template copies elements and attributes without modifications.

## Modify the Mule Configuration File

The second web service proxy example consists of one new transformer definition and two web service proxy definitions.

- Add the following to the “mule-wsproxys-config.xml” file, immediately before the closing <mule> tag:

```
...
<!--
Transformer that applies the XSL transform stored in the file
"sumDoubleTransform.xsl" to XML data in a message payload.
-->
<mule-xml:xslt-transformer
    name="doubleSumTransformer"
    xsl-file="sumDoubleTransform.xsl"/>

<!--
Abstract web service proxy that acts as a parent to
one or more child web service proxies.
Child web service proxies inherit the properties defined
in the parent web service proxy. In this case, the exception
strategy.
-->
<pattern:web-service-proxy name="exceptionStrategyProxy" abstract="true">
    <!--
        An exception strategy may be declared in a proxy,
        handling exceptions occurring while messages are passing
        through it.
    -->
    <default-exception-strategy>
        <logger level="ERROR" message="An exception occurred! "/>
    </default-exception-strategy>
</pattern:web-service-proxy>

<!--
SOAP web service proxy that has the following properties:
- Inherits from the above abstract web service proxy.
- When there is a request for the WSDL of the proxied service,
the proxy serves up a local file containing a customized WSDL.
- The proxy's inbound endpoint is declared as a child element of
the <web-service-proxy> element.
- The proxy's outbound endpoint is also declared as a child element
of the <web-service-proxy> element.
- No transformers.
The logging transformers and the unzip transformer we saw in the
previous example are now configured on the proxy's outbound
endpoint.
-->
<pattern:web-service-proxy
    name="helloWorldProxy2"
    parent="exceptionStrategyProxy"
    wsdlFile="HelloService.wsdl">
    <!-- Inbound endpoint on which the proxy receives requests. -->
    <inbound-endpoint address="http://localhost:8090/helloWorld2"/>
    <!--
        Outbound endpoint to which the proxy forwards requests.
        The responseTimeout attribute specifies a maximum time the
        endpoint waits for a response.
        The logging transformer is "applied" to requests sent over
        the endpoint.
        The following transformers are applied to response messages
        from the outbound endpoint in the order listed:
        - A transformer that unpacks the packed (zip) payload of
        response messages.
        - A transformer that applies an XSL transform to the XML
        payload of response messages (doubleSumTransformer).
        - The logging transformer.
    -->
    <outbound-endpoint
        address="http://localhost:8088/mockHelloWorldBinding"
        responseTimeout="5000"
        transformer-refs="loggingTransformer"
        responseTransformer-refs="unzipTransformer doubleSumTransformer"
```

```
loggingTransformer"/>
  </pattern:web-service-proxy>
...

```

Note that:

- There is a global transformer named “doubleSumTransformer”.  
This is the XSLT transformer that uses the XSL transform developed earlier. The XSL transform of an XSLT transformer can be stored in a file, as in this example, or entered inline, as a value of an attribute of the `<xslt-transformer>` element.
- The web service proxy named “exceptionStrategyProxy” is abstract.  
As with the other Mule patterns, web service proxies support inheritance. Abstract parent web service proxies can be declared and inherited from by one or more child web service proxies.
- The abstract web service proxy “exceptionStrategyProxy” defines an exception strategy.  
Such an exception strategy will come into effect if an exception occurs when a request or its associated response passes through the web service proxy.  
Abstract web service proxies can define other properties which are to be common among child web service proxies, such as a common outbound endpoint etc.
- The web service proxy named “helloWorldProxy2” inherits from the abstract proxy mentioned above.  
In this example, this arrangement causes the “helloWorldProxy2” to have the same exception strategy as its parent, the “exceptionStrategyProxy”.
- The “helloWorldProxy2” web service proxy has an attribute `wsdlFile`.  
Using this attribute, we can instruct the web service proxy to retrieve a local file or a document at a given URL (using the `wsdlLocation` attribute) when the WSDL of the web service being proxied is requested.
- No transformers are configured on the web service proxy.
- The inbound endpoint of the web service proxy is defined as a child element of the `<web-service-proxy>` element.  
The URL of the inbound endpoint is <http://localhost:8090/helloWorld2>.
- The outbound endpoint of the web service proxy is defined as a child element of the `<web-service-proxy>` element.
- The outbound endpoint of the web service proxy uses the `responseTimeout` attribute.  
This attribute decides for how long, in milliseconds, the endpoint will wait for a synchronous response.
- The outbound endpoint of the web service proxy applies transformations to both requests and corresponding responses, using the `transformer-refs` and `responseTransformer-refs` attributes respectively.  
Requests passed on to the underlying web service are logged, using the logging transformer seen in the previous example.  
Responses received from the underlying web service are decompressed, XSL transformed and logged.

## Modify the WSDL File

We want to be able to distinguish the WSDL stored in the file “HelloService.wsdl” from any WSDL generated by soapUI, as to make sure that the web service proxy responds requests for the WSDL with the contents of the file.

- Open the “HelloService.wsdl” and insert a comment as shown in the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    This is a local copy of the WSDL on the classpath.
-->
<wsdl:definitions targetNamespace="urn:helloWorld/sample/ivan/com"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="urn:helloWorld/sample/ivan/com"
    xmlns:types="urn:helloWorld/types"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    name="HelloWorld">
    ...

```

## Run the Web Service Proxy Example

We are now ready to test the second web service proxy example.

- Right-click the “mule-wsproxy-config.xml” Mule configuration file and select Run As -> Mule Server.
- Make sure that the mock service in soapUI is started.  
For details on setting up and starting the mock service in soapUI, please refer to the [Example Preparations](#) earlier.
- Open the “Request 1” that we used to test the mock service in the [Example Preparations](#) earlier and modify the address to which requests are to be sent to become <http://localhost:8090/helloWorld2>.  
See the detailed instructions on how to change this address in the [previous web service proxy example](#).
- Click the little green triangle in the upper-left corner of the Request 1 window.  
This will send the request to the second web service proxy, which in turn will forward the request to the mock web service.  
You should see output similar to the following in the console (excluding, of course, the colours):

```
***** Message:
org.mule.DefaultMuleMessage
{
    id=b0bee23b-8172-11e1-869b-45fdb5b2e92c
    payload=org.apache.commons.httpclient.ContentLengthInputStream
    correlationId=<not set>
    correlationGroup=-1
    correlationSeq=-1
    encoding=utf-8
    exceptionPayload=<not set>

Message properties:
    INVOCATION scoped properties:
    INBOUND scoped properties:
        Accept-Encoding=gzip,deflate
        Connection=false
        Content-Length=333
        Content-Type=text/xml; charset=utf-8
        Host=localhost:8090
        Keep-Alive=false
        MULE_ORIGINATING_ENDPOINT=endpoint.http.localhost.8090.helloWorld2
```

```

MULE_REMOTE_CLIENT_ADDRESS=/127.0.0.1:52285
User-Agent=Jakarta Commons-HttpClient/3.1
http.context.path=/helloWorld2
http.method=POST
http.request=/helloWorld2
http.request.path=/helloWorld2
http.version=HTTP/1.1
OUTBOUND scoped properties:
Accept-Encoding=gzip,deflate
Connection=false
Content-Length=333
Content-Type=text/xml; charset=utf-8
Host=localhost:8090
Keep-Alive=false
MULE_ENCODING=utf-8
MULE_ENDPOINT=http://localhost:8088/mockHelloWorldBinding
MULE_ROOT_MESSAGE_ID=b0bee23b-8172-11e1-869b-45fdb5b2e92c
MULE_SESSION=r00...
User-Agent=Jakarta Commons-HttpClient/3.1
http.context.path=/helloWorld2
http.method=POST
http.request=/helloWorld2
http.request.path=/helloWorld2
http.version=HTTP/1.1
SESSION scoped properties:
}
***** Message payload: <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:typ="urn:helloWorld/types">
<soapenv:Header/>
<soapenv:Body>
<typ:hello>
<name>Ivan</name>
<first_number>2</first_number>
<second_number>3</second_number>
</typ:hello>
</soapenv:Body>
</soapenv:Envelope>
INFO 2012-04-08 14:02:20,522 [connector.http.mule.default.receiver.04]
org.mule.transport.service.DefaultTransportServiceDescriptor: Loading default outbound
transformer: ...
***** Message:
org.mule.DefaultMuleMessage
{
id=b1b3f09f-8172-11e1-869b-45fdb5b2e92c
payload=[B
correlationId=<not set>
correlationGroup=-1
correlationSeq=-1
encoding=utf-8
exceptionPayload=<not set>

Message properties:
INVOCATION scoped properties:
INBOUND scoped properties:
Connection=false
Content-Encoding=gzip
Content-Type=text/xml; charset=utf-8
Keep-Alive=false
Server=Jetty(6.1.x)
Transfer-Encoding=chunked
http.method=POST
http.request=http://localhost:8088/mockHelloWorldBinding
http.status=200
http.version=HTTP/1.1
OUTBOUND scoped properties:
MULE_ENCODING=utf-8
MULE_SESSION=r00...
SESSION scoped properties:
}
***** Message payload: <?xml version="1.0" encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:typ="urn:helloWorld/types">
<soapenv:Header/>
<soapenv:Body>
<typ:helloResponse>
<message>Hello there!</message>

```

```
<sum>84</sum>
</typ:helloResponse>
</soapenv:Body>
</soapenv:Envelope>
```

Note that:

- The part of the log highlighted in blue is the request message.  
We can see that the SOAP request is identical to that we have seen in soapUI.
- The part of the log highlighted in green is log messages from Mule initializing its components.  
This part has been shortened, to increase readability.
- The part of the log highlighted in orange is the response message.  
We can see that the SOAP response is identical to that we have configured the mock service in soapUI to respond with.
- We can read the XML of the SOAP response.  
As in the previous web service proxy example, the “unzipTransformer” transformer decompresses compressed response messages.
- In the above output, Mule session identifier data has been shortened to avoid cluttering.
- The contents of the the <sum> element in the response SOAP message is 84.  
We thus can conclude that the XSL transform has been applied to the response message, as intended.

Next, we examine the WSDL of the service to which we send requests:

- In a web browser, issue a request to the following URL: <http://localhost:8090/helloWorld2?wsdl>
- In the web browser, you should see the WSDL with the comment we inserted in the “HelloService.wsdl” file earlier, stating that: “This is a local copy of the WSDL on the classpath.”

This concludes the web service proxy example and the chapter on Mule configuration patterns.

## Part Two – Recipes and Reference

This part contains smaller, incomplete, examples that are more focused on solving a particular task or problem and some reference information. Motivations and explanations are more scarce in this part, compared to part one.

### 1. Message Routing

This section contains recipes related to the routing of messages.

#### 1.1. Selecting Outbound Endpoint Depending on the Message

Assume that you want to send a message to one of several outgoing endpoints depending on some characteristic of the message.

One way to accomplish this is by using multiple `<filtering-router>` elements inside the `<outbound>` element of a service:

```
...
<service name="directoriesWriterService">
    <inbound>
        <vm:inbound-endpoint path="directoriesWriter" />
    </inbound>
    <outbound>
        <filtering-router>
            <!--
                The outbound endpoint that will be used if the condition of
                the filter in this filtering router is met.
            -->
            <file:outbound-endpoint
                path="directory1/"
                outputPattern="#{header:originalFilename}" />
            </file:outbound-endpoint>
            <!--
                The filter that determines the messages that will send to
                the outbound endpoint in this filtering router.
            -->
            <regex-filter pattern="some words (.*)" />
            <!--
                Any transformations to be applied before messages are
                filtered can be insert here.
            -->
        </filtering-router>
        <filtering-router>
            ...
        </filtering-router>
        <!--
            If a message does not meet the condition of any filter, it
            will be sent to the outbound endpoint specified in the
            <forwarding-catch-all-Strategy> element.
        -->
        <forwarding-catch-all-strategy>
            <file:outbound-endpoint
                path="directory-nomatch/"
                outputPattern="#{header:originalFilename}" />
            </file:outbound-endpoint>
        </forwarding-catch-all-strategy>
    </outbound>
...

```

Note that:

- The `<outbound>` element of the service contains multiple `<filtering-router>` elements.
- A `<filtering-router>` element contains one outbound endpoint, one optional filter and any number of transformers.

- The outbound endpoint in a <filtering-router> is mandatory.
- The filter in a <filtering-router> element determines whether a message will be sent to the outbound endpoint in the <filtering-router> element.  
If the message meets the condition of the filter, the message will be sent to the outbound endpoint.
- The filter in a <filtering-router> is optional.
- A <filtering-router> can contain zero or more transformers that transform messages before they are filtered.
- The <outbound> element contains a <forwarding-catch-all-strategy> element.  
If a message does not meet the condition of any of the filters in the <filtering-router> elements, it will be sent to the outbound endpoint in the catch-all strategy element.
- If no catch-all strategy element is defined in an <outbound> element and a message does not meet the condition of any of the filters in the <filtering-router> elements, the message will be ignored and a warning written to the log.

## 1.2. Routing a Message Depending on a Single Filter

In a service, we can route messages in one of two directions depending on whether or not a message meets the conditions of a filter.

```
...
<service name="someService">
    <inbound>
        <vm:inbound-endpoint path="some.input.path" />
        <!--
            A selective consumer router can apply one filter
            to incoming messages.
        -->
        <selective-consumer-router>
            <!--
                The filter determines whether or not the message is to be
                passed on to the outbound endpoint.
                If a message matches the filter conditions, it will be
                passed to the outbound endpoint.
                If a message does not match the filter conditions and there
                is a catch-all strategy defined in the <inbound> element,
                the message will be forwarded to the catch-all strategy.
                If a message does not match the filter conditions and there
                is no catch-all strategy defined in the <inbound> element,
                the message will be dropped and a message written to the log.
            -->
            <mule-xml:validation-filter
                schemaLocations ="schemas/schema2.xsd" returnResult="false" />
        </selective-consumer-router>
        <!--
            Catch-all strategy specifying where messages that do not
            match the filter conditions are to be routed.
        -->
        <forwarding-catch-all-strategy>
            <vm:outbound-endpoint path="some.no.filtermatch.path" />
        </forwarding-catch-all-strategy>
    </inbound>
    <outbound>
        <!--
            Messages arriving to the outbound endpoint match the filter conditions.
        -->
        <pass-through-router>
            <vm:outbound-endpoint path="some.filtermatch.path" />
        </pass-through-router>
    </outbound>
</service>
...
```

### 1.3. Exception-Dependent Message Routing

There are two options regarding how to do routing based on exceptions, or rather based the failure or success of one of a set of outbound endpoints.

#### The Exception-Based Router

The exception-based router is available in both Mule 2.x and Mule 3.x. The following example shows the exception-based router being used in a Mule 2.x configuration:

```
...
<service name="receiverService">
  ...
  <outbound>
    <!--
      The exception based router attempts to send the message
      to the the contained endpoints in the order listed
      until having encountered an endpoint that does not
      throw an exception.
    -->
    <exception-based-router>
      <!--
        All endpoints but the last in the exception-based
        router will be forced to be synchronous.
        The last outbound endpoint will be asynchronous unless
        the synchronous attribute is explicitly set to true
        in the <outbound-endpoint> element in the exception-
        based router. The synchronous attribute on the inbound
        endpoint to which the message is sent will not be
        considered by the router.
      -->
      <vm:outbound-endpoint path="exceptionServiceInbound"/>
      <vm:outbound-endpoint path="helloServiceInbound" synchronous="true"/>
      <!--
        Optionally, the exception-based router can be configured
        with a reply-to address, to which a message that has been
        processed by a service, that received the message from the
        exception-based router, will send the message.
      -->
      <reply-to address="vm://nextServiceURL"/>
    </exception-based-router>
  </outbound>
</service>
...
```

Note that:

- Routers, as the exception-based router, are commonly used in services.
- The exception-based router will attempt to send a message to each of the outbound endpoints in the `<exception-based-router>` element, in the order they are listed, until encountering an invocation that does not result in an exception.
- The exception-based router will force synchronous invocation of all the contained outbound endpoints except for the last.

Thus if we want the invocation of the last outbound endpoint to be synchronous, we must use the `synchronous` attribute (Mule 2.x) or the `exchange-pattern` attribute (Mule 3.x).

- The exception-based router may be configured with zero or more transformers.
- The outbound endpoints of the exception-based router may be configured with filters.
- The exception-based router may be configured with a reply-to address.

After the message has been successfully processed by one of the services listening to the outbound endpoints listed in the exception-based router, the message will be sent to the

address configured in the <reply-to> element's *address* attribute.

- If all the outbound endpoints of the exception-based router result in exceptions being thrown, an exception will be thrown.  
The exception message will indicate that the exception-based router failed to route the message.
- Exceptions thrown by services that the exception-based router attempts to invoke can be logged using a default or custom exception strategy, either on the individual service containing the exception-based router or on the model containing the service containing the exception-based router.

## The First-Successful Message Processor

The first-successful message processor is only available in Mule 3.x. This message processor is used in flows, as shown in the following example:

```
...
<flow name="ReceiverFlow">
  ...
  <!--
    The first-successful message processor attempts to send the
    message to the the contained endpoints in the order listed
    until having encountered an endpoint that succeeds.
    The exchange-pattern attribute of the outbound endpoints
    in the first-successful message processor can be set to
    "request-response" for better reliability, but may be
    "one-way" as well.
    The failureExpression attribute defines an expression used to
    determine whether an invocation of an endpoint is a failure
    or not. In this case, an endpoint invocation is a failure if
    there is an exception of the type java.lang.Exception thrown.
  -->
  <first-successful failureExpression="exception-type:java.lang.Exception">
    <vm:outbound-endpoint
      path="exceptionServiceInbound"
      exchange-pattern="request-response" />
    <vm:outbound-endpoint
      path="helloServiceInbound"
      exchange-pattern="request-response" />
  </first-successful>
</flow>
...
```

Note that:

- The first-successful message processor can appear anywhere in a flow where a message processor can appear.  
This means that it does not necessarily have to appear last in a flow.
- The *failureExpression* attribute of the <first-successful> element contains an expression that is used to determine whether the invocation of an endpoint has failed or not.  
The default expression determines failure as an exception being thrown or returned as the result of the endpoint invocation.  
The expression in the above example determines failure as an exception of the type *java.lang.Exception* having been thrown or returned.
- The first-successful message processor will attempt to send a message to each of the outbound endpoints in the <first-successful> element, in the order they are listed, until encountering an invocation that does not result in the *failureExpression* evaluating to true.
- The first-successful message processor will not force synchronous invocation of any endpoint listed in the <first-successful> element.  
However, using synchronous invocation, that is the message exchange-pattern "request-response" will increase reliability.

- If all the outbound endpoints of the first-successful message processor result in exceptions being thrown, an exception will be thrown.  
The exception message will indicate that the message processor failed to route the message.
- Exceptions thrown by services that the first-successful message processor attempts to invoke will be logged by a default or custom exception strategy.  
The exception strategy must be configured on the same flow as in which the first-successful message processor appears. Flows always have a default exception strategy, even if one has not been explicitly configured.

## 2. Filtering

This section contains recipes related to filtering of messages. Filters are used to decide whether or not to route a message to a certain service.

### 2.1. Validating XML Message Payload

Messages containing XML payload can be filtered depending on whether they pass validation against one or more XML schemas.

```
...
<!--
    A message will match the conditions of a schema validation filter if the
    XML message payload successfully validates against the XML schemas specified
    by the schemaLocations attribute.
    The returnResult attribute specifies whether the XML being the result of
    the validation should be set as the message payload after validation.
    If returnResult is set to false, validation will be slightly faster.
-->
<mule-xml:schema-validation-filter schemaLocations = "schema1.xsd,schema2.xsd"
    returnResult="false"/>
...
```

Note that:

- XML is not the only schema language that can be used when validating.
- Multiple XML schemas with the same namespace cannot be specified in the *schemaLocations* attribute.

This is due to a bug in the Xerces XML parser framework.

To validate against multiple XML schemas with the same namespace, use an <or-filter> and one <mule-xml:schema-validation-filter> for each schema.

- If the XML schema(s) specified in the *schemaLocations* attribute in turn import, or in some other way depend on, one or more other schemas, a custom resource resolver may have to be developed. See [chapter 5](#) for an example!
- Starting with Mule 3.2, a resource resolved can be specified using the *resourceResolver-ref* attribute on the <schema-validation-filter> element.

Earlier the schema validation filter had to be specified as a custom filter and the resource resolver injected as a Spring property.

## 2.2. Combining Filters

Two or more filters can be combined to act as one single filter using the standard logic operators AND and OR. Filters can be negated using the NOT-filter.

### The AND-filter

The following filter configuration combines the <payload-type-filter> and the <regex-filter> as to match only messages with payload of the type *java.lang.String* that starts with the character sequence “secret” followed by zero or more characters.

```
...
<and-filter>
    <payload-type-filter expectedType="java.lang.String" />
    <regex-filter pattern="secret (.*)"/>
</and-filter>
...
```

Note the <and-filter> element that contains two or more filters, all of which conditions must be matched in order for the AND-filter to match.

### The OR-filter

The following filter configuration combines the <message-property-filter> and the <wildcard-filter> as to match messages which either has the message property “originalFilename” with a value “matching-msg.txt” or which message payload starts with the string “the pigs”.

```
...
<or-filter>
    <message-property-filter pattern="originalFilename=matching-msg.txt" />
    <wildcard-filter pattern="the pigs*"/>
</or-filter>
...
```

Note the <or-filter> element that contains two or more filters, of which at least one filter condition must be matched in order for the OR-filter to match.

### The NOT-filter

The NOT-filter negates the result of one single filter. The following filter configuration accepts all messages which payload is not well-formed XML.

```
...
<not-filter>
    <custom-filter class="org.mule.module.xml.filters.IsXmlFilter" />
</not-filter>
...
```

Note the <not-filter> element that contains one single filter. The result of the contained filter is negated.

## 2.3. Implementing Custom Filters

In addition to the filters supplied by Mule, it is also possible to implement custom filters. The following class shows what a custom filter implemented in Java may look like:

```
package com.ivan.filters;

import org.mule.api.MuleMessage;
import org.mule.api.routing.filter.Filter;

/**
 * This class implements a custom Mule filter.
 *
 * @author Ivan Krizzan
 */
public class MyCustomFilter implements Filter
{
    /* Instance variable(s): */
    private String mSomeFilterProperty;

    /**
     * Checks the supplied message to determine whether it
     * matches the filter conditions.
     *
     * @param inMessage Non-null Mule message to check.
     * @return True if message matches filter conditions, false
     * otherwise.
     */
    @Override
    public boolean accept(final MuleMessage inMessage)
    {
        System.out.println("**** Message payload: " + inMessage.getPayload());
        System.out.println("      Filter property: " + mSomeFilterProperty);

        boolean theIsStringPayloadFlag = inMessage.getPayload().getClass().equals(
            String.class);
        return theIsStringPayloadFlag;
    }

    public void setSomeFilterProperty(final String inSomeFilterProperty)
    {
        mSomeFilterProperty = inSomeFilterProperty;
    }
}
```

The filter can then be used in a Mule configuration file:

```
...
<custom-filter class="com.ivan.filters.MyCustomFilter">
    <spring:property name="someFilterProperty" value="Hello filtered world! "/>
</custom-filter>
...
```

Note the property being injected into the filter.

### 3. ***Transforming***

This section contains recipes related to transformation of messages.

#### 3.1. Extract Part of an XML Message with XPath

To extract a part of the XML payload of a message, we can use the XPath extractor transformer. The following example shows how to apply the XPath extractor transformer inside a <file:inbound-endpoint> element.

```
...
<file:inbound-endpoint connector-ref="fileConnector"
    path=".//file-input-directory">
    <file:filename-wildcard-filter pattern="*.xml"/>

    <!--
        Must transform the XML data to DOM in order to receive
        a DOM node as result of the XPath transform.
        Otherwise we will receive the contents of the elements
        in the extracted XML fragment.
    -->
    <mule-xml:xml-to-dom-transformer returnClass="org.w3c.dom.Document" />

    <!--
        Apply an XPath expression to the message.
        The desired type of result can be specified as one of:
        STRING (default), BOOLEAN, NUMBER, NODE, NODESET
    -->
    <mule-xml>xpath-extractor-transformer
        expression="/ivan:PhoneBook/ivan:Person"
        resultType="NODE">
    <!--
        The XML data we are processing is in a namespace
        and we use a namespace prefix in the XPath expression.
        We must thus link the namespace prefix and
        the namespace URI.
    -->
    <mule-xml:namespace
        prefix="ivan"
        uri="http://www.ivan.com/schemas/addressbook"/>
</mule-xml>xpath-extractor-transformer>

    <!--
        Must transform the message from DOM to text to make
        the resulting XML readable.
    -->
    <mule-xml:dom-to-xml-transformer/>
</file:inbound-endpoint>
...
```

We usually combine the XPath extractor transformer with one or two other transformers:

- <mule-xml:xml-to-dom-transformer>  
To obtain a DOM-tree on which to apply the transform. A DOM-tree is needed in order for the result of the XPath transformer to be a XML fragment.
- <mule-xml>xpath-extractor-transformer>  
Applies an XPath expression to the DOM-tree from the previous step.
- <mule-xml:dom-to-xml-transformer>  
Optional. Used if we want the payload of the message to be an XML string.

Note that if the XML data to be processed belongs to a namespace, we must use a namespace prefix in the XPath expression and define this namespace prefix using a <mule-xml:namespace> child element in the <mule-xml>xpath-extractor-transformer>.

In Mule 3.1, globally defined namespaces, using the <mule-xml:namespace-manager> element, are not used by the XPath extractor transformer. This feature is available in Mule 3.2.

## 3.2. Transform XML Data Using XSL

A common scenario is that XML data needs to be processed in some way. Both Mule 2.x and 3.x support XSL transformation using the <xslt-transformer> from the XML module.

The following example shows how to double the number contained in the <sum> element of a SOAP message.

The SOAP message that we wish to process looks like this:

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:typ="urn:helloWorld/types">
    <soapenv:Header/>
    <soapenv:Body>
        <typ:helloResponse>
            <message>Hello there!</message>
            <sum>42</sum>
        </typ:helloResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

The XSL transformation that we wish to apply looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:typ="urn:helloWorld/types"
    exclude-result-prefixes="xs" version="2.0">

    <xsl:template match="sum">
        <sum><xsl:value-of select="text() * 2" /></sum>
    </xsl:template>

    <xsl:template match="node( ) | @* ">
        <xsl:copy>
            <xsl:apply-templates select="@* | node( )" />
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

XSL development is outside the scope of this book, but in brief the above XSL transformation copies all the elements and attributes of an XML document, except for the contents of <sum> elements, which is multiplied by two.

Finally, the definition of a global XSL transformer in a Mule configuration file looks like this, assuming the above XSL transformation has been saved in a file named “sumDoubleTransform.xsl” and the namespace prefix “mule-xml” as been configured appropriately:

```
...
<!--
    Transformer that applies the XSL transform stored in the file
    "sumDoubleTransform.xsl" to XML data in a message payload.
-->
<mule-xml:xslt-transformer
    name="doubleSumTransformer"
    xsl-file="sumDoubleTransform.xsl" />
...
```

### 3.3. Pack or Unpack Message Payload Data

Mule 2.x and 3.x has two transformers that can compress or uncompress message payload data. To uncompress, use the GZIP uncompress transformer. The following example shows the declaration of a global uncompressing transformer:

```
... <gzip-uncompress-transformer name="unzipTransformer" />
...
```

The uncompressing transformer accepts the following input types, which both are to contain compressed data:

- Byte arrays.
- Input streams.

To compress data, use the HZIP compress transformer. The following example shows the declaration of a global compressing transformer:

```
... <gzip-compress-transformer name="zipTransformer" />
...
```

The compressing transformer accepts the following input types:

- Byte arrays.
- Input streams.
- Serializable objects.

## 3.4. Custom Transformers

Custom transformers can be implemented in, for instance, Java or Groovy.

### Mule 2.x Custom Transformers

The following example shows how to implement a Mule 2.x transformer that will receive a reference to the current message:

```
package com.ivan.transformers;

import org.mule.api.MuleMessage;
import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractMessageAwareTransformer;

/**
 * Mule 2.x transformer that logs message that passes through
 * the transformer to the console. Messages are not altered in any way.
 *
 * @author Ivan Krizsan
 */
public class MessageLoggingTransformer2 extends AbstractMessageAwareTransformer
{
    @Override
    public Object transform(final MuleMessage inMessage,
                           final String inOutputEncoding) throws TransformerException
    {
        try
        {
            System.out.println("***** Message: " + inMessage.toString());
            System.out.println("***** Message payload: " +
                               inMessage.getPayloadAsString());
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
        return inMessage;
    }
}
```

Note that:

- The above transformer does not alter the message in any way.  
If it had, the altered message would have been returned by the *transform* method.
- The custom transformer class inherits from the *AbstractMessageAwareTransformer* class.  
This is a good starting-point if you are developing a message-aware Mule 2.x transformer.  
All transformers are required to implement the *org.mule.api.transformer.Transformer* interface, either directly or by inheritance. Please refer to its API documentation for a list of classes that implement this interface.
- Another good starting-point for transformer-development is the superclass to the *AbstractMessageAwareTransformer*; *org.mule.transformer.AbstractTransformer*.  
The *AbstractTransformer* class is a general base-class for all transformers.

When the transformer class has been developed, we can use it in a Mule 2.x configuration file using the <custom-transformer> element.

The following example shows a custom transformer being configured as a global transformer with the name “loggingTransformer”:

```
...
<custom-transformer name="loggingTransformer"
                    class="com.ivan.transformers.MessageLoggingTransformer2"/>
...
```

## Mule 3.x Custom Transformers

This example shows the Mule 3.x version of a transformer that will receive a reference to the current message:

```
package com.ivan.transformers;

import org.mule.api.MuleMessage;
import org.mule.api.transformer.TransformerException;
import org.mule.transformer.AbstractMessageTransformer;

/**
 * Mule 3.x transformer that logs message that passes through
 * the transformer to the console. Messages are not altered in any way.
 *
 * @author Ivan Krizsan
 */
public class MessageLoggingTransformer extends AbstractMessageTransformer
{
    @Override
    public Object transformMessage(final MuleMessage inMessage,
        final String inOutputEncoding) throws TransformerException
    {
        try
        {
            System.out.println("***** Message: " + inMessage.toString());
            System.out.println("***** Message payload: " +
                inMessage.getPayloadAsString());
        } catch (Exception theException)
        {
            theException.printStackTrace();
        }
        return inMessage;
    }
}
```

Note that:

- The above transformer does not alter the message in any way.  
If it had, the altered message would have been returned by the *transformMessage* method.
- The custom transformer class inherits from the *AbstractMessageTransformer* class.  
This is a good starting-point if you are developing a message-aware Mule 3.x transformer.  
All transformers are required to implement the *org.mule.api.transformer.Transformer* interface, either directly or by inheritance. Please refer to its API documentation for a list of classes that implement this interface.
- Another good starting-point for transformer-development is the superclass to the *AbstractMessageTransformer*; *org.mule.transformer.AbstractTransformer*.  
The *AbstractTransformer* class is a general base-class for all transformers.

When the transformer class has been developed, we can use it in a Mule 3.x configuration file using the <custom-transformer> element.

The following example shows a custom transformer being configured as a global transformer with the name “loggingTransformer”:

```
...
<custom-transformer
    name="loggingTransformer"
    class="com.ivan.transformers.MessageLoggingTransformer" />
...
```

## 4. Message Properties

This section contains recipes related to reading and manipulation of message properties, also known as message headers. Message properties are additional pieces of data that travels along with a message in a map. To retrieve or set a message property, we need to know its name.

### 4.1. Retrieving Message Properties

To retrieve a message property, a Mule expression can be used. If the original filename of a file has been inserted as a message property with the name “originalFilename”, as seen in the example in chapter 4, it can be retrieved and used as the filename of a file to be written like in the following example:

```
...  
<file:outbound-endpoint path=".//file-output-directory"  
                      outputPattern="#[header:originalFilename]" />  
...
```

Additional text and expressions may be inserted before and after the expression retrieving the original filename message property.

### 4.2. Setting Message Properties

Message properties can be set using the message properties transformer. The following example inserts the message property with the name “PROPERTYNAME” and the string value “PropertyValue” into the current message.

```
...  
<message-properties-transformer>  
  <add-message-property key="PROPERTYNAME" value="PropertyValue" />  
</message-properties-transformer>  
...
```

Note that Mule expressions can be used in the values of the *key* and *value* attributes.

Also note that a message properties transformer may contain multiple child elements that adds, removes or renames message properties.

### 4.3. Removing Message Properties

Message properties can be removed using the message properties transformer. The following example removes the message property with the name “PROPERTYNAME” from the current message.

```
...  
<message-properties-transformer>  
  <delete-message-property key="PROPERTYNAME" />  
</message-properties-transformer>  
...
```

Note that Mule expressions can be used in the value of the *key* attribute.

Also note that a message properties transformer may contain multiple child elements that adds, removes or renames message properties.

## 4.4. Renaming Message Properties

Message properties can be renamed using the message properties transformer. The following renames the message property with the name “PROPERTYNAME” to “NEWPROPERTYNAME”.

```
...<message-properties-transformer>
    <rename-message-property key="PROPERTYNAME" value="NEWPROPERTYNAME" />
</message-properties-transformer>
...
```

Note that Mule expressions can be used in the value of the *key* attribute.

Also note that a message properties transformer may contain multiple child elements that adds, removes or renames message properties.

## 4.5. Reading and Writing Message Properties to Different Scopes

Message properties can be read and written to different scopes, as listed below. Message property scopes are more strictly enforced in Mule 3.x compared to earlier versions of Mule.

- Inbound  
Message properties originating from clients requests are found in this scope.
- Outbound  
Message properties to be returned or sent out are found in this scope.
- Session  
Message properties that are to be retained throughout multiple invocations are to be placed in this scope.

The message property scope can be specified in different ways, depending on the context.

In the first example, it is specified using an attribute of a transformer:

```
...<message-properties-transformer scope="outbound">
    <add-message-property key="validated" value="true" />
</message-properties-transformer>
...
```

The following example tests for whether the property 'validated' in the outbound scope has the value 'true':

```
...<when expression="OUTBOUND:validated=true" evaluator="header">
...
```

Message properties can be retrieved using the Mule expression language. This example logs a message with the value of the 'validated' message property from the outbound scope.

```
...<logger message="Validated flag: #[header:OUTBOUND:validated]" level="ERROR" />
...
```

## **5. Expressions**

Expressions allow for retrieval of data from messages, message headers, message attachments, beans etc in, for instance, the following kinds of Mule configuration elements:

- Expression transformers.  
Example: <xpath-extractor-transformer>
- Expressions filters.  
Example: <expression-filter>
- Expression-based routers.  
Example: <expression-splitter-router>

Note that expressions are not allowed everywhere; an inbound endpoint needs to know from where to receive messages at configuration time while the evaluation of the address to which an outbound endpoint is to send a message can be deferred to runtime.

### **5.1. Evaluators**

Different kinds of evaluators are used to evaluate different kinds of expressions. For instance, there are evaluators that can access the attachments of messages, an evaluator that allows for access of message headers etc.

Below are the default Mule evaluators available in Mule 3.2.0 listed.

#### **Attachment Evaluator**

The attachment evaluator allows for access to any attachment(s) that a message may have by specifying the name of the attachment. The presence of the attachment is made optional by appending “?” to its name. If an attachment is not present, an exception will be thrown if the name does not end with “?”.

Example expression: #[attachment:mail-attachment?]

Retrieves the attachment with the name “mail-attachment” from the message.

Returns null if the attachment is not present.

Restrictions: Cannot be used in expression filters.

#### **Attachments Evaluator**

The attachments evaluator retrieves a map containing the named attachments of a message. The presence of an attachment is made optional by appending “?” to its name.

Example expression: #[attachments:foo,bar?,baz?]

Retrieves the attachments with the names “foo” (required), “bar” (optional) and “baz” (optional) from the message.

Restrictions: Cannot be used in expression filters.

## **Attachments-List Evaluator**

The attachments-list evaluator retrieves a list object, implementing `java.util.List`, containing the named attachments of a message. The presence of an attachment is made optional by appending “?” to its name. The “\*” wildcard can be used when specifying attachment names; that is “foo\*” will match “food” and “fool”. The “\*” used by itself will match any name.

Example expression: `##[attachments-list:foo?,baz*]`

Retrieves a list containing the attachment with the name “foo” (optional) and all attachment with names starting with “baz” (optional).

Restrictions: Cannot be used in expression filters.

## **Bean Evaluator**

The bean evaluator access properties from a Java bean being the payload of the current Mule message using the getters and setters of the Java bean.

### **Example:**

Assume the following Java bean class (getters and setters have been omitted to conserve space):

```
...
public class MyBean
{
    private MyBean root;
    private MyBean of;
    private String all;
    ...
    /* Getters and setters for the above instance variables have been omitted. */
}
```

The following code is used to initialize and set the payload of a Mule message:

```
...
MyBean theOf = new MyBean();
theOf.setAll("evil");
MyBean theRoot = new MyBean();
theRoot.setOf(theOf);
MyBean thePayload = new MyBean();
thePayload.setRoot(theRoot);

MuleMessage theMuleMessage = new DefaultMuleMessage(thePayload, theContext1);
...
```

In the following <logger> element, the message “evil” is retrieved from the MyBean instance which reference is stored in the variable *thePayload* created in the code above.

```
<logger level="ERROR" message="#[bean:root.of.all]" />
```

The Java code corresponding to the expression `##[bean:root.of.all]` is:

```
String theMsg = thePayload.getRoot().getOf().getAll();
```

Example expression: #[bean:root.of.all]

Retrieves an object by invoking *getRoot()* on the message payload, then retrieves an object by invoking *getOf()* on the previous result, then retrieves something by invoking *getAll()* on the previous result.

The result of the entire expression is the result obtained from invoking *getAll()*.

The expression can also be written as #[bean:root/of/all].

Restrictions:

None.

### **Endpoint Evaluator**

The endpoint evaluator allows for retrieval of properties of global endpoints.

Assume the following global endpoint declaration in a Mule configuration file:

```
...
<endpoint
    name="myGlobalEndpoint"
    address="http://ivan.com/globalEndpoint"
    exchange-pattern="request-response">
</endpoint>
...
```

Example expression: #[endpoint:myGlobalEndpoint.address]

Retrieves the “address” property of the global endpoint with the name “myGlobalEndpoint”.

Restrictions:

Cannot be used in expression filters.

Only the “address” property supported, as of Mule 3.2.0.

### **Exception-Type Evaluator**

The exception-type evaluator examines the exception payload of a message to determine whether it is of a specific type.

Example expression: #[exception-type:javax.xml.soap.SOAPException]

Evaluates to true if the exception payload of a message is of the type *javax.xml.soap.SOAPException*.

Restrictions:

Can only be used in expression filters.

In Mule 3.10 and later, the <first-successful> router can use the exception-type evaluator.

## **Function Evaluator**

The function evaluator invokes one of the functions listed below.

Function Name	Description
now	Date and time when function was invoked in a <i>java.sql.Timestamp</i> object.
date	Date when function was invoked in a <i>java.util.Date</i> object.
datestamp	String representation of the time when function was invoked represented by a <i>java.util.Date</i> object formatted with a <i>SimpleDateFormat</i> . Default date format: dd-MM-yy_HH-mm-ss.SSS Example: #[function:datestamp:dd-MM-yy]
systime	The current system time as returned by <i>System.currentTimeMillis()</i> .
uuid	String representation of an UUID.
hostname	The host name of localhost.
ip	The host address of localhost.
count	Retrieves and increments a counter global to the function evaluator of a Mule instance. The counter starts from zero each time the Mule server is started.

Example expression: #[function:systime]

Restrictions: Cannot be used in expression filters.

## **Groovy Evaluator**

The Groovy evaluator evaluates expression using the Groovy language.

The following properties are available to the Groovy script:

Property	Description
log	Logger object inheriting from the <i>SLF4JLog</i> class.
result	Result of evaluating the Groovy script.
muleContext	Object implementing the <i>MuleContext</i> interface.
payload	Payload of current Mule message.
src	Payload of current Mule message.
message	Only available if the expression is evaluated when processing a Mule message, then an object implementing the <i>MuleMessage</i> interface.
registry	Object of the type <i>MuleRegistryHelper</i> .

Example expression: #[groovy:payload.substring(0,5)]

Extracts the five first characters of the payload, assuming it is a string.

Restrictions: None.

## **Header Evaluator**

The header evaluator allows access to the specified Mule message header.

Example expression: `#[header:OUTBOUND:myHeader?]`

Accesses the optionally present message header with the name “myHeader” in the OUTBOUND scope.

The presence of a header is made optional by appending “?” to its name. If a header is not present, an exception will be thrown if the name does not end with “?”.

Restrictions:

None.

## **Headers Evaluator**

The headers evaluator allows retrieval of multiple Mule message headers, placing the result in a map object.

Example expression: `#[headers:OUTBOUND:myHeader1?, OUTBOUND:myHeader2?]`

Retrieves the optionally present headers with the names “myHeader1” and “myHeader2” both in the OUTBOUND scope.

The presence of a header is made optional by appending “?” to its name. If a header is not present, an exception will be thrown if the name does not end with “?”.

“\*” can be used as a wildcard, retrieving all message headers in a certain scope.

The default scope is OUTBOUND.

Restrictions:

Cannot be used in expressions filters.

## **Headers-List Evaluator**

The headers evaluator allows retrieval of multiple Mule message headers, placing the result in a list object.

Example expression: `#[headers-list:OUTBOUND:myHeader1?, OUTBOUND:myHeader2?]`

Retrieves the optionally present headers with the names “myHeader1” and “myHeader2” both in the OUTBOUND scope.

The presence of a header is made optional by appending “?” to its name. If a header is not present, an exception will be thrown if the name does not end with “?”.

“\*” can be used as a wildcard, retrieving all message headers in a certain scope.

The default scope is OUTBOUND.

Restrictions:

Cannot be used in expressions filters.

## **JSON Evaluator**

The JSON evaluator allows evaluation of XPath-like expressions against Mule message payload containing JSON data, extracting contents of nodes.

Assume, for instance, the following JSON data:

```
{"person": {"name": "Alistair Smith"}}
```

Also assume the following expression fed to the JSON evaluator:

```
#[$json://person/name]
```

The result produced by the JSON evaluator would be “Alistair Smith” (without quotes).

Example expression: #[\$json://person/name]

Retrieves the contents of the node “name” being a child of the node “person” from the JSON message payload.

Returns null if a node in the expression is not present.

It is also possible testing whether the contents of a JSON node is equal to or not equal to some value using expressions like this:

```
#[$json://person/name = Alistair Smith]
```

Restrictions: None.

## **JSON-Node Evaluator**

The JSON-Node evaluator allows evaluation of Xpath-like expressions against Mule message payload containing JSON data, extracting the [Jackson](#) node object.

Assume, for instance, the following JSON data:

```
{"person": {"name": "Alistair Smith"}}
```

Also assume the following expression fed to the JSON evaluator:

```
#[$json://person/]
```

The result produced by the JSON evaluator would be an object of the type *ObjectNode* containing the text node “name”.

Example expression: #[\$json://person/]

Retrieves the contents of the node “person” as a Jackson object from the JSON message payload.

Returns null if a node in the expression is not present.

Restrictions: Available in Mule 3.10 and later.

## **JXPath Evaluator**

The JXPath evaluator is deprecated and should not be used. As alternatives, consider the [Bean Evaluator](#) discussed earlier or the [XPath evaluator](#) that will be introduced later.

## **Map-Payload Evaluator**

The map-payload evaluator retrieves the value associated with a certain key from a Mule message payload of the type *java.util.Map*.

Example expression: `#[map-payload:kevinKey,nivek]`

Retrieves the values associated with the keys “kevinKey” and “nivek” from the Map object being the Mule message payload and returns a map containing the two key-value pairs.

If one single key is supplied, the object type will be that of the value object. If multiple keys are supplied, the result will be a map with the key-value pairs inserted.

Requesting the value of a key not present in the map will generate an exception.

The presence of keys can be made optional by appending “?” to the name of the key.

Restrictions:

Cannot be used with expression filters.

## **Message Evaluator**

The message evaluator allows for retrieval of properties on Mule messages. The following properties are available:

Property	Description
<code>id</code>	Unique id of the message.
<code>correlationId</code>	Correlation id of the message, or null if not set.
<code>correlationGroupSize</code>	Number of messages in a correlation group, or -1 if not known.
<code>correlationSequence</code>	Sequence or ordering number for the message, or -1 if ordering is not significant.
<code>replyTo</code>	Reply-to address for the message, or null.
<code>payload</code>	Message payload.
<code>encoding</code>	Encoding for the message, or the default encoding.
<code>exception</code>	Exception-payload of the message or null.

Example expression: `#[message:payload]`

Retrieves the payload of the current Mule message.

Restrictions:

Cannot be used with expression filters.

## **OGNL Evaluator**

The OGNL evaluator uses [OGNL](#) as the expression language. The supplied OGNL expression is applied to the payload of the Mule message.

Example expression: `#{ognl:root.of.all.equals('evil')}`

Examines whether the “all” property of the “of” property of the “root” property of the Mule message payload is equal to “evil”.

If the payload is constructed as shown in the [bean evaluator](#) example above, the example expression will evaluate to true.

Restrictions: None.

## **Payload Evaluator**

The payload evaluator retrieves the payload of the current Mule message, optionally transforming it to the supplied type.

Example expressions: `#{payload:byte[]}`, `#{payload:java.lang.String}`

Retrieves the payload of the current Mule message and, in the first example, transforms it to a byte array. In the second example, the payload is transformed to a string.

Restrictions: Cannot be used with expression filters.

## **Payload-Type Evaluator**

The payload-type evaluator allows for filtering of Mule messages with a certain type of payload.

Example expression: `<expression-filter expression="java.util.Map" evaluator="payload-type"/>`

Accepts only Mule messages which payload is of the type *java.util.Map*.

Restrictions: Can only be used with expression filters.

## **Processor Evaluator**

The processor evaluator enables invocation of a specified global message processor with the result of the nested expression as applied on the current Mule message.

A message processor can be of the following types:

- Component.
- Transformer.
- Custom processor.
- Processor chain.
- Flow.

Example expression: `#{process:myProcessor:header:INBOUND:MuleProperty2}`

Applies the processor with the name “myProcessor” to the value of the Mule message property in the inbound scope named “MuleProperty2”.

Restrictions: Available in Mule 3.1.0 and later.

## **Regex Evaluator**

The regex evaluator allows for using regular expressions in expression filters.

Example expression: <expression-filter expression="b[a-z]\*" evaluator="regex"/>

Accepts only Mule messages which payload contains at least one word that starts with the character “b”.

Restrictions: Can only be used with expression filters.

## **String Evaluator**

The string evaluator makes it possible to combine multiple expression and text. The expressions contained by the string evaluator will be evaluated and the result inserted into the string.

Example expression: #[string: The payload: #[payload:java.lang.String]]

Constructs a string consisting of “The payload: “ with the Mule message payload concatenated.

Restrictions: None to the string evaluator. Contained expressions retain their restrictions.

## **Variable Evaluator**

The variable evaluator can be used to retrieve flow variables. Note that the expression enricher with the same name looks identical to the variable evaluator but is used to store values in flow variables – the context determines whether it is the variable evaluator or the variable expression enricher.

Example expression: #[variable:myFlowVariable]

Retrieves the contents of the flow variable with the name “myFlowVariable”.

Restrictions: Available in Mule 3.1.0 and later.

The following is an example of a variable expression enricher expression used to store the result from the flow with the name “mySubFlow” in the flow variable with the name “myFlowVariable”:

```
...
<enricher target="#[variable:myFlowVariable]">
    <flow-ref name="mySubFlow"/>
</enricher>
...
```

## **Wildcard Evaluator**

The wildcard evaluator matches one or more strings that contains wildcards to the string representation of the Mule message payload or the type of the payload object.

The following wildcards can be used:

- \*  
Matches any character sequence. If the entire wildcard expression is “\*”, then matches anything.
- \*\*  
If the entire wildcard expression is “\*\*”, then matches anything.
- +  
Preceded by a classname, for instance “java.lang.String+”, matches the type of object being the Mule message payload.

Example expression: <expression-filter expression="b\*" evaluator="wildcard"/>

Restrictions: Can only be used with expression filters.

## **XPath Evaluator**

The XPath evaluator selects the text of one or more matching nodes. If there is only a single match, the result will be a string. If the result contains more than a single match, the result will be a list of strings.

The evaluator operates on the following payload types:

- String.  
String payloads are converted to DOM4J documents.
- W3C Document object.
- DOM4J Element object.
- DOM4J Document object.

Example expression: #[xpath://animal/name]

Retrieves the contents of the <name> element(s) that are children of the <animal> element, which is a root element.

Restrictions: None.

## **XPath-Node Evaluator**

The XPath-node evaluator selects one or more matching nodes. If there is only a single match, the result will be a W3C or DOM4J Document, depending on the input type. If the result contains more than a single match, the result will be a list of the aforementioned types.

The evaluator operates on the following payload types:

- String.  
String payloads are converted to DOM4J documents.
- W3C Document object.
- DOM4J Element object.
- DOM4J Document object.

Example expression: `#[xpath-node://animal/name]`

Retrieves the `<name>` element(s) that are children of the `<animal>` element, which is a root element.

Restrictions:      None.

## 6. Notifications

As mentioned in the chapter on [Mule Notifications](#), the notification mechanism in Mule is similar to the [Observer design pattern](#). There are a number of pre-defined types of notification events that are sent out in connection to different events in Mule. Below is a list of these notification event types as well as a list of the interfaces available when implementing notification listeners that receives these factory notifications.

Apart from the available notifications, Mule also supports implementing custom notifications - please refer to the Mule documentation for details.

### 6.1. Notification Event Types

In Mule, both versions 2.x and 3.x, the following pre-defined notification event types are available:

Notification Event Type	Description
CONTEXT	Mule context started, stopped etc.
MODEL	Model lifecycle state changed or components in the model registered or unregistered.
SERVICE	Service started, stopped, etc.
SECURITY	Request for authorization occurred.
ENDPOINT-MESSAGE	Message received or sent by endpoint.
COMPONENT-MESSAGE	Message processed by component.
MANAGEMENT	State of Mule instance or its resources changed.
CONNECTION	Connector connected to, released connection or failed connection attempt to resource.
REGISTRY	Event occurred on the Mule registry.
CUSTOM	Custom notification.
EXCEPTION	An exception was thrown.
TRANSACTION	Transaction begun, committed or rolled back.
ROUTING	A routing event occurred.

The notification event type values listed above are the values that are to be used in either the <notification> elements or in <disable-notification> elements. The section [Listening to Notifications](#) below show how to configure notifications in the Mule configuration file.

## 6.2. Notification Listener Interfaces

Notification listeners are observers that receive notifications when certain type(s) of events occur. By implementing one of the notification listener interfaces listed below, the listener selects what kind of events to receive. Implementing, for instance, the *ServerNotificationListener* interface will result in the listener receiving all types of events in the Mule server.

When using Mule 3.x, generics are used to specify which type of notifications the listener accepts. The following interfaces are available when implementing notification listeners.

Interface	Notification Event Type	Notes
ServerNotificationListener	-	Ancestor to all notification listener interfaces. Receives notifications when events occur in server, model and components.
MessageNotificationListener	-	Mule 2.x only. Ancestor to component and endpoint message notification listeners. Receives notification when events are sent or received by, or in, the Mule server.
ComponentMessageNotificationListener	COMPONENT-MESSAGE	Receives notification when message is processed by component.
ConnectionNotificationListener	CONNECTION	Receives notification when connector establishes connection, releases connection or connection attempt to resource fails.
CustomNotificationListener	CUSTOM	Custom notification.
EndpointMessageNotificationListener	ENDPOINT-MESSAGE	Receives notification when message received or sent by endpoint.
ExceptionNotificationListener	EXCEPTION	Receives notification when an exception was thrown.
FunctionalTestNotificationListener	-	Mule 3.x only. Receives notifications from instances of FunctionalTestComponent. See API documentation for further information.
HeartbeatNotificationListener	-	Mule 3.x only. Receives periodic notifications when the Mule server is up and running.
ManagementNotificationListener	MANAGEMENT	Receives notification when state of Mule instance or its resources changed.
MessageProcessorNotificationListener	-	Mule 3.x only. Receives notifications from message processors before and after processing a message.
ModelNotificationListener	MODEL	Receives notification when model lifecycle state changed or components in the model registered or unregistered.
MuleContextNotificationListener	CONTEXT	Receives notification when Mule context is started, stopped etc.
RegistryNotificationListener	REGISTRY	Receives notification when event occurred on the Mule registry.

RemoteDispatcherNotificationListener	-	Receives notification when requests are sent to and received from remote Mule instance.
RoutingNotificationListener	ROUTING	Receives notification when a routing event occurred.
SecurityNotificationListener	SECURITY	Receives notification when request for authorization occurs.
ServiceNotificationListener	SERVICE	Receives notification when service started, stopped, etc.
ServletContextNotificationListener	-	Mule 3.x only. Receives notifications when servlet context related to a Mule instance is initialized or destroyed (is this implemented???).
SftpTransportNotificationListener	-	Mule 3.x only. Receives notifications of SFTP operations. Does not have a corresponding notification event type.
TransactionNotificationListener	TRANSACTION	Transaction begun, committed or rolled back.

When using Mule 2.x, the *onNotification* method takes a parameter of the type *ServerNotification*. Thus all notifications which match the interface implemented by the notification listener will be delivered to the notification listener. Any filtering on the notification event type must be done programmatically in the notification listener, using the `instanceof` Java keyword.

Using Mule 3.x, the notification listener interfaces uses Java generics. When developing a notification listener, we are able to specify the exact type of notification events the listener is to receive. Mule will ensure that only notification events of the appropriate type are delivered to the listener.

For an example showing how to implement a notification listener, please see the section [Listening to Notifications](#) below!

### 6.3. Notification Events

Notification event objects are the type of objects delivered to *onNotification* methods of notification listeners. All types of notification events in Mule inherit from the class *java.util.EventObject* and the topmost class in the notification event hierarchy in Mule is *ServerNotification*. The table below lists all the default notification event classes in Mule.

Please also refer to the above section on [Notification Listener Interfaces](#) for information on when the different kinds of notification events are sent.

Notification Event Class	Source	Resource Identifier	Notes
ServerNotification			Root class of notification events.
ComponentMessageNotification	Mule message being processed by the component/flow construct.	Component name (Mule 2.x). Flow construct name (Mule 3.x).	
ConnectionNotification	Connectable resource.	Connector_name or connector_name+".dispatcher(\"+endpoint URI+)\" or connector_name+".receiver(\"+endpoint URI+)\" or lifecycle_object_name or retry_context_description.	
CustomNotification	Any object.	Any string.	
EndpointMessageNotification	Mule message sent or received by the endpoint.	Null, component or service name (Mule 2.x). Flow construct name (Mule 3.x).	
ExceptionNotification	Exception thrown.	Null (Mule 2.x). Class name of exception root cause (Mule 3.x).	
FlowConstructNotification	Flow construct name.	Flow construct name.	Mule 3.x only.
ManagementNotification	Management object that triggered notification.	Null.	
MessageProcessorNotification	MuleEvent object.	Flow construct name.	Mule 3.x only.
ModelNotification	Model object.	Model name.	
MuleContextNotification	Mule_context_domain_id+"."+Mule_context_cluster_id+"."+Mule_configuration_id.	Same as source (Mule 2.x). Mule context configuration id (Mule 3.x).	
RegistryNotification	Registry object.	Mule registry id.	Does not seem to be implemented in neither Mule 2.x nor Mule 3.x.
RemoteDispatcherNotification	Mule message.	Endpoint (of remote Mule instance) to send or receive message to/from.	
RoutingNotification	Mule message.	Null or message source URI.	
SecurityNotification	Mule 2.x: SecurityException	Mule 2.x: SecurityException string representation.	

	instance. Mule 3.x: SecurityException detailed message.	Mule 3.x: SecurityException instance class name.	
ServiceNotification	Service instance.	Service name.	
TransactionNotification	Mule 2.x: null Mule 3.x: Transaction id.	Transaction id.	

## 6.4. Listening to Notifications

The following are the steps necessary to receive notifications in a Mule application. This is only an outline of the necessary steps, for a complete example please refer to the chapter on [Mule Notifications](#) in part one of this book.

It is assumed that the kind of notification to receive is one of the factory notification types, as described in the section [Notification Listener Interfaces](#) above.

- Determine the kind of notification you want to listen to.  
In this example, we assume component message notifications.
- Select the appropriate notification listener interface and notification type from the table in the section on [Notification Listener Interfaces](#).
- Create the notification listener class.  
This class is to implement the notification listener interface selected in the previous step.  
Using Mule 3.x, we also need to specify the type of notification event the listener is to receive. For a list of available notification events, please see the above [Notification Events](#) section.

```
package com.ivan.mule;
...
/**
 * Example of a Mule 2.x notification listener class declaration.
 */
public class MyNotificationListener implements ComponentMessageNotificationListener
{
...
}
```

```
...
/**
 * Example of a Mule 3.x notification listener class declaration.
 */
public class MyNotificationListener implements
    ComponentMessageNotificationListener<ComponentMessageNotification>
{
...
}
```

- Implement the *onNotification* method in the new notification listener class.
- In the Mule configuration file, create a Spring bean that is implemented by the new notification listener class.

```
...
<spring:bean
    name="notificationListener1"
    class="com.ivan.mule.MyNotificationListener" />
...
```

- In the Mule configuration file, add a <notifications> element if not already present. The <notifications> element is an immediate child element to the root <mule> element.

```
<?xml version="1.0" encoding="UTF-8"?>
<mule ...>
  ...
  <!--
    Add the dynamic="true" attribute if notification listeners are to be
    added at runtime, after having started the Mule context.
  -->
  <notifications>
  ...
  </notifications>

  ...
</mule>
```

- In the <notifications> element, add a <notification> element as shown below. This element tells Mule that the application wants to listen for COMPONENT-MESSAGE notifications.

```
...
<notifications>
  <notification event="COMPONENT-MESSAGE" />
  ...
</notifications>
...
```

- In the <notifications> element, add a <notification-listener> element, as below. This element register a notification listener with Mule. The interface(s) implemented by the listener class decides the kind of notification events that will be sent to the listener. The value of the ref attribute contains the name of the Spring bean which is to be the notification listener.

```
...
<notifications>
  <notification event="COMPONENT-MESSAGE" />
  <notification-listener ref="notificationListener1" />
</notifications>
...
```

## 6.5. Listening to Notifications from a Specific Component

As per default, notifications from all sources are reported to a registered notification listener. It is possible to limit the reported notifications to those originating from a specific component only by using the *subscription* attribute in the <notification-listener> element.

The value of the attribute contains the name of the component, for instance a service or a flow, that is to be the only source of notifications to the listener.

Example:

```
...
<notifications>
  <notification event="COMPONENT-MESSAGE" />
  <notification-listener ref="notificationListener1" subscription="GreetingFlow" />
</notifications>
...
```

The wildcard “\*” can be used in the *subscription* attribute value, that is “\*Flow\*” will cause notifications from any component which name contains “Flow” to be accepted.

## 6.6. Disabling Notifications

It is possible to disable the delivery of notifications using the <disable-notification> element. The following options are available:

- Disabling notifications to listeners implementing a certain interface.  
Use the *interface-class* attribute of the <disable-notification> element and specify an interface.

```
...
<notifications>
  ...
    <disable-notification
      interface-class=
        "org.mule.api.context.notification.EndpointMessageNotificationListener" />
  </notifications>
...
```

- Disabling certain notification event types.  
Use the *interface* attribute of the <disable-notification> element and specify the notification type event to be disabled.

```
...
<notifications>
  ...
    <disable-notification interface="ENDPOINT-MESSAGE" />
  </notifications>
...
```

## 6.7. Registering a Notification Listener Programmatically

Notification listeners can also be registered programmatically, as will be shown in the following example.

In order to be able to register notification listeners after the Mule context has been started, the *dynamic* attribute on the <notifications> element in the Mule configuration file must be set to true.

```
...
<notifications dynamic="true">
  ...
</notifications>
...
```

The following example code shows how to programmatically register a notification listener. The filtering on the component name is optional.

```
/*
 * Programmatically register a notification listener on the
 * Mule context retrieved from the current Mule message.
 * Note that in order to be able to register notification
 * listeners when the Mule context is started, the
 * attribute dynamic on the <notifications> element in the
 * Mule configuration must be set to true.
 */
MuleMessage theMuleMsg = inNotification.getSource();
MuleContext theMuleContext = theMuleMsg.getMuleContext();

/* Create the new listener instance to be registered. */
MyMsgNotificationListener3 theNewListener = new MyMsgNotificationListener3();

try
{
    /*
     * The second parameter to the registerListener method
     * adds filtering, causing the notification listener
     * only to accept notifications originating components
     * with names matching the supplied name.
     * Note how the * wildcard may be used.
     */
    theMuleContext.registerListener(theNewListener, "*Service*");
} catch (final NotificationException theException)
{
    theException.printStackTrace();
}
```

Note that:

- According to the Mule documentation, the *dynamic* attribute of the <notifications> element must be set to true, in order to be able to register notification listeners on a Mule context that has been started.
- There are two versions of the *registerListener* method on the *MuleContext* interface. Both methods take a notification listener instance as parameter. In addition, the second method takes a string parameter. This string has the same function as the *subscription* attribute on the <notification-listener> element, allowing us to specify the name of components from which notifications will be accepted. When specifying this name, the wildcard “\*” can be used, as seen in the above example.

## 7. Mule JMX Management

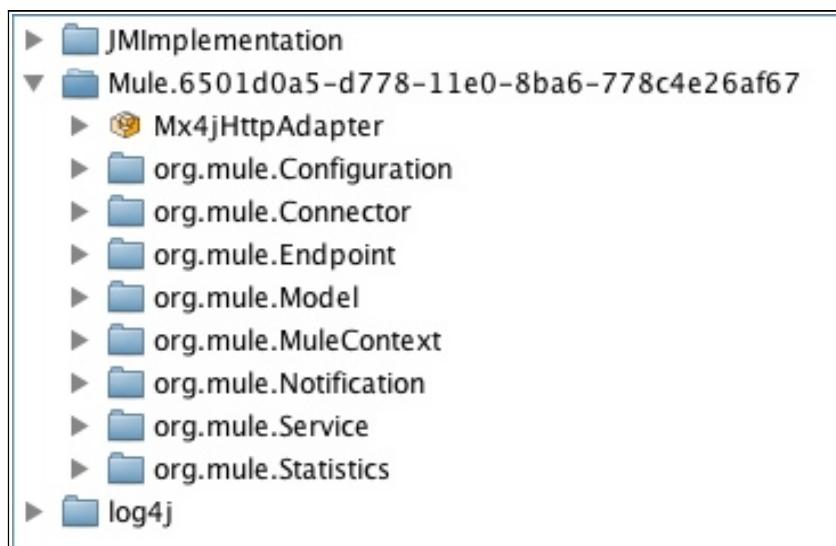
This section contains reference of the Mule JMX management options; first we'll look at the options available when using Mule 2.x and then we will look at the additional options made available when using Mule 3.x.

To look at the JMX management options on your own computer, you can use the example program in the [Monitoring Mule](#) chapter in part one of this book.

### Mule 2.x JMX Management

In this section we will look more closely at the JMX management options available when running a Mule 2.x server instance.

- Expand the domain with the name starting with “Mule” in the JConsole window.  
The result should look like this:



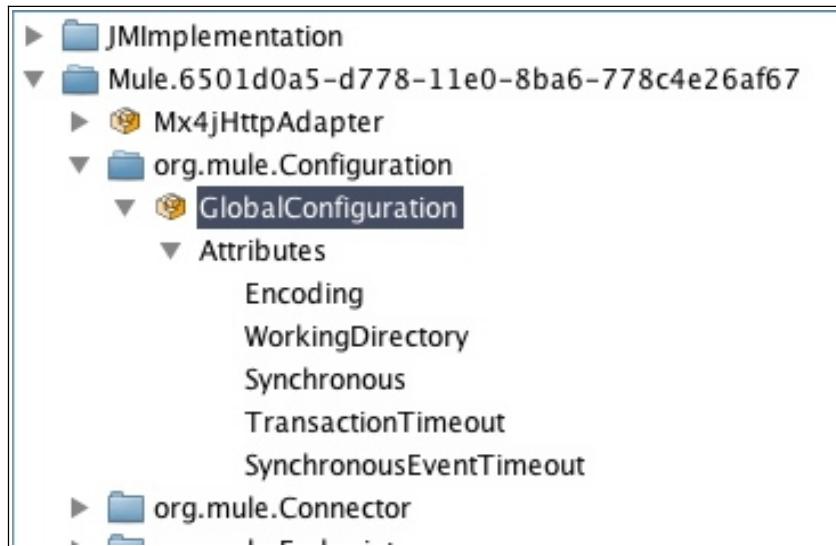
Examining a Mule 2.x server node in JConsole.

The names of the children should give a good indication of what kind of management data can be found under the child. We will look at each child except for the “Mx4jHttpAdapter” MBean.

### Mule 2.x Server Global Configuration

The node `org.mule.Configuration` contains the `GlobalConfiguration` MBean that reflects the global configuration of the Mule instance. All the information in this MBean is read-only.

- Expand the “`org.mule.Configuration`” node in JConsole and all its children.  
The result should look like this:



Mule 2.x server global configuration monitoring in JConsole.

We can see the following attributes being available in the GlobalConfiguration MBean:

Attribute Name	Writable	Description
Encoding	no	Global message encoding. Ex. UTF-8.
WorkingDirectory	no	Mule instance's working directory.
Synchronous	no	Makes all endpoints synchronous if true.
TransactionTimeout	no	Default transaction timeout in milliseconds.
SynchronousEventTimeout	no	Default timeout for synchronous events in milliseconds.

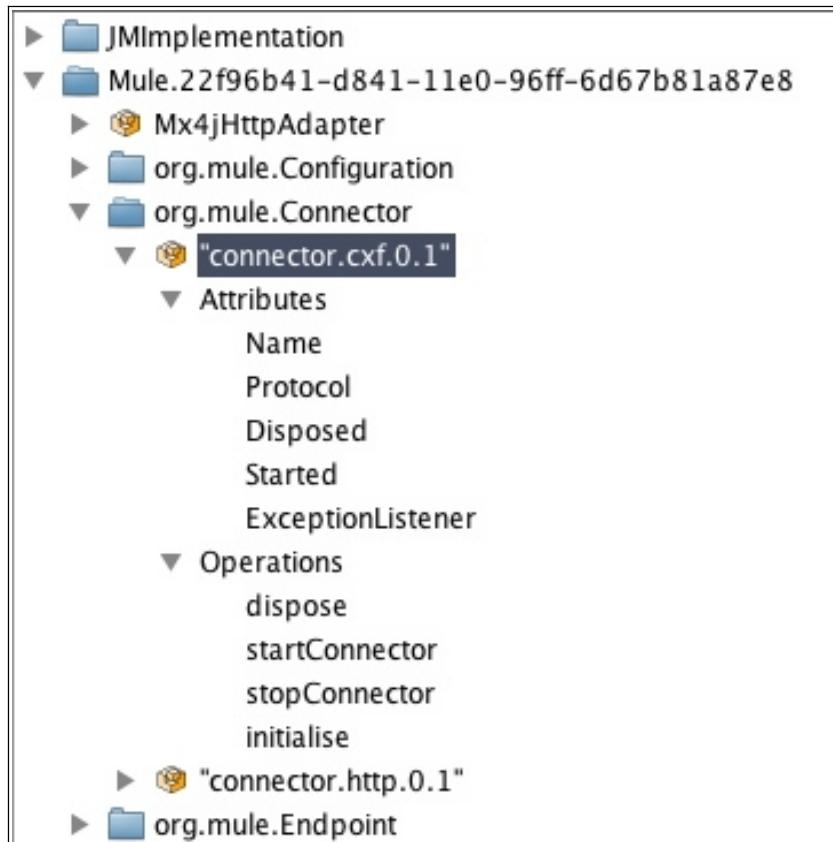
## Mule 2.x Connectors Configuration

The node org.mule.Connector allows us to view the configuration of the current connectors in the Mule instance and to apply certain operations to a connector.

- Expand the “org.mule.Connector” node in JConsole.

Also expand its first child MBean, the CXF connector, and its attributes and operations nodes.

The result should look like this:



Mule 2.x connector configuration monitoring in JConsole.

We see that the following attributes are available for a connector:

Attribute Name	Writable	Description
Name	no	Name of the connector. “connector.cxf.0.1” in this case.
Protocol	no	Name of protocol used by the connector.
Disposed	no	Connector disposed flag.
Started	no	Connector started flag.
ExceptionListener	no	Exception listener being notified of exceptions occurring in the connector.

The following operations are available on a connector:

Operation Name	Description
dispose	Life-cycle method freeing resources associated with the connector.
startConnector	Starts the connector.
stopConnector	Stops the connector.
initialize	Life-cycle method initializing the connector, allocating any resources.

See also the API documentation of the *org.mule.api.transport.Connector* interface.

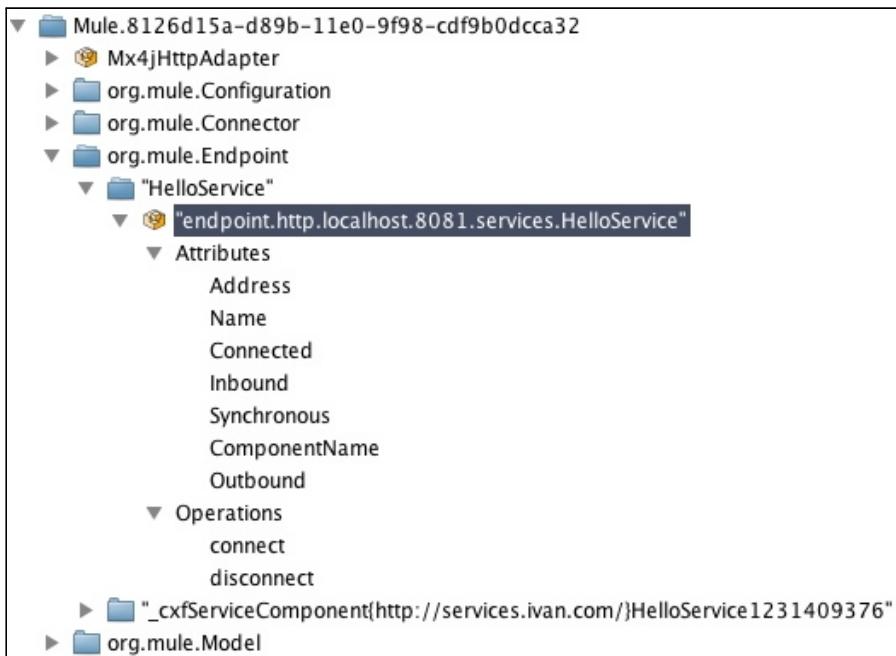
## Mule 2.x Endpoint Configuration

The node `org.mule.Endpoint` allows us to view the configuration of endpoints in the Mule instance and to apply certain operations to endpoints.

- Expand the “`org.mule.Endpoint`” node in JConsole.

Also expand its first child MBean, the endpoint with the name

“`endpoint.http.localhost.8081.services.HelloService`”, and its attributes and operations nodes. The result should look like this:



Mule 2.x server endpoint configuration monitoring in JConsole.

We see that the following attributes are available for an endpoint:

<b>Attribute Name</b>	<b>Writable</b>	<b>Description</b>
Address	no	Address of endpoint.
Name	no	Name of endpoint.
Connected	no	Endpoint connected flag.
Inbound	no	Endpoint inbound flag.
Synchronous	no	Endpoint synchronous flag.
ComponentName	no	Name of service component containing the endpoint.
Outbound	no	Outbound direction endpoint flag.

The following operations are available on an endpoint:

<b>Operation Name</b>	<b>Description</b>
connect	Connects the endpoint.
disconnect	Disconnects the endpoint.

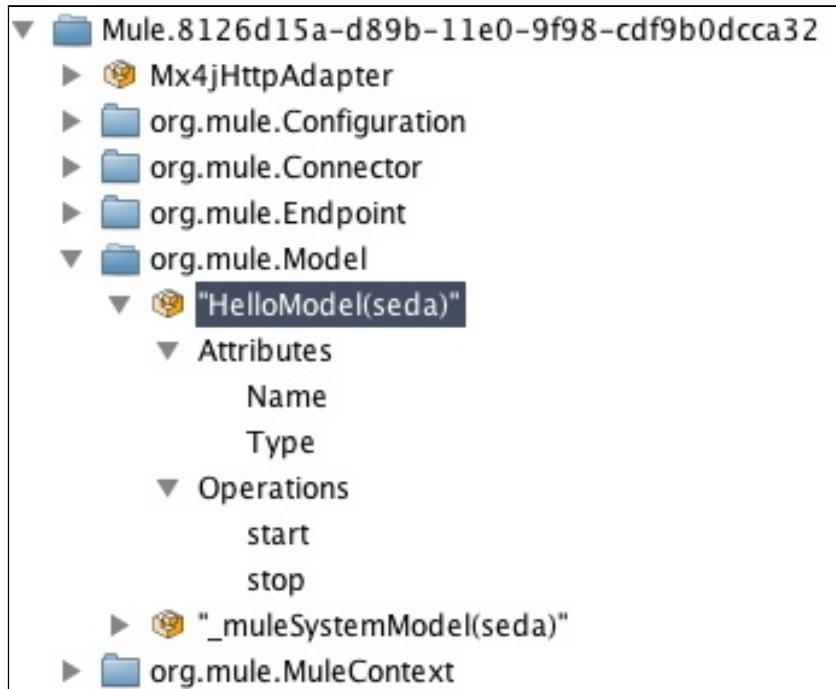
## Mule 2.x Model Configuration

The node org.mule.Model allows us to view the configuration of models in the Mule instance and to apply certain operations to these models. The first model, the “HelloModel(seda)” corresponds to the <model> element in this chapter’s Mule 2.x configuration file. “Seda” stands for “staged event-driven architecture” and is a design used internally by Mule.

- Expand the “org.mule.Model” node in JConsole.

Also expand its first child MBean, with the name “HelloModel(seda)”, and the Attributes and Operations nodes of the MBean.

The result should look like this:



Mule 2.x server model configuration monitoring in JConsole.

A model has the following attributes:

Attribute Name	Writable	Description
Name	no	Name of the node. Matches the name of the <model> element.
Type	no	Type of model. Commonly “seda” - staged event-driven architecture.

The following operations are available on a model:

Operation Name	Description
start	Starts the model.
stop	Stops the model.

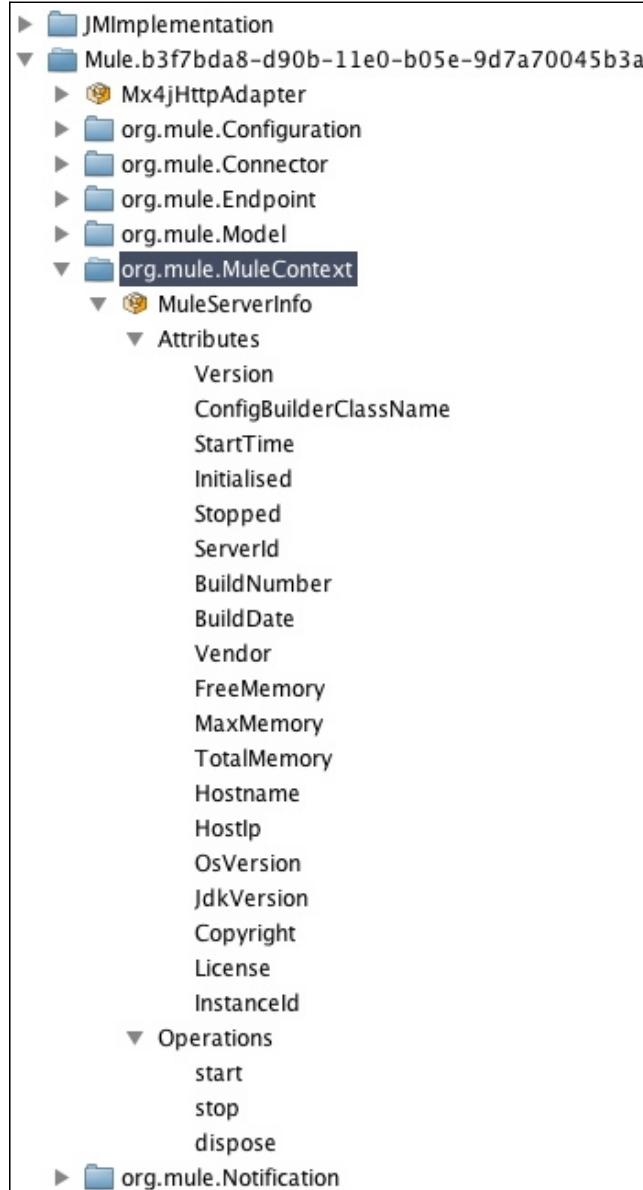
Starting and stopping the model in our example does not seem to yield any reaction.

## Mule 2.x Context Configuration

Next up is Mule contexts, found in the org.mule.MuleContext node. In this node we can find a MuleServerInfo MBean which holds information about the Mule server and the environment it runs in.

- Expand the “org.mule.MuleContext” node in JConsole.  
Also expand its first child node, the MuleServerInfo MBean, and its Attributes and Operations nodes.

The result should look like this:



Mule 2.x server information monitoring in JConsole.

The MuleServerInfo MBean has the following attributes:

<b>Attribute Name</b>	<b>Writable</b>	<b>Description</b>
Version	no	Mule version running on the server.
ConfigBuilderClassName	no	Mule configuration builder class. See the AutoConfigurationBuilder class in the Mule 2.x API documentation for details.
StartTime	no	Date object holding start time of server.
Initialised	no	Server initialized flag.
Stopped	no	Server stopped flag.
ServerId	no	Server id string.
BuildNumber	no	Mule server build number.
BuildDate	no	Mule server build date.
Vendor	no	Vendor that created the Mule server.
FreeMemory	no	Available memory in the Mule server instance.
MaxMemory	no	Maximum memory in the Mule server instance.
TotalMemory	no	Total memory in the Mule server instance.
Hostname	no	Hostname at which the Mule server instance may be accessed.
HostIp	no	Host IP address at which the Mule server instance may be accessed.
OsVersion	no	OS version the Mule server instance runs in.
JdkVersion	no	Version of the JDK on which the Mule server instances is run.
Copyright	no	Mule copyright string.
License	no	Mule license string.
InstanceId	no	Mule instance id string. Same as ServerId.

The following operations are available on a MuleServerInfo MBean:

<b>Operation Name</b>	<b>Description</b>
start	Starts the Mule context.
stop	Stops the Mule context and all agents. Does not release allocated resources, such as threads, so the JVM in which the server is running will not terminate.
dispose	Stops the Mule context and releases associated resources. The JVM in which the server is running will terminate.

Stopping the Mule context stops all services and agents running on the Mule server. We will not be able to monitor or manage the Mule server once its context has been stopped, despite the JVM in

which the server was running not being terminated.

Disposing the Mule context shuts down the Mule server and releases resources. The JVM in which the server was running will be terminated, provided there are no additional resources.

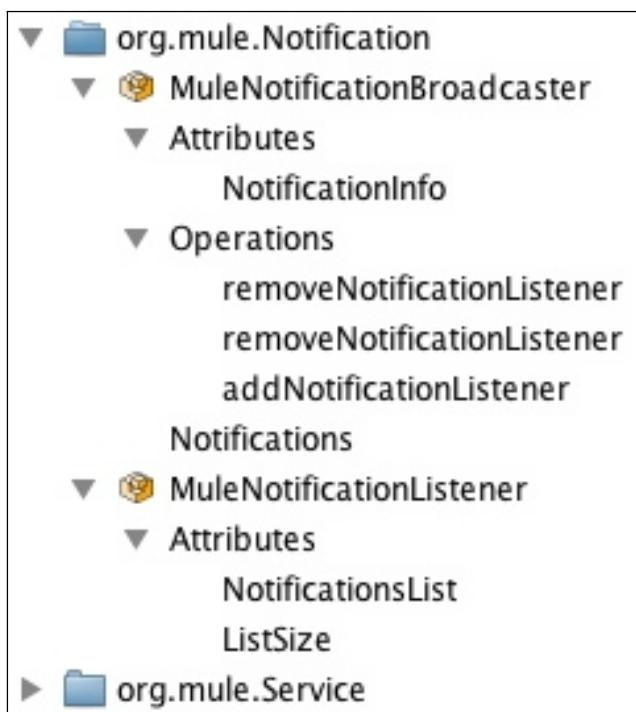
## Mule 2.x Notification Configuration

Mule provides a notification mechanism that enables agents and components in the Mule server to be notified about changes in the Mule server. For further details, please refer to the section on “Mule Server Notifications” in the Mule user documentation.

- Expand the “org.mule.Notification” node in JConsole.

Also expand its first and second child MBeans, the MuleNotificationBroadcaster and MuleNotificationListener, and their Attributes, Operations and Notifications nodes as available.

The result should look like this:



Mule 2.x notification monitoring in JConsole.

The MuleNotificationBroadcaster MBean allows for adding and removing notification listeners. These features are more usable if you are writing your own program using JMX to monitor a Mule server – inside a monitoring program, such as JConsole, there is little we can do. The MuleNotificationListener MBean maintains a list of notifications.

The MuleNotificationBroadcaster MBean has the following attributes:

Attribute Name	Writable	Description
NotificationInfo	no	An array of <i>javax.management.MBeanNotificationInfo</i> objects holding information about the different types of notifications that can be emitted.

The following operations are available on a MuleNotificationBroadcaster MBean:

<b>Operation Name</b>	<b>Description</b>
removeNotificationListener	Removes a listener from the MBean.
removeNotificationListener	Removes a listener from the MBean.
addNotificationListener	Adds a listener to the MBean.

The MuleNotificationListener MBean has the following attributes:

<b>Attribute Name</b>	<b>Writable</b>	<b>Description</b>
NotificationsList	no	List of notifications.
ListSize	no	Size of the list of notifications.

## Mule 2.x Service Configuration

The org.mule.Service node contains MBeans allowing us to monitor all the services in our Mule deployment.

- Expand the “org.mule.Service” node in JConsole.  
Also expand its first child, the “HelloService” MBean, and its Attributes and Operations nodes.

The result should look like this:



Mule 2.x service monitoring in JConsole.

A service MBean, in the above picture “HelloService”, has the following attributes:

<b>Attribute Name</b>	<b>Writable</b>	<b>Description</b>
Name	no	Name of the service.
Statistics	no	Object holding service statistics.
Stopped	no	Flag indicating whether service is stopped.
Stopping	no	Flag indicating whether service is in the process of stopping.
Paused	no	Flag indicating whether service is paused.
QueueSize	no	Size of queue holding events to be processed by the service.
QueuedEvents	no	Number of events queued to be processed by the service.
MaxQueueSize	no	Maximum number of events in queue.
AverageQueueSize	no	Total number of asynchronous events received divided by total number of queued events.
SyncEventsReceived	no	Number of synchronous events received.
AsyncEventsReceived	no	Number of asynchronous events received.
TotalEventsReceived	no	Total number of events received.
SyncEventsSent	no	Number of synchronous events sent.
AsyncEventsSent	no	Number of asynchronous events sent.
ReplyToEventsSent	no	Number of reply-to events sent. Reply-to events are messages with a reply-to URI to which the reply of the message is to be sent.
TotalEventsSent	no	Total number of events sent.
ExecutedEvents	no	Number of events executed since last component statistics reset.
ExecutionErrors	no	Number of errors during execution.
FatalErrors	no	Number of errors logged as fatal errors. Depends on the exception handling strategy.
MinExecutionTime	no	Shortest time in milliseconds spent processing an event.
MaxExecutionTime	no	Longest time in milliseconds spent processing an event.
AverageExecutionTime	no	Average time in milliseconds spent processing an event.
TotalExecutionTime	no	Total execution time in milliseconds spent processing messages.

The following operations are available on a service MBean:

<b>Operation Name</b>	<b>Description</b>
resume	Resume event processing for the service.
dispose	Stop and dispose of the service.
pause	Pause event processing for the service.
forceStop	Force stop event processing for the service.
stop	Stop the event processing service.
start	Start the event processing of the service.
clearStatistics	Clears statistics for the service. Note that not all attribute values are reset.

What is the difference between stopping and pausing a service, one may ask?

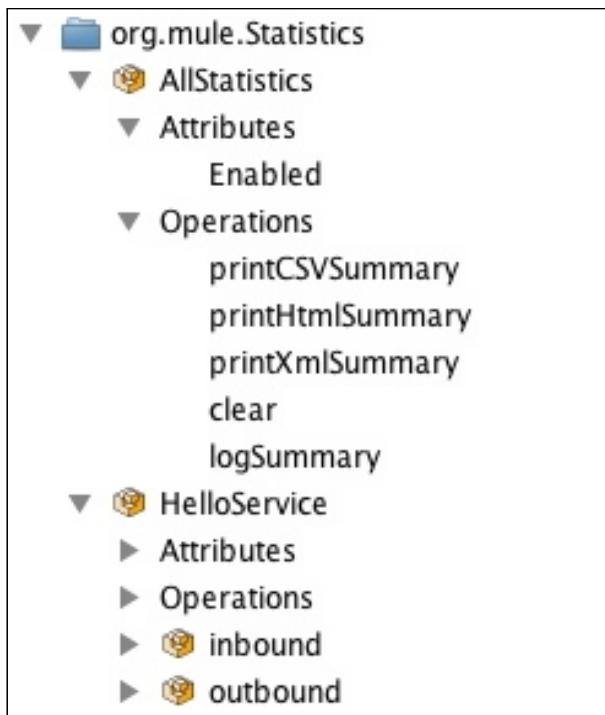
This depends on the service, but for the CXF web service in this example the HTTP listener is closed when the service is stopped. This is not the case when the service is paused – the HTTP listener remains active while the service is paused.

## Mule 2.x Statistics Configuration

The org.mule.Statistics node contains MBeans allowing us to retrieve statistics about the services in a Mule instance etc.

- Expand the “org.mule.Statistics” node in JConsole.

Also expand both its child MBeans as well as all the child nodes of the AllStatistics MBean.  
The result should look like this:



Mule 2.x statistics monitoring in JConsole.

The AllStatistics MBean has one single attribute, Enabled, which may be both read and written. Using this attribute, the accumulation of statistics can be enabled or disabled during execution of the Mule server.

In addition to the attribute, the AllStatistics MBean also has a number of operations related to all statistics of the Mule server:

Operation Name	Description
printCSVSummary	Generates a statistics summary report in CSV format.
printHtmlSummary	Generates a statistics summary report in HTML format.
printXmlSummary	Generates a statistics summary report in XML format.
logSummary	Generates a statistics summary report and outputs it to the log.
clear	Clears statistics and resets the statistics accumulation period. Note that not all attribute values are reset.

The HelloService MBean holds statistics of the HelloService service, most parts of which we have already seen in the earlier section on service configuration.

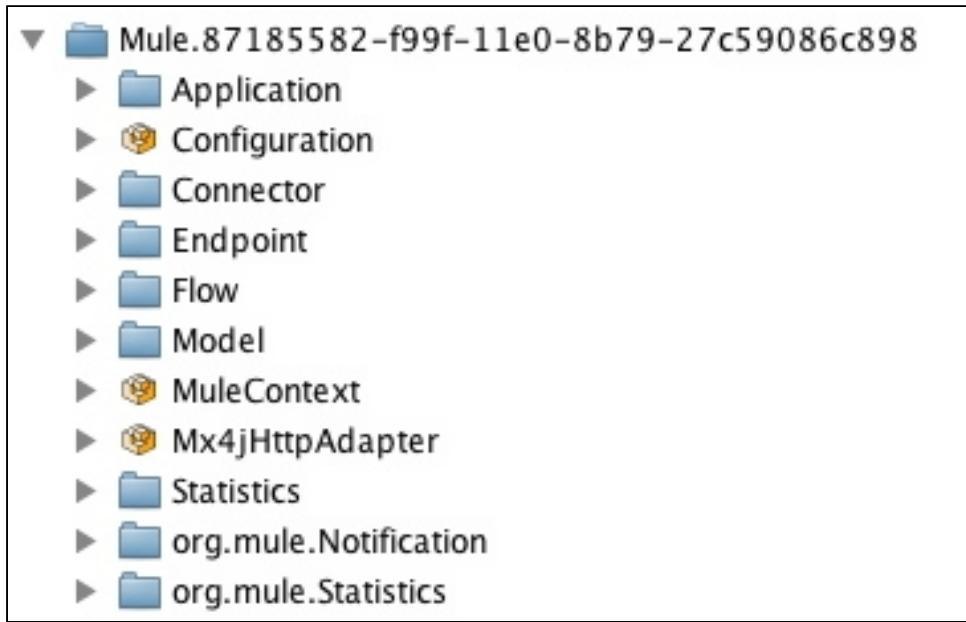
This concludes the section on Mule 2.x JMX management.

## **Mule 3.x JMX Management**

In this section we will look at the JMX management options available when running a Mule 3.x server instance. Only the management options that differ from those already discussed in connection to Mule 2.x JMX management will be described.

I use Mule 3.2 when running the example program and looking at the available management options. The available management options differ slightly between, for instance, Mule 3.1 and Mule 3.2.

- Expand the domain with the name starting with “Mule” in the JConsole window.  
The result should look like this:



Examining a Mule 3.x server node in JConsole.

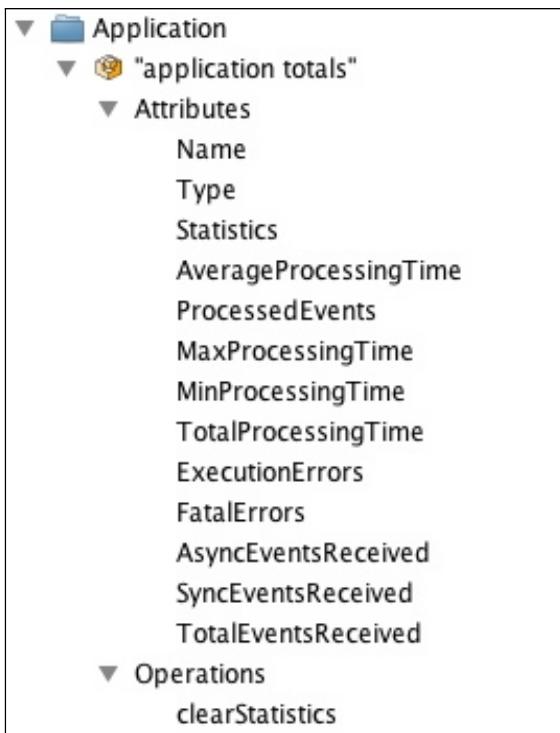
The following are different from what we saw in the Mule 2.x server:

- The names of many categories (“folders”) have been shortened.
- In Mule 3.x there is a new category named “Application”.  
This category is actually new for Mule 3.2 and contains application-level statistics.
- “Configuration” and “MuleContext” are MBeans in Mule 3.x, instead of being categories, as in Mule 2.x.
- In Mule 3.x there is a new category, the “Flow” category.  
The reason for this new category is that flows were introduced in Mule 3.x.

## Mule 3.x Application Statistics

The category contains an “application totals” MBean that contains attributes reflecting aggregated statistics of the application. The different kinds of statistics available in this MBean is also available in the “application totals” MBean under the org.mule.Statistics node.

- Expand the “Application” node, its child MBean “application totals” and all its child nodes in JConsole. The result should look like this:



Mule 3.x application level statistics monitoring in JConsole.

The following attributes are available in the “application totals” MBean:

Attribute Name	Writable	Description
Name	no	Name of the MBean. Always “application totals”.
Type	no	Type of statistics. Always “Application”.
Statistics	no	Object holding application statistics.
AverageProcessingTime	no	Average time in milliseconds spent processing an event.
ProcessedEvents	no	Number of events processed by the application.
MaxProcessingTime	no	Longest time in milliseconds spent processing an event.
MinProcessingTime	no	Shortest time in milliseconds spent processing an event.
TotalProcessingTime	no	Total time in milliseconds spent processing events.
ExecutionErrors	no	Number of errors during execution of the application.

FatalErrors	no	Number of errors logged as fatal errors in the application. Depends on the exception handling strategy.
AsyncEventsReceived	no	Number of asynchronous events received by the application.
SynchEventsReceived	no	Number of synchronous events received by the application.
TotalEventsReceived	no	Total number of events received by the application.

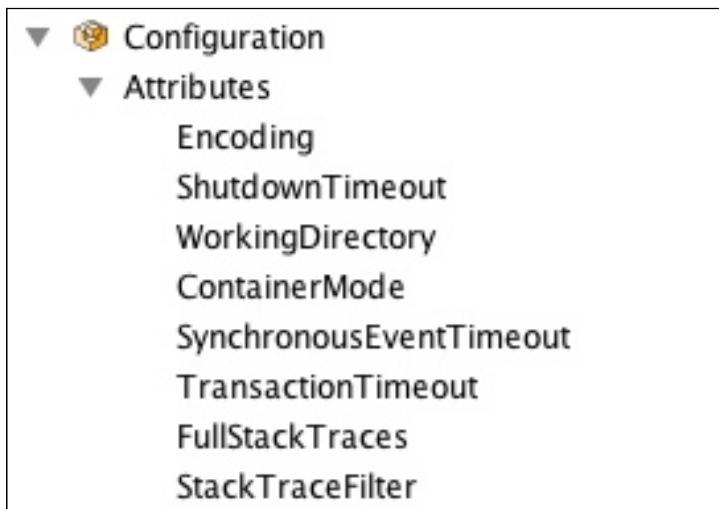
The following operation is available for the application:

Operation Name	Description
clearStatistics	Clears the statistics of the application (all statistics attributes listed above).

### Mule 3.x Server Global Configuration

The MBean Configuration contains attributes reflecting the global configuration of the Mule instance. Please also refer to the section on [Mule 2.x Server Global Configuration](#) above.

- Expand the “Configuration” MBean in JConsole and all its child nodes.  
The result should look like this:



Mule 3.x server global configuration monitoring in JConsole.

There are a few new attributes, some of which are writeable, allowing for changes in the configuration of a running Mule 3.x server:

Attribute Name	Writable	Description
ShutdownTimeout	no	Time in milliseconds to wait for messages being processed when shutting down instance. Default is 5000.
ContainerMode	no	Flag indicating whether Mule instance is running in container mode (true) or in embedded mode (false).
FullStackTraces	yes	If false, certain internal Mule-related entries are removed from stack traces to increase readability.
StackTraceFilter	yes	Comma-separated list of packages that are to be removed from sanitized stack traces.

Container mode means the JVM in which the Mule instance is running were started by launching the Mule server.

### Mule 3.x Connectors Configuration

The connectors node has changed name from org.mule.Connector to Connector and one attribute, ExceptionListener, has been removed.

For details on the remaining attributes and operations, please refer to the section on [Mule 2.x Connectors Configuration](#) above.

## Mule 3.x Endpoint Configuration

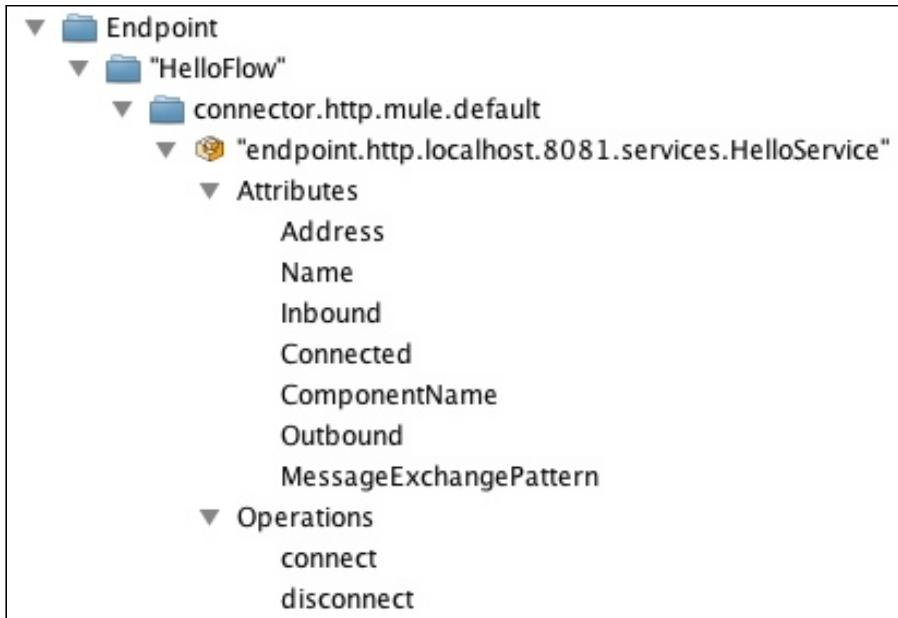
The endpoints node has changed name from org.mule.Endpoint to Endpoint. The Synchronous attribute has been removed and a new attribute, MessageExchangePattern, has been added.

The organization of the nodes and MBeans below the Endpoint node has also changed slightly, as can be seen in the picture below.

For details on the remaining attributes and operations, please refer to the section on [Mule 2.x Endpoint Configuration](#) above.

- Expand the “Endpoint” node in JConsole.

Also expand its first child node, the node with the name “connector.http.0”, the MBean with the name “endpoint.http.localhost.8081.services.HelloService”, and its attributes and operations nodes. The result should look like this:



Mule 3.x endpoint configuration monitoring in JConsole.

Note also that the immediate child of the Endpoint node is the HelloFlow node. This reflects the fact that the Mule 3.x version of the example program uses a flow instead of a model and a service.

## Mule 3.x Flow Configuration

Entirely new in Mule 3.x, compared to Mule 2.x, is the Flow node which allows us to view some statistics of flows in the Mule instance.

- Expand the “Flow” node in JConsole.

Also expand its child MBean “HelloFlow” and its attributes and operations nodes. The result should look like this:



Mule 3.x flow configuration monitoring in JConsole.

We see that the following attributes are available for a flow:

<b>Attribute Name</b>	<b>Writable</b>	<b>Description</b>
Name	no	Name of the flow.
Type	no	Always “Flow”.
Statistics	no	Object holding flow statistics.
SynchEventsReceived	no	Number of synchronous events received by the flow.
AsyncEventsReceived	no	Number of asynchronous events received by the flow.
TotalEventsReceived	no	Total number of events received by the flow.
ExecutionErrors	no	Number of errors during execution of the flow.
FatalErrors	no	Number of errors logged as fatal errors. Depends on the exception handling strategy.
ProcessedEvents	no	Number of events processed by the flow.
MinProcessingTime	no	Shortest time in milliseconds spent processing an event.
MaxProcessingTime	no	Longest time in milliseconds spent processing an event.
AverageProcessingTime	no	Average time in milliseconds spent processing an event.
TotalProcessingTime	no	Total time in milliseconds spent processing events.

The following operation is available for a flow:

<b>Operation Name</b>	<b>Description</b>
clearStatistics	Clears the statistics of the flow (all statistics attributes listed above).

### Mule 3.x Model Configuration

The model configuration node has, in Mule 3.x, changed name from org.mule.Model to Model. No changes in the contained MBeans, attributes and operations have been made. For details on the attributes and operations, please refer to the section on [Mule 2.x Model Configuration](#) above.

## Mule 3.x Context Configuration

The org.mule.MuleContext node we saw when exploring the Mule 2.x configuration has been replaced with a MBean named MuleContext. Probably due to the fact that one Mule instance always has one single context.

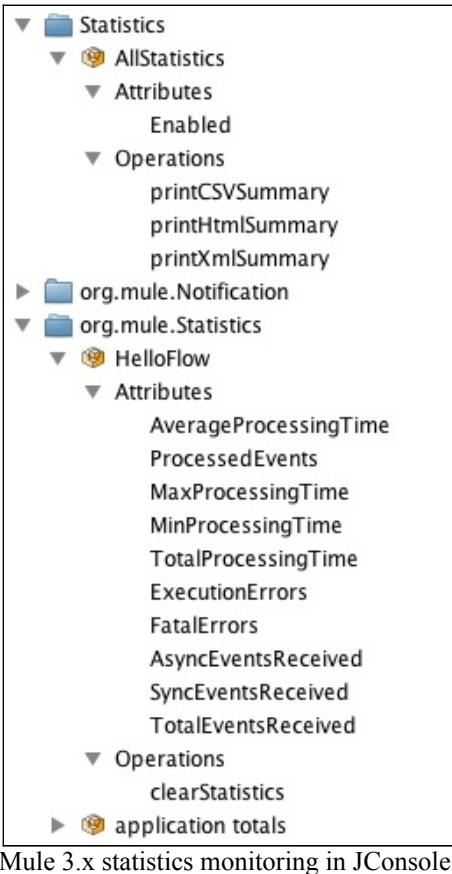
No changes in the attributes or operations available in the MBean have been made. For details, please refer to the section on [Mule 2.x Context Configuration](#) above.

## Mule 3.x Statistics Configuration

In a Mule 3.x instance, there are now two different nodes related to statistics; the org.mule.Statistics node but also the Statistics node.

- Expand the “org.mule.Statistics” and “Statistics” nodes in JConsole.  
Also expand both the AllStatistics and HelloFlow MBeans and all their children.

The result should look like this:



Mule 3.x statistics monitoring in JConsole.

The AllStatistics MBean contain the same attributes and operations, except for the clear and logSummary operations, as in Mule 2.x – please refer to the section on [Mule 2.x Statistics Configuration](#) above for details.

The MBean holding statistics for the HelloFlow is located under the org.mule.Statistics node. It duplicates the information we saw in the above section on [Mule 3.x Flow Configuration](#).

New in Mule 3.2 is the “application totals” MBean, which contain statistic information about the entire application, as discussed [earlier](#).

## **Mule 3.x Notification Configuration**

The options available concerning notification configuration and monitoring are identical with those available in Mule 2.x. Please refer to the [Mule 2.x Notification Configuration](#) section above.

This concludes the section on Mule 3.x JMX management.

## **8. Package a Mule Application**

An application using Mule can be packaged in different ways. If Mule runs embedded, that is started programmatically from the application, then the application can be packaged as a regular Java application or Java web application etc.

If, however, the application is to be deployed to a standalone Mule instance or a Mule instance running in a web container or in an application server, then application needs to be packaged along the instructions outlined in this section.

### **8.1. Package Mule 2.x Applications**

Mule 2.x applications cannot be packaged as one single unit, when deploying to a standalone Mule 2.x server. One Mule configuration file needs to be kept separate, while the rest of the artifacts, such as compiled classes, additional configuration files etc. can be packaged in a regular JAR file.

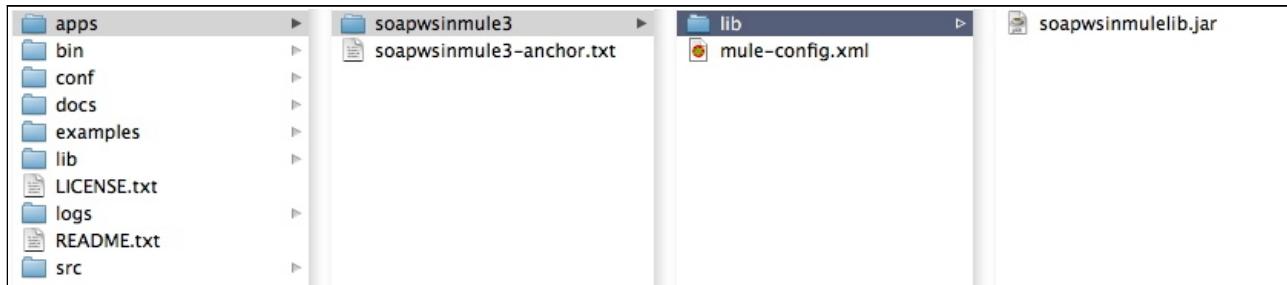
Prior to starting the standalone Mule 2.x server, the JAR file with additional artifacts must be copied to the classpath of the Mule 2.x server. The recommended location is the “lib/user” directory in the Mule home.

The Mule configuration file can be located anywhere, since we can give the complete path to the configuration file as a parameter to Mule when launching the server.

### **8.2. Package Mule 3.x Applications**

In Mule 3.x, the “apps” directory in the Mule home has been introduced as the preferred location to which to deploy Mule applications. Mule 3.x also supports the deployment of multiple applications running in one and the same standalone Mule server.

A Mule 3.x application is packaged in a directory with the following structure:



The Mule 3.x application “soapwsinmule3” deployed to the “apps” directory in the Mule home.

Note that:

- The name of the directory, “soapwsinmule3” in the above example, will become the name of the Mule application.
- The application directory contains a Mule configuration file.  
The name of the file is “mule-config.xml”, which enables Mule to find it.
- The application directory contains a “lib” directory.  
Place all the library JAR files containing the classes, configuration files etc required by the Mule application in this directory. Only the artifacts not already present in the Mule server need to be included here.
- The “apps” directory contains a file named “soapwsinmule3-anchor.txt”.  
This file will appear after the Mule application has been successfully deployed to the Mule

server. It is used to undeploy the application; delete this file and Mule will undeploy the application.

## 9. Testing

Mule contains some features that makes testing of Mule integrations significantly easier. There is, for instance, a special test component and a number of unit test-case classes.

### 9.1. Exception Component

There is no need to implement a component that throws exceptions, unless there are particular demands. Instead the `<component>` element in the test namespace can be used, as shown in the following example:

```
...
<!--
    This test component is will always throw a
    NullPointerException when invoked.
-->
<test:component throwException="true"
    exceptionToThrow="java.lang.NullPointerException"
    exceptionText="Exception thrown by test component." />
...
```

Note that:

- The test component is available in both Mule 2.x and Mule 3.x.
- The test component is implemented by the class *FunctionalTestComponent*. Please refer to the API documentation of this class for detailed information on the capabilities of the test component.
- The *throwException* attribute of the `<test:component>` only allows for static declarations of either true or false.
- Using the *exceptionToThrow* attribute, it is possible to specify the kind of exception that will be thrown when the component is invoked. The default exception type is *FunctionalTestException*.
- The *exceptionText* attribute allows us to specify the message of the exception that will be thrown.
- Test components that are configured to throw an exception will never log messages.

If the above component is invoked, we will see log output similar to this (some output omitted to conserve space):

```
ERROR 2012-02-27 06:56:53,111 [connector.http.mule.default.receiver.02]
com.ivan.exceptionhandlers.MyCustomExceptionHandler:
*****
Message          : Component that caused exception is:
DefaultJavaComponent{MySimpleRegularService.commponent}. Message payload is of type: String
Code             : MULE_ERROR--2
-----
Exception stack is:
1. Exception thrown by test component. (java.lang.NullPointerException)
   sun.reflect.NativeConstructorAccessorImpl:-2 (null)
2. Component that caused exception is: DefaultJavaComponent{MySimpleRegularService.commponent}.
   Message payload is of type: String (org.mule.component.ComponentException)
   org.mule.component.DefaultComponentLifecycleAdapter:359
   (http://www.mulesoft.org/docs/site/current3/apidocs/org/mule/component/ComponentException.html)
-----
Root Exception stack trace:
...
+ 3 more (set debug level logging or '-Dmule.verbose.exceptions=true' for everything)
```

\*\*\*\*\*

## 9.2. Return Mock Data from a Component

The test component can be configured to return static data. We use the `<component>` element from the test namespace. To return static data specified in the Mule configuration file, we use the `<return-data>` element from the test namespace:

```
...
<!--
    This test component will return the static data specified
    in the <test:return-data> child element.
-->
<test:component>
    <test:return-data>some data</test:return-data>
</test:component>
...
```

Note that:

- It is possible to use [Mule expressions](#) in the `<return-data>` element.  
Example: `<test:return-data>#[function:datestamp]</test:return-data>`

An alternative is to place the data to be returned by the component in a file and use the *file* attribute of the `<return-data>` element:

```
...
<!--
    This test component will return the contents of the
    file which name is specified using the file attribute
    of the <return-data> element.
-->
<test:component>
    <test:return-data file="message-contents-file.xml"/>
</test:component>
...
```

## 9.3. Logging Message Details

Setting the `logMessageDetails` attribute of the `<test:component>` element will cause the test component to log the payload and properties of messages sent to the component.

```
...
<!--
    This test component will log payload and properties
    of messages sent to the component.
-->
<test:component logMessageDetails="true" />
...
```

Output from the above component will look like this when sending a GET request to a HTTP service endpoint:

```
INFO 2012-02-27 18:02:59,694 [connector.http.mule.default.receiver.02]
org.mule.tck.functional.FunctionalTestComponent: Full Message payload:
/Services/Regular/test[first appender msg]

Message properties:
INVOCATION scoped properties:
INBOUND scoped properties:
Accept=text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset=ISO-8859-1,utf-8;q=0.7,*;q=0.7
Accept-Encoding=gzip, deflate
Accept-Language=en-us,en;q=0.5
Connection=true
Host=localhost:8182
Keep-Alive=true
MULE_ORIGINATING_ENDPOINT=endpoint.http.localhost.8182.Services.Regular
MULE_REMOTE_CLIENT_ADDRESS=/0:0:0:0:0:0:1%0:49398
User-Agent=Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.3)
Gecko/20090824 Firefox/3.5.3 (.NET CLR 3.5.30729)
```

```

http.context.path=/Services/Regular/
http.method=GET
http.request=/Services/Regular/test
http.request.path=/Services/Regular/test
http.version=HTTP/1.1
OUTBOUND scoped properties:
    MULE_ENCODING=UTF-8
SESSION scoped properties:

```

## 9.4. Retain a Message History

The test component used in the above examples can retain a list of messages it has received. The contents of this list can be retrieved programmatically or may be examined using a debugger.

```

...
<!--
    This test component will retain a list of all messages
    it has received.
-->
<test:component enableMessageHistory="true" />
...

```

Note that:

- The message history logging is enabled as default.
- If the message object is modified after the test component has finished executing, the corresponding message, being one and the same object, in the message history log list will also change.

Examining the message history list in the Eclipse debugger, after having hit a breakpoint set in the *FunctionalTestComponent* class, looks like this:

Name	Value
↳ this	FunctionalTestComponent (id=48)
appendString	null
doInboundTransform	true
enableMessageHistory	true
enableNotifications	true
eventCallback	null
exceptionText	"" (id=67)
exceptionToThrow	null
logger	MuleLog (id=68)
logMessageDetails	true
messageHistory	CopyOnWriteArrayList<E> (id=72)
array	Object[3] (id=87)
lock	ReentrantLock (id=89)
muleContext	DefaultMuleContext (id=79)
returnData	" #[function:datestamp]" (id=85)
throwException	false
waitTime	0

Examining the message history of an instance of the *FunctionalTestComponent* in the Eclipse debugger.

## 9.5. Introduce a Delay

The test component can also introduce a delay, simulating some time-consuming processing.

```
...
<!--
    This test component will introduce a two second delay in the message processing.
-->
<test:component waitTime="2000">
...
```

Note that:

- The delay time is to be given in milliseconds.

## 9.6. Append Text to Received Messages

The Mule test component can append text to the current message, creating a message that is a string object.

```
...
<!--
    This test component will transform the message to its
    string representation and append a string to the message.
    Mule expressions may be used in the string to be appended.
-->
<test:component
    appendString="( processed by test component at #[function:systime] )">
...
```

Note that:

- If the incoming message is not a string object, it will be transformed to its string representation prior to text being appended to the message.
- If the test component returns mock data, as discussed earlier, the mock data will replace any string message created by appending text to the received message.
- Mule expressions can, as seen in the example above, be used in the text to be appended.

## Appendix A – Prepare for Mule Development

This appendix contains step-by-step instructions on how to set up the SpringSource Tool Suite for development with Mule versions 2.x and 3.x. It assumes that you have already downloaded and installed a reasonably new version of SpringSource Tool Suite IDE.

The name “Eclipse” will be used interchangeably with “SpringSource Tool Suite” when referring to the IDE.

### 1. Download and Install Mule

The examples in this book uses two different versions of Mule; the 2.x version and the 3.x version.

- Download version 2.2.1 of the Mule standalone distribution archive from [here](#).
- Unpack the archive to the location of your choice.
- Download version 3.2.0 of the Mule standalone distribution archive from [here](#).
- Unpack the archive to the location of your choice.  
Preferably next to the unpacked 2.2.1 distribution.

In order to be able to use Mule in the standalone mode, some environment variables must be set. When setting environment variables, it is assumed that the necessary Java environment variables have already been set.

In an \*nix-based environment, either issue the following commands in a terminal window or edit the shell profile accordingly. Change the path to your Mule distribution according to its location in your computer.

```
export MULE_HOME="/Volumes/HD/Users/ivan/Applications/mule-standalone-3.2.1"
PATH=$PATH:$MULE_HOME/bin
```

In a DOS-based environment, either issue the following commands or edit the environment variables. Again, change the path to your Mule distribution according to its location on your local file system.

```
set MULE_HOME="C:\mule-standalone-3.2.1"
set PATH=%PATH%;%MULE_HOME%\bin
```

Note that in order to use another distribution when launching a standalone instance of Mule, you need to modify the MULE\_HOME environment variable to point at the appropriate Mule distribution.

### 2. Install the Eclipse Mule Plugin

The Mule plugin for the Eclipse enables creation of special Mule projects in the Eclipse IDE, provides aid when creating Mule configuration files etc. It is installed like any other Eclipse plugin, using the update site <http://dist.muleforge.org/mule-ide/updates-2.1.x/>.

The Mule IDE webpage, on which there is additional information about the plugin, can be found at: <http://www.mulesoft.org/documentation/display/MULEIDE/Home>

### **3. Configure the Mule Plugin in Eclipse**

The Eclipse Mule plugin needs to be told about the Mule distributions available on the system. The following steps describe how to configure the plugin to use the two Mule distributions we've downloaded earlier.

- Open the Eclipse preferences.
- Go to the Mule node.
- Click the Add button.
- Navigate to the directory containing the Mule 3.x standalone distribution and select it.
- Click the Add button again.
- Navigate to the directory containing the Mule 2.x standalone distribution and select it.
- Make sure the Mule 3.x distribution is the default by checking its checkbox.
- Click the Apply button in the lower right corner of the Preferences dialog.
- Click the OK button to exit the Preferences dialog.

This concludes the preparations and you should now be ready to develop with Mule in the Eclipse IDE.

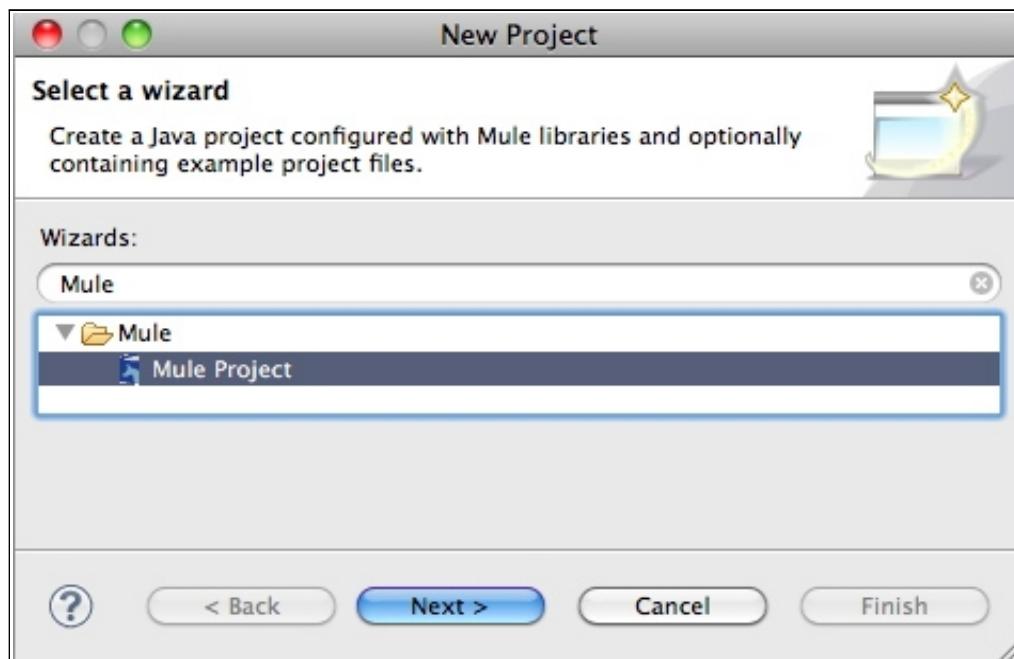
## Appendix B – Create a Mule Project

This appendix describes how to create a Mule project that will result in a standalone Java application. We will also see how to switch between Mule distributions in a Mule project.

### 1. *Create the Project*

With the Mule plugin installed, there will be a wizard for creating Mule projects.

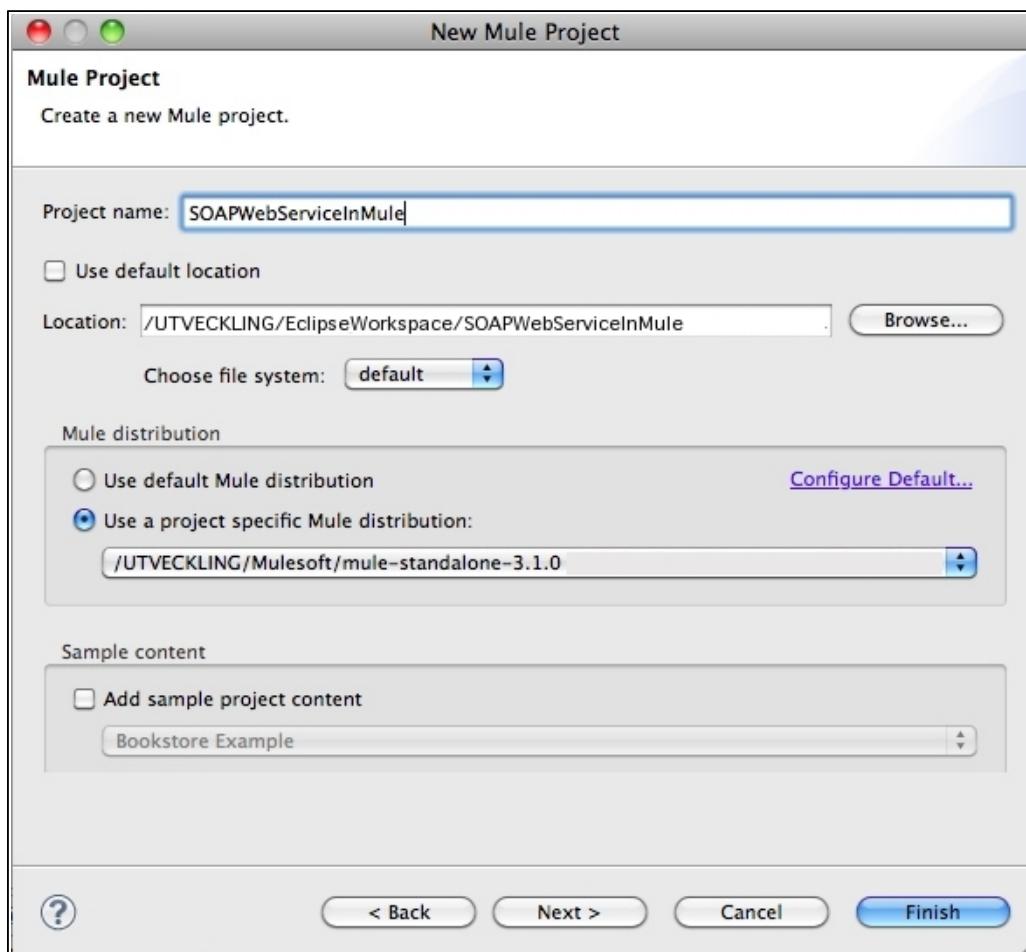
- In the File menu, select New -> Project...
- Select the Mule Project wizard in the New Project dialog and click the Next button.



Selecting the Mule Project wizard when creating a new project in Eclipse.

(continued on next page)

- Enter the name of the project, select a location (optional) and choose the Mule distribution to use in the project.
- If you intend to follow the examples in this book and use both Mule 2.x and Mule 3.x, configure the project to use the appropriate Mule 3.x distribution. Note that this does not necessarily mean Mule 3.1.0 as shown in the picture.



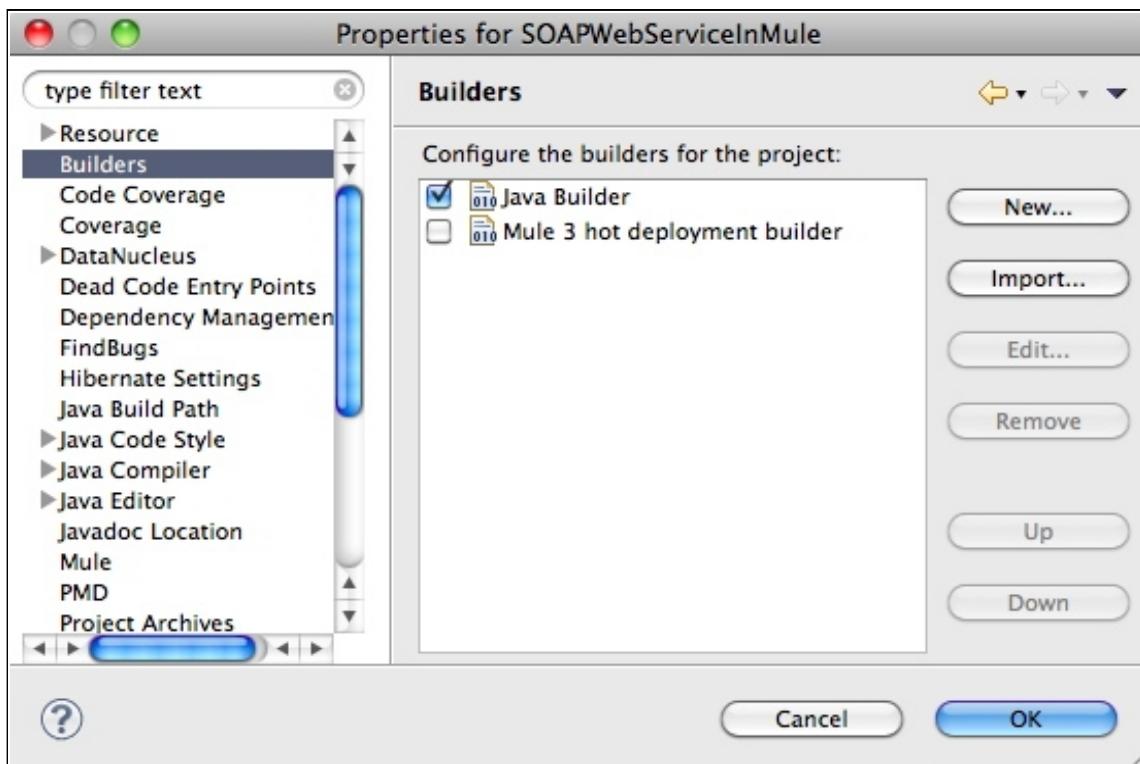
Creating a new Mule project in Eclipse; entering a project name, selecting a project location and selecting a Mule distribution to be used by the project.

- Click the Finish button.

## 2. Switch off the Mule 3 Hot Deployment Builder

On occasions, the Mule 3 hot deployment builder has caused errors for me when trying to build a Mule project. To fix such problems, just switch off the hot deployment builder for the project in question.

- Open the Project Preferences in Eclipse.
- Navigate to the Builders node on the left side.
- Uncheck the Mule 3 Hot Deployment Builder checkbox.



Switching off the Mule 3 hot deployment builder for a Mule project in Eclipse.

### 3. Create Mule Configuration Files

Mule configuration files are XML files used to configure an instance of the Mule ESB. They are just one of the ways to configure Mule, albeit the most common way. These XML files are really Spring bean configuration files with a set of special namespaces for Mule modules and transports. In this book each example will show how to use both Mule 2.x as well as Mule 3.x, so two Mule configuration files will be created for each project; one for each version. If you only want to use one version, then feel free to skip creating a second configuration file.

- In the source folder, create a Mule configuration file by, in the File menu, selecting New -> Other and then locating the Mule Configuration wizard.



Creating a Mule Configuration file; locating the Mule Configuration wizard.

- Click the Next button in the wizards-dialog.
- Click the Browse button and select the source (src) directory as the location where to save the Mule configuration file.

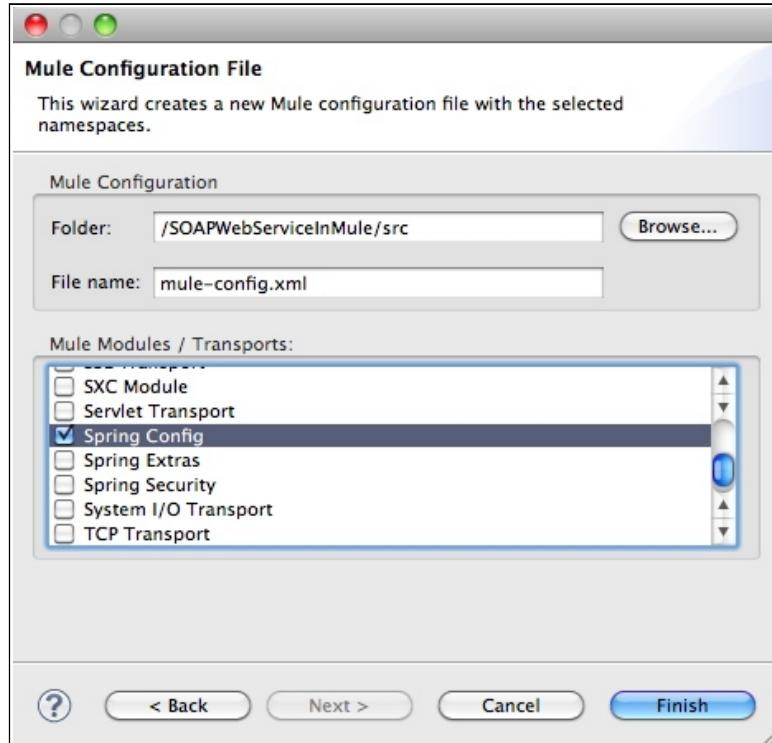
If you have a special directory for resources, like in a Maven project, then such a directory is a better choice.

- Choose a name for the Mule configuration file.

Since the examples in this book will have two different Mule configuration files, one for each Mule version used, it may be a good idea to append the Mule version number to the name of the configuration file.

The default names used in this book are “mule-config2.xml” and “mule-config3.xml”.

- Check the Mule Modules/Transports which namespaces you want to include in the Mule configuration file you are about to create. Individual projects may require individual configuration of which modules and transports to use. Such requirements will be included in the instructions for each individual project - otherwise use the default configuration.



Configuring the location and name of the Mule configuration file, as well as the namespaces for the different modules and transports to include in the configuration file.

- Click the Finish button to create the new Mule configuration file.
- Change the Mule distribution used by the project.  
The process of changing the Mule distribution of a project is described [below](#). If the project was configured to use the Mule 3.x distribution when you just created the configuration file above, then switch to the Mule 2.x distribution and vice versa.
- Repeat the above process to create a second Mule configuration file.
- Change back the Mule distribution used by the project.  
The process of changing the Mule distribution of a project is described [below](#).

#### **4. Create the Log4J Configuration File**

In order to be able to control the log written by Mule, the project needs a Log4J configuration file.

- In the source folder, create a file named “log4j.properties” by, in the File menu, selecting New -> Other and then locating the File option in the General node.
- Paste the following contents into the new file:

```
log4j.rootLogger=WARN, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%d{MM-dd HH:mm:ss}] %-5p %c{1} [%t]: %m
%n
log4j.logger.org.mule=INFO
```

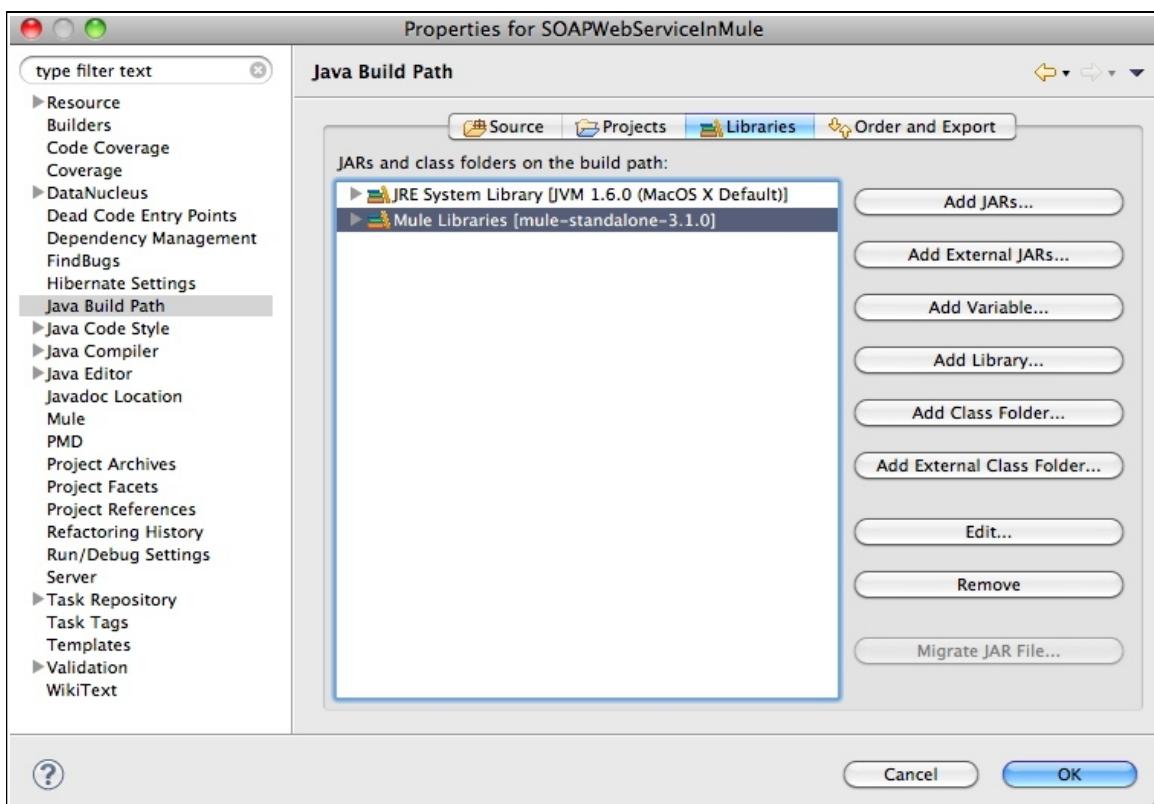
- Save the file.

This concludes creation of a Mule project and we are now ready to start developing in the project.

## 5. Change the Mule Distribution of a Project

This section is not relevant when just having created a new Mule project, but is provided for completeness sake. It will come in handy once we start developing the examples and want to develop for both Mule 2.x and Mule 3.x. In order to do this we need to be able to change the Mule distribution a project uses. This is accomplished in the Eclipse project preferences:

- Open the Project Preferences in Eclipse.
- Navigate to the Java Build Path node on the left side.
- Click the Libraries tab.
- Select the Mule Libraries.

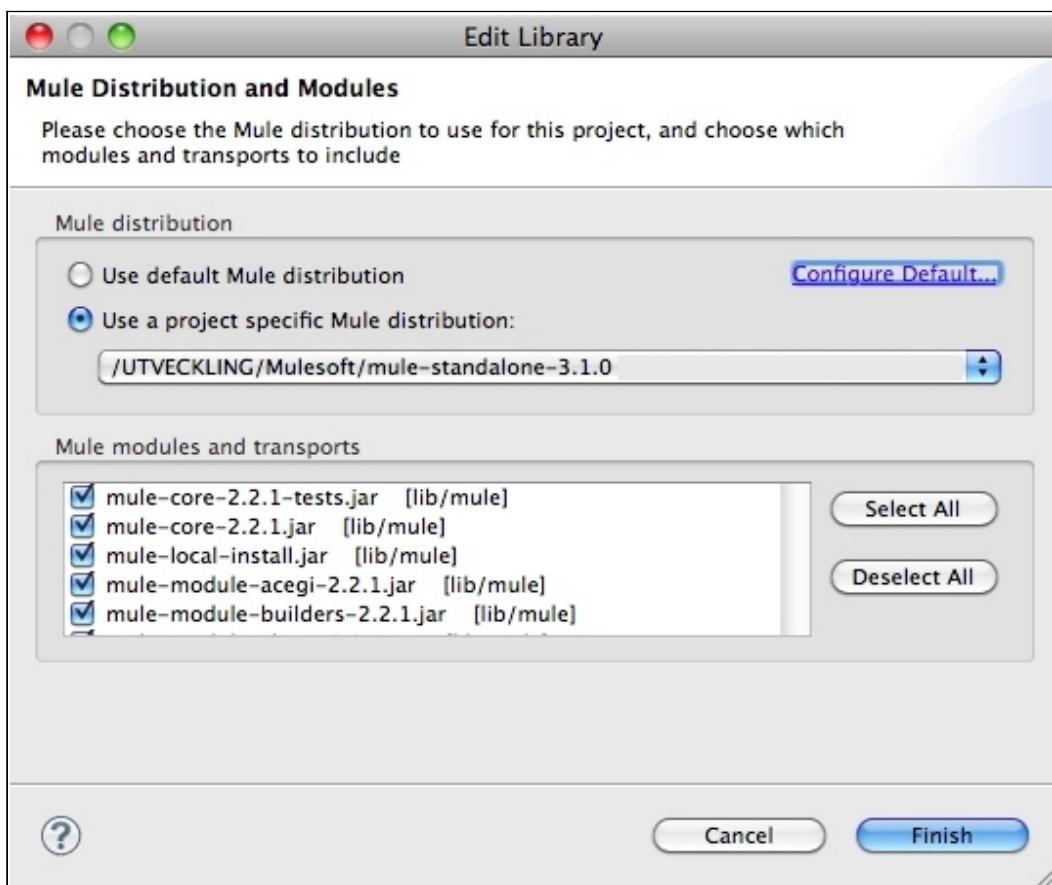


Locating the Mule libraries in an Eclipse project.

- Click the Edit button.

(continue on next page)

- In the new dialog that appears, select the Mule distribution you want to switch to. As far as I have experienced, enabling or disabling modules and transports is of little use; they all seem to be enabled once having exited the dialog.



Selecting the Mule distribution to use in an Eclipse project.

The project now uses a different Mule distribution. Remember to use Mule configuration files appropriate for the version in question!

## Appendix C – Enabling Maven Dependency Management for an Eclipse Project

The SpringSource Tool Suite development environment comes with the Maven m2eclipse plugin preinstalled. This plugin enables us to, among other things, enable dependency management for an existing project.

This should be a trivial operation, were it not for the fact that the plugin modifies the build path and compiler settings of the project in the process.

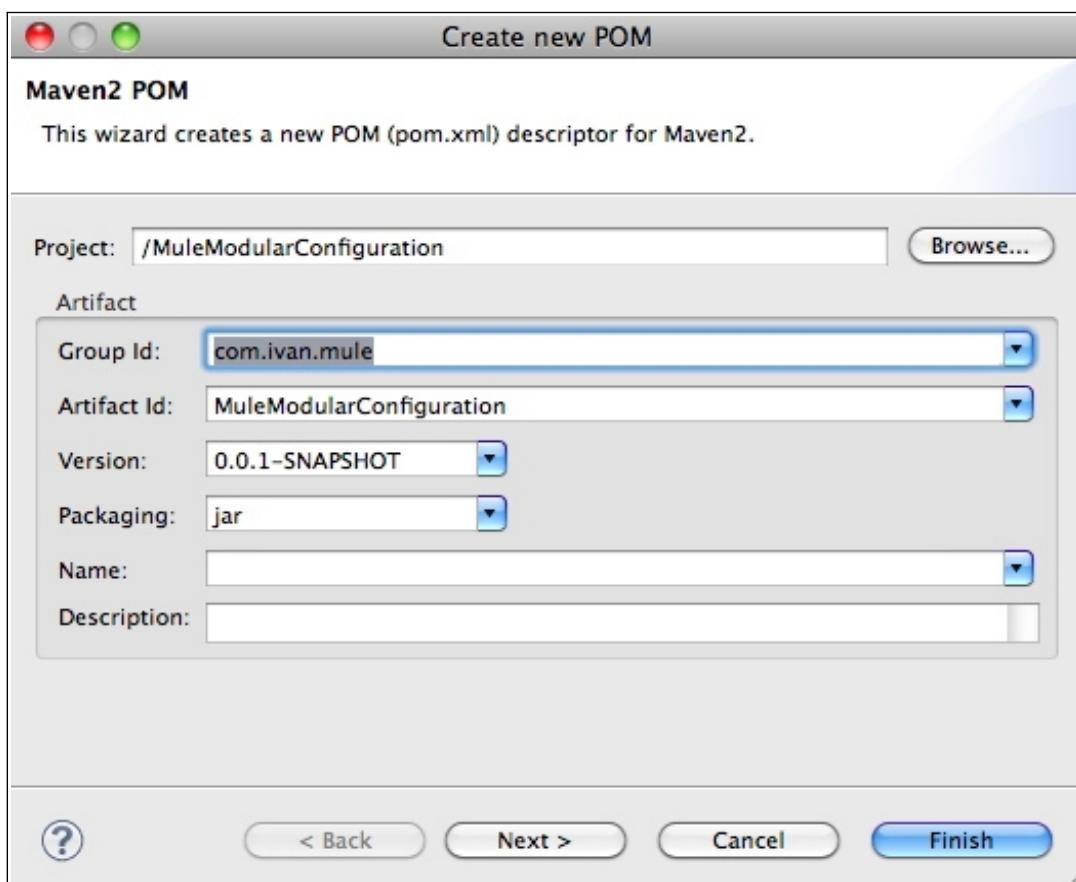
Below are instructions on how to enable dependency management for a project and restore the other settings:

- Right-click the project in the Eclipse Package Explorer and select Maven -> Enable Dependency Management.



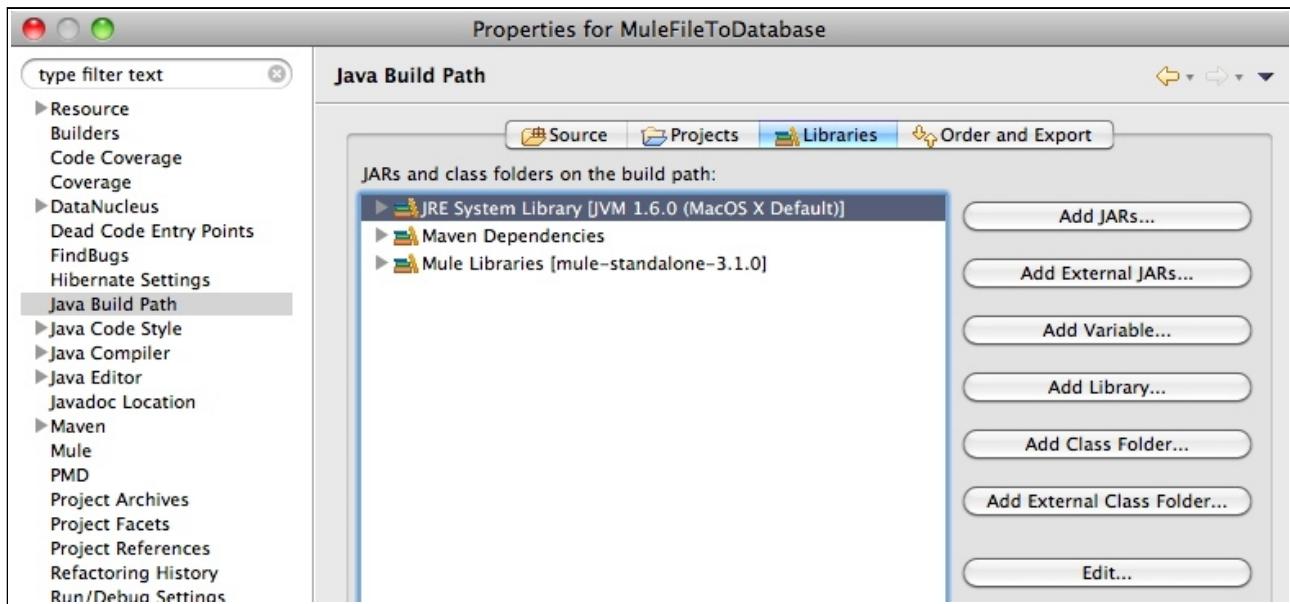
Enabling Maven dependency management for the example project.

- Enter the data in the POM-creation dialog.  
As a minimum, enter the group id - “com.ivan.mule” in my case.



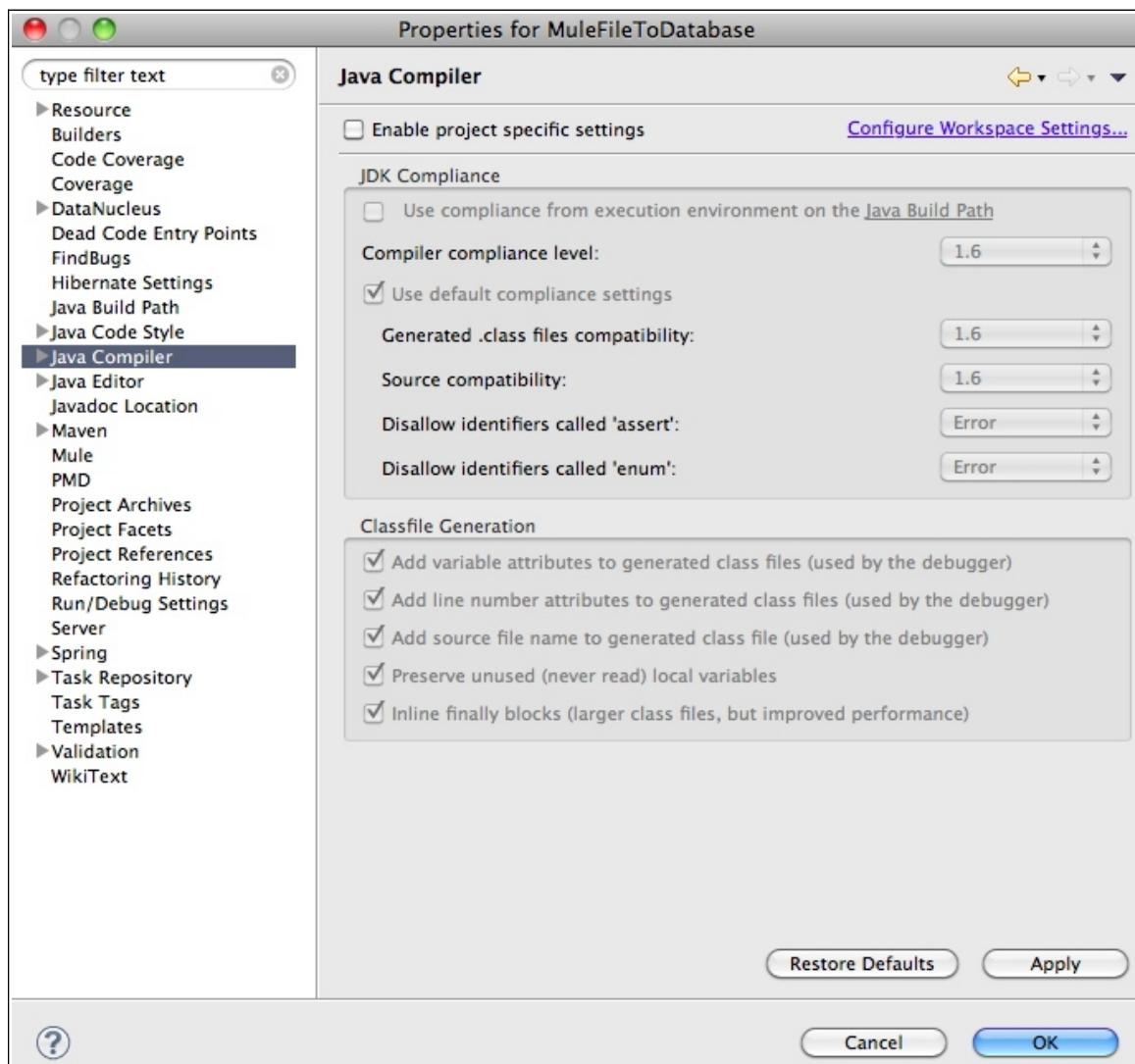
Entering data in the POM-creation dialog when enabling Maven dependency management for a project.

- Click the Finish button.  
A pom.xml file should appear in the project, as can be seen in the Package or Project Explorer.
- Right-click on the project and open the project's Properties.  
Alternatively, click the project and hold the option key while hitting the return key.
- In the Java Build Path node, in the Libraries tab, change the JRE System Library to the latest JRE you have installed.  
This is accomplished by selecting the JRE System Library entry, clicking the Edit button and selecting the workspace default JRE.  
When enabling Maven dependency management, the JRE is automatically switched to the version 1.4 JRE and we need to switch it back to be able to use Mule.



Having switched the JRE System Library of a project with Maven dependency management.

- Still in the project's Properties, navigate to the Java Compile node and disable the use of project specific settings.  
Again, this is caused by enabling the Maven dependency management.



Having disabled Java Compiler settings of a project with Maven dependency management.

- Click Apply and then OK to exit the project properties dialog.

We have now completed enabling Maven dependency management for a project and restored the settings modified in the process.

## Appendix D – Mule Standalone Server

This appendix describes basic management of a Mule standalone server, such as starting and stopping it. There is also a section for OS X users describing how to fix the Mule installation so that it can be run as a standalone Mule server on an 64-bit Macintosh computer.

### 1. **Mule Standalone Server on OS X**

If you intend to run Mule with Macintosh OS X on a 64-bit computer, there is a good chance that you will run into problems due to no 64-bit version of the Tanuki wrapper having been included. This issue can be fixed the following way:

- In a web browser, go to the URL  
<http://wrapper.tanukisoftware.com/doc/english/download.jsp#stable>
- Download the latest Delta Pack community version.  
In my case this is version 3.5.15 and the direct link is:  
<http://wrapper.tanukisoftware.com/download/3.5.13/wrapper-delta-pack-3.5.13.tar.gz>
- Expand the downloaded archive.
- From the “lib” directory in the root of the unpacked Delta Pack archive, copy all files whose names start with “libwrapper-macosx” to the “lib/boot” directory in the Mule directory.
- Delete the JAR file in the “lib/boot” directory in the Mule directory which name starts with “wrapper” and ends with “jar” (there may or may not be a version number in the filename). Remember the name of the JAR file, since the new wrapper JAR file must have the same name as the removed JAR file otherwise you won't be able to start Mule from within Eclipse.
- From the “lib” directory in the root of the unpacked Delta Pack, copy the “wrapper.jar” file to the “lib/boot” directory in the Mule directory.  
Rename the “wrapper.jar” file to have the same name as the JAR file removed in the previous step.
- From the “bin” directory in the root of the unpacked Delta Pack, copy all the files with names start with “wrapper-macosx-universal” to the “lib/boot/exec” directory in the Mule directory.

If you now attempt to start Mule by issuing the “./mule” command in the “bin” directory of the Mule directory, there will be console output similar to the following:

```
...
Running Mule...
--> Wrapper Started as Console
Java Service Wrapper Community Edition 32-bit 3.5.13
Copyright (C) 1999-2011 Tanuki Software, Ltd. All Rights Reserved.
http://wrapper.tanukisoftware.com

Launching a JVM...
...
```

Note that the message indicates the use of the 32-bit edition of the Tanuki wrapper. However, it is the 64-bit version of the JVM that is started, in which Mule later is run.

## **2. Mule Standalone Server Basic Management**

This section shows basic management of a standalone Mule server, including starting and stopping the server as well as deploying and undeploying applications in the server.

Before being able to run a standalone instance of Mule, it must be installed and the appropriate environment variables must be set. Please refer to [appendix A](#) for details on how to do this.

Unless noted otherwise, the instructions below apply to both Mule 2.x and Mule 3.x.

### **2.1. Start and Stop a Mule Server**

A standalone Mule server can be started either as a foreground process, which writes log messages to the console in which Mule was started, or as a background process. In the latter case, log messages are only written to log file(s).

The following table lists terminal commands used to start a server instance, stop a server instance and query a running Mule instance for status.

Terminal Command	Options	Description
<code>./mule -config [one or more Mule configuration files]</code>		Start the Mule server as a foreground process. Mule 3.x: Starts all applications deployed to the “apps” directory.
<code>./mule -app [application name]</code>		Mule 3.x only. Starts the Mule server as a foreground process running only the named application. The application must be present in the “apps” directory.
<code>./mule start</code>	<code>-config [one or more Mule configuration files]</code>	Start the Mule server as a background process. Mule 3.x: Starts all applications present in the “apps” directory.
<code>./mule stop</code>		Stop a Mule server background process.
<code>./mule restart</code>		Restarts a running Mule server background process.
<code>./mule status</code>		Output information about the MULE_HOME environment variable and whether there is a Mule server background process running and, if there is, its process id.

## 2.2. Deploy and Undeploy a Mule Application

When using Mule 2.x, deployment and undeployment are limited to when starting and stopping the entire Mule server. It is not possible to modify a running Mule 2.x application while it is running.

With Mule 3.x, it is not only possible to deploy a Mule application to a running Mule server, but it is also possible to make modifications to a running Mule application and have the modifications applied on the fly. This feature is called hot deployment.

### Mule 2.x Deployment

When the Mule 2.x application has been packaged according to the instructions in the section [Package Mule 2.x Applications](#), the application can be deployed according to the following steps:

- Copy the JAR file(s) required by the application to the “lib/user” directory in the Mule home directory.
- Launch the Mule 2.x server with the path to the Mule configuration file of the application as parameter according to the instructions in the previous section.

### Mule 3.x Deployment

Mule 3.x application deployment has been simplified, since all the artifacts of an application can be kept in one and the same place – a directory. In addition, Mule 3.x allows for hot deployment, which enables changes to a running application to come into effect without restarting the Mule server.

With the Mule 3.x application packaged according to the instructions in the section [Package Mule 3.x Applications](#), the application may be deployed as follows:

- Copy the application directory to the “apps” directory in the Mule home directory.
- If the Mule 3.x server is not already started, then start it.  
When starting the Mule 3.x server, there is no need to supply the path to the configuration file(s) of the application. Mule 3.x will search the “apps” directory and find configuration files of Mule applications.

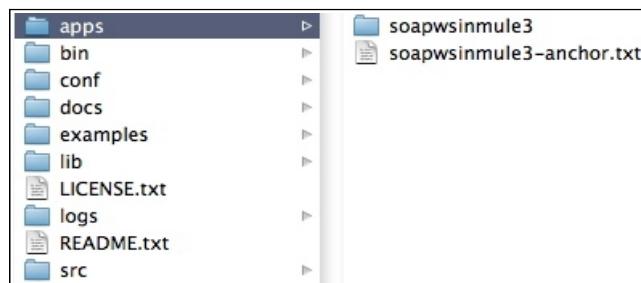
## 2.3. Undeploy a Mule Application

In Mule 2.x, the only way to undeploy a Mule application is to stop the server and remove the application artifacts; JAR file(s) and configuration files.

In Mule 3.x, an anchor-file will appear in the “apps” directory next to a successfully deployed application.

Anchor-file for a successfully deployed Mule 3.x application.

Deleting the anchor-file of an application undeploys the application in question.



## Appendix E – Database Access from within Eclipse

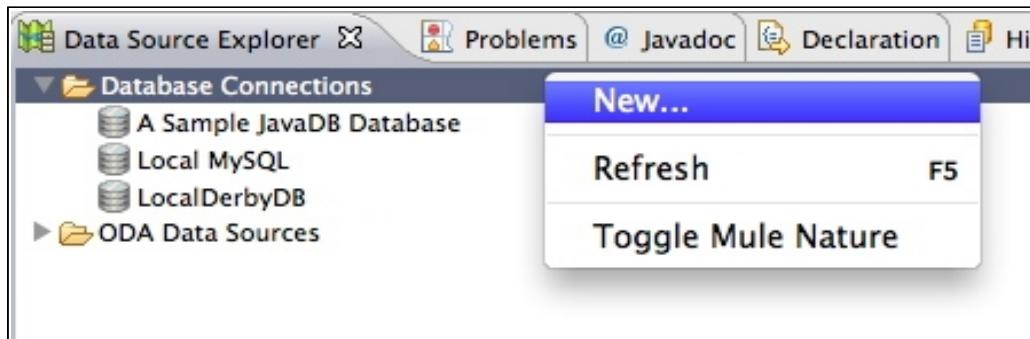
This appendix describes how to setup and perform access to a PostgreSQL database running on the local computer from within Eclipse.

### 1. Data Source Creation

In order to be able to, for example, view data in a database from within the Eclipse IDE, we need to configure a Data Source in Eclipse.

The example in this appendix shows how to configure a data source for the PostgreSQL database, but the procedure is the same regardless of the database used.

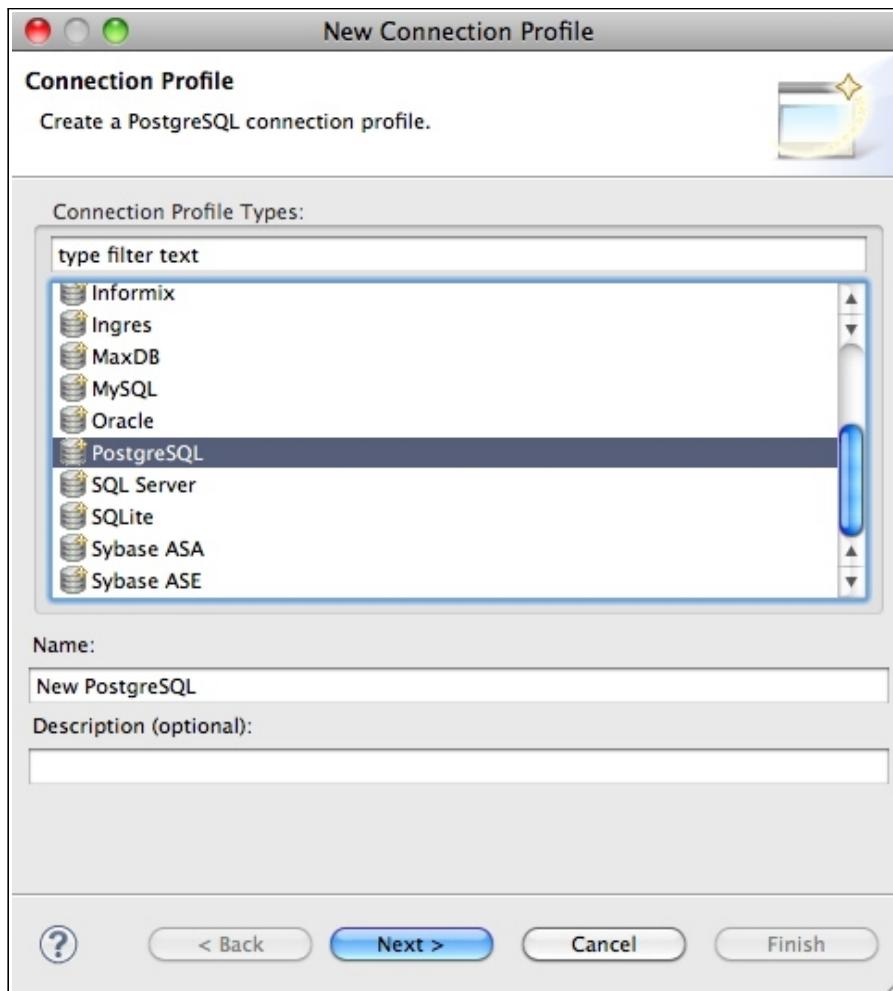
- Download the JDBC driver JAR for the database in question.  
The PostgreSQL driver JAR can be found at <http://jdbc.postgresql.org/>.  
Since we will be using Java 6 or later, we chose the JDBC4 driver.
- In Eclipse, open the Data Source Explorer view.
- Right-click on the Database Connections folder in the Data Source Explore and select New.



Creating a new database connection in the Eclipse Data Source Explorer.

(continued on next page)

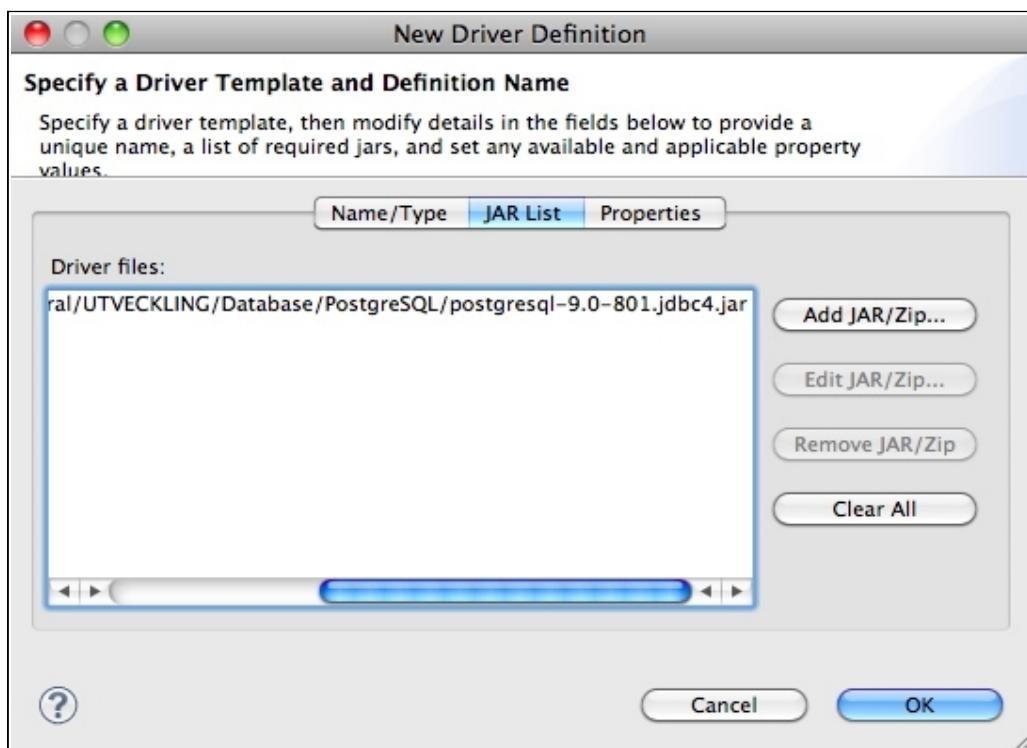
- Select the database type, PostgreSQL in our case, and enter a name for the data source.



Selecting the database type and data source name when creating a new data source in Eclipse.

- Click the Next button.
- Click the  symbol located to the right of the drivers popup-menu to create a new driver definition.
- In the New Driver Definition dialog that appears, select the PostgreSQL JDBC Driver type in the Name/Type tab.
- In the JAR List tab, select the existing entry and click Remove JAR/Zip.

- Still in the JAR List tab, click the Add JAR/Zip button and select the PostgreSQL JDBC JAR.

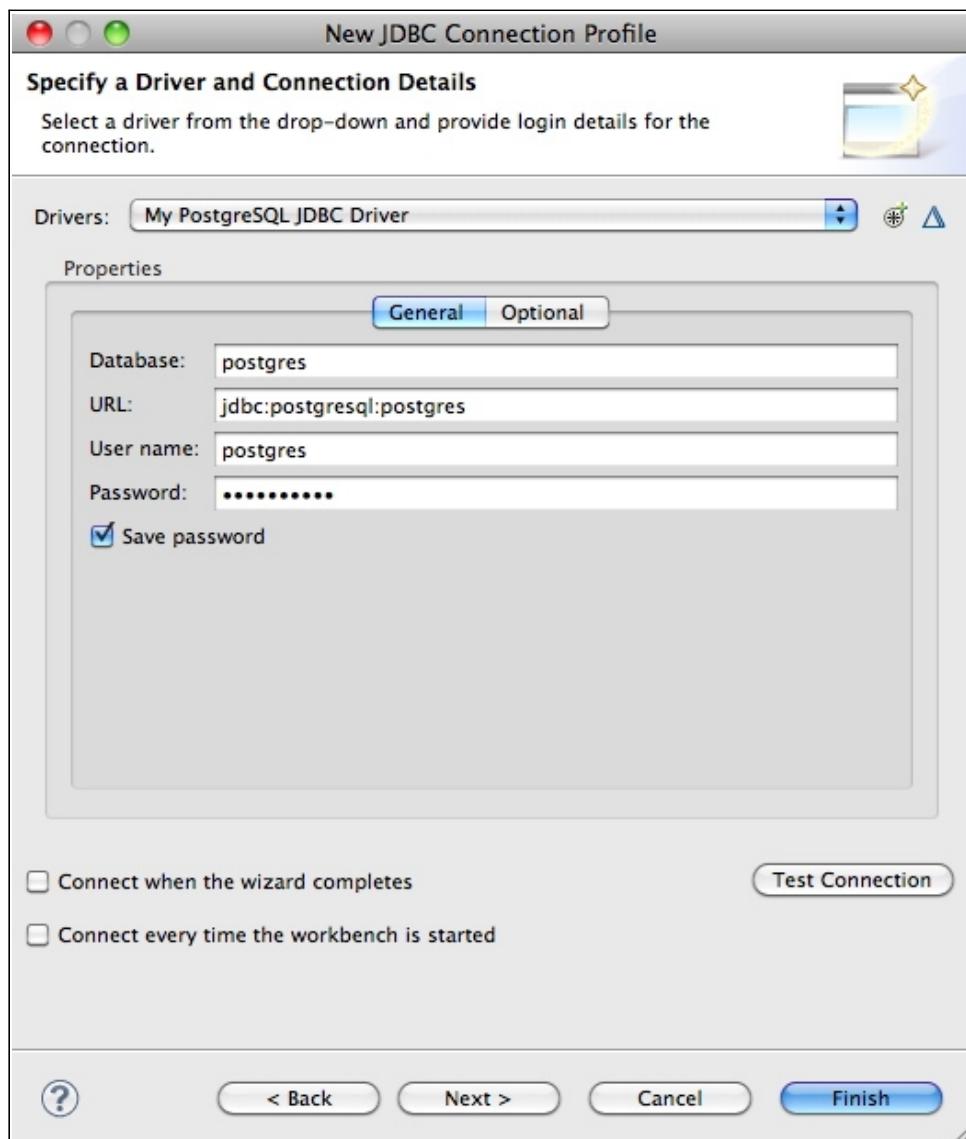


Having selected the database driver JAR file when creating new database driver definition in Eclipse.

- Click the OK button to finish creation of the driver definition.

(continued on next page)

- Continue in the New JDBC Connection Profile dialog by entering user name and password. These are preferably the administrator username and password, as entered when installing PostgreSQL. If you want to avoid entering the password every time you want to connect to the server, check the “Save password” checkbox.



Entering final connection details when creating a new data source in Eclipse.

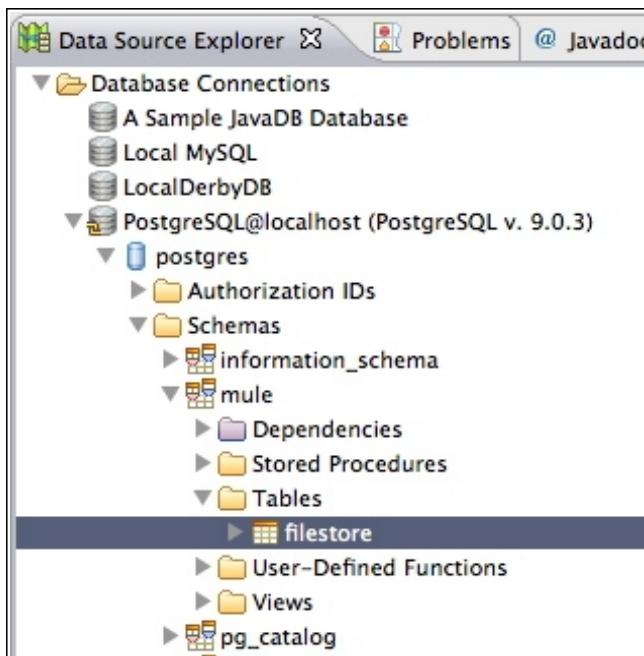
- To ensure that the data source configuration is correct, click the Test Connection button. If the connection test failed, change the settings and repeat the test until it succeeds.
- Finish creation of the new data source by clicking the Finish button.

## 2. Data Access

This section contains a very brief description on how to query a database from within Eclipse. It assumes that a data source has been configured for the database in question, as described in the previous section.

As an example, we will look at data produced by the example program from [chapter 4](#):

- In Eclipse, open the Data Source Explorer view.
- Expand the Database Connections node.
- Right-click on the database connection for the database which you want to examine and select Connect.
- Navigate down to the database table that you want to examine:



Navigating down to database table to examine in the Eclipse Data Source Explorer.

- Right-click the name of the database table (“filestore” in the above figure) and select Data -> Edit.

An editor should appear, listing the current contents of the database table:

filestore		
id [SERIAL]	message [TEXT(2147483647)]	time_stamp [TIMESTAMP]
0	<project	2011-03-24 18:26:00.69
1	<?xml version="1.0" encoding="	2011-03-24 18:31:55.15

Displaying contents of database table in Eclipse.

This concludes the appendix on database access from within Eclipse.