|  |
| --- |
| Primer on Apache camel |
| A brief overview to Apache Camel |
|  |
|  |
|  |

|  |
| --- |
| This document provides the brief overview of the apache camel, covering the briefly about camel component, SOA related EIP patterns used heavily in Camel. |

Table of Contents

[First Step 2](#_Toc395783108)

[Meeting Camel 2](#_Toc395783109)

[Introduction 2](#_Toc395783110)

[Why use Camel? 2](#_Toc395783111)

[Camel’s message model 4](#_Toc395783112)

[Camel’s architecture 7](#_Toc395783113)

[Camel concepts 8](#_Toc395783114)

[Key points 9](#_Toc395783115)

[POM Dependency of Camel 10](#_Toc395783116)

[Routing with Camel 11](#_Toc395783117)

[Types of Consumers 11](#_Toc395783118)

[JMS overview 12](#_Toc395783119)

[Route Builder 13](#_Toc395783120)

[Creating routes in Java 14](#_Toc395783121)

[Adding a Processor 16](#_Toc395783122)

[Bean injection and Spring 17](#_Toc395783123)

[How to Configure Camel to Use a JMS Provider 18](#_Toc395783124)

[Routing and EIP Points 20](#_Toc395783125)

[Core Camel 29](#_Toc395783126)

[Using the getIn and getOut methods on exchanges 29](#_Toc395783127)

[The Direct component 29](#_Toc395783128)

[Transforming data with Camel 30](#_Toc395783129)

[Data transformation overview 30](#_Toc395783130)

[Transforming data using EIPs and Java 31](#_Toc395783131)

[Transforming XML 40](#_Toc395783132)

[Transforming with data formats 44](#_Toc395783133)

[Transforming with templates 53](#_Toc395783134)

[About Camel type converters. 54](#_Toc395783135)

[Using beans with Camel 57](#_Toc395783136)

# First Step to Apache Camel

## Meeting Camel

### Introduction

At the core of the Camel framework is a routing engine, integration framework or more precisely a routing engine builder. It allows you to define your own routing rules, decide from which sources to accept messages, and determine how to process and send those messages to other destinations. Camel uses an integration language that allows you to define complex Routing rules, akin to business processes.

Camel offers higher-level abstractions that allow you to interact with various systems using the same API regardless of the protocol or data type the systems are using.

Components in Camel provide specific implementations of the API that target different protocols and data types. Out of the box, Camel comes with support for over 80Protocols and data types.

Its extensible and modular architecture allows you to implement and seamlessly plug in support for your own protocols, proprietary or not.

Apache ServiceMix and ActiveMQ, already use Camel as a way to carry out enterprise integration

### Why use Camel?

Camel features

* Routing and mediation engine
* Enterprise integration patterns (EIPs)
* Domain-specific language (DSL)
* Extensive component library
* Payload-agnostic router
* Modular and pluggable architecture
* POJO model
* Easy configuration
* Automatic type converters
* Lightweight core
* Test kit
* Vibrant community

Let’s dive into the details of each of these features.

**ROUTING AND MEDIATION ENGINE**

The core feature of Camel is its routing and mediation engine. A routing engine will selectively move a message around, based on the route’s configuration.

In Camel’s Case, routes are configured with a combination of enterprise integration patterns (EIP’s) and a domain-specific language, both of which we’ll describe next.

**ENTERPRISE INTEGRATION PATTERNS (EIPS)**

Camel is heavily based on EIPs. Although EIPs describe integration problems and solutions and also provide a common vocabulary, the vocabulary isn’t formalized. Camel tries to close this gap by providing a language to describe the integration solutions. There’s almost a one-to-one relationship between the patterns described in Enterprise Integration Patterns and the Camel DSL.

**DOMAIN-SPECIFIC LANGUAGE (DSL)**

Camel’s domain-specific language (DSL) is a major contribution to the integration space. A few other integration frameworks currently feature a DSL (and some allow you to use XML to describe routing rules), but unlike Camel their DSLs are based on custom languages.

Camel is unique because it offers multiple DSLs in regular programming languages such as Java, Scala, Groovy, and it also allows routing rules to be specified in XML.

The purpose of the DSL is to allow the developer to focus on the integration problem rather than on the tool—the programming language. Although Camel is written mostly in Java, it does support mixing multiple programming languages. Each language has its own strengths, and you may want to use different languages for different tasks. You have the freedom to build a solution your own way with as few constraints as possible.

Here are some examples of the DSL using different languages and staying functionally equivalent:

**Scenario:** Moving files into Queue: Order, whenever files is placed in the data/inbox

* Java DSL

*from("file:data/inbox").to("jms:queue:order");*

* Spring DSL

*<route>*

*<from uri="file:data/inbox"/>(from defines the source path)*

*<to uri="jms:queue:order"/> (to defines the destination path)*

*</route>*

* Scala DSL

*from "file:data/inbox" -> "jms:queue:order"*

These examples are real code, and they show how easily you can route files from a folder to a JMS queue. Because there’s a real programming language underneath, you can use the existing tooling support, such as code completion and compiler error detection.

**EXTENSIVE COMPONENT LIBRARY**

Camel provides an extensive library of more than 80 components. These components enable Camel to connect over transports, use APIs, and understand data formats.

**PAYLOAD-AGNOSTIC ROUTER**

Camel can route any kind of payload—you aren’t restricted to carrying XML payloads. This freedom means that you don’t have to transform your payload into a canonical format to facilitate routing.

**MODULAR AND PLUGGABLE ARCHITECTURE**

Camel has a modular architecture, which allows any component to be loaded into Camel, regardless of whether the component ships with Camel, is from a third party, or is your own custom creation.

**POJO MODEL**

Beans (or POJOs) are considered first-class citizens in Camel, and Camel strives to let you use beans anywhere and anytime in your integration projects. This means that in many places you can extend Camel’s built-in functionality with your own custom code.

**EASY CONFIGURATION**

The convention over configuration paradigm is followed whenever possible, which minimizes configuration requirements. In order to configure endpoints directly in routes, Camel uses an easy and intuitive URI configuration.

For example, you could configure a file consumer to scan recursively in a subfolder and include only a .txt file, as follows:

*from("file:data/inbox?recursive=true&include=\*.txt")...*

**AUTOMATIC TYPE CONVERTERS**

Camel has a built-in type-converter mechanism that ships with more than 150 converters. You no longer need to configure type-converter rules to go from byte arrays tostrings, for example. And if you find a need to convert to types that Camel doesn’t support, you can create your own type converter. The best part is that it works under the hood, so you don’t have to worry about it. The Camel components also leverage this feature; they can accept data in most types and convert the data to a type they’re capable of using. This feature is one of the top favorites in the Camel community. You may even start wondering why it wasn’t provided in Java itself! Chapter 3 covers more about type converters.

**LIGHTWEIGHT CORE**

Camel’s core can be considered pretty lightweight, with the total library coming in at about 1.6 MB and only having a dependency on Apache Commons Logging and Fuse- Source Commons Management. This makes Camel easy to embed or deploy anywhere you like, such as in a standalone application, web application, Spring application, JavaEE application, JBI container, OSGi bundle, Java Web Start, or on the Google App engine. Camel was designed not to be a server or ESB but instead to be embedded in whatever platform you choose.

**TEST KIT**

Camel provides a Test Kit that makes it easier for you to test your own Camel applications. The same Test Kit is used extensively to test Camel itself, and it includes more than 6,000 unit tests. The Test Kit contains test-specific components that, for example, can help you mock real endpoints. It also contains setup expectations that Camel can use to determine whether an application satisfied the requirements or failed.

**VIBRANT COMMUNITY**

Camel has an active community. This is essential if you intend to use any open source project in your application. Inactive projects have little community support, so if you run into issues, you’re on your own. With Camel, if you’re having any trouble, users and developers alike will come to your aid promptly.

Now that you’ve seen the main features that make up Camel, we’ll get a bit more hands on by looking at the Camel distribution and trying out an example.

### Camel’s message model

In Camel, there are two abstractions for modeling messages,

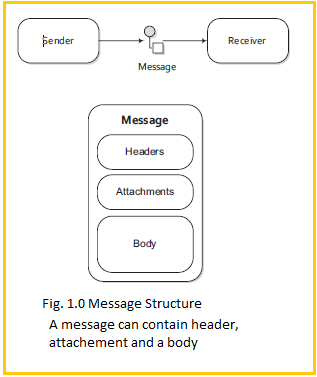
* org.apache.camel.Message —

The fundamental entity containing the data being carried and routed in Camel

* org.apache.camel.Exchange —

The Camel abstraction for an exchange of messages. This exchange of messages has an “in” message and as a reply, an “out” message.

#### Message

******

Messages are the entities used by systems to communicate with each other when using messaging channels. Messages flow in one direction from a sender to a receiver.

Messages have a body (a payload), headers, and optional attachments, as illustrated in figure 1.0.

Messages are uniquely identified with an identifier of type java.lang.String. The identifier’s uniqueness is enforced and guaranteed by the message creator, its protocol dependent, and it doesn’t have a guaranteed format.

For protocols that don’t define a unique message identification scheme, Camel uses its own UID generator.

**HEADERS AND ATTACHMENTS**

Headers are values associated with the message, such as sender identifiers, hints about content encoding, authentication information, and so on.

Headers are name-value pairs; the name is a unique, case-insensitive string, and the value is of type **java.**

**lang.Object**. This means that Camel imposes no constraints on the type of the headers

Headers are stored as a map within the message. A message can also have optional attachments, which are typically used for the web service and email components.

**BODY**

The body is of type **java.lang.Object.** That means that a message can store any kind of content. It also means that it’s up to the application designer to make sure that the receiver can understand the content of the message. When the sender and receiver use different body formats, Camel provides a number of mechanisms to transform the data into an acceptable format, and in many cases the conversion happens automatically with type converters, behind the scenes.

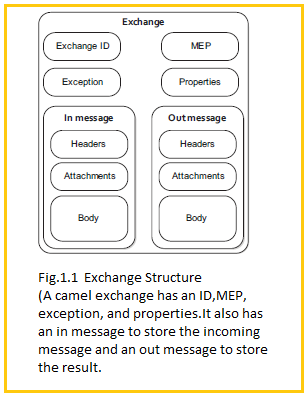
**FAULT FLAG**

***Messages*** also have a fault flag. Some protocols and specifications, such as WSDL and JBI, distinguish between output and fault messages. They’re both valid responses to invoking an operation, but the latter indicates an unsuccessful outcome. In general, faults aren’t handled by the integration infrastructure. They’re part of the contract between the client and the server and are handled at the application level during routing; messages are contained in an exchange.

#### Exchange

An exchange in Camel is the message’s container during routing. An exchange also provides support for the various types of interactions between systems, also known as message exchange patterns (MEPs). MEPs are used to differentiate between one-way and request-response messaging styles. The Camel exchange holds a pattern property that can be either

* **InOnly**—A one-way message (also known as an Event message). For example, JMS messaging is often one-way messaging.
* **InOut**—A request-response message. For example, HTTP-based transports are often request reply, where a client requests to retrieve a web page, waiting for the reply from the server.



***Exchange ID***- A unique ID that identifies the exchange. Camel will generate a default unique ID, if you don’t explicitly set one.

***MEP*** - A pattern that denotes whether you’re using the InOnly or InOut messaging style. When the pattern is InOnly, the exchange contains an in message. For InOut, an out message also exists that contains the reply message for the caller.

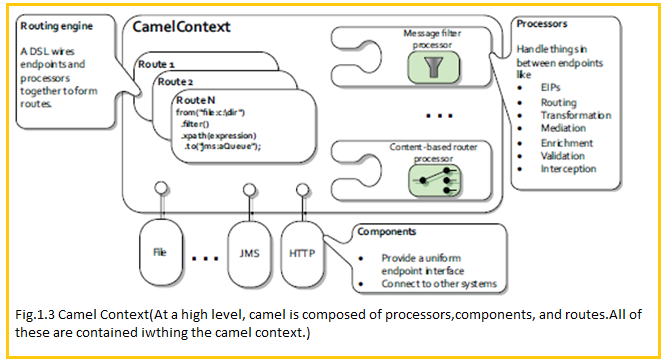
***Exception*** -If an error occurs at any time during routing, an Exception will be set in the exception field.

***Properties***  - Similar to message headers, but they last for the duration of the entire exchange. Properties are used to contain global-level information, whereas message headers are specific to a particular message. Camel itself will add various properties to the exchange during routing. You, as a developer, can store and retrieve properties at any point during the lifetime of an exchange.

***In message—***this is the input message, which is mandatory. The in message contains the request message.

***Out message—***this is an optional message that only exists if the **MEP** is InOut. The out message contains the reply message.

### Camel’s architecture



* The routing engine uses routes as specifications for where messages are routed.

Routes are defined using one of Camel’s domain-specific languages (DSLs).

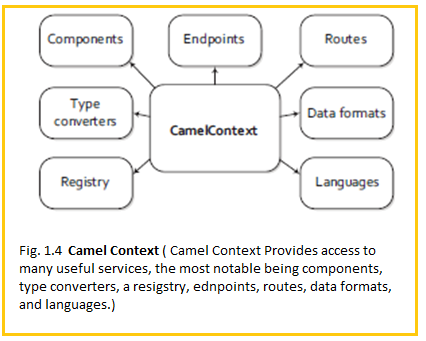
* Processors are used to transform and manipulate messages during routing and also to

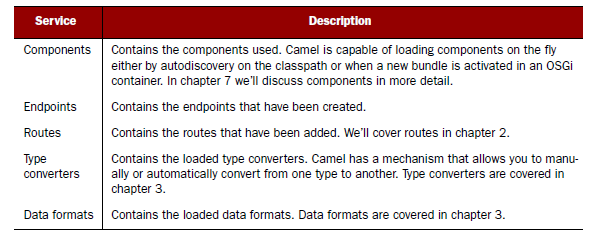
Implement all the EIP patterns, which have corresponding keywords in the DSL languages.

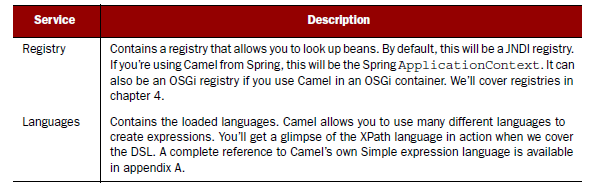
* Components are the extension points in Camel for adding connectivity to other systems.

To expose these systems to the rest of Camel, components provide an endpoint interface.

### Camel context concepts







### Key points

* Where Camel Context is Camel’s runtime system, which keeps all the pieces together.
* Camel’s routing engine is what actually moves messages under the hood
* Routes are obviously a core abstraction for Camel.
* Each route in Camel has a unique identifier that’s used for logging, debugging, monitoring

and starting and stopping routes.

* Routes also have exactly one input source for messages, so they’re effectively tied to an input endpoint.
* To define a route, a DSL is used.
* To wire processors and endpoints together to form routes, Camel defines a DSL
* In Camel, DSL means a fluent Java API that contains methods named for EIP terms.
* Example

Here, in a single Java statement, you define a route that consumes files from a file endpoint.

Messages are then routed to the filter EIP, which will use an XPath predicate to

test whether the message is a test order or not. If a message passes the test, it’s forwarded

to the JMS endpoint. Messages failing the filter test will be dropped

**Java DSL:**

*from("file:data/inbox")(file is component of Camel)*

*.filter().xpath("/order[not(@test)]")(filter is an EIP)*

*.to("jms:queue:order")(JMS is component of camel)*

**Spring DSL:**

*<route>*

*<from uri="file:data/inbox"/>*

*<filter>*

*<xpath>/order[not(@test)]</xpath>*

*<to uri="jms:queue:order"/>*

*</filter>*

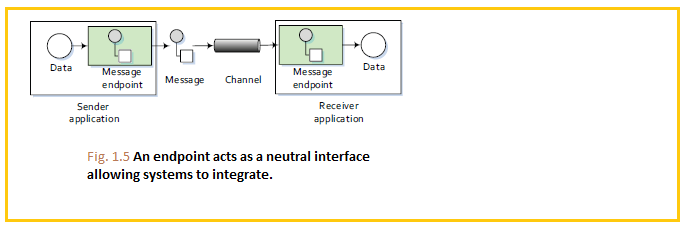
*</route>*

* The processor is a core Camel concept that represents a node capable of using, creating, or modifying an incoming exchange. During routing, exchanges flow from one processor to another; as such, you can think of a route as a graph having specialized processors as the nodes, and lines that connect the output of one processor to the input of another. Many of the processors are implementations of EIPs, but one could easily implement their own custom processor and insert it into a route.
* Components are the main extension point in Camel. To date, there are over 80 components

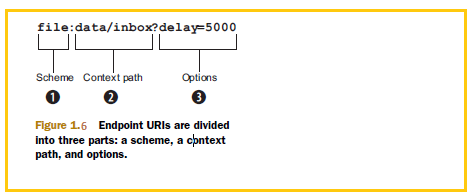
in the Camel ecosystem that range in function from data transports, to DSLs,

data formats, and so on.

* An endpoint is the Camel abstraction that models the end of a channel through Which a system can send or receive messages.



* Explanation:



In Camel, you configure endpoints using URIs, such as file:data/inbox?delay=5000, and you

also refer to endpoints this way. At runtime, Camel will look up an endpoint based on the URI

notation. Figure 1.9 shows how this works.

* The scheme **1** denotes which Camel component handles that type of endpoint. In this case, the scheme of file selects the FileComponent.The FileComponent then works as a factory creating the FileEndpoint based on the remaining parts of the URI.
* The context path data/inbox **2** tells the FileComponent that the starting folder is data/inbox.
* The option**,**delay=5000 **3** indicates that files should be polled at a 5 second interval.

### POM Dependency of Camel

Add the following in pom.xml under dependencies tag

**If using Camel core**

*<dependency>*

*<groupId>org.apache.camel</groupId>*

*<artifactId>camel-core</artifactId>*

*<version>${camel-version}</version>*

*</dependency>*

**If using FTP**

*<dependency>*

*<groupId>org.apache.camel</groupId>*

*<artifactId>camel-ftp</artifactId>*

*<version>2.5.0</version>*

*</dependency>*

**If using JMS**

*<dependency>*

*<groupId>org.apache.camel</groupId>*

*<artifactId>camel-jms</artifactId>*

*<version>2.5.0</version>*

*</dependency>*

**If using ActiveMQ**

*<dependency>*

*<groupId>org.apache.activemq</groupId>*

*<artifactId>activemq-core</artifactId>*

*<version>5.3.2</version>*

*</dependency>*

**If Using Log4j**

*<dependency>*

*<groupId>org.slf4j</groupId>*

*<artifactId>slf4j-log4j12</artifactId>*

*<version>5.3.2</version>*

*</dependency>*

## Routing with Camel

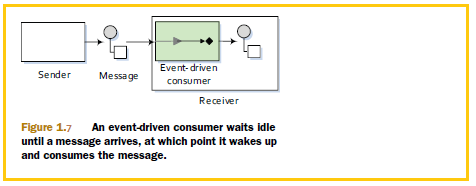
### Types of Consumers

* In Camel there are two kinds of consumers: event-driven consumers and polling consumers.

**EVENT-DRIVEN CONSUMER**

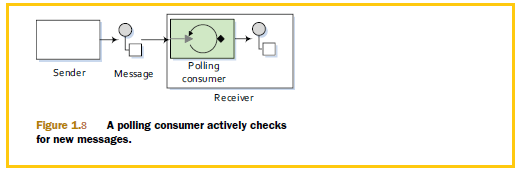
This kind of consumer is mostly associated with client-server architectures and web services. It’s also referred

to as an asynchronous receiver in the EIP world. An event-driven consumer listens on a particular messaging channel, usually a TCP/IP port or a JMS queue, and waits for a client to send messages to it. When a message arrives, the consumer wakes up and takes the message for processing.

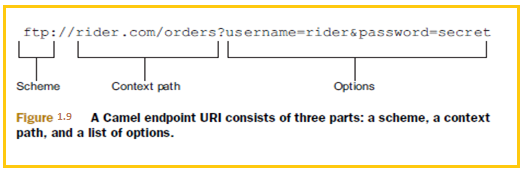


**POLLING CONSUMER**

In contrast to the event-driven consumer, the polling consumer actively goes and fetches messages from a particular source, such as an FTP server. The polling consumer is also known as a synchronous receiver in EIP lingo, because it won’t poll for more messages until it has finished processing the current message. A common flavor of the polling consumer is the scheduled polling consumer, which polls at scheduled intervals. File, FTP, and email transports all use scheduled polling consumers.

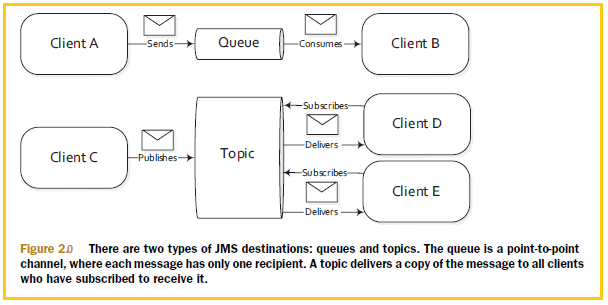


* from("ftp://rider.com/orders?username=rider&password=secret")

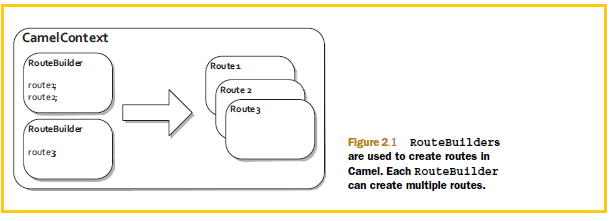


### JMS overview

* **JMS (Java Message Service)** is a Java API that allows you to create, send, receive, and read messages. It also mandates that messaging is asynchronous and has specific elements of reliability, like guaranteed and once-and-only-once delivery. JMS is the defacto messaging solution in the Java community.
* JMS providers are usually referred to as brokers because they manage the communication between a message producer and a message consumer.
* Queues are strictly point-to-point, where each message has only one consumer.
* Topics operate on a publish/subscribe scheme; a single message may be delivered to many consumer. if they have subscribed to the topic

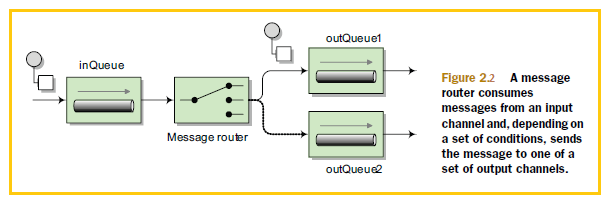


### Route Builder



Routing happens in many aspects of everyday life. When you mail a letter, for instance, it may be routed through several cities before reaching its final address. An email you send will be routed through many different computer network systems before reaching its final destination. In all cases, the router’s function is to selectively move the message forward.

In the context of enterprise messaging systems, routing is the process by which a message is taken from an input queue and, based on a set of conditions, sent to one of several output queues, as shown in figure 2.2.



**Note**:

* Input and output queues are unaware of the conditions in between them.
* The conditional logic is decoupled from the message consumer and producer.
* A route may have many processing components that modify the message or send it to another location, or it may have none.
* Some endpoints can have an intimidating list of endpoint URI properties . For instance, the JMS component has about 60 options, many of which are only used in specific JMS scenarios. Camel always tries to provide built-in defaults that fit most cases
* Refer: <http://camel.apache.org/jms.html>. ( For JMS Properties)
* *to("jms:queue:incomingOrders")*

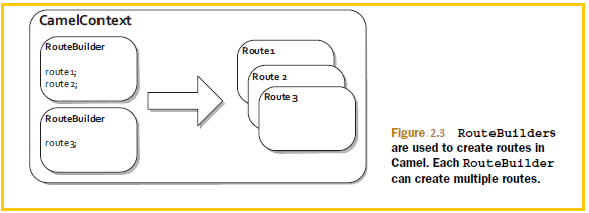
This can be read as sending message to the JMS queue named incomingOrders

### Creating routes in Java

The addRoutes method of the CamelContext accepts a RoutesBuilder, not just a RouteBuilder. The RoutesBuilder interface has a single method defined:

void addRoutesToCamelContext(CamelContext context) throws Exception;

This means that you could use your own custom class to build Camel routes. The most common way to build routes, though, is to use the RouteBuilder class, which implements RoutesBuilder. The RouteBuilder class also gives you access to Camel’s Java DSL for route creation.



The abstract **org.apache.camel.builder.RouteBuilder** class in Camel is one that you’ll see frequently. You’ll need to use it any time you create a route in Java. To use the RouteBuilder class, you extend a class from it and implement the configure method, like this:

**class** MyRouteBuilder **extends** RouteBuilder {

**public** **void** configure() **throws** Exception {

...

}

}

You then need to add the class to the CamelContext with the addRoutes method:

CamelContext context = **new** DefaultCamelContext();

context.addRoutes(**new** MyRouteBuilder());

Alternatively, you can combine the RouteBuilder and CamelContext configuration by adding an anonymous RouteBuilder **class** directly into the CamelContext, like **this**:

CamelContext context = **new** DefaultCamelContext();

context.addRoutes(**new** RouteBuilder() {

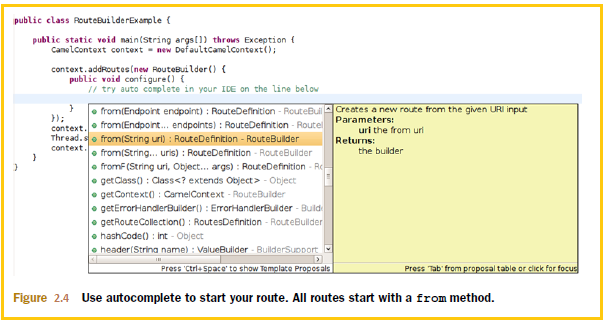
**public** **void** configure() **throws** Exception {

...

}

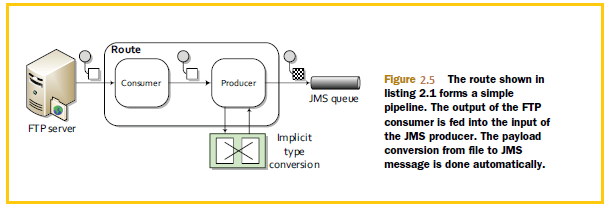
});

Within the configure method, you define your routes using the Java DSL



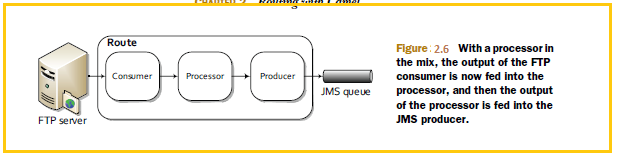
Ctrl-space in Eclipse) in the *configure* method, you’ll be presented with a number of methods. To start a route, you should use the from method.

The flow of messages in this simple route can be viewed as a basic pipeline, where the output of the consumer is fed into the producer as input. This is depicted in figure 2.5.



One thing you may have noticed is that we didn’t do any conversion from the FTP file type to the JMS message type—this was done automatically by Camel’s Type Converter facility. You can force type conversions to occur at any time during a route, but often you don’t have to worry about them at all

### Adding a Processor



The *Processor* interface in Camel is an important building block of complex routes. It’s a simple interface, having a single method:

*public void process(Exchange exchange) throws Exception;*

This gives you full access to the message exchange, letting you do pretty much whatever you want with the payload or headers.

All EIPs in Camel are implemented as processors.

You can even add a simple processor to your route inline, like so:

*from("ftp://rider.com/orders?username=rider&password=secret").*

*process(new Processor() {*

*public void process(Exchange exchange) throws Exception {*

*System.out.println("We just downloaded: "*

*+ exchange.getIn().getHeader("CamelFileName"));*

*}*

*}).*

*to("jms:incomingOrders");*

This route will now print out the filename of the order that was downloaded before sending it to the JMS queue.

The output of the FTP consumer is fed into the processor as input; the processor doesn’t modify the message payload or headers, so the exchange moves on to the JMS producer as input.

**NOTE**

Many components, like the FileComponent and the FtpComponent, set useful headers describing the payload on the incoming message. In the previous example, you used the CamelFileName header to retrieve the filename of the file that was downloaded via FTP. The component pages of the online documentation contain information about the headers set for each individual component. You’ll find information about the FTP component at <http://camel.apache.org/ftp2.html>.

### Bean injection and Spring

Creating an application from beans using Spring is pretty simple. All you need are a few Java beans (classes), a Spring XML configuration file, and an ApplicationContext. The ApplicationContext is similar to the CamelContext, in that it’s the runtime container for Spring.

Consider an application that prints out a greeting followed by your username. In this application you don’t want the greeting to be hardcoded, so you can use an interface to break this dependency. Consider the following interface:

**public** **interface** Greeter {

**public** String sayHello();

}

This **interface** is implemented by the following classes:

**public** **class** EnglishGreeter **implements** Greeter {

**public** String sayHello() {

**return** "Hello " + System.*getProperty*("user.name");

}

}

**public** **class** DanishGreeter **implements** Greeter {

**public** String sayHello() {

**return** "Davs " + System.*getProperty*("user.name");

}

}

You *can* now create a greeter application as follows:

**public** **class** GreetMeBean {

**private** Greeter greeter;

**public** **void** setGreeter(Greeter greeter) {

**this**.greeter = greeter;

}

**public** **void** execute() {

System.*out*.println(greeter.sayHello());

}

}

This application will output a different greeting depending on how you configure it.

To configure this application using Spring, you could do something like this:

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

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

<bean id=*"myGreeter"* class=*"camelinaction.EnglishGreeter"* />

<bean id=*"greetMeBean"* class=*"camelinaction.GreetMeBean"*>

<property name=*"greeter"* ref=*"myGreeter"* />

</bean>

</beans>

This XML file instructs spring to do the following:

* Create an instance of EnglishGreeter and names the bean myGreeter
* Create an instance of GreetMeBean and names the bean greetMeBean
* Set the reference of the greeter property of the GreetMeBean to the bean named myGreeter

This configuring of beans is called wiring.

In order to load this XML file into Spring, you can use the ClassPathXmlApplicationContext, which is a concrete implementation of the ApplicationContext that’s provided with the Spring framework. This class loads Spring XML files from a location specified on the classpath.

Here is the final version of **GreetMeBean:**

**public class GreetMeBean {**

**...**

**public static void main(String[] args) {**

**ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");**

**GreetMeBean bean = (GreetMeBean) context.getBean("greetMeBean");**

**bean.execute();**

**}**

**}**

The ClassPathXmlApplicationContext you instantiate here loads up the bean definitions you saw previously in the beans.xml file. You then call getBean on the context to look up the bean with the greetMeBean ID in the Spring registry. All beans defined in this file are accessible in this way.

### How to Configure Camel to Use a JMS Provider

* To connect Camel to a specific JMS provider, you need to configure Camel’s JMS component

with an appropriate ConnectionFactory.

* you can create an ActiveMQConnectionFactory that points to the location of the running ActiveMQ broker:
* ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("vm://localhost");
* The vm://localhost URI means that you should connect to an embedded broker named “localhost” running inside the current JVM.
* The vm transport connector in ActiveMQ creates a broker on demand if one isn’t running already, so it’s very handy for quickly testing JMS applications; for production scenarios, it’s recommended that you connect to a broker that’s already running.
* Next, when you create your CamelContext, you can add the JMS component as follows:

CamelContext context = new DefaultCamelContext();(it will initialize all the camel stuff)

context.addComponent("jms",JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));

**Overall code: JAVA DSL**

**import** javax.jms.ConnectionFactory;

**import** org.apache.activemq.ActiveMQConnectionFactory;

**import** org.apache.camel.CamelContext;

**import** org.apache.camel.Exchange;

**import** org.apache.camel.Processor;

**import** org.apache.camel.builder.RouteBuilder;

**import** org.apache.camel.component.jms.JmsComponent;

**import** org.apache.camel.impl.DefaultCamelContext;

/\*\*

\* A route that polls an FTP server for new orders, downloads them, converts the order

\* file into a JMS Message and then sends it to the JMS incomingOrders queue hosted

\* on an embedded ActiveMQ broker instance.

\*

\* **@author** janstey

\*

\*/

**public** **class** FtpToJMSWithProcessorExample {

**public** **static** **void** main(String args[]) **throws** Exception {

// create CamelContext

CamelContext context = **new** DefaultCamelContext();

// connect to embedded ActiveMQ JMS broker

ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory("vm://localhost");

context.addComponent("jms", JmsComponent.*jmsComponentAutoAcknowledge*(connectionFactory));

// add our route to the CamelContext

context.addRoutes(**new** RouteBuilder() {

@Override

**public** **void** configure() {

from("ftp://rider.com/orders?username=rider&password=secret").

process(**new** Processor() {

**public** **void** process(Exchange exchange) **throws** Exception {

System.*out*.println("We just downloaded: " + exchange.getIn().getHeader("CamelFileName"));

}

}).

to("jms:incomingOrders");

}

});

// start the route and let it do its work

context.start();

Thread.sleep(10000);

// stop the CamelContext

context.stop();

}

}

**Above configuration in Spring DSL**

**camel-context.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!--

Licensed to the Apache Software Foundation (ASF) under one or more

contributor license agreements. See the NOTICE file distributed with

this work for additional information regarding copyright ownership.

The ASF licenses this file to You under the Apache License, Version 2.0

(the "License"); you may not use this file except in compliance with

the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software

distributed under the License is distributed on an "AS IS" BASIS,

WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and

limitations under the License.

-->

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

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

*http://camel.apache.org/schema/spring*

[*http://camel.apache.org/schema/spring/camel-spring.xsd*](http://camel.apache.org/schema/spring/camel-spring.xsd)*"*>

//configuring the JMS

<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">

<property name="connectionFactory">

<bean class="org.apache.activemq.ActiveMQConnectionFactory">

<property name="brokerURL" value="vm://localhost" />

</bean>

</property>

</bean>

//Processor definition

<bean id=*"downloadLogger"* class=*"camelinaction.DownloadLogger"*/>

//Routing logic

<camelContext xmlns="http://camel.apache.org/schema/spring">

<route>

<from uri="file:src/data?noop=true"/>

<process ref="downloadLogger"/>

<to uri="jms:incomingOrders"/>

</route>

</camelContext>

</beans>

### Routing and EIP Points

#### Using Multiple Router’s

You may recall that in the Java DSL each Java statement starting with a from creates a new route. You can also create multiple routes with the Spring DSL. To do this, simply add an additional route element within the camelContext element.

For example, move the *DownloadLogger* bean into a second route, after the order gets sent to the *incomingOrders* queue:

<?xml version="1.0" encoding="UTF-8"?>

<camelContext xmlns="http://camel.apache.org/schema/spring">

<route>

<from uri="file:src/data?noop=true" />

<to uri="jms:incomingOrders" />

</route>

<route>

<from uri="jms:incomingOrders" />

<process ref="downloadLogger" />

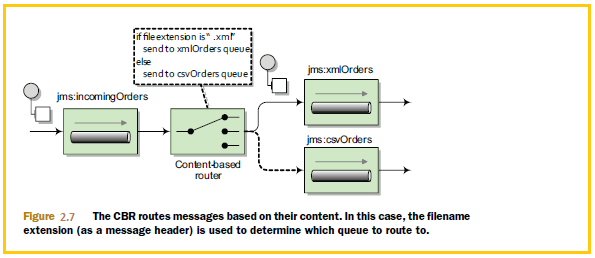
</route>

</camelContext>

Now you are consuming the message from the *incomingOrders* queue in the second route. So, the downloaded message will be printed after the order is sent to the queue.

#### Content-based router

Content-Based Router (CBR) is a message router that routes a message to a destination based on its content. The content could be a message header, the payload data type, part of the payload itself—pretty much anything in the Message exchange.



**Code Sample :**

**import** javax.jms.ConnectionFactory;

**import** org.apache.activemq.ActiveMQConnectionFactory;

**import** org.apache.camel.CamelContext;

**import** org.apache.camel.Exchange;

**import** org.apache.camel.Processor;

**import** org.apache.camel.builder.RouteBuilder;

**import** org.apache.camel.component.jms.JmsComponent;

**import** org.apache.camel.impl.DefaultCamelContext;

/\*\*

\* A set of routes that watches a directory for new orders, reads them, converts the

\* order file into a JMS Message and then sends it to the JMS incomingOrders queue

\* hosted on an embedded ActiveMQ broker instance.

\* From there a content-based router is used to send the order to either the

\* xmlOrders or csvOrders queue.

\*

\* **@author** janstey

\*

\*/

**public** **class** OrderRouter {

**public** **static** **void** main(String args[]) **throws** Exception {

// create CamelContext

CamelContext context = **new** DefaultCamelContext();

// connect to embedded ActiveMQ JMS broker

ConnectionFactory connectionFactory = **new** ActiveMQConnectionFactory("vm://localhost");

context.addComponent("jms",JmsComponent.*jmsComponentAutoAcknowledge*(connectionFactory));

// add our route to the CamelContext

context.addRoutes(**new** RouteBuilder() {

@Override

**public** **void** configure() {

// load file orders from src/data into the JMS queue

from("file:src/data?noop=true").to("jms:incomingOrders");

// content-based router

from("jms:incomingOrders")

.choice()

.when(header("CamelFileName").endsWith(".xml"))

.to("jms:xmlOrders")

.when(header("CamelFileName").regex("^.\*(csv|csl)$"))

.to("jms:csvOrders")

.otherwise().to("jms:badOrders");

// test that our route is working

from("jms:xmlOrders").process(**new** Processor() {

**public** **void** process(Exchange exchange) **throws** Exception {

System.*out*.println("Received XML order: "

+ exchange.getIn().getHeader("CamelFileName"));

}

});

from("jms:csvOrders").process(**new** Processor() {

**public** **void** process(Exchange exchange) **throws** Exception {

System.*out*.println("Received CSV order: "

+ exchange.getIn().getHeader("CamelFileName"));

}

});

}

});

// start the route and let it do its work

context.start();

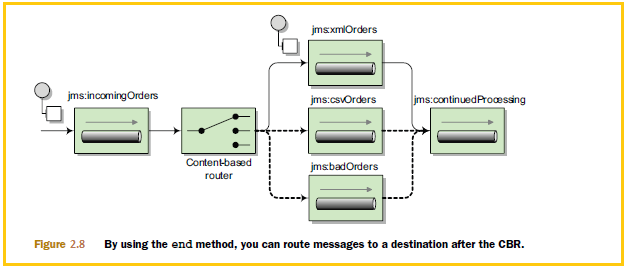
Thread.*sleep*(10000);

// stop the CamelContext

context.stop();

}

}



**Same above condition in Spring DSL**

<?xml version="1.0" encoding="UTF-8"?>

<route>

<from uri="jms:incomingOrders" />

<choice>

<when>

<simple>${header.CamelFileName} regex '^.\*xml$'</simple>

<to uri="jms:xmlOrders" />

</when>

<when>

<simple>${header.CamelFileName} regex '^.\*(csv|csl)$'</simple>

<to uri="jms:csvOrders" />

</when>

<otherwise>

<to uri="jms:badOrders" />

<stop />

</otherwise>

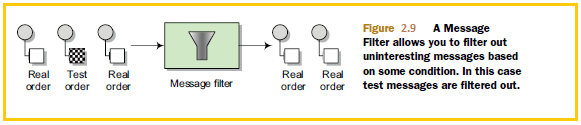
</choice>

<to uri="jms:continuedProcessing" />

</route>

#### Using message filters.

Message Filter allows you to filter out uninteresting messages based on some condition.



<?xml version="1.0" encoding="UTF-8"?>

<order name="motor" amount="1" customer="foo" test="true"/>

from("jms:xmlOrders").filter(xpath("/order[not(@test)]"))

.process(new Processor() {

public void process(Exchange exchange) throws Exception {

System.out.println("Received XML order: "

+ exchange.getIn().getHeader("CamelFileName"));

}

});

<route>

<from uri="jms:xmlOrders"/>

<filter>

<xpath>/order[not(@test)]</xpath>

<process ref="orderLogger"/>

</filter>

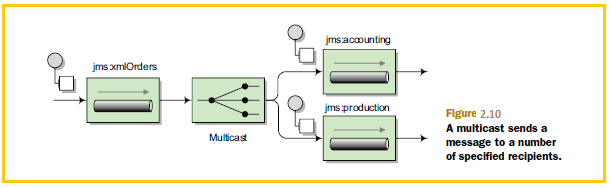
</route>

<?xml version="1.0" encoding="UTF-8"?>

<order name="motor" amount="1" customer="foo" test="true"/>

#### Using multicasting

Sending message to different destinations at the same time



With Camel, you can use the multicast method in the Java DSL to implement this solution:

from("jms:xmlOrders").multicast().to("jms:accounting", "jms:production");

from("jms:xmlOrders")

.multicast().stopOnException()

.parallelProcessing().executorService(executor)

.to("jms:accounting", "jms:production");

<?xml version="1.0" encoding="UTF-8"?>

<route>

<from uri="jms:xmlOrders" />

<multicast>

<to uri="jms:accounting" />

<to uri="jms:production" />

</multicast>

</route>

**Note**: For dealing with responses from services invoked in a multicast, an aggregator is used.

By default, the multicast sends message copies sequentially. In the preceding example, a message is sent to the accounting queue and then to the production queue. But what if you wanted to send them in parallel?

#### Parallel Multicasting

This will set up the multicast to distribute messages to the destinations in parallel

Sending messages in parallel using the multicast involves only one extra DSL method: parallelProcessing

from("jms:xmlOrders")

.multicast().parallelProcessing()

.to("jms:accounting", "jms:production");

A default thread pool size of 10 is used if you don’t specify anything else

If you want to change this default, you can set the underlying **java.util.concurrent.ExecutorService** that’s used to launch new asynchronous message sends by using the **executorService**

DSL method here’s an example:

ExecutorService executor = Executors.newFixedThreadPool(16);

from("jms:xmlOrders")

.multicast().parallelProcessing().executorService(executor)

.to("jms:accounting", "jms:production");

from("jms:xmlOrders")

.multicast().stopOnException()

.parallelProcessing().executorService(executor)

.to("jms:accounting", "jms:production");

<route>

<from uri="jms:xmlOrders"/>

<multicast parallelProcessing="true" executorServiceRef="executor">

<to uri="jms:accounting"/>

<to uri="jms:production"/>

</multicast>

</route>

<bean id="executor" class="java.util.concurrent.Executors" factory-method="newFixedThreadPool">

<constructor-arg index="0" value="16"/>

</bean>

By default, the multicast will continue sending messages to destinations even if one fails.

In your application, though, you may consider the whole process as failed if one destination fails. What do you do in this case?

**STOPPING THE MULTICAST ON EXCEPTION**

To enable this feature, use the stopOnException method as follows:

from("jms:xmlOrders")

.multicast().stopOnException()

.parallelProcessing().executorService(executor)

.to("jms:accounting", "jms:production");

<route>

<from uri="jms:xmlOrders"/>

<multicast stopOnException="true" parallelProcessing="true" executorServiceRef="executor">

<to uri="jms:accounting"/>

<to uri="jms:production"/>

</multicast>

</route>

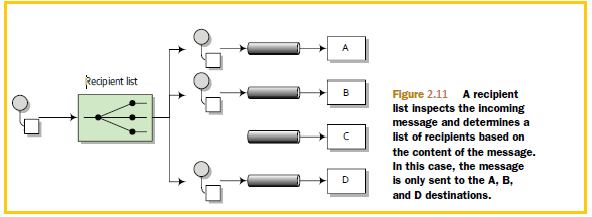
<bean id="executor" class="java.util.concurrent.Executors" factory-method="newFixedThreadPool">

<constructor-arg index="0" value="16"/>

</bean>

#### Using recipient lists

A recipient list first inspects the incoming message, then generates a list of desired recipients dynamically based on the message content, and sends the message to those recipients.



Parallelizing only top tier customers’ orders to production, all other orders would have to go to accounting first, thereby not bogging down production.

**Note**: that the recipient list is different from the multicast because the list of recipients is dynamic

For example, the following route will take the list of recipients from a header named

recipients, where each recipient is separated from the next by a comma:

from("jms:xmlOrders")

.recipientList(header("recipients"));

You need some way of determining whether the message is from a top-tier customer or not.

A simple solution could be to add a custom processor to do this:

from("jms:xmlOrders")

.setHeader("customer", xpath("/order/@customer"))

.process(new Processor() {

public void process(Exchange exchange) throws Exception {

String recipients = "jms:accounting";

String customer =

exchange.getIn().getHeader("customer", String.class);

if (customer.equals("honda")) {

recipients += ",jms:production";

}

exchange.getIn().setHeader("recipients", recipients);

}

})

.recipientList(header("recipients"));

The processor now sets the recipients header to "jms:accounting, jms:production"

only if the customer is at the gold level of support. The check for gold-level support here is greatly simplified—ideally you’d query a database for this check. Any other orders will be routed only to accounting, which will send them to production after the checks are complete.

The Spring DSL version of this route follows a very similar layout:

<route>

<from uri="jms:xmlOrders" />

<setHeader headerName="customer">

<xpath>/order/@customer</xpath>

</setHeader>

<process ref="calculateRecipients" />

<recipientList>

<header>recipients</header>

</recipientList>

</route>

**RECIPIENT LIST ANNOTATION**

Rather than using the recipientList method in the DSL, you can add a @Recipient-List annotation to a method in a plain Java class (a Java bean). This annotation tells Camel that the annotated method should be used to generate the list of recipients from the exchange. This behavior only gets invoked, however, if the class is used with Camel’s bean integration.

For example, replacing the custom processor you used in the previous section with an annotated bean results in a greatly simplified route:

from("jms:xmlOrders").bean(RecipientListBean.class);

Now all the logic for calculating the recipients and sending out messages is captured

in the RecipientListBean class, which looks like this:

public class RecipientListBean {

@RecipientList

public String[] route(@XPath("/order/@customer") String customer) {

if (isGoldCustomer(customer)) {

return new String[] {"jms:accounting", "jms:production"};

} else {

return new String[] {"jms:accounting"};

}

}

private boolean isGoldCustomer(String customer) {

return customer.equals("honda");

}

}

Notice that the return type of the bean is a list of the desired recipients. Camel will take this list and send a copy of the message to each destination in the list.

You also have access to Camel’s bean-binding annotations, which allow you to extract data from the message

using expressions, so you don’t have to manually explore the exchange. This example uses the @XPath bean-binding annotation to grab the customer attribute of the order element in the body.

#### Using the wireTap method

WireTap method is used for passing the message to different destination without breaking or disturbing the existing flow or route.

We can use this in case of auditing the incoming request. It’s a fire and forgets type fashion

By using the wireTap method in the Java DSL, you can send a copy of the exchange to a secondary destination without affecting the behavior of the rest of the route:

from("jms:incomingOrders")

.wireTap("jms:orderAudit")

.choice()

.when(header("CamelFileName").endsWith(".xml"))

.to("jms:xmlOrders")

.when(header("CamelFileName").regex("^.\*(csv|csl)$"))

.to("jms:csvOrders")

.otherwise()

.to("jms:badOrders");

The preceding code sends a copy of the exchange to the orderAudit queue, and the original exchange continues on through the route, as if you hadn’t used a wire tap at all. Camel doesn’t wait for a response from the wire tap because the wire tap sets the message exchange pattern (MEP) to InOnly. This means that the message will be sent

to the orderAudit queue in a fire-and-forget fashion—it won’t wait for a reply.

In the Spring DSL, you can configure a wire tap just as easily:

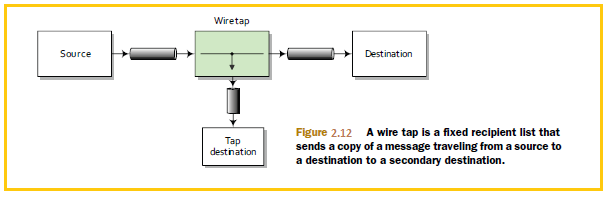
<route>

<from uri="jms:incomingOrders"/>

<wireTap uri="jms:orderAudit"/>

..

The wire tap is a pretty useful monitoring tool, but it leaves most of the work up to you.



# Core Camel

## Using the getIn and getOut methods on exchanges

The Camel Exchange defines two methods for retrieving messages: getIn and getOut. The getIn method returns the incoming message, and the getOut method accesses the outbound message.

There are two scenarios where the Camel end user will have to decide among using these methods:

* A read-only scenario, such as when you’re logging the incoming message
* A write scenario, such as when you’re transforming the message

In the second scenario, you’d assume getOut should be used. That’s correct according to theory, but in practice there’s a common pitfall when using getOut: the incoming message headers and attachments will be lost. This is often not what you want,so you must copy the headers and attachments from the incoming message to the outgoing message, which can be tedious. The alternative is to set the changes directly

on the incoming message using getIn, and not to use getOut at all. This is the practice we use in this book.

## The Direct component

The example here uses the Direct component (http://camel.apache.org/direct) as the input source for the route (from("direct:start")). The Direct component provides direct invocation between a producer and a consumer. It only allows connectivity from within Camel, so external systems can’t send messages directly to it. This component is used within Camel to do things such as link routes together or for testing.

## Transforming data with Camel

### Data transformation overview

Data *transformation* is a broad term that covers two types of transformation:

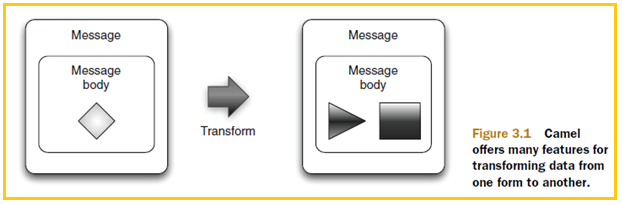
* *Data format transformation*—The data format of the message body is transformed from one form to another.

For example, a CSV record is formatted as XML.

* *Data type transformation*—The data type of the message body is transformed from one type to another.

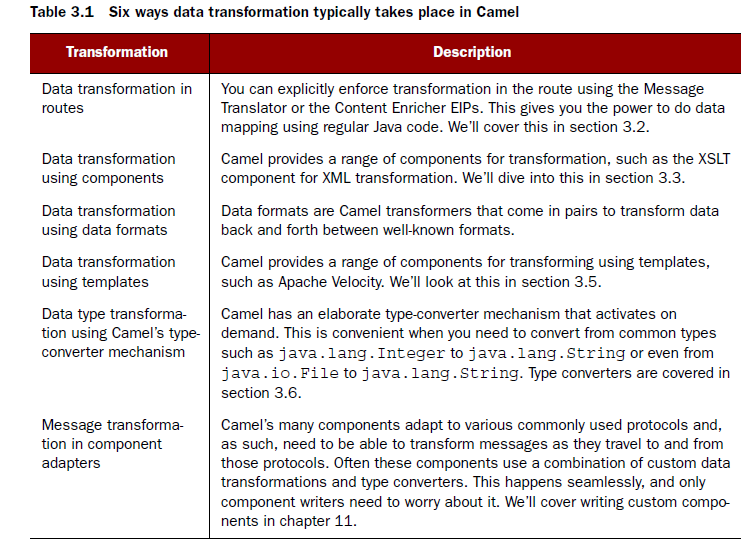
For example a **java.lang.String** is transformed into a **javax.jms.TextMessage.**

Camel has a built in type-converter mechanism that can automatically convert between types, which greatly reduces the need for end users to deal with type transformations.



#### Data transformation with Camel

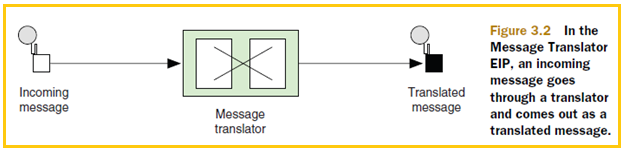
In Camel, data transformation typically takes places in the six ways



### Transforming data using EIPs and Java

Data mapping is the process of mapping between two distinct data models, and it’s a key factor in data integration. Camel provides great freedom in data mapping because it allows you to use Java code.

#### Using the Message Translator EIP



This pattern covers translating a message from one format to another. It’s the equivalent of the Adapter pattern from the Gang of Four book.

Camel provides three ways of using this pattern:

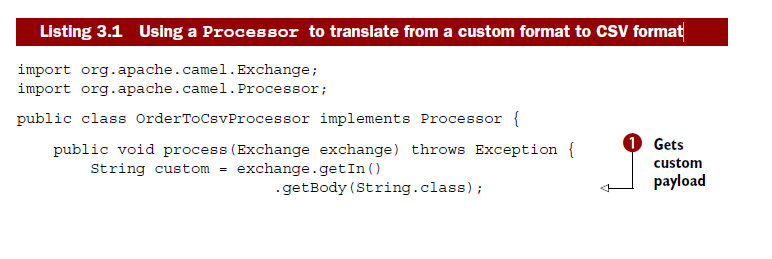
* Using a Processor
* Using beans
* Using <transform>

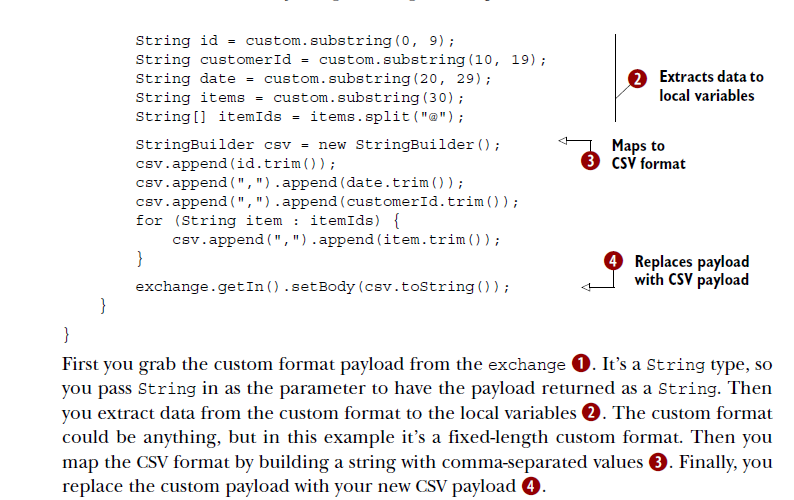
**TRANSFORMING USING A PROCESSOR**

The Camel *Processor* is an interface defined in **org.apache.camel.Processor** with a single method:

public void process(Exchange exchange) throws Exception;

The Processor is a low-level API where you work directly on the Camel Exchange instance. It gives you full access to all Camel’s moving parts from the CamelContext, which you can obtain Exchange using the getCamelContext method.





You can use the **OrderToCsvProcessor** in a Camel route as follows:

from("quartz://report?cron=0+0+6+\*+\*+?")

.to("http://riders.com/orders/cmd=received&date=yesterday")

.process(new OrderToCsvProcessor())

.to("file://riders/orders?fileName=report-${header.Date}.csv");

The preceding route uses Quartz to schedule a job to run once a day at 6 a.m. It then invokes the HTTP service to retrieve the orders received yesterday, which are returned in the custom format. Next, it uses **OrderToCSVProcessor** to map from the custom format to CSV format before writing the result to a file

The equivalent route in Spring XML is as follows:

<bean id="csvProcessor" class="camelinaction.OrderToCsvProcessor"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">

<route>

<from uri="quartz://report?cron=0+0+6+\*+\*+?"/>

<to uri="http://riders.com/orders/cmd=received&amp;date=yesterday"/>

<process ref="csvProcessor"/>

<to uri="file://riders/orders?fileName=report-${header.Date}.csv"/>

</route>

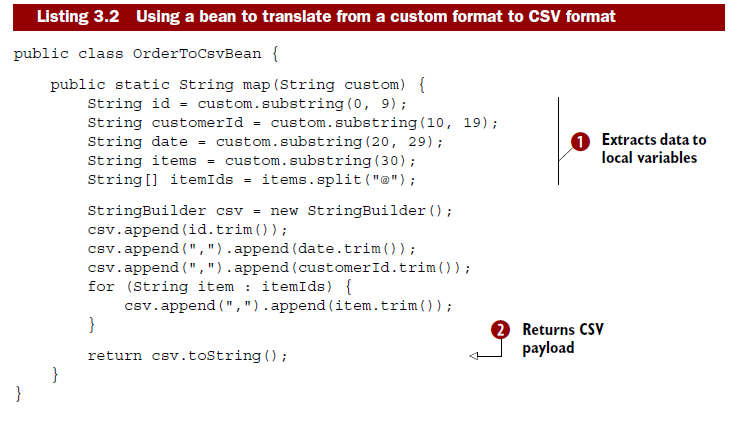
</camelContext>

**Disadvantage:**  Using a processor has one disadvantage: you’re required to use the Camel API. In the

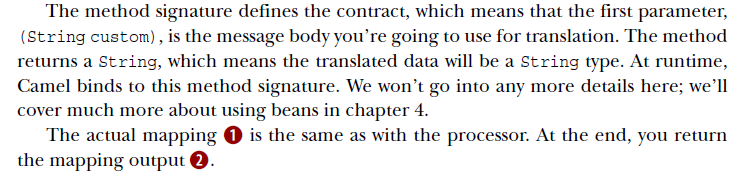
next section, we’ll look at how to avoid this by using a bean

**TRANSFORMING USING BEANS**

Using beans is a great practice because it allows you to use any Java code and library you wish. Camel imposes no restrictions whatsoever. Camel can invoke any bean you choose, so you can use existing beans without having to rewrite or recompile them.



The first noticeable difference between processor and Bean, bean doesn’t use any Camel imports. This means your bean is totally independent of the Camel API. The next difference is that you can name the method signature in this case it’s a static method named map.



You can use OrderToCsvBean in a Camel route as shown here:

from("quartz://report?cron=0+0+6+\*+\*+?")

.to("http://riders.com/orders/cmd=received&date=yesterday")

.bean(new OrderToCsvBean())

.to("file://riders/orders?fileName=report-${header.Date}.csv");

The equivalent route in Spring XML is as follows:

<bean id="csvBean" class="camelinaction.OrderToCsvBean"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">

<route>

<from uri="quartz://report?cron=0+0+6+\*+\*+?"/>

<to uri="http://riders.com/orders/cmd=received&amp;date=yesterday"/>

<bean ref="csvBean"/>

<to uri="file://riders/orders?fileName=report-${header.Date}.csv"/>

</route>

</camelContext>

Another advantage of using beans over processors for mappings is that unit testing is much easier.

As we doesn’t require any Camel stuff to pass while unit testing. Beans are very useful for doing

Message transformation

**TRANSFORMING USING THE TRANSFORM() METHOD FROM THE JAVA DSL**

**Transform** () is a method in the Java DSL that can be used in Camel routes to transform messages. By allowing the use of expressions, transform() permits great flexibility, and using expressions directly within the DSL can sometimes save time.

Suppose you need to prepare some text for HTML formatting by replacing all line breaks with a <br/> tag. This can be done with a built-in Camel expression that searches and replaces using regular expressions:

from("direct:start")

.transform(body().regexReplaceAll("\n", "<br/>"))

.to("mock:result");

What this route does is use the **transform**() method to tell Camel that the message should be transformed using an expression. Camel provides what is known as the Builder pattern to build expressions from individual expressions. This is done by chaining together method calls, which is the essence of the Builder pattern.

In this example, you combine **body**() and **regexReplaceAll**(). The expression should be read as follows: take the body and perform a regular expression that replaces all new lines (\n) with <br/> tags. Now you’ve combined two methods that conform to a compound Camel expression.

Camel also allows you to use custom expressions. This is useful when you need to be in full control and have Java code at your fingertips. For example, the previous example could have been implemented as follows:

from("direct:start")

.transform(new Expression() {

public <T> T evaluate(Exchange exchange, Class<T> type) {

String body = exchange.getIn().getBody(String.class);

body = body.replaceAll("\n", "<br/>");

body = "<body>" + body + "</body>";

return (T) body;

}

})

.to("mock:result");

**TRANSFORMING USING <TRANSFORM> FROM SPRING XML**

Using <**transform**> from Spring XML is a bit different than from Java DSL because the XML DSL isn’t as powerful. In Spring XML, the Builder pattern expressions aren’t available because with XML you don’t have a real programming language underneath

<bean id="htmlBean" class="camelinaction.HtmlBean"/>(1)

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

<route>

<from uri="direct:start"/>

<transform>

<method bean="htmlBean" method="toHtml"/>(2)

</transform>

<to uri="mock:result"/>

</route>

</camelContext>

First, you declare a regular spring bean to be used to transform the message 1. Then, in the route, you use <**transform**> with a <**method**> call expression to invoke the bean 2

The implementation of the htmlBean is very straightforward:

public class HtmlBean {

public static String toHtml(String body) {

body = body.replaceAll("\n", "<br/>");

body = "<body>" + body + "</body>";

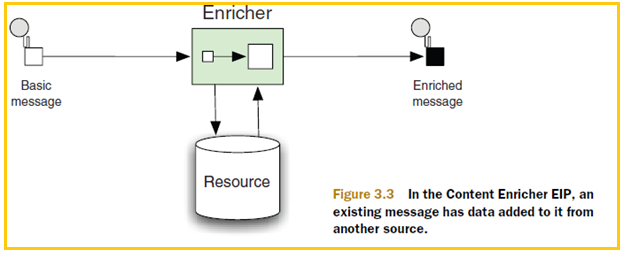
return body;

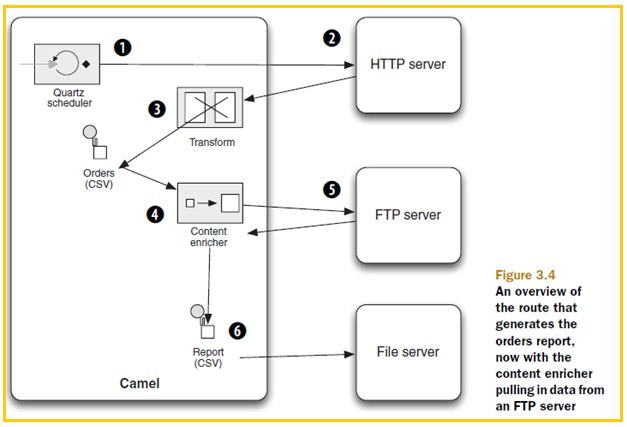
}

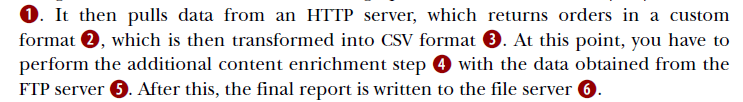
}

#### Using the Content Enricher EIP

This pattern documents the scenario where a message is enriched with data obtained from another resource







Camel provides two operations in the DSL for implementing the pattern:

■ **pollEnrich**—This operation merges data retrieved from another source using a consumer.

■ **enrich**—this operation merges data retrieved from another source using a producer.

The difference between **pollEnrich** and **enrich**

The difference between **pollEnrich** and enrich is that the former uses a consumer and the latter a producer to retrieve data from the source. Knowing the difference is important: the file component can be used with both, but using enrich will write the message content as a file; using **pollEnrich** will read the file as the source, which is most likely the scenario you’ll be facing when enriching with files. The HTTP component only works with enrich; it allows you to invoke an external HTTP service and use its reply as the source.

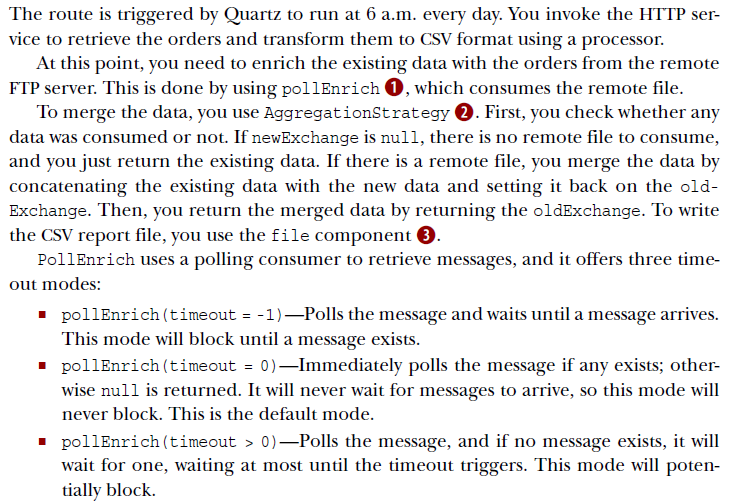
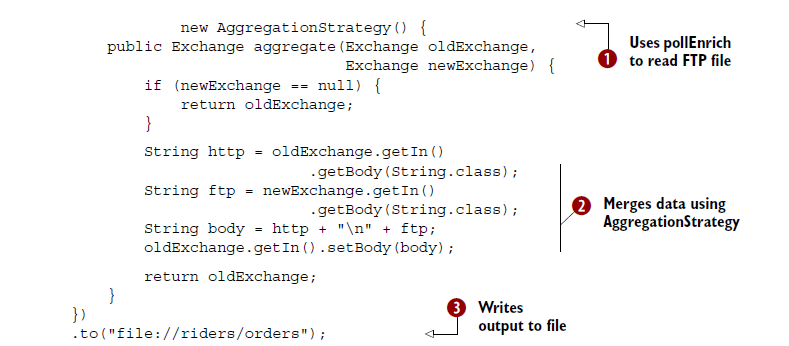
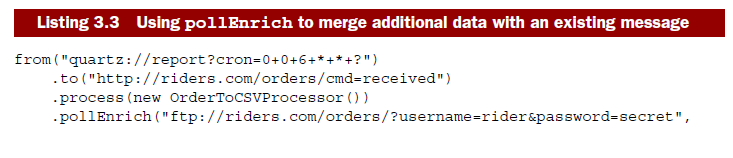
Camel uses the **org.apache.camel.processor.AggregationStrategy** interface to merge the result from the source with the original message, as follows:

Exchange aggregate(Exchange oldExchange, Exchange newExchange);

This aggregate method is a callback that you must implement. The method has two parameters: the first, named oldExchange, contains the original exchange; the second, newExchange, is the enriched source. Java code and return the merged result. This may sound a bit confusing,

**ENRICHING USING POLLENRICH**

Shows how you can use pollEnrich to retrieve the additional orders from the remote FTP server and aggregate this data with the existing message using Camel’s AggregationStrategy

.

It’s a best practice to either use timeout = 0 or to assign a timeout value when using **pollEnrich** to avoid waiting indefinitely if no message arrives.

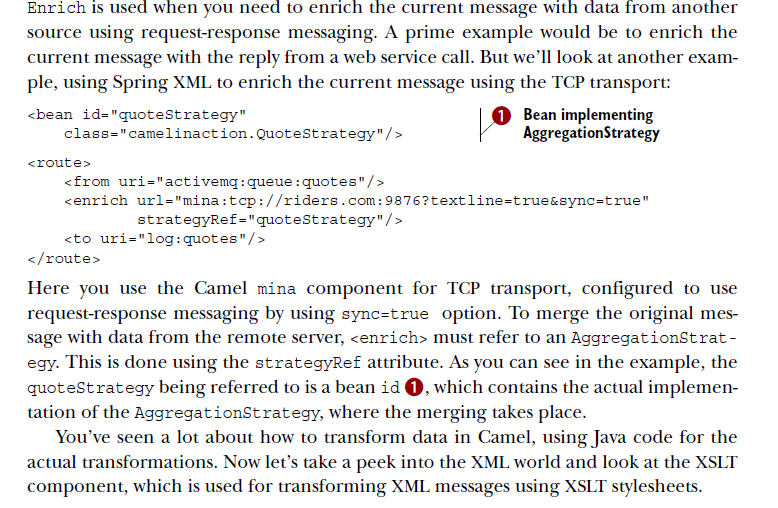
Enrich and **pollEnrich** can’t access information in the current exchange

Neither enrich nor **pollEnrich** can leverage any information from the current exchange.

This means, for example, that you can’t store a filename header on the exchange for **pollEnrich** to use to select a particular file. This may change in the future if the Camel team can find a solution.

**ENRICHING USING ENRICH**

Enrich is used when you need to enrich the current message with data from another source using request-response messaging. A prime example would be to enrich the current message with the reply from a web service call. But we’ll look at another example, using Spring XML to enrich the current message using the TCP transport:



### Transforming XML

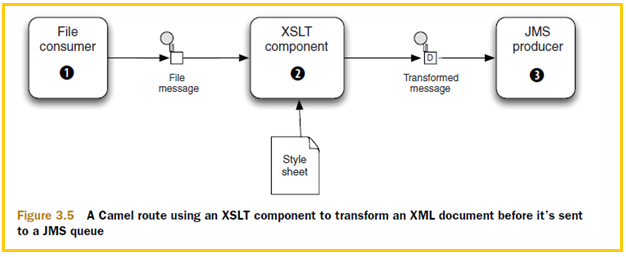
Camel provides two ways to perform XML transformations:

* *XSLT component*—for transforming an XML payload into another format using XSLT stylesheets
* *XML marshaling*—For marshaling and unmarshaling objects to and from XML

#### Transforming XML with XSLT

XSL Transformations (XSLT) is a declarative XML-based language used to transform XML documents into other documents. For example, XSLT can be used to transform XML into HTML for web pages or to transform an XML document into another XML document with a different structure. XSLT is powerful and versatile, but it’s also a complex language that takes time and effort to fully understand and master. Think twice before deciding to pick up and use XSLT.

Camel provides XSLT as a component in camel-spring.jar because it leverages Spring’s resource loading. This means greater flexibility in loading stylesheets because spring enables them to be loaded from various locations, such as the classpath, file paths, and over HTTP.



Using the XSLT component is straightforward because it’s just another Camel component.

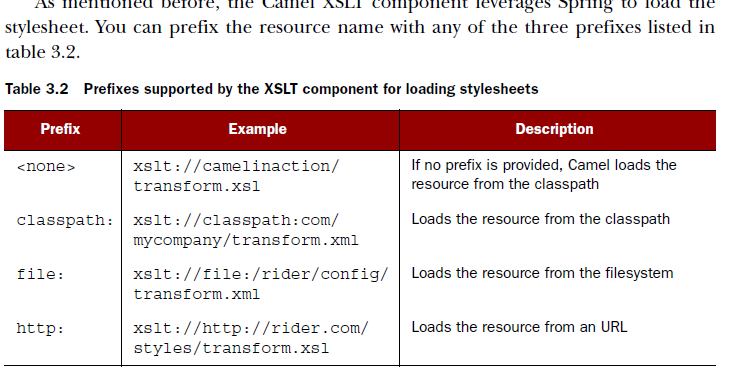
The following route shows an example of how you could use it; this route is also illustrated in figure 3.5.

from("file://rider/inbox")

.to("xslt://camelinaction/transform.xsl")

.to("activemq:queue:transformed")

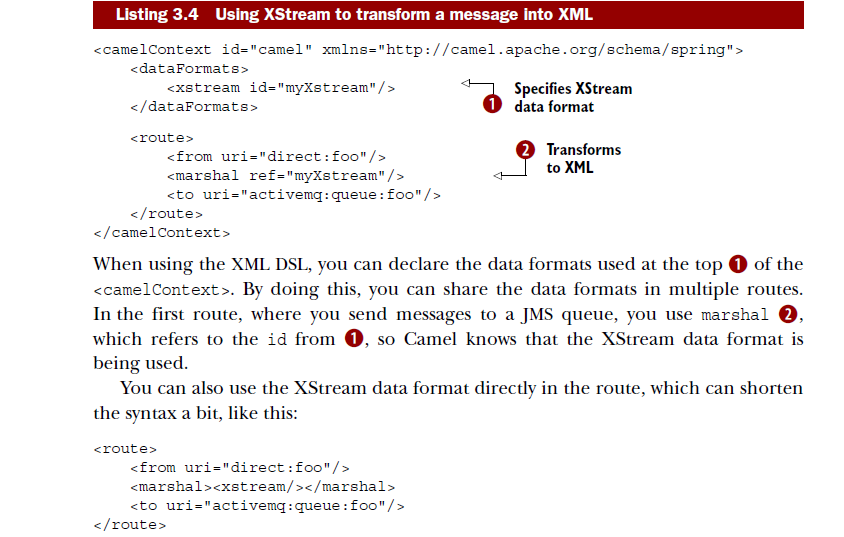
The file consumer picks up new files and routes them to the XSLT component, which transforms the payload using the stylesheet. After the transformation, the message is routed to a JMS producer, which sends the message to the JMS queue. Notice in the preceding code how the URL for the XSLT component is defined: xslt://camelinaction/transform.xsl. The part after the scheme is the URI location of the stylesheet to use. Camel will look in the classpath by default.



#### Transforming XML with object marshaling

**TRANSFORMING USING XSTREAM**

The classpath and the XStream library itself. Suppose you need to send messages in XML format to a shared JMS queue, which is then used to integrate two systems. Let’s look at how this can be done.



The same route is a bit shorter to write in the Java DSL, because you can do it with one line per route:

from("direct:foo").marshal().xstream().to("uri:activemq:queue:foo");

Yes, using XStream is that simple. And the reverse operation, unmarshaling from XML to an object, is just as simple:

<?xml version="1.0" encoding="UTF-8"?>

<route>

<from uri="activemq:queue:foo" />

<unmarshal ref="myXstream" />

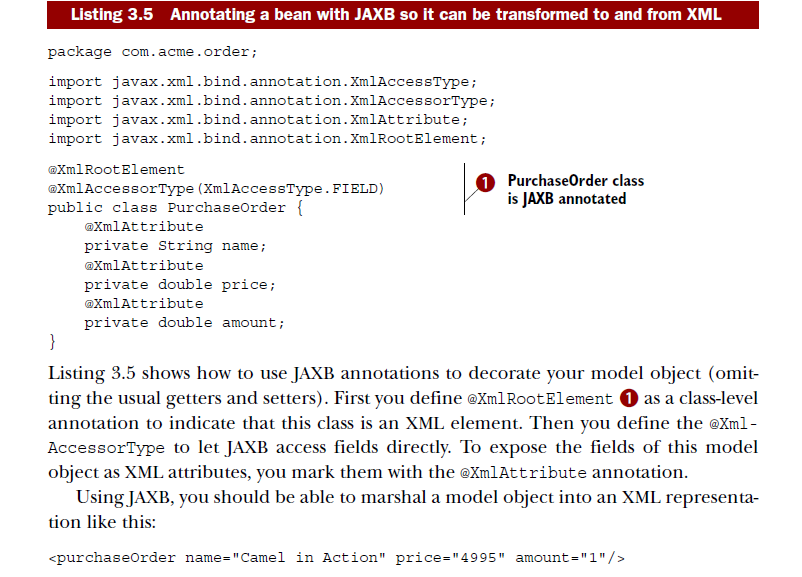
<to uri="direct:handleFoo" />

</route>

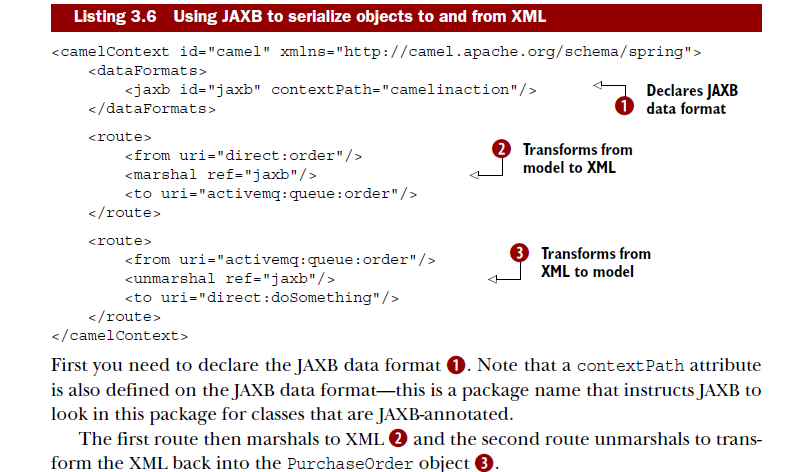
You’ve now seen how easy it is to use XStream with Camel. Let’s take a look at using JAXB with Camel.

**TRANSFORMING USING JAXB**

JAXB (Java Architecture for XML Binding) is a standard specification for XML binding, and it’s provided out of the box in the Java runtime. Like XStream, it allows you to serialize objects to XML and back again. It’s not as simple, but it does offer more bells and whistles for controlling the XML output. And because it’s distributed in Java, you don’t need any special JAR files on the classpath. Unlike XStream, JAXB requires that you do a bit of work to declare the binding between Java objects and the XML form. This is often done using annotations. Suppose you define a model bean to represent an order, as shown in listing 3.5, and you want to transform this into XML before sending it to a JMS queue. Then you want to transform it back to the order bean again when consuming from the JMS queue. This can be done as shown



Shows how you can use JAXB in routes to transform the PurchaseOrder object to XML before it’s sent to a JMS queue, and then back again from XML to the PurchaseOrder object when consuming from the same JMS queue.

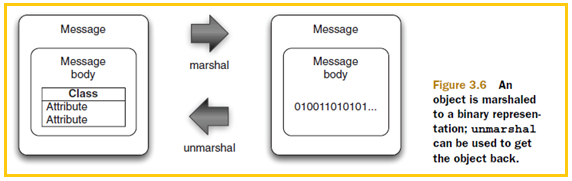


**NOTE:** To tell JAXB which classes are JAXB-annotated, you need to drop a special jaxb.index file into the context path. It’s a plain text file in which each line lists the class name. In the preceding example, the file contains a single line with the text PurchaseOrder.

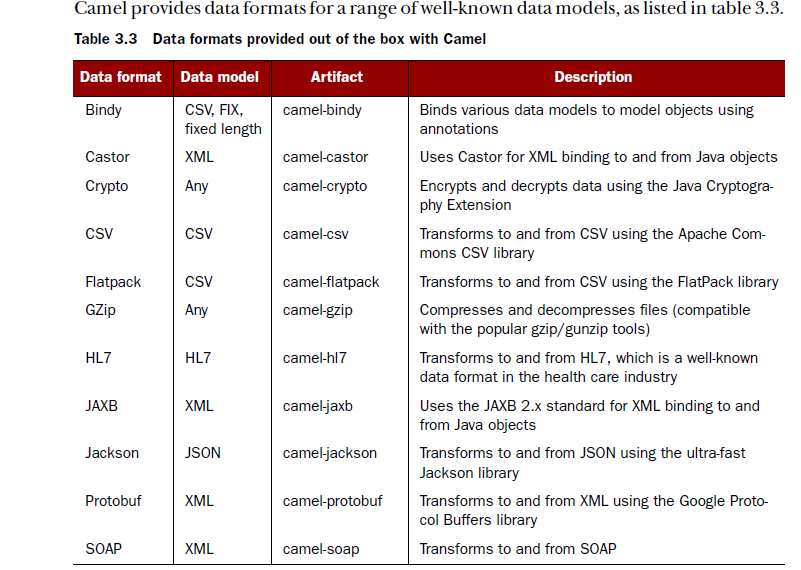
### Transforming with data formats

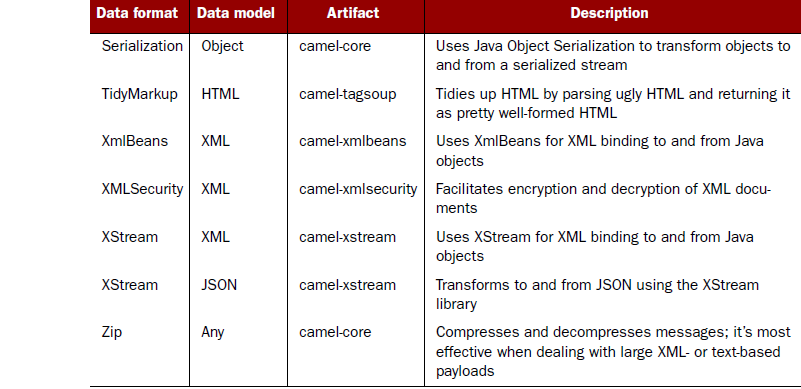
In Camel, data formats are pluggable transformers that can transform messages from one form to another and vice versa. Each data format is represented in Camel as an interface in **org.apace.camel.spi.DataFormat** containing two methods:

* marshal—For marshaling a message into another form, such as marshaling Java objects to XML, CSV, EDI, HL7, or other well-known data models
* unmarshal—For performing the reverse operation, which turns data from wellknown formats back into a message



#### Data formats provided with Camel





#### Using Camel’s CSV data format

The camel-csv data format is capable of transforming to and from CSV format. It leverages Apache Commons CSV to do the actual work.

Suppose you need to consume CSV files, split out each row, and send it to a JMS queue.

Sounds hard to do, but it’s possible with little effort in a Camel route:

from("file://rider/csvfiles")

.unmarshal().csv()

.split(body()).to("activemq:queue.csv.record");

All you have to do is unmarshal the CSV files, which will read the file line by line and store all lines in the message body as a **java.util.List**<List> type. Then you use the splitter to split up the body, which will break the **java.util.List**<List<String>> into rows (each row represented as another List<String> containing the fields) and send each row to the JMS queue. You may not want to send each row as a List type to the JMS queue, so you can transform the row before sending, perhaps using a processor

<?xml version="1.0" encoding="UTF-8"?>

<camelContext xmlns="http://camel.apache.org/schema/spring" id="camel">

<route>

<from uri="file://rider/csvfiles" />

<unmarshal>

<csv />

</unmarshal>

<split>

<simple>body</simple>

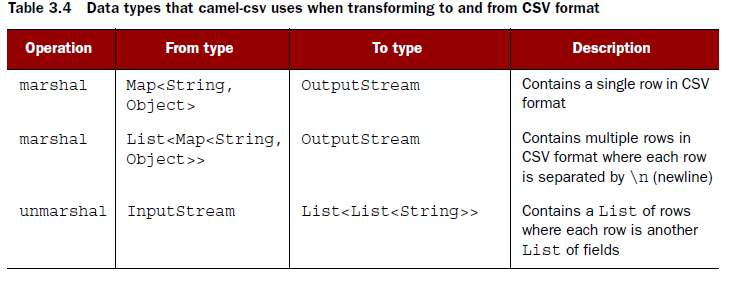
<to uri="activemq:queue.csv.record" />

</split>

</route>

</camelContext>

The noticeable difference is how you tell <split> that it should split up the message body. To do this you need to provide <split> with an Expression, which is what the splitter should iterate when it performs the splitting. To do so, you can use Camel’s built-in expression language called Simple



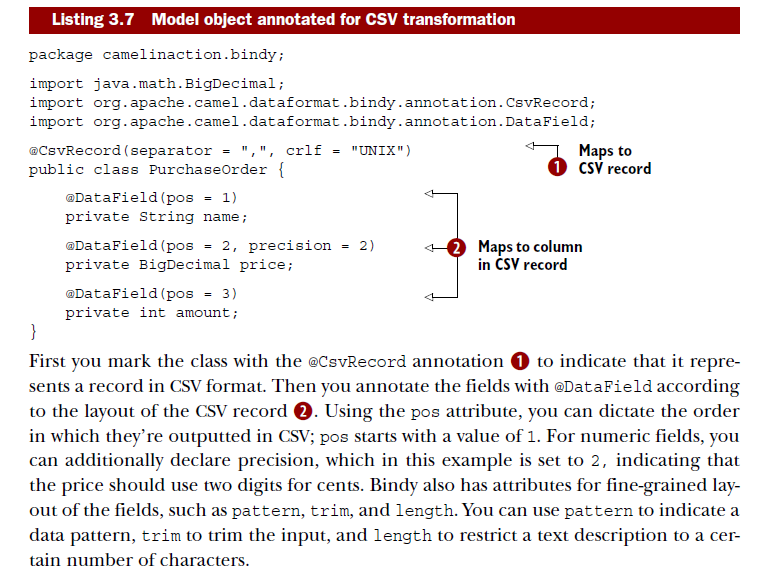
One problem with camel-csv is that it uses generic data types, such as Maps or Lists, to represent CSV records.

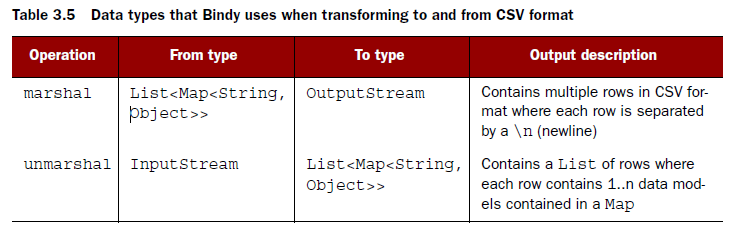
#### Using Camel’s Bindy data format

The two existing CSV-related data formats (camel-csv and camel-flatpack) are older libraries that don’t take advantage of the new features in Java 1.5, such as annotations and generics. In light of this deficiency, Charles Moulliard stepped up and wrote the camel-bindy component to take advantage of these new possibilities. It’s capable of binding CSV, FIX, and fixed-length formats to existing model objects using annotations.

This is similar to what JAXB does for XML.

Suppose you have a model object that represents a purchase order. By annotating the model object with camel-bindy annotations, you can easily transform messages between CSV and Java model objects





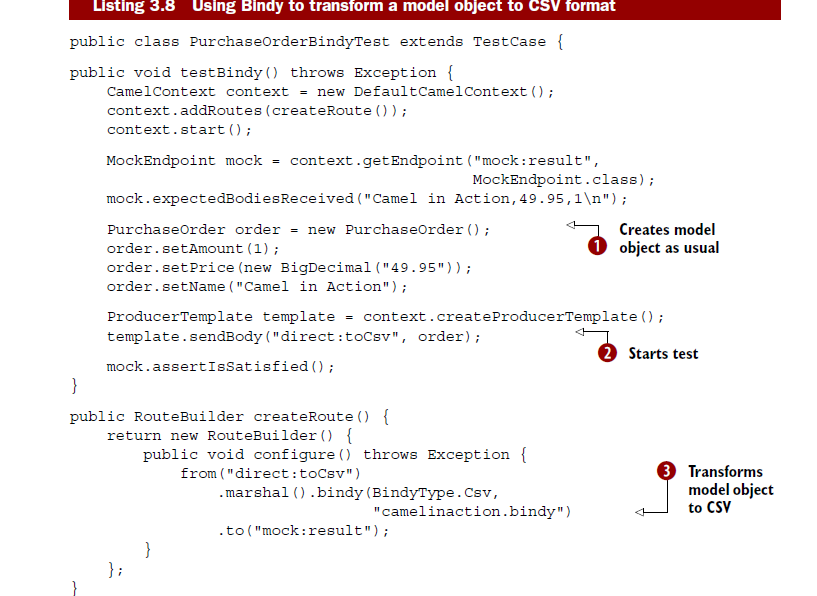
The important thing to notice in table 3.5 is that Bindy uses a Map<String, Object> to epresent a CSV row.

you can have multiple model objects with the CSV record being scattered across those objects.

For example, you could have fields 1 to 3 in one model object, fields 4 to 9 in another, and fields 10 to 12 in a third.

The map entry <String, Object> is distilled as follows:

* Map key (String)—must contain the fully qualified class name of the model object
* Map value (Object)—must contain the model object



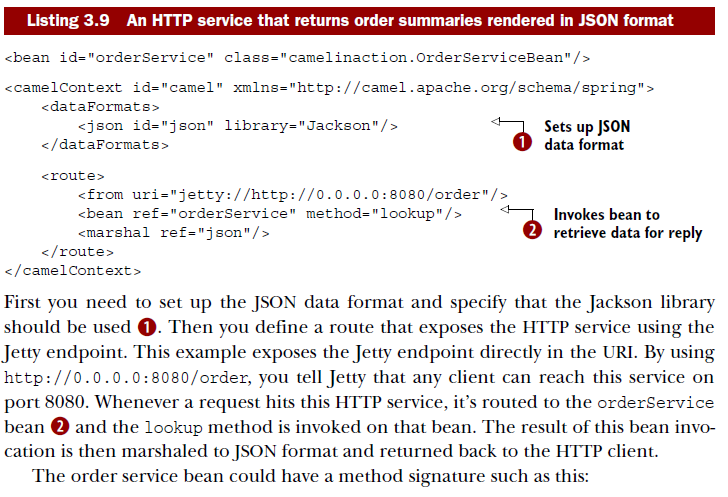
In this listing, you first create and populate the order model using regular Java setters 1. Then you send the order model to the route by sending it to the direct:toCsv endpoint 2 that is used in the route. The route will then marshal the order model to CSV using Bindy 3. Notice how Bindy is configured to use CSV mode via BindyType.Csv. To let Bindy know how to map to order model object, you need to provide a package name that will be scanned for classes annotated with Bindy annotations. This is the same solution JAXB offers.

**NOTE** Listing 3.8 uses MockEndpoint to easily test that the CSV record is as expected. Chapter 6 will covered testing with Camel, and you’ll learn all about using MockEndpoints.

CSV is only one of the well-known data formats that Bindy supports. Bindy is equally capable of working with fixed-length and FIX data formats, both of which follow the same principles as CSV.

#### Using Camel’s JSON data format

JSON (JavaScript Object Notation) is a data-interchange format, and Camel provides two components that support the JSON data format: camel-xstream and camel-jackson. In this section, we’ll focus on camel-jackson because Jackson is a very popular JSON library. Back at Rider Auto Parts, you now have to implement a new service that returns order summaries rendered in JSON format. Doing this with Camel is fairly easy, because Camel has all the ingredients needed to brew this service.



The order service bean could have a method signature such as this:

public PurchaseOrder lookup(@Header(name = "id") String id)

Notice that the service bean can return a POJO that the JSON library is capable of marshaling.

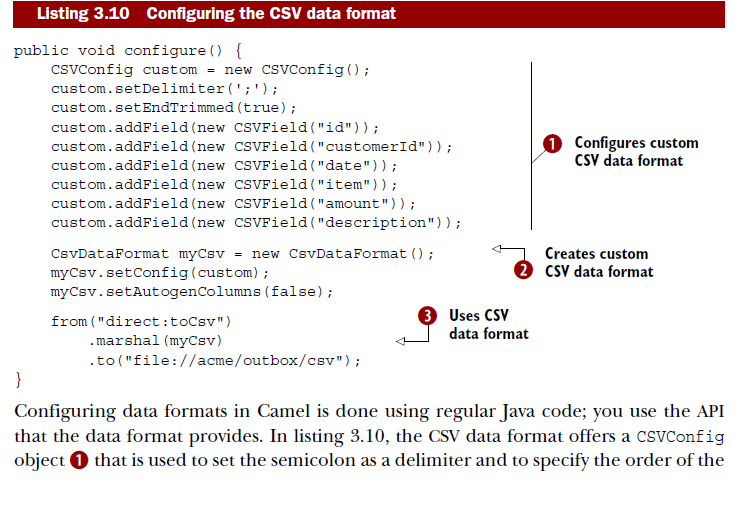
For example, suppose you used the PurchaseOrder and had JSON output as follows:

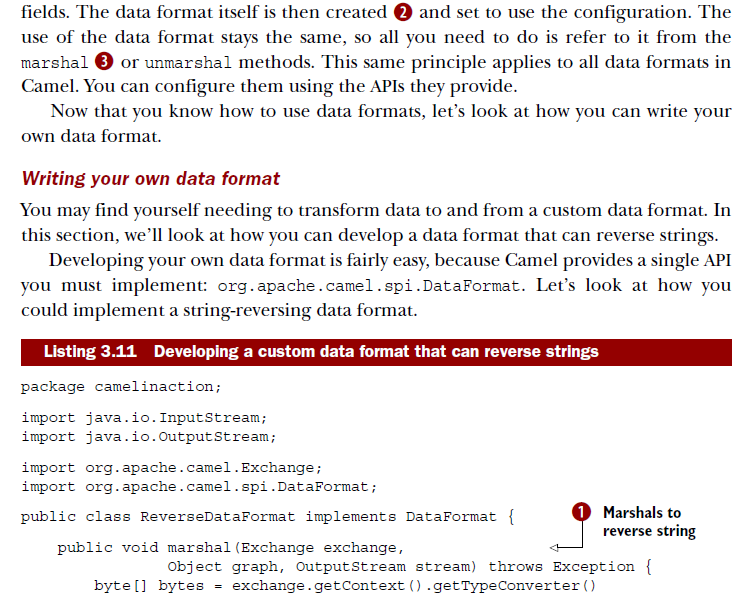
{"name":"Camel in Action","amount":1.0,"price":49.95}

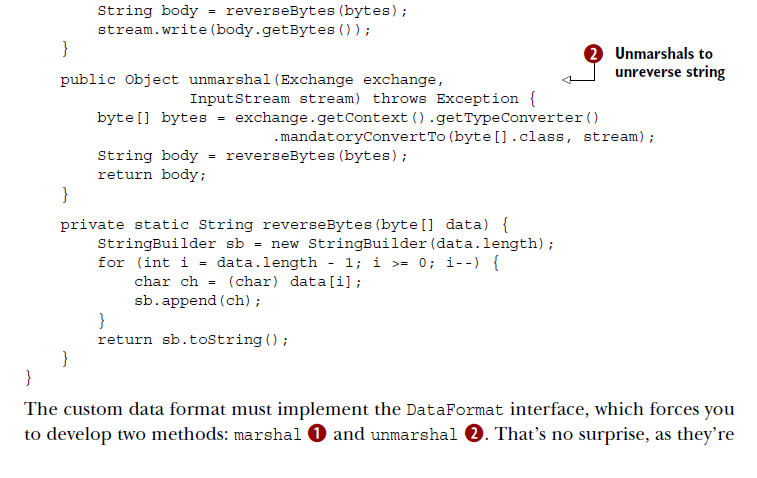
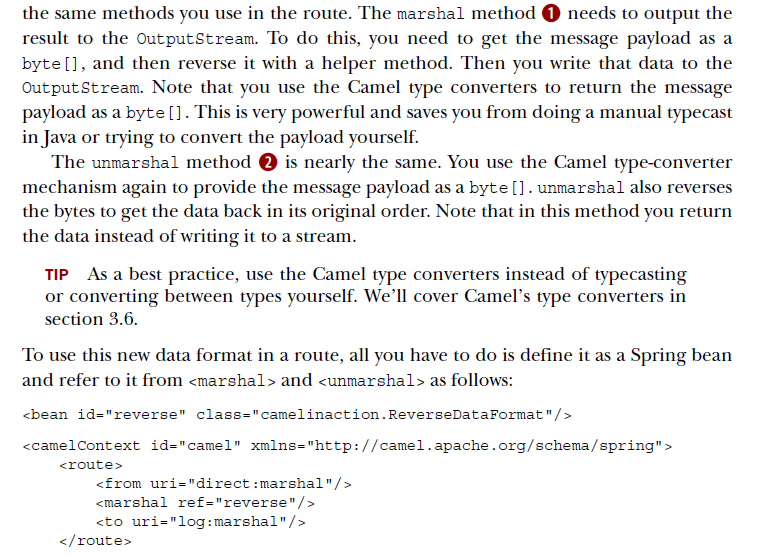
The HTTP service itself can be invoked by an HTTP Get request with the id of the order as a parameter: <http://0.0.0.0:8080/order/service?id=123>.

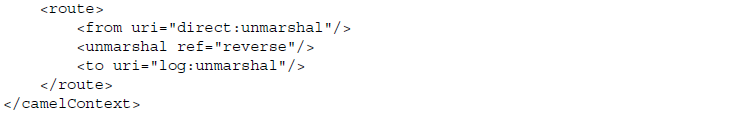
Notice how easy it is with Camel to bind the HTTP id parameter as the String id parameter with the help of the @Header annotation.

#### Configuring Camel data formats/Writing your own data format









### Transforming with templates

Camel provides slick integration with two different template languages:

* *Apache Velocity*—Probably the best known templating language (<http://camel.apache.org/velocity.html>)
* *FreeMarker*—Another popular templating language that may be a bit more advanced than Velocity (<http://camel.apache.org/freemarker.html>)

#### Using Apache Velocity

Rider Auto Parts has implemented a new order system that must send an email reply when a customer has submitted an order. Your job is to implement this feature.

The reply email could look like this:

Dear customer

Thank you for ordering X piece(s) of XXX at a cost of XXX.

This is an automated email, please do not reply.

There are three pieces of information in the email that must be replaced at runtime with real values. What you need to do is adjust the email to use the Velocity template language, and then place it into the source repository as src/test/resources/email.vm:

Dear customer

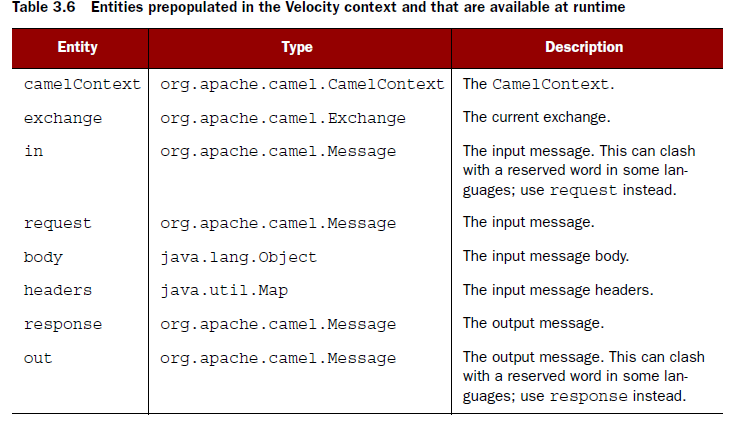
Thank you for ordering ${body.amount} piece(s) of ${body.name} at a cost of

${body.price}.

This is an automated email, please do not reply.

Notice that we’ve inserted ${ } placeholders in the template, which instructs Velocity to evaluate and replace them at runtime. Camel prepopulates the Velocity context with a number of entities that are then available to Velocity. Those entities are listed in table 3.6.

**NOTE** The entities in table 3.6 also apply for other templating languages, such as FreeMarker



Using Velocity in a Camel route is as simple as this:

from("direct:sendMail")

.setHeader("Subject", constant("Thanks for ordering"))

.setHeader("From", constant("donotreply@riders.com"))

.to("velocity://rider/mail.vm")

.to("smtp://mail.riders.com?user=camel&password=secret");

All you have to do is route the message to the Velocity endpoint that’s configured with the template you want to use, which is the rider/mail.vm file that’s loaded from the classpath by default. All the template components in Camel leverage the Spring resource loader, which allows you to load templates from the classpath, file paths, and

other such locations

### About Camel type converters.

Camel provides a built-in type-converter system that automatically converts between well-known types. This system allows Camel components to easily work together without having type mismatches. And from the Camel user’s perspective, type conversions are built into the API in many places without being invasive. For example, you used it

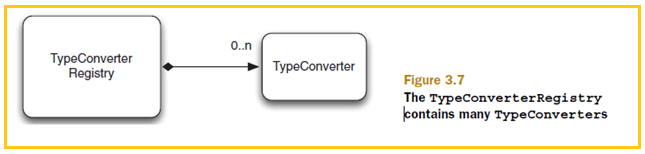
in listing 3.1:

String custom = exchange.getIn().getBody(String.class);

The getBody method is passed the type you want to have returned. Under the covers, the type-converter system converts the returned type to a String if needed.

#### How the Camel type-converter mechanism works

To understand the type-converter system, you first need to know what a type converter in Camel is. Figure 3.7 illustrates the relationship between the TypeConverterRegistry and the TypeConverters it holds



The TypeConverterRegistry is where all the type converters are registered when Camel is started. At runtime, Camel uses the TypeConverterRegistry’s lookup method to look up a suitable

TypeConverter: TypeConverter lookup(Class<?> toType, Class<?> fromType);

By using the TypeConverter, Camel can then convert one type to another using TypeConverter’s convertTo method, which is defined as follows:

<T> T convertTo(Class<T> type, Object value);

NOTE Camel implements about 150 or more type converters out of the box, which are capable of converting to and from the most commonly used types.

LOADING TYPE CONVERTERS INTO THE REGISTRY

On startup, Camel loads all the type converters into the TypeConverterRegistry by using a classpath-scanning solution. This allows Camel to pick up not only type converters from camel-core but also from any of the other Camel components, including your Camel applications.

To scan and load the type converters, Camel uses **org.apache.camel.impl.converter. AnnotationTypeConverterLoader**. To avoid scanning zillions of classes, it reads a service discovery file in the META-INF folder: META-INF/services/org/apache/camel/TypeConverter. This is a plain text file that has a list of packages that contain Camel type converters. The special file is needed to avoid scanning every possible JAR and all their packages, which would be time consuming. This special file tells Camel whether or not the JAR file contains type converters. For example, the file in camel core contains the following three entries:

org.apache.camel.converter

org.apache.camel.component.bean

org.apache.camel.component.file

The AnnotationTypeConverterLoader will scan those packages and their subpackages for classes that have been annotated with @Converter, and it searches within them for public methods that are annotated with @Converter. Each of those methods is considered a type converter.

This is best illustrated with an example. The following code is a snippet from IOConverter class from camel-core JAR:

@Converter

public final class IOConverter {

@Converter

public static InputStream toInputStream(URL url) throws IOException {

return url.openStream();

}

}

Camel will go over each method annotated with @Converter and look at the method signature.

The first parameter is the from type, and the return type is the to type—in this

example you have a TypeConverter that can convert from a URL to an InputStream. By

doing this, Camel loads all the built-in type converters, including those from the Camel components in use.

Now that you know how the Camel type converters are loaded, let’s look at using them.

#### Using Camel type converters

As we mentioned, the Camel type converters are used throughout Camel, often automatically.

But you might want to use them to force a specific type to be used in a route,

such as before sending data back to a caller or a JMS destination. Let’s look at how to

do that.

Suppose you need to route files to a JMS queue using javax.jmx.TextMessage. To

do so, you can convert each file to a String, which forces the JMS component to use

TextMessage. This is easy to do in Camel—you use the convertBodyTo method, as

shown here:

from("file://riders/inbox")

.convertBodyTo(String.class)

.to("activemq:queue:inbox");

If you’re using Spring XML, you provide the type as an attribute instead, like this:

<?xml version="1.0" encoding="UTF-8"?>

<route>

<from uri="file://riders/inbox" />

<convertBodyTo type="java.lang.String" />

<to uri="activemq:queue:inbox" />

</route>

You can omit the java.lang. prefix on the type, which can shorten the syntax a bit:

<convertBodyTo type="String"/>.

Another reason for using convertBodyTo is to read files using a fixed encoding

such as UTF-8. This is done by passing in the encoding as the second parameter:

from("file://riders/inbox")

.convertBodyTo(String.class, "UTF-8")

.to("activemq:queue:inbox");

**TIP** If you have trouble with a route because of the payload or its type, try

using .convertBodyTo(String.class) at the start of the route to convert to a

String type, which is a well-supported type. If the payload cannot be converted

to the desired type, a NoTypeConversionAvailableException exception

is thrown.

#### Writing your own type converter

Suppose you wanted to write a custom type converter that can convert a byte[] into a PurchaseOrder model object

