- Common middleware services, such as databases, message-oriented middleware, and web servers

- Popular enterprise information systems, such as SAP/R3 and SalesForce

- Popular web sites and services, such as Amazon and Twitter • Apache Karaf, an OSGi container

- JBoss Enterprise Application Platform (EAP), a certified Enterprise Java (Java EE) application server

- Spring Boot, a popular framework for developing microservices

INTRODUCING APACHE CAMEL

The core component that enables agile integration in Red Hat Fuse is *Apache Camel*, an open source library that implements enterprise integration patterns (EIP).

Enterprise Integration Patterns (EIP) are proven solutions to recurring integration problems. EIPs describe design considerations and common issues, and are brought to life using messaging technologies. EIPs establish a technology-independent vocabulary and a visual notation to design and document integration solutions.

Besides ready-to-use implementations of most EIPs, the Apache Camel community provides connectors to:

Apache Camel also provides facilities for data transformation, supporting popular data formats such as XML, JSON, CSV, and HL7.

While Camel provides a lot of the heavy lifting with respect to integration development, Red Hat Fuse provides the reliability, performance, scalability, security, and high availability for integration solutions. Further, Red Hat Fuse subscriptions provides a certified set of Camel libraries and components, together with enterprise-level support with strict service-level agreements.

INTRODUCING RED HAT FUSE DISTRIBUTIONS AND RUNTIMES

Red Hat Fuse is provided in the form of three different distributions:

Fuse Standalone

This is the traditional distribution of Fuse that supports any operating system with a certified Java Virtual Machine.

This distribution is supported on the following application containers or runtimes:

- Apache Karaf, an OSGi container

- JBoss Enterprise Application Platform (EAP), a certified Enterprise Java (Java EE) application server

- Spring Boot, a popular framework for developing microservices

Fuse on OpenShift

This distribution supports deploying integration applications as containerized applications on OpenShift, which is Red Hat's enterprise distribution of Kubernetes.

All three runtimes supported for Fuse Standalone are supported by Fuse on OpenShift, providing a migration path from legacy deployments to containerized deployments.

Red Hat Fuse 7 provides a unified technology stack for all supported runtimes, which is built upon the following components:

- Apache Camel: Enterprise Integration Patterns (EIP)

- Apache CXF: SOAP-based Web Services

- AMQ clients: connect with an external AMQ broker

- Naranaya: distributed transaction manager

- Undertow: lightweight and high-performance web container based on asynchronous I/O

A Red Hat Fuse 7 subscription includes entitlements to two other Red Hat Middleware products:
- JBoss Enterprise Application Platform (EAP)

• Red Hat AMQ, a messaging server

To use Red Hat Fuse on OpenShift, you must have a subscription to Red Hat OpenShift Container Platform or any other edition of OpenShift, such as OpenShift.io.

SOLVING INTEGRATION PROBLEMS USING FUSE

To illustrate how Fuse and Camel helps solve integration problems, consider a typical order management system: The system includes a number of endpoints and some business and integration logic. Vendors from traditional stores use legacy, scheduled *extract transform load* (ETL) processes to place orders.

These ETL processes drop files containing many orders per file on the file system to be processed later and saved into a *relational database management system* (RDBMS). Data from the RDBMS is in turn used to process order fulfilment and other internal business processes.

Different vendors generate orders using different formats, which require specific ETL processes for each, so they comply to the RDBMS database schema. For example, one vendor sends orders in a CVS format, produced by a mainframe application, while other vendor sends orders using a XML format produced by an *Enterprise Resource Planning* (ERP) package.

Vendors that use newer web-based storefronts place orders in real time using a REST API supported by an off-the-shelf online shopping application.

Further, these orders trigger events within your system, which starts back-end processes, such as payment confirmation and shipping of products, over a messaging middleware using *Java Message Service* (JMS).

Each new vendor requires a different set of data transformations and may use a different networking protocol. In the past, updating an integration application to support a new vendor was a time-consuming process that impacted the business ability to make new partnerships and lead to lost opportunities. Red Hat Fuse allows implementing integration services that are stateless, fault-tolerant, and largely reusable. A new integration application can be quickly added to match a new vendor requirement, without impacting the integration applications already deployed to serve current vendors.

The following figure shows a high-level overview of the set of integration applications used to address the order management system:
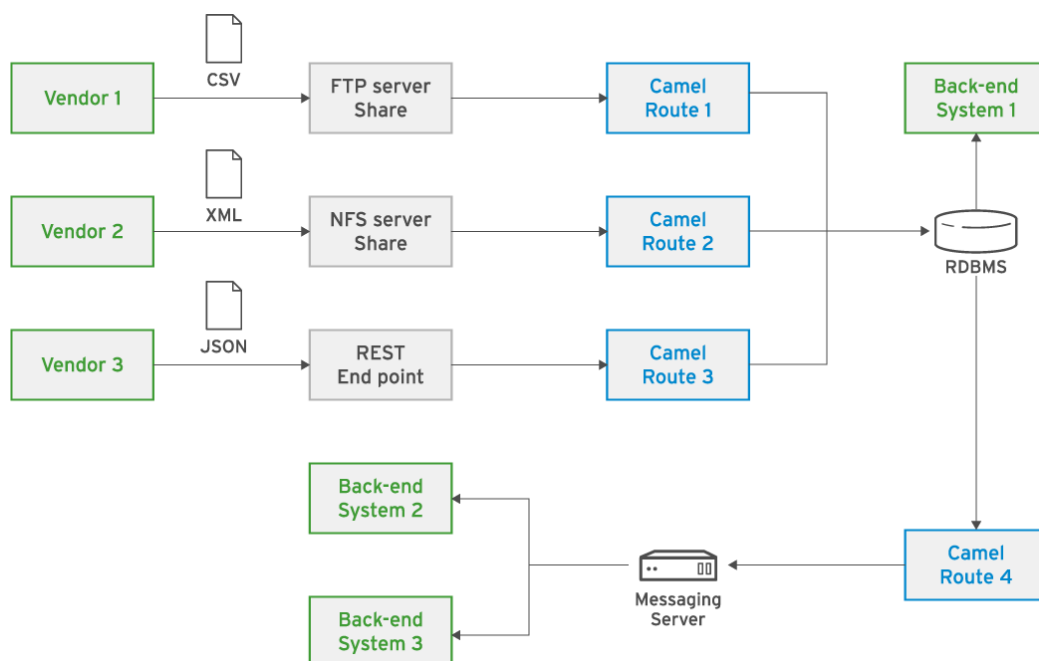
• Receive data from FTP servers and file systems, transform, and save to the RDBMS

• Receive data from REST endpoints, transform the data, and then saves it to the RDBMS

• Watches for new records on the RDBMS and dispatch notification to the back-end systems

A developer designs and implements an integration application as a set of Camel *routes* that:
A developer builds Camel routes by connecting components that receive, marshal, transform, and send data. Different components deal with different data sources (file systems, RDBMS), data formats (CSV, XML, JSON), and data sinks (RDBMS, JMS).
These routes could all be deployed as a single application or multiple applications on the same Fuse runtime, or they could be deployed as independent services over different Fuse runtimes. Depending on the deployment architecture, you can view the set of Camel routes as a single integration application or as a loosely coupled set of integration applications.

PROBLEMS SOLVED BY ENTERPRISE INTEGRATION PATTERNS AND CAMEL

While every software application is distinct and unique, many technical problems are similar or common among all applications. To avoid such pitfalls and to approach application design in
a reliable way, software developers use *design patterns*. Java developers are used to design patterns, such as Singleton and Factory, that solve technical problems and provide a common vocabulary to describe solutions and their applicability.
Organizations deal with integration problems involving enterprise software by leveraging *Enterprise Integration Patterns*. In 2003, Hohpe and Woolf published the book *Enterprise Integration Patterns* (EIP) describing 65 separate patterns that represent common approaches for designing integration solutions. These EIPs provide the framework and inspiration for Apache Camel's approach to integration.
In their book, Hohpe and Woolf describe EIPs using the concept of *messaging*. Messaging in the context of EIPs and Camel is distinct from *message-oriented middleware* (MOM), such as Red Hat AMQ and *Java Message Service* (JMS) from *Java Enterprise Edition* (Java EE). Messaging in EIPs is a high-level concept that describes information flow from one system to another managed by EIPs.
Traditional integration applications approach integration by implementing XML-based *Simple Object Access Protocol* (SOAP) web services. The contemporary approach to application integration leverages messaging and EIPs using JSON-based *representation state transfer* (REST) HTTP APIs. Camel is able to use JMS, REST, and other messaging technologies with its implementation of EIPs.

DESCRIBING ENTERPRISE INTEGRATION PATTERNS

The following is a review of some of the Enterprise Integration Patterns (EIPs) used in this course. This is not an exhaustive list of all EIPs that Camel provides, rather this list illustrates some of the most common EIPs.

Splitter

This pattern separates a list or collection of items in a message payload into individual messages. This can be used when you need to process smaller, rather than larger, bulk messages.
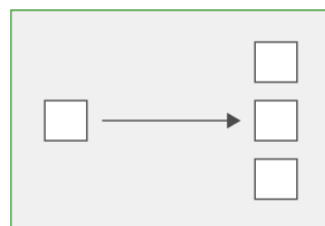

Figure 1.2: Splitter EIP

A benefit of the Splitter EIP is that when a message is split into chunks that can be processed independently of each other, an error within the split message does not cause all message processing to stop. A single split message can be retried, or an error response can be sent back to originating system while remaining messages are processed despite the error. Further, a
Splitter is useful for consuming messages in parallel, as discussed in detail in a later chapter about Concurrency.
An example of a Splitter EIP in action is an online order system that sends orders in batches.
The Splitter EIP is used to split individual orders from a single large batch message. With the messages split, each order is checked out from inventory. With an additional Splitter, each item in an order is individually validated to ensure that the item is in stock. Because of the Splitter, all of this processing occurs in parallel and simplifies the business logic by allowing each system to work with smaller messages.

Message Channel

The Message Channel pattern represents point-to-point communication between two applications or systems, with the additional benefits of providing asynchronous communication and decoupled endpoints using a messaging system.
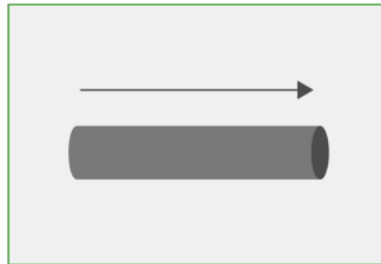

Figure 1.3: Message Channel EIP

Asynchronous and decoupled communication is typically implemented with the help of a message-oriented middleware, such as Red Hat AMQ, allowing an application to provide quick responses and continue servicing requests when there is a failure in some dependent system.

A message channel is a common pattern to use in e-commerce applications, such as an online store. When a user checks out the shopping cart, there is no need to wait for payment confirmation, checking inventory levels, and shipping. All these processes can work independently of each other, using message channels. Checking out the shopping cart puts a message in a channel, where it waits for the payment process. Then the payment process puts a message in another channel, where it waits for the inventory process. If a payment processor is unavailable, this does not prevent processing orders that require a different payment processor. Orders that require the unavailable payment processor keep waiting in a queue until that payment processor is available again.

Normalizer

This pattern processes messages from systems using different formats and converts the into a common format.
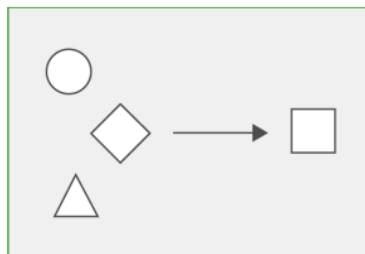

Figure 1.4: Normalizer EIP

For example, normalizer is useful for a company consumes orders from multiple vendors. Some vendors might use legacy mainframe systems and submit orders as CSV files. Other vendors use *enterprise resource planning (ERP)* systems that submit orders as XML files. And many vendors are switching to newer REST-based systems that submit orders as JSON files.

A normalizer converts all of these orders to the data format expected by the company's internal order processing system. A benefit of this approach is that the enterprise does not need to change their business logic as they expand to include new vendors that send orders in a specific format.
Rather, they only need to update their integration solution to support additional formats.

Transactional Client

A Transactional Client calls to another system to ensure delivery and ensure data integrity during the transaction.
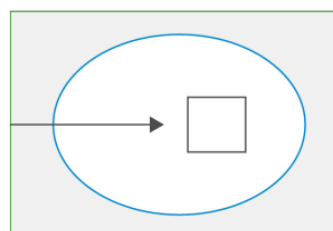
Figure 1.5: Transactional Client EIP

Many developers are familiar with the concept of a *transaction*, or *single unit of work*, from relational databases. However, transactions also apply to non-relational databases and messaging systems.

Consider a process that reads data from a message queue and saves the data to a database. One concern in this approach is that if the process reads the message from the queue, but the database writes fail, the message is lost because it is no longer available in the queue. To solve this, you want the message to remain in the queue in the event of a database write failure, so it can be read again later. Further, you want to ensure that if the write is successful to the database, that the message is removed from the queue.

Wrapping these three operations (reading a message from a queue, writing to a database, and removing a message from a queue) into a single transaction guarantee that either all operations succeed, or the system rolls back the entire operation to maintain data integrity.

Messaging Gateway

The Messaging Gateway pattern delegates the responsibility of messaging to a dedicated messaging layer so that business applications do not need to contain additional logic specific to messaging. The dedicated messaging layer becomes the source or destination of messages from EIPs.
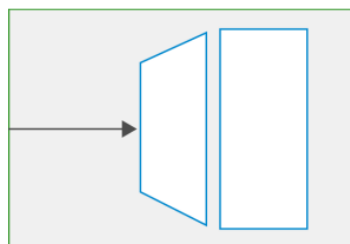

Figure 1.6: Messaging Gateway EIP

For example, a web server that throttles traffic and authenticates clients as they submit HTTP requests destined to business applications without regard for the request contents is an example of a messaging gateway. The gateway is able to process each request before passing the message to the business application without the business application needing to contain the added complexity of messaging protocols and logic.

This kind of functionality is the motivation for many *API Gateway* products in the market, such as Red Hat 3Scale. Red Hat Fuse provides an easy way to implement simple internal API gateways by combining EIPs, while Red Hat 3Scale provides built-in features, such as metering and billing, that simplifies implementing an API gateway for external users. These two products can be combined for more complex gateway scenarios.

Content Based Router

A Content Based Router pattern routes messages to an endpoint dynamically based on the content of the message's body or header.
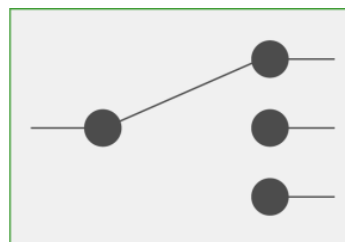

Figure 1.7: Content Based Router EIP

This pattern is useful in the following scenario: an online store, which used to sell only products from its own inventory, starts a partnership with local business to sell their items. Some orders are to be processed by the internal order processing system, but others need to be sent to external systems for fulfillment.

A content-based router would look at the products included in each order. If these products come from the store's own inventory, they are sent to the internal order fulfillment system. If the products come from the external vendor, they are sent to the external vendor system.

Aggregator

The Aggregator pattern collapses multiple messages into a single message. This is helpful to aggregate many business processes into a single event message to be delivered to another client or to re-join a list of messages which were previously split using the Split pattern.
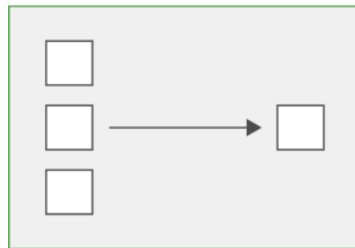

Figure 1.8: Aggregator EIP

The aggregator pattern makes efficient use of information systems that perform better when they process data in batches instead of as individual messages.

Consider a checking account system that processes transfers from other banks to customers' checking accounts. Because communication happens over slow WAN links, and there is no requirement that the funds transfer occur in real time, the checking accounts systems use an aggregator to batch transfer messages that send funds to the same bank. If a transfer operation is made to wait for too long, it is sent alone in a batch containing a single message.

COMPOSING MULTIPLE EIPS INTO A CAMEL INTEGRATION APPLICATION

A typical integration application relies on more than just a single EIP. One of the challenges of being a good Camel developer is understanding which patterns to use to provide the most reliable and efficient integration system.

Consider the example of an online web store that sells goods from external vendors. The same order could contain items from different vendors. A Splitter EIP generates multiple smaller orders, each one destined to a single vendor, and a Content Based Router EIP submits the individual smaller orders to the correct vendor.

A Camel route composes multiple EIPs into a pipeline that has a message source and one or more message destinations.
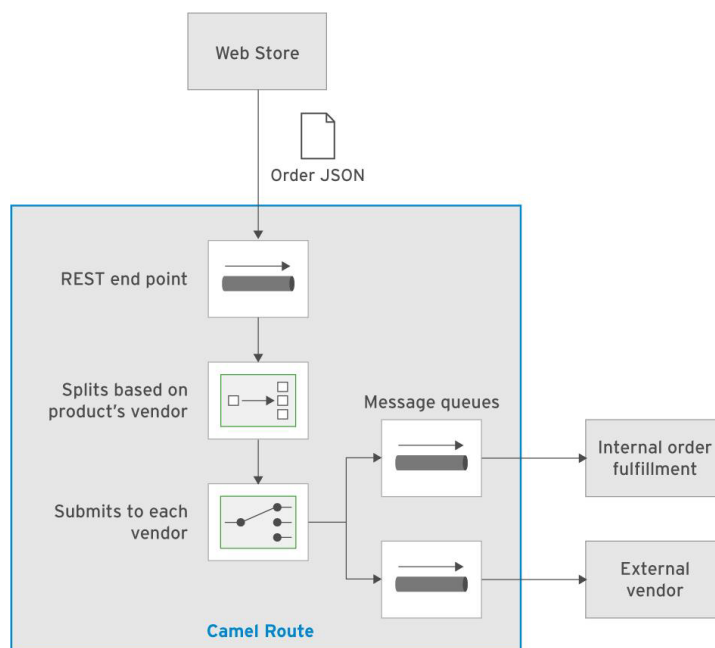
Figure 1.9: Composing multiple EIPs Into a Camel Route

A Camel route in the online web store example receives messages from a REST API end point, passes each message though a Splitter, then to a Content Router, which in turn submits each message to a Message Channel specific to each external vendor.

DESCRIBING CAMEL CONCEPTS

DESCRIBING THE CAMEL ARCHITECTURE

Apache Camel is a Java library that implements Enterprise Integration Patterns (EIPs) and provides a solid foundation to develop integration systems for both legacy and new applications. Apache Camel works with different messaging systems, ranging from older mainframe-based applications, passing through the WS-* standards world, to the new HTTP API-based style of development. Camel also works with message-oriented middleware (MOM) that can be accessed using the Java Message Service (JMS) API, such as Red Hat AMQ.

Not all inputs and outputs to a Camel route need to come from an actual messaging system. Apache Camel can also process data from file systems, FTP servers, databases, and other data stores. Camel encapsulates data originating from these data stores as messages, so these data stores can act as source or destination of messages in an integration application.

Camel's architecture is based on a small number of Java classes and interfaces: CamelContext, Route, Endpoint, Component, Exchange, Message, Processor and Predicate.
The following section discusses concepts related to these classes and interfaces at a very high level, not the syntax around them. All these concepts are revisited during this course, along with coding examples.

Context, Routes, and Components

Integration applications based on Camel define a context that aggregates one or more routes and zero or more components.

A context provides the environment where running routes and components run. Apache Camel provides integration with many application frameworks and runtimes so most developers do not need to know how to initialize and start a Camel context.

A route describes an integration scenario, where a message flows from a source to one or more destination endpoints and may be transformed along the way. An endpoint is a name, usually using URI syntax, to a specific source or destination for messages.

A component provides configuration for endpoints and encapsulates the logic required to work with different sources or destinations of messages, such as FTP servers, JMS Queues, or relational databases. Multiple endpoints can reuse the same component, and multiple instances of the same component can be defined in the same context.

For example, many endpoints can use the File component, which accesses files from the local file system. Each endpoint can specify a different file system folder. Another example is the JDBC component, which can access a relational database. Multiple instances of the JDBC component can connect to different database servers using different datasources.

A context only needs to describe components that require additional, explicit configuration, such as a database server, which requires at least a server address and login credentials. Many components take all the configuration they need from an endpoint URI, such as the path for a file system folder.

Camel routes are defined using *Domain Specific Language* (DSL). The Apache Camel project provides a number of DSLs optimized for programming languages, such as Scala and Groovy. Red Hat Fuse provides production support for the following two DSLs:

Java DSL
Uses a fluent style of concatenated method calls to define routes and components. A route contains no procedural code, therefore if you need to embed complex conditional or transformation logic inside a route, you can invoke a Java Bean.

XML DSL

Uses XML elements inside a Spring bean or an OSGi Blueprint configuration file to define routes and components. Though the XML DSL is not as complete or as flexible as the Java DSL, the XML DSL does allow non-developers to create routes using a graphical tool if desired.

These two DSLs can be mixed: the same integration application can define some routes using the Java DSL, and others using the XML DSL. Components configured using the XML DSL can be used in routes defined using the Java DSL, and vice versa.

Exchanges, Messages, and Endpoints

A route connects at least two endpoints: a *consumer* and one or more *producers*. A consumer defines the origin of messages the route receives, and a producer defines a destination for messages that the route sends.
These terms can be counter-intuitive for users accustomed to messaging systems. To avoid confusion, remember they follow the point of view of the route: a route consumes messages that it receives from an endpoint, and produces messages that it sends to another endpoint.
A consumer provides a route with an *exchange*, which flows through the route until it reaches a producer. An exchange contains one *inbound* and optionally one *outbound* message. Notice that these are not input and output messages. They are request and response messages. The inbound and outbound denotation is relative to the consumer endpoint's point of view.
Depending on the component type of an endpoint, the Camel route may not produce an outbound message, that is, a response to the consumer. The message a route sends to a producer is always the inbound message of the exchange.

Processors, Predicates and Expressions

Between the consumer and producer endpoints, a number of *processors* can manipulate the exchange contents. Processors implement most of the EIPs provided by Camel. Most processors manipulate only the inbound message, as this is the message the route sends to a producer. An example processor takes a message in XML format and transforms it into a different XML format using a XSLT transformation.

Many processors make use of *predicates* and *expressions*. Predicates allow you to create conditions based on a message's body, headers, and properties. Expressions extract data from
a message. Camel supports a number of expression languages, such as XPath, JSONpath, and
MVEL.

MVEL is very similar to the Java EE *expression language* (EL) used by *Java Server Pages* (JSP) and the *Java Server Faces* (JSF) framework and provides a simple way to access properties of Java Beans.
Predicates allow a developer to define conditions, such as a message that does not have an empty body, or a message that contains a given header. Expressions are more flexible, but the expression language must match the data format of a message. For example, MVEL only works for messages that contain Java Beans, and XPath only works for XML data.

Extending and Supporting Camel

You can implement custom components, processors, and predicates to encapsulate complex logic and use them inside a route. With this functionality you can extend Camel's feature set to meet almost any requirement, adding new integration patterns and connectors to new message sources or destinations.
These custom components, processors, and predicates can be glued to routes using many approaches, for example a dependency injection framework, such as the Spring framework or Context and Dependency Injection (CDI) from Java EE.
Camel itself is agnostic to any application framework or runtime. Be aware that the Apache Camel community may provide a number of integrations with programming languages, frameworks, and information systems that are not mature enough for production usage. This is one reason why Red Hat Fuse does not include or support all of Apache Camel, but only a large subset of the components.
The parts of Camel that are included in the Red Hat Fuse product receive full production support from Red Hat. Subscription users can open support tickets and rely on Red Hat to provide a solution according to the user's service-level agreement.
Be careful to not abuse Camel's power: Camel's features are sufficient to implement many application scenarios, and you may end up implementing business logic inside a Camel route. Remember that Camel's goal is not to replace existing applications, nor to implement new applications. Camel's goal is to integrate existing and new applications, moving data between them in an easy and flexible way.

CHAPTER 2: CREATING ROUTES

READING AND WRITING FILES

CAMEL ROUTES

In Camel, a *route* describes the path of a message from one *endpoint* (the origin) to another endpoint (the destination). Routes are a critical aspect of Camel because they define integration between endpoints. With the help of components, Routes are able to move, transform, and split messages. Traditionally, integration implementation requires lots of complicated and unnecessary coding. With Camel, routes are defined in a few simple, human-readable lines of code in either Java DSL or Java XML.

A route starts with a *consumer*, which receives the data from a point of origin. The term "consumer" is slightly counter-intuitive because it is often used to refer to the destination of a message. With Camel, consider that the consumer is referring to where and how the initial

message is being picked up. The origin endpoint is any Camel component, such as a location on the file system, a JMS queue, or even a tweet from Twitter. The route then directs the message to the *producer*, which sends data to a similarly configured destination. By abstracting the integration code, developers are able to implement EIPs that manipulate or transform the data within the Camel route without requiring changes to either the origin or the destination.

Figure 2.1: Route Example

In this example, a consumer receives a CSV file from a file server. The file server is the origin while the CSV file is the message. The route transforms the data into the JSON format, and the producer routes the JSON to a web system through an HTTP request.

CamelContext

To implement a Camel route, the route must be attached to a `CamelContext` instance. The `CamelContext` loads all resources like components, endpoints, and converters in the runtime system to execute the routes.

A common way to instantiate a `CamelContext` and attach routes to it is by using the Spring Framework. Camel provides integration with this framework, including an XML configuration dialect to use inside Spring beans configuration files.

Alternatively, you can instantiate a new `CamelContext` using the Java language:

```
DefaultCamelContext camelContext = new DefaultCamelContext(registry);
camelContext.setName("camelContext");
camelContext.addRoutes(new FileRouter()); camelContext.start();
```

The Camel-specific configuration inside a Spring beans configuration uses the XML DSL. A Spring beans configuration can also refer to routes defined using the Java DSL by declaring `RouteBuilder` objects as Spring beans.

JAVA DSL

In Java DSL routes in Camel are created by extending the `org.apache.camel.builder.RouteBuilder` class and overriding the `configure` method. A route is composed of two endpoints: a consumer and a producer. In Java DSL, this is represented by the `from` method for the consumer and the `to` method for the producer. Inside the overridden `configure` method, use the `from` and `to` methods to define the route.

```
public class FileRouteBuilder extends RouteBuilder {
@Override
public void configure() throws Exception {
from("file:orders/incoming")
.to("file:orders/outgoing");
}
}
```

In this example, the consumer uses the **file** component to read all files from the **orders/ incoming** path and the producer uses the **file** component to move to the **orders/outgoing** path.
Inside a **RouteBuilder** class, each route can be uniquely identified using a **routeId** method. Naming a route makes it easy to verify route execution in the logs, and also simplifies the process of creating unit tests. Multiple routes can be created by calling the **from** method many times.

```
public void configure() throws
Exception {
from("file:orders/incoming")
.routeId("route1")
.to("file:orders/outgoing");

from("file:orders/new")
.routeId("routefinancial")
.to("file:orders/financial");
}
```

Each component can specify configuration attributes. You can add attributes by appending a
? after the origin or destination. Refer to the Camel documentation for component-specific attributes.

```
public void configure() throws Exception {

from("file:orders/incoming?include=order.*xml")
.to("file:orders/outgoing/?fileExist=Fail");

}
```

The route consumes only files with a name starting with "order", and it is an XML file.
Throws an exception if a given file already exists.

To enable a route, you need to register it as a Spring bean inside a Spring beans configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

<bean class="com.redhat.training.jb421.FileRouteBuilder"
id="fileRouteBuilder"/>

<camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">
<routeBuilder ref="fileRouteBuilder"/>

</camelContext>
```

Creates a new Spring bean.
Enables the route in the **CamelContext**.

XML DSL

Method names from the Java DSL map directly to XML elements in the Spring DSL in most cases. However, due to syntax differences between Java and XML, sometimes the name and the structure of elements are different in Spring DSL. Refer to Camel documentation for the correct structure of the route methods.

To use the Spring DSL, declare a `camelContext` element, using the custom Camel Spring namespace, inside a Spring beans configuration file. Inside the `camelContext` element, declare one or more `route` elements starting with a `from` element and usually ending with a `to` element. These `from` and `to` elements are similar to the Java DSL `from` and `to` methods.

LAB:

READING MESSAGES FROM AN FTP SERVER

MESSAGE EXCHANGE PATTERNS

The message exchange pattern describes the behavior of the communication between the producer and the consumer. Two of the primary message exchange patterns are:

Synchronous (InOut)
Consumer sends the request and consumer waits for a response.

Asynchronous (InOnly)
Consumer sends the request and does not wait for a response.

To support these patterns, Camel uses an *exchange* object. The exchange object encapsulates a received message, known as `inMessage`, and stores its metadata. When using the `InOut` pattern, the exchange also encapsulates a reply message, also known as the `outMessage`.
Camel supports several message exchange patterns, including `InOnly` and `InOut`. These patterns serve different purposes, and one may be more appropriate than the other, depending on the functionality of your route.
By default, many Camel components, such as jms, file, or ftp, use `InOnly` exchanges unless otherwise specified. However, this can vary from one component to another, so you should check the documentation for the components you are using in your route for their default exchange pattern.

THE `InOnly` EXCHANGE PATTERN

Messages in routes that use the `InOnly` pattern are sometimes referred to as event messages. These messages are a one-way communication and, as such, only the `inMessage` will ever have a value on the message exchange. The `InOnly` pattern is appropriate to use when the producer in your route does not need any reply information from the route consumer.
For example, a route consumes an order file from an FTP server and produces a message to the financial system. The consumer does not need to wait for a reply from the producer.
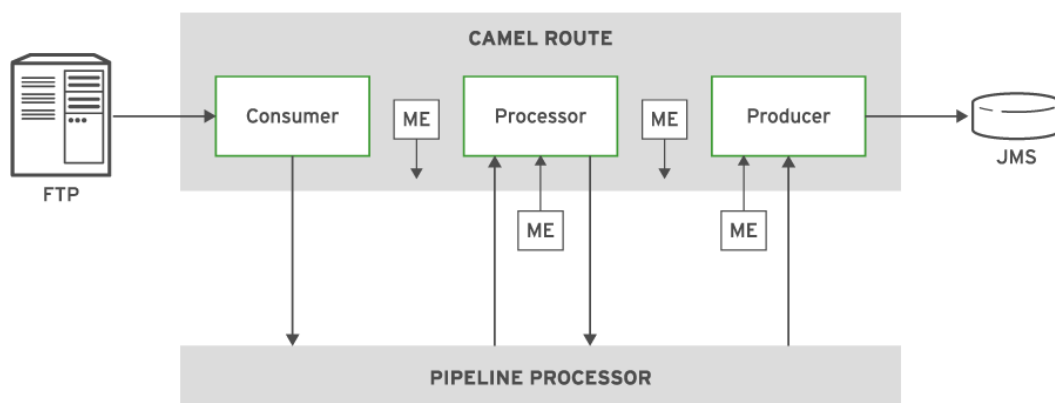

Figure 2.2: Camel route and pipeline processor

To access the **inMessage** object on the Camel exchange at runtime, use the **getIn** method provided by the **Exchange** interface as shown below:

```
Message inMessage = exchange.getIn();
```

You should access the **inMessage** object of the exchange object when you want to manipulate the original message, such as adding new headers or changing the original content.

THE **InOut** EXCHANGE PATTERN

Messages in routes using the **InOut** pattern are referred to as *request-reply* messages. The **InOut** pattern is used when a route's consumer requires a response from the producer before proceeding. This can be useful to report the results of some action that the route has taken, or to make a callback when the route is complete.

For example, the route consumes an order request from a client. The producer sends a message to the bank system to approve the transaction, and needs to wait for a reply message with the transaction status. If the transaction is approved, the route needs to produce a new message to the shipment route. If not, the route needs to produce a new message that sends an email to the client asking for a new payment transaction.
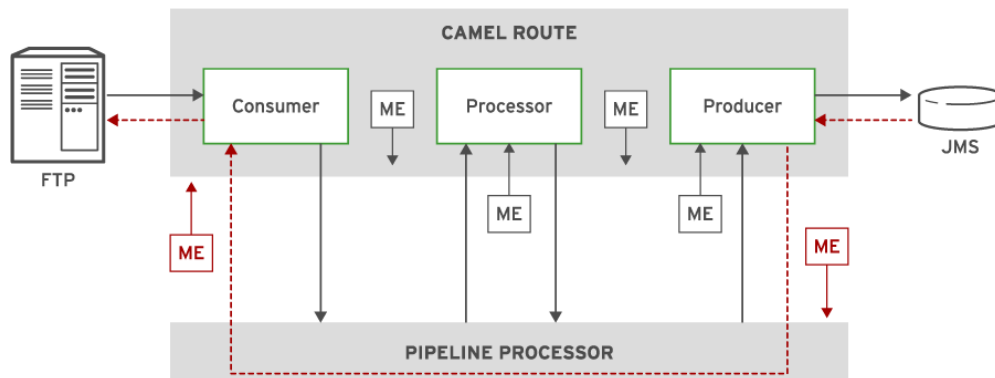

Figure 2.3: Camel route with request and reply messages

To access the **outMessage** object on the Camel exchange at runtime, use the **getOut** method provided by the **Exchange** interface as listed:

```
Message outMessage = exchange.getOut();
```

You should access the **outMessage** of the exchange object when you want to manipulate the reply message.

MESSAGES IN CAMEL

A message is a uniquely identified data structure for storing data from a producer to a consumer. The following are the primary components of messages:

Headers
The message header contains metadata such as information about the sender of the message and security details in a **java.util.Map** collection. Use the **setHeader** method to add a new header.
Placing custom metadata in a message header is a useful approach for dynamically routing messages or providing additional ways to categorize messages.

Attachments
Contain file-based contents to be transmitted to a receiver. For example, you can attach an image in the email component.

Body
The message body contains the data to be processed by the receiver or manipulated by Camel processors. This is usually JSON or XML data, but can be an **Object** of any type.

Fault flag
The fault flag is a property that developers can use to identify whether the message contained an error during processing.

ACCESSING COMPONENT AND EIP DOCUMENTATION

Extensive documentation for all Camel components and EIPs are available directly from `camel.apache.org`. Red Hat also maintains documentation for all supported Camel components and EIPs, which can be found in the Red Hat Fuse documentation at `https:// access.redhat.com/documentation/en-us/red_hat_fuse/7.1/`.

While some of information available from these two sources is mostly identical, be aware that:
• The Camel community documentation may contain changes that have not yet been incorporated by the Red Hat Fuse product.

• Some Camel components may not be included as part of the Red Hat Fuse product. They can be added by the developer, but its use is not supported by Red Hat. This might be an important consideration for production environments.

• The Camel community documentation does not contain details about packaging and deploying to the supported Apache Karaf, JBoss EAP, and Spring Boot containers. These containers are discussed in more detail in a later chapter in this course.

The component documentation provides useful information about component syntax and attributes. When reviewing the documentation for components, carefully observe the following information:

URI format
A component URI format specifies the format required when defining an endpoint. This typically involves a specific scheme to the URI such as `ftp:` or `file:` as well as other required and optional values to be included in the URI to control component behavior. The URI always begins with the name of the Camel component. For example, the ftp component URI format:

```
ftp://[username@]hostname[:port]/foldername[?options]
```

For the FTP component, **hostname** is a required options, but the **username** and **port** are optional.

URI options
The URI options are where most of the specific configuration parameters for a component are defined. These often have default values, so be sure to review the expected component behaviour.

Message Headers
Camel components frequently provide special headers that are automatically populated when that component is used. For example, in the ftp component, you have a message header containing the original file name.

Required Dependencies
Most components are not included in **camel-core**. This section of the documentation informs you about required dependencies for those components that need to be added to the **pom.xml** file.

LOGGING IN CAMEL

Camel provides two logging features: the *log component*, and the *log EIP*.

The log component is used as a producer endpoint to output log events using SLF4J logging. Logging is a important feature to debug and collect information about routes. Output is delegated to the first logging API available in the class path. For example, the Apache log4J or **java.util.logging** API is used, based on which API is available first in the class path.
The log component URI format is:

```
log:loggingCategory[?options]
```

For the log component, the `loggingCategory` option is the name of the logging category to use. Usually, this is the Java package name.

The following are examples of using the `log` component:

• Set the logging name and level explicitly:

```
from("activemq:input")
.to("log:com.foo?level=DEBUG")
.to("activemq:output");
```

• Format the output of the information from the exchange:

```
from("activemq:input")
.to("log:com.foo?showAll=true&multiline=true&level=DEBUG")
.to("activemq:output");
```

The log EIP, provided by the `log` DSL method, is another way to add logging to a Camel route. You should use the log EIP when you want to log custom messages. It also uses the SLF4J logging facade and messages are formatted using the *Simple expression language* (Simple EL), covered in detail later in this section.

Some examples of using the `log` DSL method are:

• Set the logging name and level explicitly using the Java DSL

```
from("activemq:input")
.log(LoggingLevel.DEBUG, "com.foo", "Got a message")
.to("activemq:output");
```

• Set the logging name using the XML DSL:

```
<from uri="activemq:input"/>
<log loggingLevel="DEBUG" loggerRef="com.foo" message="Got a message"/>
<to uri="activemq:output"/>
```

USING SIMPLE EXPRESSION LANGUAGE

Log, filter, and other EIPs can use the Camel Simple EL to get the current exchange attributes (body, headers, properties) from a message. For example:

```
from("file:in")
.log("message body:${body}")
.to("file:out")
```

Simple expressions are delimited by a dollar sign followed by braces: `${..}`. It is similar to the Java Server Pages (JSP) EL. Objects such as the route parent Camel context, the current exchange, the input and output messages, among other convenience objects, have names predefined by Camel. Refer to Camel Simple EL documentation for a list of all available variables.

You can use Simple EL to specify dynamic URI options for your endpoints, which refer to specific data from the exchange. For example, using the `CamelFileHost` header in the file name option of the file component in the following XML DSL:

```
<to uri="file:orders?fileExist=Append&amp;fileName=${header.CamelFileHost}"/>
```

RUNNING CAMEL USING MAVEN

Camel provides a plug-in for Maven to initialize the `CamelContext` object using the Camel Spring integration libraries. This is beneficial because you can easily test your routes without having to initialize the `CamelContext` in a `main` method.

Using the Camel Maven plug-in, the application can be started using the Maven goal `camel:run` that starts an instance of `CamelContext` inside Maven in a separate thread.

Any Camel project using Maven can use this feature by adding the following to the `pom.xml` file:

• The Spring Framework integration dependency.
• The plug-in reference.

The Spring Framework integration dependency can be referenced using the following statement:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-spring</artifactId>
</dependency>
```

The Camel Maven plug-in is activated by adding the following to the POM:

```
<plugin>
<groupId>org.apache.camel</groupId>
<artifactId>camel-maven-
plugin</artifactId>
</plugin>
```

MESSAGE EXCHANGES IN CAMEL

Camel uses exchange objects to store messages processed through Camel routes. Camel exchanges also store routing information and properties. Exchange objects are composed of the following:

• *Exchange ID*: A unique identifier generated by Camel. • *Message Exchange Pattern (MEP)*: Describes how messages are exchanged from a sender to a receiver. The pattern can be an `InOnly` request (contains only one message) or an `InOut` request (containing two messages, one from the sender and another from the receiver).

• *Exception*: Stores any errors generated during the exchange processing.

• *Properties*: Stores metadata from the exchange processing, such as processing status. The route code can use properties to safely store information not meant to be processed by endpoints.

• *In message*: The request sent by the sender.

• *Out message*: The answer sent by the receiver.
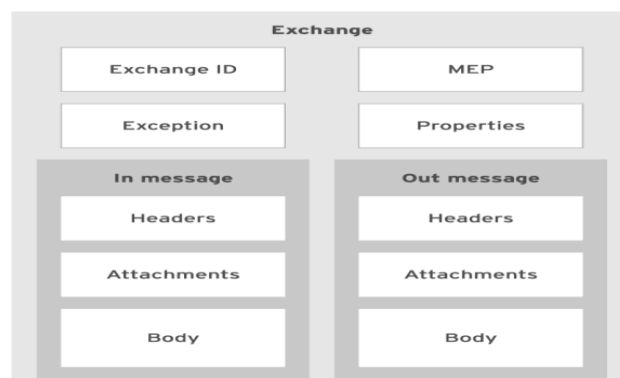
FILTERING MESSAGES

INTRODUCING CAMELCONTEXT

The **CamelContext** class is the Camel runtime environment that provides important features, such as:
When using the **camel:run** Maven goal or a runtime environment with Camel support, such as Red Hat Fuse, the **camelContext** is automatically started.

• *Simple extension support*: Camel works with components that can be injected by using popular *inversion of control* (IoC) frameworks based on either XML configuration or annotations such as Spring and CDI. These frameworks allow developers to integrate various components in a loosely coupled type-safe manner. Additionally, by using these industry standard techniques, developers can leverage existing solutions in their Camel routes.

• *Life cycle*: A **CamelContext** instance must be started to make its routes available for execution. Sometimes the routes must be stopped for maintenance purposes, and they can be suspended and resumed. The **CamelContext** instances can also be stopped to shut down each route cleanly and avoid any processing problems.

• *Services*: Services can be added to the **CamelContext** that provide important features such as persistence, configuration, or integration with other frameworks.

The **org.apache.camel.impl.DefaultCamelContext** class is used to instantiate a **CamelContext** instance. However, the **CamelContext** instance needs to be started explicitly to enable processing the routes. You can start the **CamelContext** in the **main** method.

```
CamelContext context = new DefaultCamelContext();
context.start();
```

To shut down the context gracefully, invoke the **stop** method when the route execution is completed in the **main** method.

```
context.stop();
```

CREATING A ROUTEBUILDER TO BUILD ROUTES

Use the **RouteBuilder** abstract class provided by Camel API to develop routes. Because of the dynamic nature of routes, it has many methods that are often chained together. This results in a route that is easily read and understood by a human. In Java programming, this technique is known as a *fluent interface*; essentially creating a domain-specific language.
The fluent interface from the **RouteBuilder** class provides a clean and direct code to define routes and create helper objects used in your Camel routes, such as enterprise integration pattern implementations.
Normally, the starting point to create a route is to extend a **RouteBuilder** class and override the **configure** method, for example:

```
public class FileRouteBuilder extends
RouteBuilder {
@Override
public void configure() throws Exception
{
...
}
}
```

Alternatively, a Spring beans configuration file can declare a route. To achieve this goal, a **route** element is declared, nested in the **camelContext** element. For example:

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

Routes defined this way use the Camel XML domain specific language (DSL), also known as Spring DSL.

IMPORTANT
Multiple routes can be declared in the `configure` method of a `RouteBuilder`
implementation.

A `camelContext` element can also contain references to routes defined using the Java DSL, by declaring the Java class as a Spring bean and referencing it inside the `camelContext` element:

```
<bean class="com.example.MyRouteBuilder" id="myRouteBuilder"/>

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

<routeBuilder ref="myRouteBuilder"/>

</camelContext>
```

You can use both XML DSL and Java DSL routes inside the same `camelContext` element.

DECLARING ENDPOINTS USING COMPONENTS

Data flows from a consumer to one or more producers in Camel. Each consumer or producer is an endpoint, which abstracts the end of a channel through which messages can be sent or received.

Declaring an endpoint is done using URI syntax:

<scheme>://<context/path>?<options>

For example, to connect to a file system, you must use the `file` component, which is associated with the `file:` URI prefix:

file://folder?fileExist=Append

Camel parses this URI string to instantiate an `Endpoint` and configure it according to the URI options. In the previous example, if the destination file already exists in the file system, it is not overwritten but appended. If multiple options are needed, the each one is separated by an ampersand (&).
A component is an endpoint factory and the main abstraction in Camel. A `Component` class is responsible for the following tasks:

• Connecting to the integration system
• Marshaling and unmarshaling data to Camel Message standards

Components may be explicitly configured and added to the `CamelContext`, injected by an IoC framework, or auto-discovered using URIs.

IMPLEMENTING THE FILTER EIP

Camel implements most of the enterprise integration patterns and provides them as methods in the `RouteBuilder` class or elements in the XML DSL.
Camel defines the message filter pattern to remove some messages during route execution based on the content of the Camel message.
To filter messages sent to a destination, use the following Java DSL:

```
from("<Endpoint URI>")
.filter(<filter>)
.to("<Endpoint URI>");
```

The filter must be a Camel *predicate*, which evaluates to either **true** or **false** based on the message content. The filter drops any messages that evaluate to **false** and the remainder of the route is not processed. Predicates can be created using expressions, which Camel evaluates at runtime.

Because of the number of data formats supported by integration systems, a set of technologies can be used to filter information. For example, in an XML-based message, XPath can be used to identify fields in an XML file.

To evaluate if a certain value is available at a specific XML element, use the following syntax:

filter(xpath("/body/title/text() = 'Hello World'"))

Likewise, the **Simple** expression language can be used to filter Java objects.

```
<scheme>://<context/path>?<options>
file://folder?fileExist=Append
from("<Endpoint URI>")
.filter(<filter>)
.to("<Endpoint URI>");

from("direct:a")
.filter(simple("${header.foo} == 'bar'"))
.to("direct:b")
.filter().xpath("/person[@name='James']")
.to("mock:result");
```

INTRODUCING EXPRESSIONS

Camel uses *expressions* to look for information inside messages. They support a large number of data formats, including Java-based data and common data format exchanges (XML, JSON, SQL, and so on).

An expression language used to identify Java objects is the Simple EL. It uses a syntax that resembles the Java Server Pages expression languages to search for attributes in an object. For example, in a class named **Order**, with an **address** attribute that contains a ZIP code, the Simple expression language to search for the zip code 33212 is:

${order.address.zipCode = '33212'}

IMPLEMENTING PREDICATES

Predicates in Camel are essentially expressions that must return a boolean value. This is often used to look for a certain value in an **Exchange** instance. Predicates can be leveraged by Camel in conjunction with expression languages to customize routes and filter data in a route. For example, to use the Simple expression language in a filter, the Simple expression must be called inside a route calling the **simple** method. Similarly, to use an XPath expression, there is a method named **xpath**.
The following XML is used as an example to explain the XPath syntax.

```
<order>
<orderId>100</orderId>
<shippingAddress>
<zipCode>22322</zipCode>
</shippingAddress>
</order>
```

To navigate in an XML file, XPath separates each element with a forward slash (/). Therefore, to get the text within the **<orderId>** element, use the following XPath expression:

/order/orderId/text()

To get the **zipCode** from the previous XML, use the following expression:

/order/orderId/shippingAddress/zipCode/text()

To get all XML contents where **zipCode** is *not* 23221, use the following expression:

/order/orderId/shippingAddress/[not(contains(zipCode,'23221'))]

To use the expression as a predicate in a Camel route, the **xpath** method parses the XPath expression and returns a boolean:

```
xpath("/order/orderId/shippingAddress/[not(contains(zipCode,'23221'))]")
```

ROUTING MESSAGES FROM JMS

THE CONTENT BASED ROUTER EIP IN CAMEL

The *Content Based Router (CBR)* EIP allows routing messages to the correct destination based on the message contents.

The Camel CBR implementation is similar to a programming language construct: the **choice** DSL element contains multiple **when** DSL elements and optionally an **otherwise** DSL element:

```xml
<route>
<from uri="schema:origin"/>
<choice>
        <when>
        <!-- predicate1 -->
        <to uri="schema:destination1"/>
        </when>
        <when>
        <!-- predicate2 -->
        <to uri="schema:destination2"/>
        </when>
        <!-- other when elements -->
        <otherwise>
        <!-- exception flow or destination -->
        </otherwise>
</choice>
</route>
```

Each **when** DSL element requires a predicate that, when true, triggers sending the message to its destination. If the predicate is false, the route flow moves to the next **when** element. The predicate can use Simple EL, XPath expressions, or any other expression language supported by Camel.

The **otherwise** DSL element allows having a destination for messages that fail to match any of the **when** predicates. The **otherwise** DSL element usually starts an exception flow.

CBR Using the Java DSL

The general Content Based Router (CBR) template using the Java DSL is:

```java
from("schema:origin")
.choice()
.when(predicate1)
.to("schema:destination1")
.when(predicate2)
.to("schema:destination2")
...
.otherwise()
.to("schema:destinationN");
```

INTRODUCTION TO RED HAT AMQ 7

Red Hat AMQ 7 is a lightweight, flexible, and reliable messaging platform that simplifies both the integration of application components as well as workload scaling. Based on the upstream projects Apache ActiveMQ and Apache Qpid, Red Hat AMQ integrates enterprise applications by providing a standards-based platform for application components and subsystems to communicate in a loosely coupled manner regardless of language or platform.

The AMQ Broker is a Message Oriented Middleware (MOM) component based on the ActiveMQ Artemis messaging implementation that provides queuing, message persistence, and manageability. The broker supports a variety of messaging styles such as point-to-point, publish- subscribe, store, and forward as well as multiple languages and platforms.

The main benefits of using a messaging system such as Red Hat AMQ include the fact that you can easily decouple the integrations between your systems such that their is little to no direct communication between programs. Additionally, messaging enables communications between systems to be asynchronous and event-driven. Messaging also supports additional features such as message priorities, message security, as well as data integrity and persistence.

Despite the similar characteristics this messaging system has with Camel, Camel provides more features, such as allowing integration with other components and using multiple enterprise integration patterns.

AMQ QUEUES WITH CAMEL

Using MOMs for integration purposes has been very common since the mainframe days because MOMs decouples applications from each other. The main MOM abstraction is the *queue*, which stores messages sent by a publisher until they are received by a consumer. The publisher and the consumer do not know about each other, and the publisher does not wait for a reply from the consumer. Guaranteeing message delivery is the MOM's responsibility.

The Java Message System API allows applications to connect to any MOM that provides a **JMSConnectionFactory**, in a similar way to connecting to relational databases using the JDBC API and a **DataSource**.

Camel provides a generic jms component and a specialized activemq component that share most implementation code and configuration options. Both use the JMS API to access the MOM.
The general URI syntax for the activemq component is as follows:

activemq:queue:*queue_name*

Using the Camel activemq component requires registering a **JmsConfiguration** with the Camel context. To do that using a Spring beans configuration file, declare an **activemq** bean with a **configuration** property.

```
<bean class="org.apache.activemq.camel.component.ActiveMQComponent" id="activemq">
      <property name="configuration" ref="jmsConfig"/>
</bean>
```

To create a **JmsConfiguration**, you need to register a JMS **JMSConnectionFactory** with the Camel context. To do that using a Spring beans configuration file, declare a **jmsConfig** bean with a **connectionFactory** property.

```
<bean class="org.apache.camel.component.jms.JmsConfiguration" id="jmsConfig">
      <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

To create a **JMSConnectionFactory**, you need to register a **ActiveMQConnectionFactory** with the Camel context. To do that using a Spring beans configuration file, declare a **connectionFactory** bean with a **brokerURL** property and a property to trust all packages.

The *activemq_server_uri* provides connection parameters and credentials to access an ActiveMQ broker.

```
<bean class="org.apache.activemq.spring.ActiveMQConnectionFactory" id="connectionFactory">
<property name="brokerURL" value="activemq_server_uri"/>
<property name="trustAllPackages" value="true"/>
</bean>
```

The *activemq_server_uri* provides connection parameters and credentials to access an ActiveMQ broker.

CAMEL MESSAGE REDELIVERY

Connecting to network resources such as MOMs, FTP servers, and other middleware, involves the risk of experiencing failures. Camel deals with this by using a *redelivery policy* component that
stores exchanges in memory and retries sending messages a number of times, usually with a delay between attempts.
A redelivery policy can be attached to the entire Camel context or to a specific route, by means of an exception handling configuration, or a more powerful error handing configuration.
The Camel redelivery policy is independent of the route endpoints and can also handle transient errors from EIPs in the middle of a route.

CONFIGURING EXCHANGE HEADERS

THE DIRECT COMPONENT

The direct component connects a producer endpoint directly to a consumer endpoint in the same Camel context. This component is less of an integration feature and more of a way to break a route into small pieces. The direct component is intended to make it easier to handle large workflows in smaller chunks as well as reuse common routing logic for different consumers.

For example, you have multiple routes to consume data from many sources. All of them needs to convert the content in the JSON format and send to the database. Instead of replicating the code that converts and sends the content to the database for each route, you can create a route with the direct component that performs the conversion and sends the results to the database. Then, each route can produce a message to the new route using the direct component.

The direct URI syntax is:

```
direct:name[?options]
```

The **_name_** is a handle defined by the developer to connect producers and consumers in the same Camel context, for example:

```
<route id="route1">
      <from uri="file:input"/>
      <to uri="direct:mylogic"/>
</route>
<route id="route2">
      <from uri=""direct:mylogic""/>
      <!-- do something here -->
      <to uri=""file:output"/>
</route>
```

Produces a message exchange to a direct component called `myLogic`.

Receives a message exchange from a direct component called `myLogic`.

The direct component provides synchronous invocation of any consumers when a producer sends a message exchange. A synchronous invocation means that the producer waits for the execution of the consumer. The following diagram depicts one Camel route sending a message to another using a direct endpoint named **B**. Notice that the same thread is used to execute both routes, this is why direct components are considered synchronous.
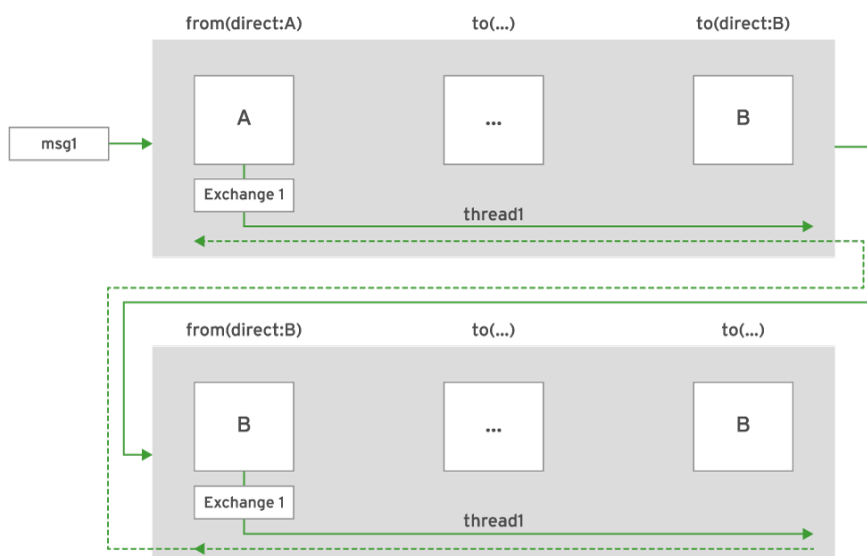
The following example demonstrates using the direct component with Java DSL:

```
from("activemq:queue:order.in")
.to("bean:orderService?method=validate")
.to("direct:processOrder");
from("direct:processOrder")
.to("bean:orderService?method=process")
.to("activemq:queue:order.out");
```

In this example, the Camel route receives messages on an ActiveMQ queue named **order.in**. The first route validates those messages using a bean, and then sends the messages to a second Camel route using a direct endpoint named **processOrder**. Camel then uses the same thread to execute the second route, using a bean to process the message before sending the result to a second ActiveMQ queue named **order.out**.

The direct component has a few limitations:

• Message passing is synchronous.

• There can be only a single consumer for each direct URI.

• Producer and consumer have to be part of the same Camel context.

THE PROCESSOR INTERFACE

The **org.apache.camel.Processor** interface provides a generic way to write custom logic to manipulate an exchange. This can be useful for creating new headers or modifying a message body; for example, adding new content such as the date that the message was processed.

You can use processors to implement the **Message Translator** and **Event Driven Consumer** EIPs.

Implementing the **Processor** interface requires implementing a single method named **process**:

void process(Exchange exchange) throws Exception

The exchange argument allows access to both the input and output messages and the parent Camel context. **Processor** implementation classes usually take advantage of other Camel features, such as data type converters and fluent expression builders.

To use a processor inside a route, insert the **process** element. For example:

```
<bean class="com.example.MyProcessor" id="myProcessor"/>
...
<route id="route1">
<from uri="file:inputFolder"/>
<process beanRef="myProcessor"/>
<to uri="activemq:outputQueue"/>
</route>
```

The same example using Java DSL:

```
.from("file:inputFolder")
.process(new com.example.MyProcessor())
.to("activemq:outputQueue");
```

Before writing your own **Processor** implementation, check first whether there are components, EIP, or other Camel features that might provide the same result with less custom code, for example:

• The **transform** DSL method allows changing a message body using any expression language supported by Camel.

• The **setHeader** DSL method allows changing header values.

• The **bean** DSL method allows calling any Java bean method from inside a route.

Compared to Java Beans, a Camel processor is preferred when there is a need to call Camel APIs from the custom Java code. A Java Bean is preferred when a transformation can reuse code that has no knowledge of Camel APIs.

DYNAMIC ENDPOINTS

An endpoint URI may include a few dynamic components, such as simple expressions to provide an optional value. Sometimes this is not sufficient if an application requires the ability to route to destinations using different components based on message contents or other runtime data.

In some application containers, Camel might cache and reuse endpoint instances in a way that the dynamic part is not re-evaluated. This could lead to messages being delivered to the wrong destination.

For those scenarios, Camel provides the **toD** DSL method, which stands for *dynamic to*, or *dynamic producer*. The URI is re-evaluated for each message sent and might even have different schema and components for each message. By default, the **toD** method uses the Simple EL to resolve the dynamic endpoint URI. For example, to send a message to an endpoint defined by a header, you can:

```
<route>
  <from uri="direct:start"/>
  <toD uri="${header.foo}"/>
</route>
```

In Java DSL:

```
from("direct:start")
  .toD("${header.foo}");
```

You can also prefix the URI with a value because by default the URI is evaluated using the Simple EL, which supports string concatenation.

```
<route>
  <from uri="direct:start"/>
  <toD uri="mock:${header.foo}"/>
</route
```

In Java DSL:

```
from("direct:start")
  .toD("mock:${header.foo}");
```

CUSTOM AND PRE-DEFINED MESSAGE HEADERS

Most Camel components, when used as endpoints, provide metadata as message headers, and also behave differently based on received headers. Header names and their meaning depends on the component used. Carefully check the component and middleware documentation before writing routing logic that relies on a header value because they can change.

The following example uses a simple expression to access a header provided by the file component:

```
.from("file:incoming")
.log("${header.CamelFileName}")
...;
```

Camel provides the **setHeader** DSL method to set header values. For example:

```
...
.setHeader("CamelFileName", "destinationFile.txt")
.to("file:outgoing");
```

Headers defined by Camel components usually have the **Camel** prefix.

It is a common practice to create custom headers to store values for later use inside a route, or for debugging purposes. For example, you can create a header that contains an attribute from the message exchange and then route the message based on this attribute. Any header whose meaning is not known by a Camel component is ignored.

CHAPTER 3 TRANSFORMING DATA

GOAL

Convert messages between data formats using implicit and explicit transformation.

OBJECTIVES

• Invoke data transformation automatically and explicitly using a variety of different techniques.

• Transform messages using additional data formats, custom type converters, and the Message Translator pattern.

• Merge multiple messages using the Aggregator pattern and customizing the output to a traditional data format.

• Access a database with JDBC and JPA.

TRANSFORMING DATA IN A CAMEL ROUTE

One of the most common problems when integrating disparate systems is that those systems do not use a consistent data format. For example, you might receive new product information from a vendor in CSV format, but your production information system only accepts JSON data. This
difference in data format means you need to transform the data from one system before another system can correctly process it. For this reason, Camel is designed to handle these differences by providing support for dozens of different data formats, including the ability to transform data from one format to another. The data formats that Camel supports include JSON, XML, CSV, flat files, EDI, and many others.

Marshaling Data

Marshaling is the process where Camel converts the message payload from a memory-based format (for example, a Java object) to a data format suitable for storage or transmission (XML or JSON, for example). To perform marshaling, Camel uses the `marshal` method, which requires a `DataFormat` object as a parameter.

Data Formats

Data Formats in Camel are the various forms that your data can be represented, either in binary or text. The data formats that Camel supports include standard JVM serialization, XML, JSON, CSV data, and others. Each data format has a class you need to instantiate and optionally configure before you can use it in a route. For example, the `JaxbDataFormat` class allows you to specify the package where your XML model classes are present. Once you instantiate the `JaxbDataFormat` class, it uses your model classes, complete with JAXB annotations as the blueprint for how to marshal and unmarshal your XML data to your Java model classes. Additionally, data format classes support a number of custom options, such as "pretty print", or other configurations that affect the behavior of marshaling or unmarshaling. These options allow you to have more control over the marshal and unmarshal behavior such as how to handle missing elements, what type of encoding to use, namespace prefixing, and more.

Unmarshaling Data

Unmarshaling is the opposite process of marshaling, where Camel converts the message payload from a data format suitable for transmission such as XML or JSON to a memory-based format, typically a Java object. To perform unmarshaling explicitly in a Camel route, use the `unmarshal` method in the Java DSL. Similar to the marshaling, the method requires a `DataFormat` object as a parameter.

TRANSFORMING XML DATA USING JAXB

There are multiple options for working with XML data in Camel. JAXB is a very popular XML framework and is the XML marshaling library focused on throughout this course. To use JAXB with Camel, include the `camel-jaxb` library as a dependency of your project:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-jaxb</artifactId>
</dependency>
```

The `camel-jaxb` library also provides the JAXB annotations that you need to annotate your model class. These annotations are used to tell JAXB how to unmarshal the XML data into the Java objects correctly. JAXB matches fields on the model class to elements and attributes contained in the XML data using the information provided by the JAXB annotations.
Given the following XML content:

<order id="10" description="N2PENCIL" value="1.5" tax="0.15"/>

The following JAXB model class contains the necessary annotations to marshal the XML to a Java object:

```
package com.redhat.training;
 import java.io.Serializable;
 import javax.xml.bind.annotation.*;

  @XmlRootElement
  @XmlAccessorType(XmlAccessType.FIELD)
public class Order implements Serializable{

private static final long serialVersionUID = 5851038813219503043L;
@XmlAttribute
 private String id;
@XmlAttribute
 private String description;
@XmlAttribute
private double value;
 @XmlAttribute
private double tax;
 }
```

Note that this model class implements the **java.io.Serializable** interface which required by JAXB to execute the marshaling and unmarshaling process.

In conjunction with the **camel-jaxb** library, you must also include JAXB annotations to instruct Camel which XML fields map to which properties in your Java model classes. In the previous example, each field in the Java class represents an attribute on the root **Order** element. By default, if no alternate name is specified in the JAXB annotation parameters, the marshaller uses the name of the field directly to map the XML data.

In the following sample route, an external system places XML data in the body of a message and then sends the message to an ActiveMQ queue named **itemInput**. The Camel route consumes the messages, and JAXB immediately unmarshals the XML data and replaces the contents of the exchange body with the corresponding Java object. The route then sends the Java object data in a message to an ActiveMQ queue named **itemOutput**.

For an XML-based route, declare the JAXB **dataFormat** and then use it to unmarshal XML data as shown in the following example:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
<dataFormats>
<jaxb id="jaxb" contextPath="com.mycompany.training"/>
</dataFormats>
<route>
<from uri="activemq:queue:itemInput"/>
<unmarshal ref="jaxb"/>
<to uri="activemq:queue:itemOutput"/>
</route>
</camelContext>
```

The **contextPath** parameter refers to which package JAXB should scan for model classes that are annotated with JAXB annotations.
Convert the XML data in the body of the exchange into an instance of the corresponding model class and replace the exchange body with that model class instance.

Similarly, the following example demonstrates marshalling Java objects into XML using the Java DSL:

```
from("activemq:queue:itemInput)
   .marshal().jaxb()
   .to("activemq:queue:itemOutput);
```

Notice that in the previous example Java DSL route, no JAXB data format is instantiated. This is possible because `camel-jaxb` automatically finds all classes that contain JAXB annotations if no context path is specified, and uses those annotations to unmarshal the XML data.

MARSHALING AND UNMARSHALING JSON DATA IN A CAMEL ROUTE WITH JACKSON

Similar to XML, Camel offers multiple libraries for working with JSON data. Like JAXB, Jackson is a very popular framework for working with JSON data in Java and this course focuses on
this approach for working with JSON data. To use Jackson with Camel, you need to include the `camel-jackson` library as a dependency of your project.

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-jackson</artifactId>
</dependency>
```

Similar to JAXB, Jackson also provides a set of annotations you use to control the mapping of JSON data into your model classes.
Given the following JSON data:

```
{
"_id": "1",
"value": 5.00,
"tax": 0.50,
"description": "Sample text",
...
```

The following example is a Jackson-annotated model class which you can use to marshal the JSON data:

```
package com.redhat.training;
...
public class Order implements Serializable{
private static final long serialVersionUID = 5851038813219503043L;

@JsonProperty("_id")
private String id;
@JsonIgnore
private String description;
@JsonProperty
private double value;
 @JsonProperty
 private double tax;

}
```

Use the `@JsonProperty` annotation to explicitly declare a field in the model class, and optionally include a name for Jackson to use when marshaling and unmarshaling JSON data for that property.
Use the `@JsonIgnore` annotation to make Jackson ignore a field entirely when marshaling an instance of the model class into JSON data.

For an XML-based Camel route configuration, declare a Jackson `dataFormat` and then unmarshal JSON data, as in this example:

```
<camelContext id="camel"
xmlns="http://camel.apache.org/schema/spring">
<dataFormats>
<json id="jackson" prettyPrint="true" library="Jackson"/>
</dataFormats>
<route>
<from uri="file:inbox"/>
<unmarshal ref="jackson"/>
 <to uri="activemq:queue:itemInput"/>
</route>
</camelContext>
```

In the previous example route, JSON files are consumed from the **inbox** directory, and are unmarshaled into Java objects. Those Java objects are then sent to the **itemInput** queue.

The following is a Java DSL example of using Jackson to marshal JSON data before writing the JSON data to a file in the **outbox** directory:

```
from("queue:activemq:queue:itemInput")
.marshal().json(JsonLibrary.Jackson)
.to("file:outbox")
```

TRANSFORMING DIRECTLY BETWEEN XML AND JSON DATA USING THE CAMEL-XMLJSON MODULE

As discussed previously, Camel supports data formats to perform both XML and JSON-related conversions. However, both of these transformations require a Java model class object either as an input (marshaling) or they produce a Java object as output (unmarshaling). The **camel-xmljson** data format provides the capability to convert data from XML to JSON directly and vice versa without needing to use intermediate Java objects. Directly transforming XML to JSON is preferred as there is significant performance overhead involved in doing extra transformations.

When you use the **camel-xmljson** module, the terminology "marshaling" and "unmarshaling" are not as obvious since there is no Java objects involved. To resolve this, the module defines XML as the high-level format or the equivalent of what your Java model classes typically represent, and JSON as the low-level format more suitable for transmission or storage. This designation is mostly arbitrary for the purpose of defining the marshal and unmarshal terms.

This implies that the terms marshal and unmarshal are defined as follows:

marshaling
Converting from XML to JSON

unmarshaling
Converting from JSON to XML

To use the **XmlJsonDataFormat** class in your Camel routes you need to add the following dependencies to your POM file:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-xmljson</artifactId>
</dependency>

<dependency>
<groupId>xom</groupId>
<artifactId>xom</artifactId>
<version>1.2.5</version>
</dependency>
```

The following example Camel route includes the use of the **XmlJsonDataFormat**:

```
XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();
// From XML to JSON from("direct:marshal")
.marshal(xmlJsonFormat)
.to("direct:json");
// From JSON to XML from("direct:unmarshal")
.unmarshal(xmlJsonFormat)
.to("direct:xml");
```

IMPLEMENTING THE WIRE TAP PATTERN IN A CAMEL ROUTE

The wire tap EIP is an integration pattern that creates a duplicate copy of each message that the route processes and then forwards the duplicate message to a second destination without
interrupting the flow from the source to the original destination. The following diagram illustrates this pattern:
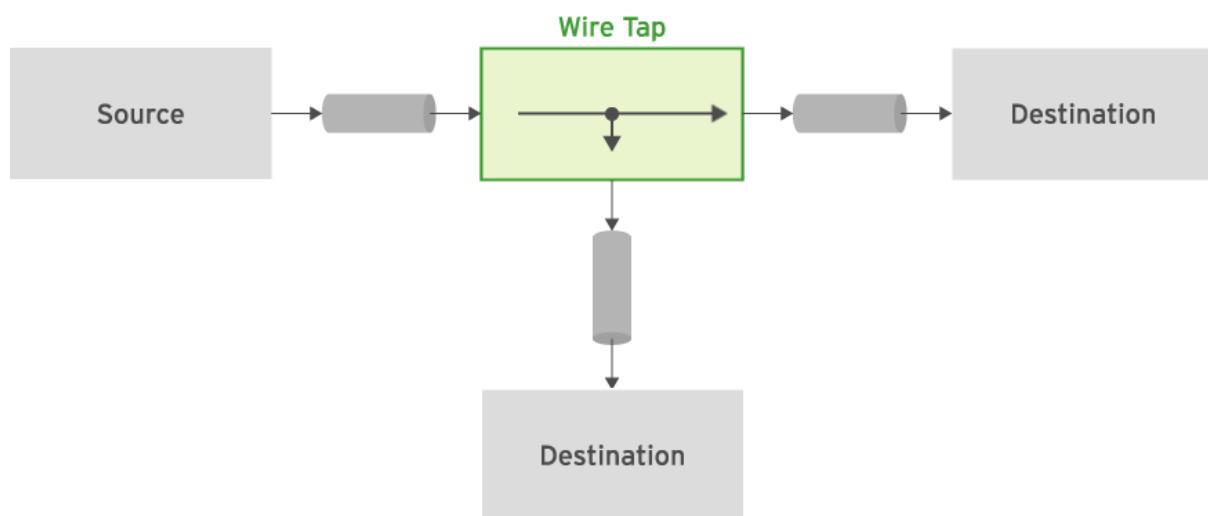
This functionality is most useful for inspecting messages as they travel along a Camel route without impacting the route execution itself. Often Camel developers use wire taps to debug routes by checking the message content matches expectations mid-route. For instance, you could use a wire tap to write message body content to a file mid-route, or notify an external system with real-time updates asynchronously.
To implement this pattern, Camel provides the **wireTap** component, which you can use in your routes to duplicate message exchanges and send a copy of the exchange to a second destination. The following Camel route demonstrates the use of this component:

```
from("activemq:queue:orders.in")
.wireTap("file:backup")
.to("direct:start");
```

In this example, a copy of every exchange received from the **orders.in** queue is written to a file in the **backup** folder using a **wireTap**.

INTRODUCING MOCK ENDPOINTS IN CAMEL ROUTES

During the development of a Camel route, it is often necessary to create a mock destination to represent and behave like an external system with which you have not yet integrated. You can use a mock as a place holder any time you are integrating with other development teams in parallel, and real integration points are not ready.

To use a mock endpoint in a route definition, use the URI prefix `mock,` as the following Spring DSL example demonstrates:

```
<route>
  <from uri="direct:inventoryUpdate"/>
  <to uri="log:com.mycompany.inventory?level=DEBUG"/>
  <to uri="mock:fufillmentSystem"/>
</route>
```

In this example, the mock component is providing a placeholder for an integration with an external fulfillment system that has not been implemented yet.

In addition, mocks can be used when testing route functionality. The mock component provides a powerful declarative testing mechanism, which is similar to jMock or Mockito testing frameworks. Mock endpoints allow you to define expectations for any endpoint before running your unit tests. When the test executes, the Camel route processes messages that are then sent to one or more endpoints. By using a mock you can then assert the expectations defined by your test case were met to ensure the system worked as expected.

This allows you to test various things, for example:

• The correct number of messages are received by each endpoint.

• The correct payloads are received, in the right order.

• The order that the messages arrived at an endpoint using some `Expression` to create an order testing function.

• The messages match some kind of `Predicate` such as containing specific headers with certain values, or that parts of the messages like the body match some predicate, for example an XPath or XQuery expression.

IMPLEMENTING THE MESSAGE TRANSLATOR PATTERN USING TRANSFORM

The message translator pattern provides a mechanism to make updates or transformations to your Camel message exchanges to comply with requirements of the various integration endpoints used by your system. This pattern is usually used to adapt the message payload from one system to another, whether it is an entirely different data format, or just the addition of an extra field, or a tweak to the file delimiter. Message translation is often necessary for integration systems that interface between many different applications, including legacy applications, proprietary software solutions, and external vendor systems. Typically each of these systems use their own data model. This means, for instance, that each system may have a slightly different notion of the *order* entity, including fields that define an *order* and which other entities an *Order* has relationships with. For example, the accounting system may be more interested in the order's sales tax numbers while the fulfillment system stores order item SKUs and quantity.

In Camel, one method of implementing the message translator pattern is using the `transform` DSL method. The `transform` method makes inline changes to the `Message` instance to update its contents or format. In the following code, the route changes all of the commas (,) from a file to semicolons (;) to create a semicolon-separated file:

```
package com.redhat.training;
import org.apache.camel.builder.RouteBuilder; public
class JavaRouteBuilder extends RouteBuilder {
@Override
public void configure() throws Exception {

from("file:in")
  .transform(body()
  .regexReplaceAll(",", ";")
  )
  to("file:out");
}
}
```

transform method call to start the update of file content to a different format
body method call to update the contents from the message's body (the file content). A change to the header is
also possible, calling the **header** method instead
Looks for all commas (,) in the file and changes each occurrence to a semicolon (;)

The **transform** method supports using Simple EL, as shown in the previous example, as well as using the
constant method to overwrite the entire message body, as demonstrated in the following route:

```
from("direct:start")
 .log("Article name is ${body}")
 .choice()
  .when().simple("${body} contains 'Camel'")
    .transform(constant("Yes"))
  .otherwise()
    .transform(constant("No"))
 .end();
 .to("stream:out")
```

In this example, the body of the message is set to either **Yes** or **No** based on whether the message body content
contains the string **Camel**.

INTRODUCING CAMEL-BINDY FOR TRANSLATING CSV FILES TO JAVA OBJECTS

Camel allows marshaling and unmarshaling to and from traditional text formats, such as a CSV or flat file, to Java
objects, using the **camel-bindy** library. Bindy uses a similar approach as JAXB to map Java objects to text format
and vice versa.
The following route transforms the data from a CSV format to a Java object:

```
DataFormat bindy = new
BindyCsvDataFormat("com.redhat.training");
from("file:in")
.unmarshal(bindy)
.to("jms:queue:orders");
```

Scan for classes available at **com.redhat.training** using camel-bindy annotations and unmarshal from CSV
files to Java objects. This class also includes a constructor that takes a single class name instead of a package, as
shown below.

DataFormat bindy = new BindyCsvDataFormat(Order.class)

The **camel-bindy** module processes the contents from a CSV file and transforms them into a
List<Map<String,Object>>. For each position of this **List**, there will be a record in the CSV file, which is
represented as a **Map**.

For each record in the **Map**, there will be a Java object retrieved from the CSV file. This allows the mapping of multiple objects from a record in the CSV file.

Mapping Java object fields to variables using **camel-bindy** is done using annotations similarly to JPA. The following code shows some important annotations.

```
package com.redhat.training;

@CsvRecord(separator=",",crlf="UNIX")
public class Order implements Serializable{

@DataField(pos=1)
private int id;
@DataField(pos=2)
private String description;
@DataField(pos=3)
private double price;
@DataField(pos=4) private double tax;
}
```

The **@CsvRecord** annotation marks the actual class as managed by Bindy. In this case, the annotation specifies that it will parse a CSV file separated by a comma (**separator** parameter) where the line breaks (**crlf** parameter) are UNIX-compliant. The separator parameter accepts one of **WINDOWS**, **MAC**, or **UNIX** as its parameter, allowing you to ensure you are parsing your files correctly based on the operating system platform where the CSV files were created.

The class implements the **java.io.Serializable** interface to allow marshaling processing.

The **DataField** annotation requires a **pos** parameter to map the position where an attribute will be read in a CSV file.

TRANSFORMING XML DATA USING THE XSLT COMPONENT

Another common mechanism for transforming XML data is XSLT (Extensible Stylesheet Language Transformations). XSLT can be used to transform XML, altering elements, values and attributes to match a different schema, or even to transform XML data to another format such as HTML.

In Camel, the xslt component provides the ability to process a message using an XSLT template. The basic URI format of the xslt component is as follows:

xslt:templateName[?options]

Here, **templateName** is the URI to an XSLT template that is available on the class path, or a complete URL to a remote XSLT template. Two examples are listed, as follows:

xslt:file:///tmp/transform.xsl # will use the file located at /tmp/transform.xsl

xslt:http://example.com/xslt/foo.xsl # will use the remote resource at the specified URL

An XSLT endpoint in a route would appear similar to the following:

```
from("activemq:myQueue").
to("xslt:file:///tmp/transform.xsl");
```