3

Transforming data with Camel

This chapter covers

- Transforming data by using EIPs and Java
- Transforming XML data
- Transforming by using well-known data formats
- Writing your own data formats for transformations
- Understanding the Camel type-converter mechanism

The preceding chapter covered routing, which is the single most important feature any integration kit must provide. This chapter looks at the second most important feature: data or message transformation.

Just like the real world, where people speak different languages, the IT world speaks different protocols. Software engineers regularly need to act as mediators between various protocols when IT systems must be integrated. To address this, the data models used by the protocols must be transformed from one form to another, adapting to whatever protocol the receiver understands. Mediation and data transformation are key features in any integration kit, including Camel.

In this chapter, you'll learn all about how Camel can help you with your data transformation challenges. We'll start with a brief overview of data transformation in Camel and then look at transforming data into any custom format you may have. Next we'll look at Camel components that are specialized for transforming XML data and other well-known data formats. We end the chapter by looking into Camel's type-converter mechanism, which supports, implicitly and explicitly, type conversion.

After reading this chapter, you'll know how to tackle any data transformation you're faced with and which Camel solution to use.

3.1 Data transformation overview

Camel provides many techniques for data transformation, and we'll cover them shortly. But let's start with an overview of data transformation in Camel. *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, `java.lang.String` is transformed into `javax.jms.TextMessage`.

Figure 3.1 illustrates the principle of transforming a message body from one form into another. This transformation can involve any combination of format and type transformations. In most cases, the data transformation you'll face with Camel is format transformation: you have to mediate between two protocols. 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.

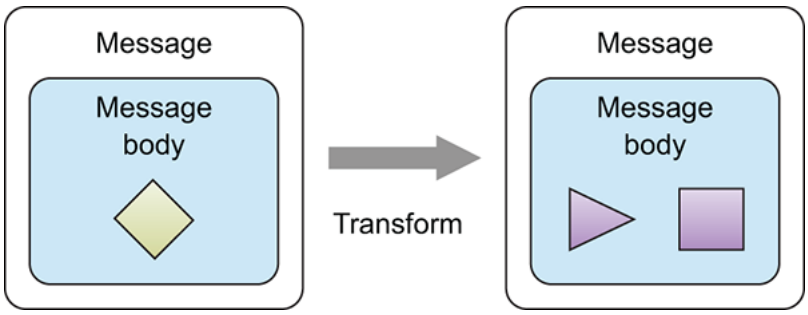

Figure 3.1 Camel offers many features for transforming data from one form to another.

Camel has many data-transformation features. We introduce them in the following section, and then present them one by one. After reading this chapter, you'll have a solid understanding of how to use Camel to transform your data. In Camel, data transformation typically takes place in the six ways listed in table 3.1.

Table 3.1 Six ways data transformation typically takes place in Camel

Transformation	Description
Data transformation using EIPs and Java	You can explicitly enforce transformation in the route by using the Message Translator or the Content Enricher EIPs. This gives you the power to do data mapping by using regular Java code. We cover this in section 3.2.
Data transformation using components	Camel provides a range of components for transformation, such as the XSLT component for XML transformation. We dive into this in section 3.3.
Data transformation using data formats	Data formats are Camel transformers that come in pairs to transform data back and forth between well-known formats. Section 3.4 covers this topic.
Data transformation using templates	Camel provides a range of components for transforming by using templates, such as Apache Velocity. We'll look at this in section 3.5.
Data type transformation using Camel's type-converter mechanism	Camel has an elaborate type-converter mechanism that activates on demand. This is convenient when you need to convert from common types such as `java.lang.Integer` to `java.lang.String` or even from `java.io.File` to `java.lang.String`. Section 3.6 covers type converters.
Message transformation in component adapters	Camel's many components adapt to various commonly used protocols and, as such, need to be able to transform messages as they travel to and from those protocols. Often these components use a combination of

Transformation	Description
	custom data transformations and type converters. This happens seamlessly, and only component writers need to worry about it. Chapter 8 covers writing custom components.

This chapter covers the first five of these data transformation methods. We'll leave the last one for chapter 8 because it applies only to writing custom components.

3.2 Transforming data by using EIPs and Java

Data mapping, the process of mapping between two distinct data models, is a key factor in data integration. There are many existing standards for data models, governed by various organizations or committees. As such, you'll often find yourself needing to map from a company's custom data model to a standard data model.

Camel provides great freedom in data mapping because it allows you to use Java code. You aren't limited to using a particular data-mapping tool that may at first seem elegant but turns out to make things impossible.

In this section, you'll look at mapping data by using `Processor`, a Camel API. Camel can also use Java beans for mapping, which is a good practice because it allows your mapping logic to be independent of the Camel API.

3.2.1 Using the Message Translator EIP

The Message Translator EIP is illustrated in figure 3.2.

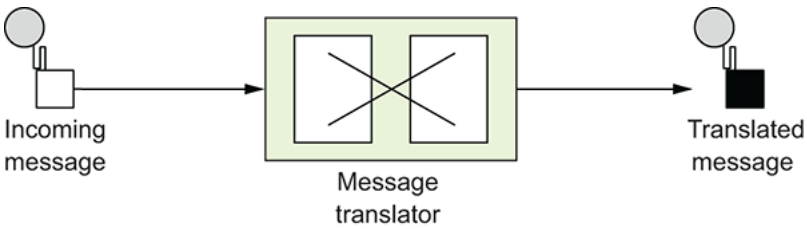

Figure 3.2 In the Message Translator EIP, an incoming message goes through a translator and comes out as a translated message.

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.

NOTE The Gang of Four book is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley Professional, 1994). See the "Design Patterns" Wikipedia article for more information: http://en.wikipedia.org/wiki/Design_Patterns_(book).

Camel provides three ways of using this pattern:

- Using `Processor`
- Using Java beans
- Using `<transform>`

We'll look at them each in turn.

Transforming using Processor

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

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

`Processor` is a low-level API in which you work directly on the Camel `Exchange` instance. It gives you full access to all of Camel's moving parts from the `CamelContext`, which you can obtain from the `Exchange` by using the `getContext` method.

Let's look at an example. At Rider Auto Parts, you've been asked to generate daily reports of newly received orders to be outputted to a CSV file. The company uses a custom format for order entries, but to make things easy, they already have an HTTP service that returns a list of orders for whatever date you input. The challenge you face is mapping the returned data from the HTTP service to a CSV format and writing the report to a file.

Because you want to get started on a prototype quickly, you decide to use the Camel `Processor`, as shown in the following listing.

Listing 3.1 Using `Processor` to translate from a custom format to a CSV format

```
import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class OrderToCsvProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        String custom = exchange.getIn()
                .getBody(String.class);  (1)
```

(1) Gets custom payload

```
        String id = custom.substring(0, 10);  (2)
```

(2) Extracts data to local variables

```
        String customerId = custom.substring(10, 20);  (2)
        String date = custom.substring(20, 30);  (2)
        String items = custom.substring(30);  (2)
        String[] itemIds = items.split("@");  (2)
        StringBuilder csv = new StringBuilder();  (3)
```

(3) Maps to CSV format

```
        csv.append(id.trim());  (3)
        csv.append(",").append(date.trim());  (3)
        csv.append(",").append(customerId.trim());  (3)
        for (String item : itemIds) {  (3)
            csv.append(",").append(item.trim());  (3)
        }  (3)
        exchange.getIn().setBody(csv.toString());  (4)
```

(4) Replaces payload with CSV payload

```
    }
}
```

First you grab the custom format payload from the `exchange` ❶. It's a `String` type, so you pass `String` in as the parameter to have the payload returned as a string. Then you extract data from the custom format to the local variables ❷. The custom format could be anything, but in this example, it's a fixed-length custom format. Then you map the CSV format by building a string with comma-separated values ❸. Finally, you replace the custom payload with your new CSV payload ❹.

You can use `OrderToCsvProcessor` from listing 3.1 in a Camel route as follows:

```
from("quartz2://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 XML is as follows:

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

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="quartz2://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>
```

You can try this example yourself; we've provided a little unit test with the book's source code. Go to the chapter3/transform directory and run

these Maven goals:

```
mvn test -Dtest=OrderToCsvProcessorTest
mvn test -Dtest=SpringOrderToCsvProcessorTest
```

After the test runs, a report file is written in the target/orders/received directory.

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.

In two scenarios, the Camel end user will have to decide which method to use:

- 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 by using `getIn`, and not to use `getOut` at all. This is the practice we use most often in this book.

Using a processor has one disadvantage: you're required to use the Camel API. In the next section, you'll learn 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 following listing shows using a bean instead of `Processor`.

Listing 3.2 Using a bean to translate from a custom format to CSV format

```
public class OrderToCsvBean {
    public static String map(String custom) {
    String id = custom.substring(0, 10);    (1)
```

(1) Extracts data to local variables

```
    String customerId = custom.substring(10, 20);    (1)
    String date = custom.substring(20, 30);    (1)
    String items = custom.substring(30);    (1)
    String[] itemIds = items.split("@");    (1)
    StringBuilder csv = new StringBuilder();    (1)
        csv.append(id.trim());
        csv.append(",").append(date.trim());
        csv.append(",").append(customerId.trim());
        for (String item : itemIds) {
            csv.append(",").append(item.trim());
        }
    return csv.toString();    (2)
```

(2) Returns CSV payload

```
    }
}
```

The first noticeable difference between listings 3.1 and 3.2 is that listing 3.2 doesn't use any Camel imports. Your bean is totally independent of the Camel API. The next difference is that you can name the method signature in listing 3.2—in this case, it's a static method named `map`.

The method signature defines the contract, which means that the first parameter `(String custom)` is the message body you're going to use for translation. The method returns a string, which means the translated data

will be a `String` type. At runtime, Camel binds to this method signature. We won't go into any more details here; chapter 4 covers much more about using beans.

The mapping ❶ is the same as with the processor. At the end, you return the mapping output ❷.

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

```
from("quartz2://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 XML is as follows:

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

<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="quartz2://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>
```

You can try this example from the chapter3/transform directory by using the following Maven goals:

```
mvn test -Dtest=OrderToCsvBeanTest
mvn test -Dtest=SpringOrderToCsvBeanTest
```

This generates a test report file in the target/orders/received directory.

Another advantage of using beans over processors for mappings is that unit testing is much easier. For example, listing 3.2 doesn't require the use of Camel at all, as opposed to listing 3.1, where you need to create and pass in an `Exchange` instance.

We'll leave the beans for now, because they're covered extensively in the next chapter. But you should keep in mind that beans are 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. Let's look at a little example.

Suppose you need to prepare text for HTML formatting by replacing all line breaks with a `<br/>` tag. You can do this 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 the Builder pattern to build compound expressions from individual expressions. This is done by chaining together method calls, which is the essence of the Builder pattern.

NOTE For more information on the Builder pattern, see the Wikipedia article: http://en.wikipedia.org/wiki/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.

You can run this example from chapter3/transform directly by using the following Maven goal:

```
mvn test -Dtest=TransformTest
```

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 allows connectivity only 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.

For more information on the Direct component and other types of in-memory messaging, see chapter 6.

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've 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");
```

As you can see, this code uses an inlined Camel `Expression` that allows you to use Java code in its `evaluate` method. This follows the same principle as the Camel `Processor` you saw before.

Now let's see how to transform data using the XML DSL.

Transforming using <transform> from the XML DSL

Using `<transform>` from the XML DSL is a bit different from the Java DSL because the XML DSL isn't as powerful. In the XML DSL, the Builder pattern expressions aren't available because with XML you don't have a

real programming language underneath. What you can do instead is invoke a method on a bean or use scripting languages.

Let's see how this works. The following route uses a method call on a bean as the expression:

```
<bean id="htmlBean" class="camelinaction.HtmlBean"/>    ❶
```

❶ Does the transformation

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <transform>
          <method bean="htmlBean" method="toHtml"/>    ❷
```

❷ Invokes toHtml method on bean

```
        </transform>
        <to uri="mock:result"/>
    </route>
</camelContext>
```

First, you declare a regular Spring bean to be used to transform the message ❶. Then, in the route, you use `<transform>` with a `<method>` call expression to invoke the bean ❷.

The implementation of the `htmlBean` is straightforward:

```
public class HtmlBean {
    public static String toHtml(String body) {
        body = body.replaceAll("\n", "<br/>");
        body = "<body>" + body + "</body>";
        return body;
```

```
    }
}
```

You can also use scripting languages as expressions in Camel. For example, you can use Groovy, MVFLEX Expression Language (MVEL), JavaScript, or Camel's own scripting language, called Simple (explained in appendix A). We won't go into detail on how to use the other scripting languages at this point, but you can use the Simple language to build strings with placeholders. It pretty much speaks for itself—we're sure you'll understand what the following transformation does:

```
<transform>
    <simple>Hello ${body} how are you?</simple>
</transform>
```

You can try the XML DSL transformation examples provided in the book's source code by running the following Maven goals from the chapter3/transform directory:

```
mvn test -Dtest=SpringTransformMethodTest
mvn test -Dtest=SpringTransformScriptTest
```

We're done covering the Message Translator EIP, so let's look at the related Content Enricher EIP.

3.2.2 Using the Content Enricher EIP

The Content Enricher EIP is illustrated in figure 3.3. This pattern documents the scenario in which a message is enriched with data obtained from another resource.

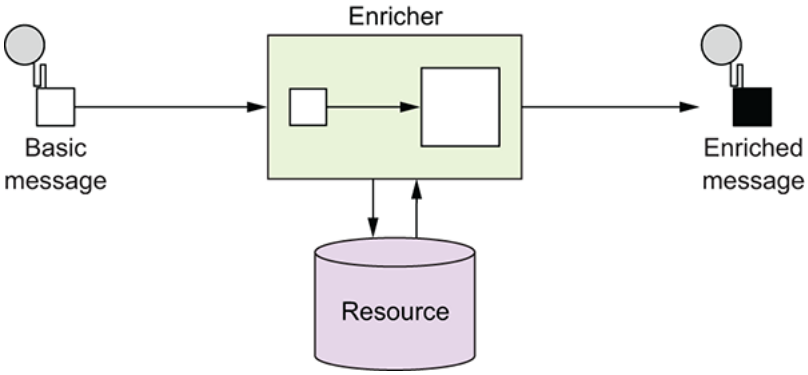

Figure 3.3 In the Content Enricher EIP, an existing message has data added to it from another source.

To help understand this pattern, let's turn back to Rider Auto Parts. It turns out that the data mapping you did in listing 3.1 wasn't sufficient. Orders are also piled up on an FTP server, and your job is to somehow merge this information into the existing report. Figure 3.4 illustrates the scenario.

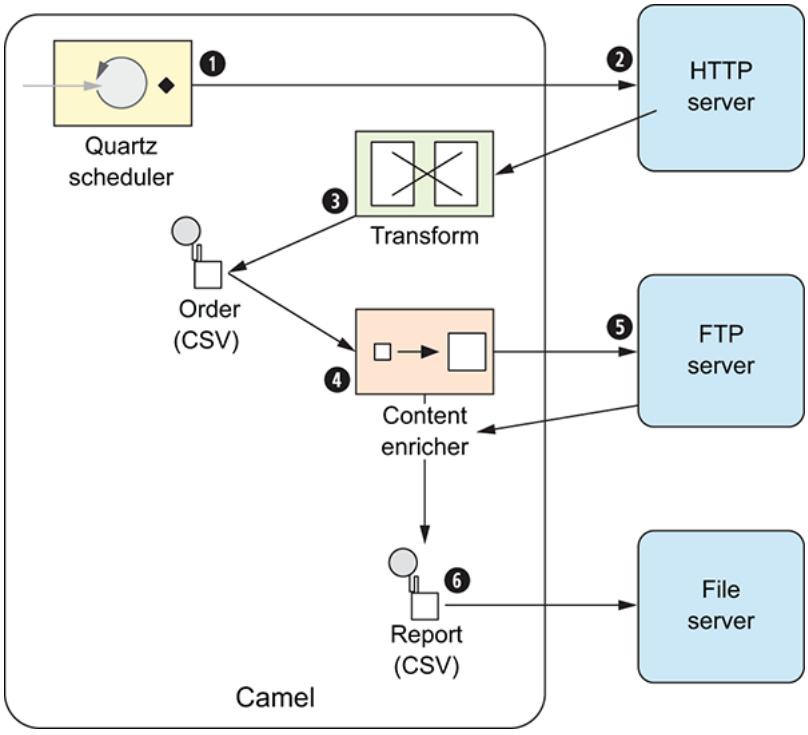

Figure 3.4 An overview of the route that generates the orders report, now with the content enricher pulling in data from an FTP server

A scheduled consumer using Quartz starts the route every day at 6 a.m. ❶. It then pulls data from an HTTP server, which returns orders in a custom format ❷, which is then transformed into CSV format ❸. At this point, you have to perform the additional content enrichment step ❹ with the data obtained from the FTP server ❺. After this, the final report is written to the file server ❻.

Before you dig into the code and see how to implement this, you need to take a step back and look at how the Content Enricher EIP is implemented in Camel. Camel provides two methods in the DSL for implementing the pattern:

- `pollEnrich`—This method merges data retrieved from another source by using a consumer.
- `enrich`—This method merges data retrieved from another source by 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 uses 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 works only 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.aggregate.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. Your task is to enrich the message by using Java code and return the merged result. Let's see this in action.

To solve the problem at Rider Auto Parts, you need to use `pollEnrich` because it's capable of polling a file from an FTP server.

Enriching using pollEnrich

The following listing shows how to use `pollEnrich` to retrieve the additional orders from the remote FTP server and aggregate this data with the existing message by using Camel's `AggregationStrategy`.

Listing 3.3 Using `pollEnrich` to merge additional data with an existing message

```
from("quartz2://report?cron=0+0+6+*+*+?")
    .to("http://riders.com/orders/cmd=received&date=yesterday")
    .process(new OrderToCsvProcessor())
    .pollEnrich("ftp://riders.com/orders/?username=rider&password=secret
        new AggregationStrategy() {     (1)

(1) Uses pollEnrich to read FTP file

        public Exchange aggregate(Exchange oldExchange,
                                  Exchange newExchange) {
            if (newExchange == null) {
                return oldExchange;
            }
        String http = oldExchange.getIn()     (2)

(2) Merges data using AggregationStrategy

                .getBody(String.class);     (2)
        String ftp = newExchange.getIn()     (2)
                .getBody(String.class);     (2)
        String body = http + "\n" + ftp;     (2)
        oldExchange.getIn().setBody(body);     (2)
            return oldExchange;
        }
    })
    .to("file://riders/orders");     (3)
```

❸

Writes output to file

The route is triggered by Quartz to run at 6 a.m. every day. You invoke the HTTP service to retrieve the orders and transform them to CSV format by using a processor.

At this point, you need to enrich the existing data with the orders from the remote FTP server. This is done by using `pollEnrich` ❶, which consumes the remote file.

To merge the data, you use `AggregationStrategy` ❷. First, you check whether any data was consumed. If `newExchange` is `null`, there's no remote file to consume, and you just return the existing data. If there's a remote file, you merge the data by concatenating the existing data with the new data and setting it back on the `oldExchange`. Then, you return the merged data by returning the `oldExchange`. To write the CSV report file, you use the `file` component ❸.

TIP Both `enrich` and `pollEnrich` can accept dynamic URIs, as discussed in chapter 2, section 2.5.1.

`PollEnrich` uses a polling consumer to retrieve messages, and it offers three time-out modes:

- `pollEnrich(timeout = -1)`—Polls the message and waits until a message arrives. This mode blocks until a message exists.
- `pollEnrich(timeout = 0)`—Immediately polls the message if any exists; otherwise, `null` is returned. It never waits for messages to arrive, so this mode never blocks. This is the default mode.
- `pollEnrich(timeout > 0)`—Polls the message, and if no message exists, it waits for one, waiting at most until the time-out triggers. This mode potentially blocks.

It's a best practice to either use `timeout = 0` or assign a time-out value when using `pollEnrich` to avoid waiting indefinitely if no message

arrives.

Now let's take a quick look at how to use `enrich` with the XML DSL; it's a bit different from using the Java DSL. You use `enrich` when you need to enrich the current message with data from another source using request-reply messaging. A prime example is to enrich the current message with the reply from a web service call. But let's look at another example, using XML to enrich the current message via the TCP transport:

```
<bean id="quoteStrategy"
    class="camelinaction.QuoteStrategy"/>    ❶
```

❶ Bean implementing AggregationStrategy

```
<route>
    <from uri="jms:queue:quotes"/>
    <enrich url="netty4:tcp://riders.com:9876?textline=true&amp;sync=tru
            strategyRef="quoteStrategy"/>
    <to uri="log:quotes"/>
</route>
```

Here you use the Camel `netty4` component for the TCP transport, configured to use request-reply messaging by using the `sync=true` option. To merge the original message with data from the remote server, `<enrich>` must refer to an `AggregationStrategy`. This is done using the `strategyRef` attribute. As you can see in the example, the `quoteStrategy` being referred to is a `bean` `id` ❶, which contains the implementation of the `AggregationStrategy`, where the merging takes place.

You've seen a lot about how to transform data in Camel, using Java code for the transformations. Now let's take a peek into the XML world and look at the XSLT component, which is used for transforming XML messages into another format by using XSLT stylesheets.

3.3 Transforming XML

Camel provides two ways to perform XML transformations:

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

Both of these are covered in the following subsections.

3.3.1 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 the XSLT component as part of camel-core.jar, so you don't need any other dependencies. Using the XSLT component is straightforward because it's just another Camel component. The following route shows an example of how to use it; this route is also illustrated in figure 3.5:

```
from("file://rider/inbox")
    .to("xslt://camelinaction/transform.xsl")
    .to("jms:queue:transformed")
```

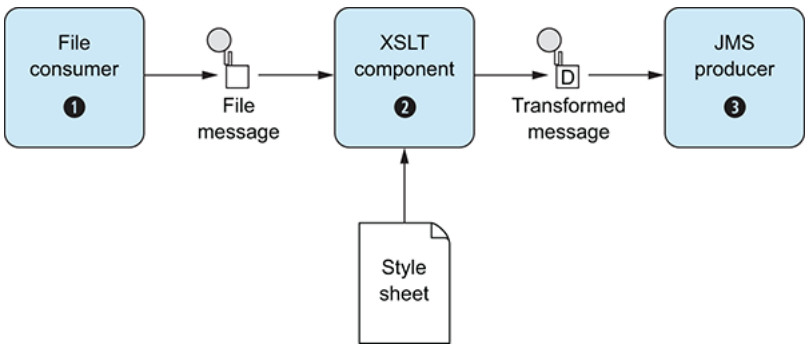

Figure 3.5 A Camel route using an XSLT component to transform an XML document before it's sent to a JMS queue

The file consumer picks up new files and routes them to the XSLT component, which transforms the payload by 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 URI 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. To look elsewhere, you can prefix the resource name with any of the prefixes listed in table 3.2.

Table 3.2 Prefixes supported by the XSLT component for loading stylesheets

Prefix	Example	Description
`<none>`	`xslt://camelinaction/transform.xsl`	If no prefix is provided, Camel loads the resource from the classpath.
`file:`	`xslt://file:/rider/config/transform.xml`	Loads the resource from the filesystem.
`http:`	`xslt://http://rider.com/styles/transform.xsl`	Loads the resource from a URL.
`ref:`	`xslt://ref:resourceId`	Look up the resource from the registry.
`bean:`	`xslt://bean:nameOfBean.methodName`	Look up a bean in the registry and call a method which returns the resource.

Let's leave the XSLT world now and take a look at how to do XML-to-object marshaling with Camel.

3.3.2 Transforming XML with object marshaling

Any software engineer who has worked with XML knows that it's a challenge to use the low-level XML API that Java offers. Instead, people often prefer to work with regular Java objects and use marshaling to transform between Java objects and XML representations.

In Camel, this marshaling process is provided in ready-to-use components known as *data formats*. Section 3.4 covers data formats in full detail, but you'll take a quick look at the XStream and JAXB data formats here as we cover XML transformations using marshaling.

Transforming using XStream

XStream is a simple library for serializing objects to XML and back again. To use it, you need camel-xstream.jar on 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. The following listing shows how this can be done.

Listing 3.4 Using XStream to transform a message into XML

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
    <xstream id="myXstream"/>    ❶
```

❶ Specifies XStream data format

```
    </dataFormats>
    <route>
        <from uri="direct:foo"/>
    <marshal ref="myXstream"/>    ❷
```

❷ Transforms to XML

```
        <to uri="jms:queue:foo"/>
    </route>
</camelContext>
```

When using the XML DSL, you can declare the data formats used at the top ❶ of the `<camelContext>`. By doing this, you can share the data formats in multiple routes. In the first route, where you send messages to a JMS queue, you use `marshal` ❷, which refers to the `id` from ❶, so Camel knows that the XStream data format is being used.

You can also use the XStream data format directly in the route, which can shorten the syntax a bit, like this:

```
<route>
    <from uri="direct:foo"/>
    <marshal><xstream/></marshal>
    <to uri="jms:queue:foo"/>
</route>
```

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

```
from("direct:foo").marshal().xstream().to("jms:queue:foo");
```

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

```
<route>
    <from uri="jms: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

Java Architecture for XML Binding (JAXB) 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 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 in listings 3.5 and 3.6.

Listing 3.5 Annotating a bean with JAXB so it can be transformed to and from XML

```
package camelinaction;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement     ❶
```

❶ PurchaseOrder class is JAXB annotated

```
@XmlAccessorType(XmlAccessType.FIELD)     ❶
public class PurchaseOrder {     ❶
    @XmlAttribute
    private String name;
    @XmlAttribute
    private double price;
    @XmlAttribute
    private double amount;
}
```

Listing 3.5 shows how to use JAXB annotations to decorate your model object (omitting the usual getters and setters). First you define `@XmlRootElement` ❶ as a class-level annotation to indicate that this class is an XML element. Then you define the `@XmlAccessorType` to let JAXB access fields directly. To expose the fields of this model object as XML attributes, you mark them with the `@XmlAttribute` annotation.

Using JAXB, you should be able to marshal a model object into an XML representation like this:

```
<purchaseOrder name="Camel in Action" price="6999" amount="1"/>
```

The following listing 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.

Listing 3.6 Using JAXB to serialize objects to and from XML

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
    <jaxb id="jaxb" contextPath="camelinaction"/>  ❶
```

❶ Declares JAXB data format

```
    </dataFormats>
    <route>
        <from uri="direct:order"/>
    <marshal ref="jaxb"/>  ❷
```

❷ Transforms from model to XML

```
        <to uri="jms:queue:order"/>
    </route>
    <route>
```

```
        <from uri="jms:queue:order"/>
    <unmarshal ref="jaxb"/>    ❸
```

❸ Transforms from XML to model

```
        <to uri="direct:doSomething"/>
    </route>
</camelContext>
```

First you need to declare the JAXB data format ❶. Note that a `contextPath` attribute is also defined on the JAXB data format; this is a package name that instructs JAXB to look in this package for classes that are JAXB annotated. The first route then marshals to XML ❷, and the second route unmarshals to transform the XML back into the `PurchaseOrder` object ❸.

You can try this example by running the following Maven goal from the chapter3/order directory:

```
mvn test -Dtest=PurchaseOrderJaxbTest
```

NOTE To tell JAXB which classes are JAXB annotated, you need to drop a special jaxb.index file into each package in the classpath containing the POJO classes. 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`.

That's the basis of using XML object marshaling with XStream and JAXB. Both are implemented in Camel via data formats that are capable of transforming back and forth between various well-known formats.

3.4 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.apache.camel.spi.DataFormat` containing two methods:

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

You may already have realized that these two functions are opposites; one is capable of reversing what the other has done, as illustrated in figure 3.6.

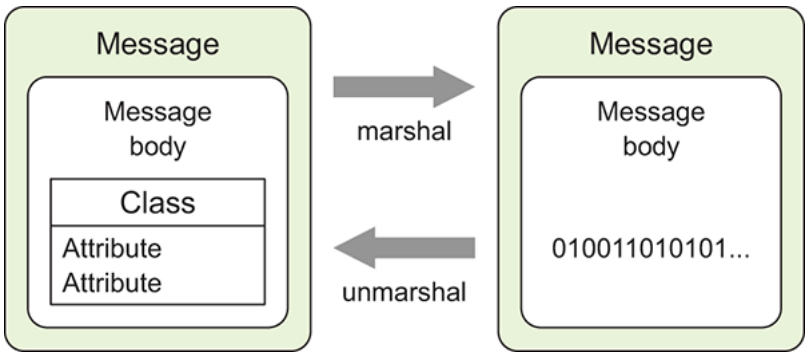

Figure 3.6 An object is marshaled to a binary representation; `unmarshal` can be used to get the object back.

We touched on data formats in section 3.3, where we covered XML transformations. This section covers data formats in more depth and using data types other than XML, such as CSV and JSON. We'll even look at how to create your own data formats. We'll start our journey by briefly looking at the data formats Camel provides out of the box.

3.4.1 Data formats provided with Camel

Camel provides data formats for a range of well-known data models, some of which are listed in table 3.3.

Table 3.3 Selection of data formats provided out of the box with Camel

Data format	Data model	Artifact	Description
Avro	Binary Avro format	camel-avro	Supports serializing and deserializing messages by using Apache Avro
Base64	Base64 string	camel-base64	Can encode and decode into a base64 string
Bindy	CSV, FIX, fixed length	camel-bindy	Binds various data models to model objects by using annotations
Crypto	Any	camel-crypto	Encrypts and decrypts data by using the Java Cryptography Extension
CSV	CSV	camel-csv	Transforms to and from CSV by using the Apache Commons CSV library
GSon	JSON	camel-gson	Transforms to and from JSON by using the Google GSON library
GZip	Any	camel-gzip	Compresses and decompresses files (compatible with the popular gzip/gunzip tools)

Data format	Data model	Artifact	Description
HL7	HL7	camel-hl7	Transforms to and from HL7, which is a well-known data format in the health-care industry
JAXB	XML	camel-jaxb	Uses the JAXB 2.x standard for XML binding to and from Java objects
Jackson	JSON	camel-jackson	Transforms to and from JSON by using the ultra-fast Jackson library
PGP	Any	camel-crypto	Encrypts and decrypts data by using PGP
Protobuf	XML	camel-protobuf	Transforms to and from XML by using the Google Protocol Buffers library
SOAP	XML	camel-soap	Transforms to and from SOAP
Serialization	Object	camel-core	Uses Java Object Serialization to transform objects to and from a serialized stream
Syslog	RFC3164, RFC5424	camel-syslog	Transforms between RFC3164/RFC5424

Data format	Data model	Artifact	Description
			messages and SyslogMessage model objects
XMLSecurity	XML	camel-xmlsecurity	Facilitates encryption and decryption of XML documents
XStream	XML	camel-xstream	Uses XStream for XML binding to and from Java objects
XStream	JSON	camel-xstream	Transforms to and from JSON by using the XStream library
Zip	Any	camel-core	Compresses and decompresses messages, and is most effective when dealing with large XML- or text-based payloads
Zip file	Zip file	camel-zipfile	Compresses and decompresses zip files

Camel provides more than 40 data formats out of the box. You can read more about these data formats at the Camel website (http://camel.apache.org/data-format.html). We've picked three to cover in the following section. They're among the most commonly used, and what you learn about those will also apply to the remainder of the data formats.

3.4.2 Using Camel's CSV data format

The camel-csv data format is capable of transforming to and from CSV format. It uses Apache Commons CSV to do the 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("jms: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.

The same example in XML is a bit different, as shown here:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="file://rider/csvfiles"/>
        <unmarshal><csv/></unmarshal>
        <split>
            <simple>body</simple>
            <to uri="jms:queue:csv.record"/>
        </split>
    </route>
</camelContext>
```

The noticeable difference is in the way 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 (see appendix A), which knows how to do that.

NOTE The Splitter EIP is fully covered in chapter 5.

This example is in the source code for the book, in the chapter3/order directory. You can try the examples by running the following Maven goals:

```
mvn test -Dtest=PurchaseOrderCsvTest
mvn test -Dtest=PurchaseOrderCsvSpringTest
```

At first, the data types that the CSV data format uses may seem confusing. They're listed in table 3.4.

Table 3.4 Data types that camel-csv uses when transforming to and from CSV format

Operation	From type	To type	Description
`marshal`	`Map<String,Object>`	`OutputStream`	Contains a single row in CSV format.
`marshal`	`List<Map<String,Object>>`	`OutputStream`	Contains multiple rows in CSV format; each row is separated by `\n` (newline).
`unmarshal`	`InputStream`	`List<List<String>>`	Contains a `List` of rows; each row is another `List` of fields.

One problem with camel-csv is that it uses generic data types, such as `Map` or `List`, to represent CSV records. Often you'll already have model objects to represent your data in memory. Let's look at using model objects with the camel-bindy component.

3.4.3 Using Camel's Bindy data format

Two of the 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 by 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, as shown in the following listing.

Listing 3.7 Model object annotated for CSV transformation

```
package camelinaction.bindy;

import java.math.BigDecimal;
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator = ",", crlf = "UNIX")    (1)
```

(1) Maps to CSV record

```
public class PurchaseOrder {
    @DataField(pos = 1)    (2)
```

(2) Maps to column in CSV record

```
    private String name;
  @DataField(pos = 2, precision = 2)    ❷
    private BigDecimal price;
  @DataField(pos = 3)    ❷
    private int amount;
}
```

First you mark the class with the `@CsvRecord` annotation ❶ to indicate that it represents a record in CSV format. Then you annotate the fields with `@DataField` according to the layout of the CSV record ❷. Using the `pos` attribute, you can dictate the order in which they're output in CSV; `pos` starts with a value of `1`. For numeric fields, you can additionally declare precision, which in this example is set to `2`, indicating that the price should use two digits for cents. Bindy also has attributes for fine-grained layout of the fields, such as `pattern`, `trim`, and `length`. You can use `pattern` to indicate a data pattern, `trim` to trim the input, and `length` to restrict a text description to a certain number of characters.

Before you look at how to use Bindy in Camel routes, the data types Bindy expects to use are listed in table 3.5.

Table 3.5 Data types that Bindy uses when transforming to and from CSV format

Operation	From type	To type	Output description
`marshal`	`List<Map<String, Object>>`	`OutputStream`	Contains multiple rows in CSV format; each row is separated by `\n` (newline).
`unmarshal`	`InputStream`	`List<Map<String, Object>>`	Contains a `List` of rows; each row contains 1 ... n data models contained in a `Map`.

The important thing to notice in table 3.5 is that Bindy uses `Map<String, Object>` to represent a CSV row. At first, this may seem odd. Why doesn't it use a single model object for that? The answer is that 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

If this seems confusing, don't worry. The following listing should make it clearer.

Listing 3.8 Using Bindy to transform a model object to CSV format

```
public void testBindy() throws Exception {
    CamelContext context = new DefaultCamelContext();
    context.addRoutes(createRoute());
    context.start();
    MockEndpoint mock = context.getEndpoint("mock:result",
                                            MockEndpoint.class);
    mock.expectedBodiesReceived("Camel in Action,69.99,1\n");
    PurchaseOrder order = new PurchaseOrder();    (1)
```

(1) Creates model object as usual

```
    order.setAmount(1);
    order.setPrice(new BigDecimal("69.99"));
    order.setName("Camel in Action");
    ProducerTemplate template = context.createProducerTemplate();
    template.sendBody("direct:toCsv", order);    (2)
```

(2) Starts test

```
    mock.assertIsSatisfied();
}

public RouteBuilder createRoute() {
    return new RouteBuilder() {
        public void configure() throws Exception {
            from("direct:toCsv")
            .marshal().bindy(BindyType.Csv,    (3)
```

(3) Transforms model object to CSV

```
                    camelinaction.bindy.PurchaseOrder.class)    (3)
                .to("mock:result");
```

```
            }
        };
    }
```

In listing 3.8, you first create and populate the order model by using regular Java setters ❶. Then you send the order model to the route by sending it to the `direct:toCsv` endpoint ❷ used in the route. The route will then marshal the order model to CSV by using Bindy ❸. Notice that Bindy is configured to use CSV mode via `BindyType.Csv`. To let Bindy know how to map the order model object, you need to provide a class annotated with Bindy annotations, as in listing 3.7.

NOTE Listing 3.8 uses `MockEndpoint` to easily test that the CSV record is as expected. Chapter 9 covers testing with Camel, and you'll learn all about using `MockEndpoint`.

You can try this example from the chapter3/order directory by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderBindyTest
```

The source code for the book also contains a *reverse* example of how to use Bindy to transform a CSV record into a Java object. You can try it by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderUnmarshalBindyTest
```

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.

It's now time to leave CSV and look at a more modern format: JSON.

3.4.4 Using Camel's JSON data format

JavaScript Object Notation (*JSON*) is a data-interchange format, and Camel provides six components that support the JSON data format: camel-xstream, camel-gson, camel-jackson, camel-boon, camel-fastjson, camel-

johnzon. This section focuses on camel-jackson because Jackson is a 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 following listing shows how to ramp up a prototype.

Listing 3.9 An HTTP service that returns order summaries rendered in JSON format

```
<bean id="orderService" class="camelinaction.OrderServiceBean"/>

<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
      <json id="json" library="Jackson"/>    ❶
```

❶ Sets up JSON data format

```
    </dataFormats>
    <route>
        <from uri="jetty://http://0.0.0.0:8080/order"/>
      <bean ref="orderService" method="lookup"/>    ❷
```

❷ Invokes bean to retrieve data for reply

```
        <marshal ref="json"/>
    </route>
</camelContext>
```

First you need to set up the JSON data format and specify that the Jackson library should be used ❶. Then you define a route that exposes the HTTP service using the Jetty endpoint. This example exposes the Jetty endpoint directly in the URI. By using `http://0.0.0.0:8080/order`, you tell Jetty that any client can reach this service on port 8080. Whenever a request

hits this HTTP service, it's routed to the `orderService` bean ❷, and the `lookup` method is invoked on that bean. The result of this bean invocation is then marshaled to JSON format and returned to the HTTP client.

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

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

This signature allows you to implement the lookup logic as you wish. You'll learn more about the `@Header` annotation in chapter 4, when we cover how bean parameter binding works in Camel.

Notice that the service bean can return a POJO that the JSON library is capable of marshaling. For example, suppose you used the `PurchaseOrder` from listing 3.7 and had JSON output as follows:

```
{"name":"Camel in Action","amount":1.0,"price":69.99}
```

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.

You can try this example yourself from the chapter3/order directory by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderJSONTest
```

So far, we've used data formats with their default settings. But what if you need to configure the data format, for example, to use another splitter character with the CSV data format? That's the topic of the next section.

3.4.5 Configuring Camel data formats

In section 3.4.2, you used the CSV data format, but this data format offers many additional settings. The following listing shows how to configure the CSV data format.

Listing 3.10 Configuring the CSV data format

```
public void configure() {
    CsvDataFormat myCsv = new CsvDataFormat()
    .setDelimiter(';')    ❶
```

❶ Creates and configures a custom CSV data format

```
    .setHeader(new String[]{    ❶
    "id", "customerId", "date", "item", "amount", "description"});    ❶

    from("direct:toCsv")
    .marshal(myCsv)    ❷
```

❷ Uses CSV data format

```
        .to("file://acme/outbox/csv");
}
```

Configuring data formats in Camel is typically done directly on `DataFormat` itself; sometimes you may also need to use the API that the third-party library under the hood provides. In listing 3.10, the CSV data format nicely wraps the third-party API so you can just configure the `DataFormat` directly ❶. Here you set the semicolon as a delimiter and specify the order of the fields ❶. The use of the data format stays the same, so all you need to do is refer to it from the `marshal` ❷ or `unmarshal` methods. This same principle applies to all data formats in Camel.

TIP You can learn how to create your own data format in chapter 8, section 8.5.

You've learned all about data formats, and now it's time to say goodbye to data formats and take a look at using templating with Camel for data

transformation. Templating is extremely useful when you need to generate automatic reply emails.

3.5 Transforming with templates

Camel provides slick integration with two template languages:

- *Apache Velocity*—Probably the best-known templating language (http://camel.apache.org/velocity.html)
- *Apache FreeMarker*—Another great templating language from Apache (http://camel.apache.org/freemarker.html)

These two templating languages are fairly similar to use, so we discuss only Velocity here.

3.5.1 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.
```

Three pieces of information in the email must be replaced at runtime with real values. You need to 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
This is an automated email, please do not reply.
```

Notice that you insert `${ }` placeholders in the template, which instructs Velocity to evaluate and replace them at runtime. Camel prepopulates the Velocity context with numerous entities that are then available to Velocity. Those entities are listed in table 3.6.

NOTE The entities in table 3.6 also apply to other templating languages, such as FreeMarker.

Table 3.6 Entities that are prepopulated in the Velocity context and available at runtime

Entity	Type	Description
`camelContext`	`org.apache.camel.CamelContext`	The `CamelContext`.
`exchange`	`org.apache.camel.Exchange`	The current exchange.
`in`	`org.apache.camel.Message`	The input message. This can clash with a reserved word in some languages; use `request` instead.
`request`	`org.apache.camel.Message`	The input message.
`body`	`java.lang.Object`	The input message body.
`headers`	`java.util.Map`	The input message headers.
`response`	`org.apache.camel.Message`	The output message.
`out`	`org.apache.camel.Message`	The output message. This can clash with a reserved word in some

Entity	Type	Description
		languages; use `response` instead.

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 use the same resource loader, which allows you to load templates from the classpath, file paths, and other such locations. You can use the same prefixes listed in table 3.2.

You can try this example by going to the chapter3/order directory in the book's source code and running the following Maven goal:

```
mvn test -Dtest=PurchaseOrderVelocityTest
```

We'll now leave data transformation and look at type conversion. Camel has a powerful type-converter mechanism that removes all need for boilerplate type-converter code.

3.6 Understanding 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.

In this section, you'll take a look at the insides of the type-converter system. We'll explain how Camel scans the classpath on startup to register type converters dynamically. We'll also show how to use it from a Camel route, and how to build your own type converters.

3.6.1 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 `TypeConverterRegistry` and the `TypeConverter`s it holds.

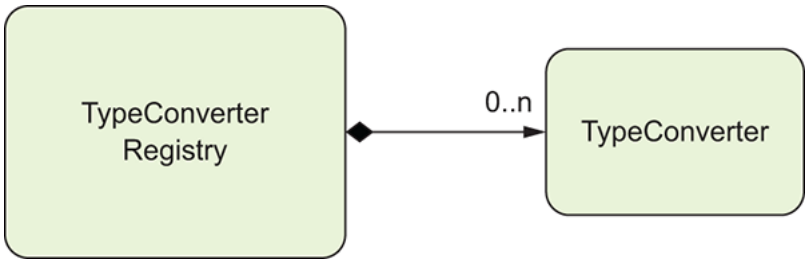

Figure 3.7 The `TypeConverterRegistry` contains many `TypeConverter`s

`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 `TypeConverter`, Camel can then convert one type to another by using `TypeConverter`'s `convertTo` method, which is defined as follows:

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

NOTE Camel implements about 350 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 type converters not only from camel-core, but also from any of the other Camel components, including your Camel applications. You'll see this in section 3.6.3 when you build your own type converter.

Camel uses `org.apache.camel.impl.converter.AnnotationTypeConverterLoader` to scan and load the type converters. 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 fully qualified class names and 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 the JAR file contains type converters. For example, the file in camel-cxf contains the following entries:

```
org.apache.camel.component.cxf.converter.CxfConverter
org.apache.camel.component.cxf.converter.CxfPayloadConverter
```

`AnnotationTypeConverterLoader` loads those classes that have been annotated with `@Converter`, and then searches within them for public methods that are annotated with `@Converter`. Each of those methods is considered a type converter. Yes, the class `@Converter` annotation is a bit of overkill when we've already defined the class name in the TypeConvertertext file. We need this because we can also specify package names, which could include many classes. For example, a package name of `org.apache.camel.component.cxf.converter` also could have been provided in the TypeConverter text file and would have included

`CxfConverter` and `CxfPayloadConverter`. Using the fully qualified class name is preferred, though, because Camel loads them more quickly.

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

```
@Converter
public final class IOConverter {
    @Converter
    public static InputStream toInputStream(URL url) throws IOException
        return IOHelper.buffered(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.

TIP Type converters can also be loaded into the registry manually. This is often useful if you need to quickly add a type converter into your application or want full control over when it will be loaded. You can find more information in the online documentation (http://camel.apache.org/type-converter.html).

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

3.6.2 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 by using `javax.jms.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("jms:queue:inbox");
```

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

```
<route>
    <from uri="file://riders/inbox"/>
    <convertBodyTo type="java.lang.String"/>
    <to uri="jms:queue:inbox"/>
</route>
```

You can omit the `java.lang.` prefix on the type, which can shorten the syntax: `<convertBodyTo type="String"/>`.

Another reason for using `convertBodyTo` is to read files by 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("jms: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 can't be converted to the desired type, a `NoTypeConversionAvailableException` exception is thrown.

That's all there is to using type converters in Camel routes. Before we wrap up this chapter, though, let's take a look at how to write your own type converter.

3.6.3 Writing your own type converter

Writing your own type converter is easy in Camel. You already saw what a type converter looks like in section 3.6.1, when you looked at how type converters work.

Suppose you want to write a custom type converter that can convert a `byte[]` into a `PurchaseOrder` model object (an object you used in listing 3.7). As you saw earlier, you need to create an `@Converter` class containing the type-converter method, as shown in the following listing.

Listing 3.11 A custom type converter to convert from `byte[]` to `PurchaseOrder` type

```
@Converter
public final class PurchaseOrderConverter
    @Converter
    public static PurchaseOrder toPurchaseOrder(byte[] data,
                                                Exchange exchange) {
        TypeConverter converter = exchange.getContext()
                .getTypeConverter();   (1)
```

(1) Grabs TypeConverter to reuse

```
        String s = converter.convertTo(String.class, data);
        if (s == null || s.length() < 30) {
            throw new IllegalArgumentException("data is invalid");
        }
    s = s.replaceAll("##START##", "");   (2)
```

(2) Converts from String to PurchaseOrder

```
    s = s.replaceAll("##END##", "");   (2)
    String name = s.substring(0, 9).trim();   (2)

    String s2 = s.substring(10, 19).trim();   (2)
```

```
        BigDecimal price = new BigDecimal(s2);    ❷

        price.setScale(2);    ❷
        String s3 = s.substring(20).trim();    ❷
        Integer amount = converter    ❷
                .convertTo(Integer.class, s3);    ❷
        return new PurchaseOrder(name, price, amount);    ❷
        }
}
```

The `Exchange` gives you access to the `CamelContext` and thus to the parent `TypeConverter` ❶, which you use in this method to convert between strings and numbers. The rest of the code is the logic for parsing the custom protocol and returning the `PurchaseOrder` ❷. Notice that you can use `converter` to easily convert between well-known types.

All you need to do now is add the service discovery file, named TypeConverter, in the META-INF directory. As explained previously, this file contains the fully qualified name of the `@Converter` class.

If you `cat` the TypeConverter file, you'll see this:

```
$ cat src/main/resources/META-INF/services/org/apache/camel/TypeConverte
camelinaction.PurchaseOrderConverter
```

This example can be found in the chapter3/converter directory of the book's source code, which you can try by using the following Maven goal:

```
mvn test -Dtest=PurchaseOrderConverterTest
```

Returning null values

By default, a `null` return value from a type converter isn't valid. Camel considers `null` as a "miss" and adds the pair of types you're trying to convert to a blacklist so they won't be tried again. For example, if our previous example returned `null`, the conversion from `byte[]` to `PurchaseOrder` would be blacklisted. If `null` is a valid return value for your conversion, you can force Camel to accept it by using the `allowNull` option on the `@Converter` annotation. For example, if the example in listing 3.11 required a `null` return value, you could do something like this:

```
@Converter(allowNull = true)
public static PurchaseOrder toPurchaseOrder(byte[] data,
                                            Exchange exchange) {
...
```

Adding type converters to camel-core

If you're on track to becoming a star Camel rider and want to write a shiny new type converter for the camel-core module, you may notice that type-converter loading is handled differently there. The META-INF/services/org/apache/camel/TypeConverter file specifies the `org.apache.camel.core` package, which doesn't exist. It's just a dummy package name. The type converters for camel-core are specified directly in `org.apache.camel.impl.converter.CorePackageScanClassResolver`, and you can add them there. And that completes this chapter on transforming data with Camel.

3.7 Summary and best practices

Data transformation is the cornerstone of any integration kit; it bridges the gap between various data types and formats. It's also essential in today's industry, because more and more disparate systems need to be integrated to support the ever-changing businesses and world we live in.

This chapter covered many of the possibilities Camel offers for data transformation. You learned how to format messages by using EIPs and beans. You also learned that Camel provides special support for transforming XML documents by using XSLT components and XML-capable data formats. Camel provides data formats for well-known data models, which you learned to use, and it even allows you to build your own data formats. We also took a look into the templating world, which can be used to format data in specialized cases, such as generating email bodies. Finally, we looked at how the Camel type-converter mechanism works and learned that it's used internally to help all the Camel components work together. You learned how to use it in routes and how to write your own converters.

Here are a few key tips you should take away from this chapter:

- *Data transformation is often required*—Integrating IT systems often requires you to use different data formats when exchanging data. Camel can act as the mediator and has strong support for transforming data in any way possible. Use the various features in Camel to aid with your transformation needs.
- *Java is powerful*—Using Java code isn't a worse solution than using a fancy mapping tool. Don't underestimate the power of the Java language. Even if it takes 50 lines of grunt boilerplate code to get the job done, you have a solution that can easily be maintained by fellow engineers.
- *Prefer to use beans over processors*—If you're using Java code for data transformation, you can use beans or processors. Processors are more dependent on the Camel API, whereas beans allow loose coupling. Chapter 4 covers how to use beans.

This chapter, along with chapter 2, covered two crucial features of integration kits: routing and transformation. The next chapter dives into the world of Java beans, and you'll see how Camel can easily adapt to and use your existing beans. This allows a higher degree of reuse and loose coupling, so you can keep your business and integration logic clean and apart from Camel and other middleware APIs.