6

Using components

This chapter covers

- An overview of Camel components
- Working with files and databases
- Messaging with JMS
- Networking with Netty
- Working with databases
- In-memory messaging
- Automating tasks with the Quartz and Scheduler components
- Sending and receiving email

So far, we've touched on only a handful of ways that Camel can communicate with external applications, and we haven't gone into much detail on most components. It's time to take your use of the components you've already seen to the next level, and to introduce new components that will enable your Camel applications to communicate with the outside world.

First, we'll discuss exactly what it means to be a component in Camel. We'll also show you how components are added to Camel. Then, although we can't describe every component—that would at least triple the length of this book—we'll present the most commonly used ones.

Table 6.1 lists the components covered in this chapter and the URLs for their official documentation.

Table 6.1 Components covered in this chapter

Component function	Component	Camel documentation reference
File I/O	File	http://camel.apache.org/file2.html
	FTP	http://camel.apache.org/ftp2.html
Asynchronous messaging	JMS	http://camel.apache.org/jms.html
Networking	Netty4	http://camel.apache.org/netty4.html
Working with databases	JDBC	http://camel.apache.org/jdbc.html
	JPA	http://camel.apache.org/jpa.html
In-memory messaging	Direct	http://camel.apache.org/direct.html
	Direct-VM	http://camel.apache.org/direct-vm.html
	SEDA	http://camel.apache.org/seda.html
	VM	http://camel.apache.org/vm.html
Automating tasks	Scheduler	http://camel.apache.org/scheduler.html
	Quartz2	http://camel.apache.org/quartz2.html

Let's start with an overview of Camel components.

6.1 Overview of Camel components

Components are the primary extension point in Camel. Over the years since Camel's inception, the list of components has grown. As of version 2.20.1, Camel ships with more than 280 components, and dozens more are available separately from other community sites.[1] These components allow you to bridge to many APIs, protocols, data formats, and so on. Camel saves you from having to code these integrations yourself; thus it achieves its primary goal of making integration easier.

---

[1] See appendix B for information on some of these community sites.

What does a Camel component look like? Well, if you think of Camel routes as highways, components are roughly analogous to on- and off-ramps. A message that travels down a route needs to take an off-ramp to get to another route or external service. If the message is headed for another route, it then needs to take an on-ramp to get onto that route.

From an API point of view, a Camel component is simple, consisting of a class implementing the `Component` interface, shown here:

```
public interface Component extends CamelContextAware {
    Endpoint createEndpoint(String uri) throws Exception;
    boolean useRawUri();
}
```

The main responsibility of a component is to be a factory for endpoints. To do this, a component also needs to extend `CamelContextAware`, which means it holds a reference to `CamelContext`. `CamelContext` provides access to Camel's common facilities, such as the registry, class loader, and type converters. This relationship is shown in figure 6.1.

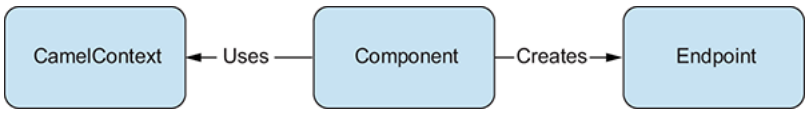

Figure 6.1 A component creates endpoints and may use the `CamelContext`'s facilities to accomplish this.

Components are added to a Camel runtime in two main ways: by manually adding them to `CamelContext` and through autodiscovery.

6.1.1 Manually adding components

You've seen the manual addition of a component already. In chapter 2, you had to add a configured JMS component to `CamelContext` to use `ConnectionFactory`. This was done using the `addComponent` method of the `CamelContext` interface, as follows:

```
CamelContext context = new DefaultCamelContext();
context.addComponent("jms",
    JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));
```

In this example, you add a component created by the `JmsComponent.jmsComponent-AutoAcknowledge` method and assign it a name of `jms`. This component can be selected in a URI by using the `jms` scheme.

6.1.2 Autodiscovering components

The other way components can be added to Camel is through autodiscovery. The autodiscovery process is illustrated in figure 6.2.

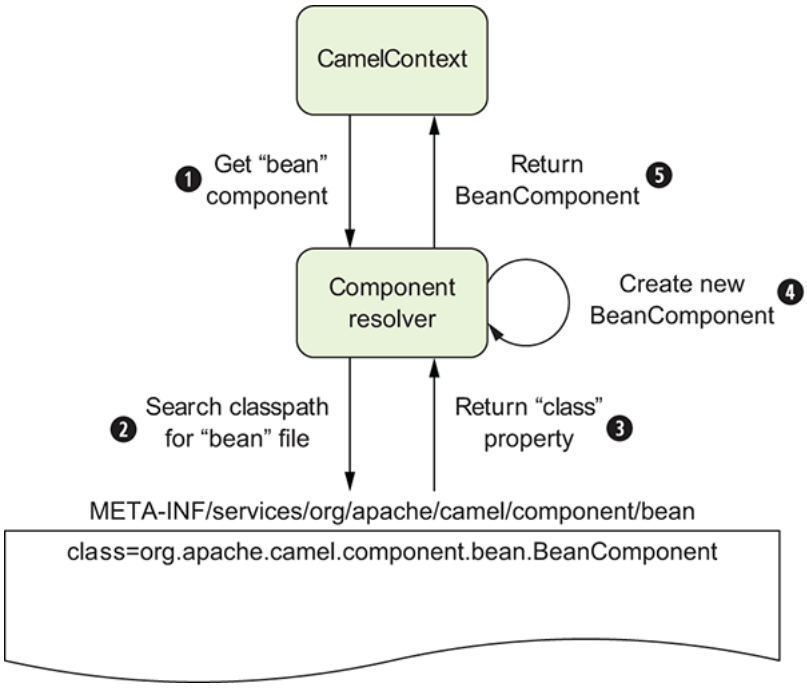

Figure 6.2 To autodiscover a component named "bean," the component resolver searches for a file named "bean" in a specific directory on the classpath. This file specifies that the component class that will be created is `BeanComponent`.

Autodiscovery is the way the components that ship with Camel are registered. To discover new components, Camel looks in the META-INF/services/org/apache/camel/component directory on the classpath for

files. Files in this directory determine the name of a component and the fully qualified class name.

As an example, let's look at the Bean component. It has a file named bean in the META-INF/services/org/apache/camel/component directory that contains a single line:

```
class=org.apache.camel.component.bean.BeanComponent
```

This `class` property tells Camel to load up the `org.apache.camel.component.bean.BeanComponent` class as a new component, and the filename gives the component the name of Bean.

TIP We discuss how to create your own Camel component in section 8.4 in chapter 8.

Most of the components in Camel are in separate Maven modules from the camel-core module, because they usually depend on third-party dependencies that would bloat the core. For example, the Atom component depends on Apache Abdera to communicate over Atom. You wouldn't want to make every Camel application depend on Abdera, so the Atom component is included in a separate camel-atom module.

The camel-core module has 24 useful components built in, though. These are listed in table 6.2.

Table 6.2 Components in the camel-core module

Component	Description	Camel documentation reference
Bean	Invokes a Java bean in the registry. You saw this used extensively in chapter 4.	http://camel.apache.org/bean.html
Browse	Allows you to browse the list of exchanges that passed through a browse endpoint. This can be useful for testing, visualization, or debugging.	http://camel.apache.org/browse.html
Class	Creates a new Java bean based on a class name and invokes the bean similar to the bean component.	http://camel.apache.org/class.html
ControlBus	Allows you to control the lifecycle of routes and gather performance	http://camel.apache.org/controlbus-component.ht

Component	Description	Camel documentation reference
	statistics by sending messages to a ControlBus endpoint. Based on the Control Bus EIP pattern.	
DataFormat	A convenience component that allows you to invoke a data format as a component.	http://camel.apache.org/dataformat-component.html
DataSet	Allows you to create large numbers of messages for soak or load testing.	http://camel.apache.org/dataset.html
Direct	Allows you to synchronously call another endpoint with little overhead. Section 6.6 covers this component.	http://camel.apache.org/direct.html
Direct-VM	Allows you to synchronously call another	http://camel.apache.org/direct-vm.html

Component	Description	Camel documentation reference
	endpoint in the same JVM. Section 6.6 covers this component.	
File	Reads or writes to files. Section 6.2 covers this component.	http://camel.apache.org/file2.html
Language	Executes a script against the incoming exchange by using one of the languages supported by Camel.	http://camel.apache.org/language.html
Log	Logs messages to various logging providers.	http://camel.apache.org/log.html
Mock	Tests that messages flow through a route as expected. You'll see the Mock component in action in chapter 9.	http://camel.apache.org/mock.html

Component	Description	Camel documentation reference
Properties	Allows you to use property placeholders in endpoint URIs. You've seen this already in section 2.5.2.	http://camel.apache.org/properties.html
Ref	Looks up endpoints in the registry.	http://camel.apache.org/ref.html
REST	Used for hosting or calling REST services. See section 10.2 for more information on how this is used with Camel's Rest DSL.	http://camel.apache.org/rest.html
REST API	Used to provide Swagger API docs for REST endpoints created with Camel's Rest DSL. Covered in section 10.3.	https://github.com/apache/camel/blob/master/cam core/src/main/docs/rest-api-component.adoc

Component	Description	Camel documentation reference
SEDA	Allows you to asynchronously call another endpoint in the same `CamelContext`. Section 6.6 covers this component.	http://camel.apache.org/seda.html
Scheduler	Sends out messages at regular intervals. You'll learn more about the Scheduler component and a more powerful scheduling endpoint based on Quartz in section 6.7.	http://camel.apache.org/scheduler.html
Stub	Allows you to stub out real endpoint URIs for development or testing purposes.	http://camel.apache.org/stub.html

Component	Description	Camel documentation reference
Test	Tests that messages flowing through a route match expected messages pulled from another endpoint.	http://camel.apache.org/test.html
Timer	Sends out messages at regular intervals.	http://camel.apache.org/timer.html
Validator	Validates the message body by using the JAXP Validation API.	http://camel.apache.org/validation.html
VM	Allows you to asynchronously call another endpoint in the same JVM. Section 6.6 covers this component.	http://camel.apache.org/vm.html
XSLT	Transforms a message by using an XSLT template.	http://camel.apache.org/xslt.html

Now let's look at each component from table 6.1 in detail. We'll start with the File component.

6.2 Working with files: File and FTP components

It seems that in integration projects, you always end up needing to interface with a filesystem somewhere. You may find this strange, as new systems often provide nice web services and other remoting APIs to serve as integration points. The problem is that in integration, you often have to deal with older legacy systems, and file-based integrations are common.

For example, you might need to read a file that was written by another application—it could be sending a a command to be executed, an order to be processed, data to be logged, or anything else. This kind of information exchange, illustrated in figure 6.3, is called a *file transfer* in EIP terms.

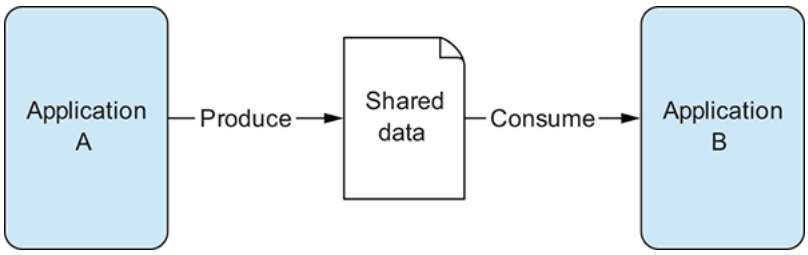

Figure 6.3 A file transfer between two applications is a common way to integrate with legacy systems.

Another reason that file-based integrations are so common is that they're easy to understand. Even novice computer users know something about filesystems.

Even though they're easy to understand, file-based integrations are difficult to get right. Developers commonly have to battle with complex I/O APIs, platform-specific filesystem issues, different file encodings, concurrent access, and the like.

Camel has extensive support for interacting with filesystems. This section covers how to use the File component to read files from and write them to the local filesystem. This section also covers advanced options for file processing and ways to access remote files with the FTP component.

6.2.1 Reading and writing files with the File component

As you saw before, the File component is configured through URI options. Table 6.3 shows some common options; for a complete listing, see the on-

line documentation (http://camel.apache.org/file2.html).

Table 6.3 Common URI options used to configure the File component

Option	Default value	Description
`delay`	`500`	Specifies the number of milliseconds between polls of the directory.
`recursive`	`false`	Specifies whether to recursively process files in all subdirectories of this directory.
`noop`	`false`	Specifies file-moving behavior. By default, Camel moves files to the .camel directory after processing them. To stop this behavior and keep the original files in place, set the `noop` option to `true`.
`fileName`		Uses an expression to set the filename used. For consumers, this acts as a filename filter; in producers, it's used to set the name of the file being written.
`fileExist`	`Override`	Specifies what a file producer will do if the same filename already exists. Valid options are `Override`, `Append`, `Fail`, `Ignore`, `Move`, and `TryRename`. `Override` causes the file to be replaced. `Append` adds content to the file. `Fail` causes an exception to be thrown. If `Ignore` is set, an exception won't be thrown, and the file won't be written. `Move` causes the existing file to be moved. `Move` also requires the `moveExisting` option to be set to an expression used to compute the filename to move to. `TryRename` causes the file rename to be attempted. `TryRename`

Option	Default value	Description
		applies only to temporary files specified by the `tempFileName` option.
`delete`	`false`	Specifies whether Camel will delete the file after processing. By default, Camel won't delete the file.
`move`	`.camel`	Specifies the directory to which Camel moves files after it's done processing them.
`include`		Specifies a regular expression. Camel processes only those files that match this expression. For example, `include=.*xml$` would include all files with the .xml extension.
`exclude`		Specifies a regular expression. Camel excludes files based on this expression.

Let's first see how Camel can be used to read files.

Reading files

As you've seen in previous chapters, reading files with Camel is straightforward. Here's a simple example:

```
public void configure() {
    from("file:data/inbox?noop=true").to("stream:out");
}
```

This route reads files from the data/inbox directory and prints the contents of each to the console. The printing is done by sending the message to the `System.out` stream, accessible by using the Stream component. As stated in table 6.3, the `noop` flag tells Camel to leave the original files as

is. This is a convenience option for testing, because it means that you can run the route many times without having to repopulate a directory of test files.

To run this yourself, change to the chapter6/file directory in the book's source code and run this command:

```
mvn test -Dtest=FilePrinterTest
```

What if you remove the `noop` flag and change the route to the following?

```
public void configure() {
    from("file:data/inbox").to("stream:out");
}
```

This uses Camel's default behavior, which is to move the consumed files to a special camel directory (though the directory can be changed with the `move` option); the files are moved after the routing has completed. This behavior was designed so that files wouldn't be processed over and over, but it also keeps the original files around in case something goes wrong. If you don't mind losing the original files, you can use the `delete` option listed in table 6.3.

By default, Camel also locks any files that are being processed. The locks are released after routing is complete.

Both of the two preceding routes consume any file not beginning with a period, so they ignore files such as .camel, .m2, and so on. You can customize which files are included by using the `include` and `exclude` options.

Writing files

You just saw how to read files created by other applications or users. Now let's see how Camel can be used to write files. Here's a simple example:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:"/>
  <to uri="file:data/outbox"/>
</route>
```

This example uses the Stream component to accept input from the console. The `stream:in` URI instructs Camel to read any input from `System.in` on the console and create a message from that. The `promptMessage` option displays a prompt, so you know when to enter text. The `file:data/outbox` URI instructs Camel to write out the message body to the data/outbox directory.

To see what happens firsthand, you can try the example by changing to the chapter6/file directory in the book's source code and executing the following command:

```
mvn camel:run
```

When this runs, you'll see an `Enter something:` prompt. Enter text into the console and press Enter, like this:

```
Enter something:Hello
```

The example keeps running until you press Ctrl-C. The text (in this case, `Hello`) is read in by the Stream component and added as the body of a new message. This message's body (the text you entered) is then written out to a file in the data/outbox directory (which will be created if it doesn't exist).

If you run a directory listing on the data/outbox directory now, you'll see a single file that has a rather strange name:

```
ID-ghost-43901-1489018386363-0-1
```

Because you didn't specify a filename to use, Camel chose a unique filename based on the message ID.

To set the filename that should be used, you can add a `fileName` option to your URI. For example, you could change the route so it looks like this:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:"/>
  <to uri="file:data/outbox?fileName=prompt.txt"/>
</route>
```

Now, any text entered into the console will be saved into the prompt.txt file in the data/outbox directory.

Camel will by default overwrite prompt.txt, so you now have a problem with this route. If text is frequently entered into the console, you may want new files created each time, so they don't overwrite the old ones. To implement this in Camel, you can use an expression for the filename. You can use the Simple expression language to put the current time and date information into your filename:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:"/>
  <to uri="file:data/outbox?fileName=${date:now:yyyyMMdd-hh:mm:ss}.txt"/
</route>
```

The `date:now` expression returns the current date, and you can also use any formatting options permitted by `java.text. SimpleDataFormat`.

Now if you enter text into the console at 2:00 p.m. on January 10, 2009, the file in the data/outbox directory will be named something like this:

```
20090110-02:00:53.txt
```

The simple techniques for reading from and writing to files discussed here will be adequate for most of the cases you'll encounter in the real world. For the trickier cases, many configuration possibilities are listed in the online documentation.

We've started slowly with the File component, to get you comfortable with using components in Camel. Next we present the FTP component, which builds on the File component but introduces messaging across a network. After that, we'll get into more complex topics.

6.2.2 Accessing remote files with the FTP component

Probably the most common way to access remote files is by using FTP, and Camel supports three flavors of FTP:

- Plain FTP mode transfer
- Secure FTP (SFTP) for secure transfer

- FTP Secure (FTPS) for transfer with the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) cryptographic protocols enabled

The FTP component inherits all the features and options of the File component, and it adds a few more options, as shown in table 6.4. For a complete listing of options for the FTP component, see the online documentation (http://camel.apache.org/ftp2.html).

Table 6.4 Common URI options used to configure the FTP component

Option	Default value	Description
`username`		Provides a username to the remote host for authentication. If no username is provided, anonymous login is attempted. You can also specify the username by prefixing `username@` to the hostname in the URI.
`password`		Provides a password to the remote host to authenticate the user. You can also specify the password by prefixing the hostname in the URI with `username:password@`.
`binary`	`false`	Specifies the transfer mode. By default, Camel transfers in ASCII mode; set this option to `true` to enable binary transfer.
`disconnect`	`false`	Specifies whether Camel will disconnect from the remote host right after use. The default is to remain connected.
`maximumReconnectAttempts`	`3`	Specifies the maximum number of attempts

Option	Default value	Description
		Camel will make to connect to the remote host. If all these attempts are unsuccessful, Camel will throw an exception. A value of `0` disables this feature.
`reconnectDelay`	`1000`	Specifies the delay in milliseconds between reconnection attempts.

Because the FTP component isn't part of the camel-core module, you need to add a dependency to your project. If you use Maven, you add the following dependency to your POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-ftp</artifactId>
  <version>2.20.1</version>
</dependency>
```

To demonstrate accessing remotes files, let's use the Stream component as in the previous section to interactively generate and send files over FTP. A route that accepts text on the console and then sends it over FTP would look like this:

```
<route>
  <from uri="stream:in?promptMessage=Enter something:" />
  <to uri="ftp://rider:secret@localhost:21000/target/data/outbox"/>
</route>
```

This is a Spring-based route; Spring makes it easy to hook start and stop methods to an embedded FTP server. This FTP endpoint URI specifies that Camel should send the message to an FTP server on the `localhost` listen-

ing on port `21000`, using `rider` as the username and `secret` as the password. It also specifies that messages are to be stored in the data/outbox directory of the FTP server.

To run this example, change to the chapter6/ftp directory and run this command:

```
mvn camel:run
```

After Camel has started, you need to enter something into the console:

```
Enter something:Hello
```

The example keeps running until you press Ctrl-C.

You can now check to see whether the message made it into the FTP server. The FTP server's root directory was set up to be the current directory of the application, so you can check data/outbox for a message:

```
$ cat target/data/outbox/ID-ghost-43901-1489018386363-0-1
Hello
```

As you can see, using the FTP component is similar to using the File component.

Now that you know how to do the most basic of integrations with files and FTP, let's move on to more advanced topics, such as JMS and web services.

6.3 Asynchronous messaging: JMS component

JMS messaging is an incredibly useful integration technology. It promotes loose coupling in application design, has built-in support for reliable messaging, and is by nature asynchronous. As you saw in chapter 2, when you looked at JMS, it's also easy to use from Camel. This section expands on the coverage in chapter 2 by going over some of the more commonly used configurations of the JMS component.

Camel doesn't ship with a JMS provider; you need to configure Camel to use a specific JMS provider by passing in a `ConnectionFactory` instance.

For example, to connect to an Apache ActiveMQ broker listening on port 61616 of the local host, you could configure the JMS component like this:

```
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://localhost:61616"/>
    </bean>
  </property>
</bean>
```

The tcp://localhost:61616 URI passed in to `ConnectionFactory` is JMS provider-specific. In this example, you're using the `ActiveMQConnectionFactory`, so the URI is parsed by ActiveMQ. The URI tells ActiveMQ to connect to a broker by using TCP on port 61616 of the local host.

If you want to connect to a broker over another protocol, ActiveMQ supports connections over VM, SSL, UDP, multicast, MQTT, AMQP, and so on. Throughout this section, we'll demonstrate JMS concepts using ActiveMQ as the JMS provider, but any provider could be used here.

THE ACTIVEMQ COMPONENT

By default, a JMS `ConnectionFactory` doesn't pool connections to the broker, so it spins up new connections for every message. The way to avoid this is to use connection factories that use connection pooling.

For convenience to Camel users, ActiveMQ ships with the ActiveMQ component, which automatically configures connection pooling for improved performance. The ActiveMQ component is used as follows:

```
<bean id="activemq"
  class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

When using this component, you also need to depend on the activemq-camel module from ActiveMQ:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
  <version>5.15.2</version>
</dependency>
```

This module contains the Camel ActiveMQ component.

Camel's JMS component has a daunting list of configuration options—more than 80 to date. Many of these are seen in only specific JMS usage scenarios. The common ones are listed in table 6.5.

TIP Apache Kafka is another popular asynchronous messaging project. We cover this in section 17.4 of chapter 17.

To use the JMS component in your project, you need to include the camel-jms module on your classpath as well as any JMS provider JARs. If you're using Maven, the JMS component can be added with the following dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
  <version>2.20.1</version>
</dependency>
```

Table 6.5 Common URI options used to configure the JMS component

Option	Default value	Description
`clientId`		Sets the JMS client ID, which must be unique among all connections to the JMS broker. The client ID set in `ConnectionFactory` overrides this one if set. Needed only when using durable topics.
`concurrentConsumers`	`1`	Sets the number of consumer threads to use. It's a good idea to increase this for high-volume queues, but it's not advisable to use more than one concurrent consumer for JMS topics, because this will result in multiple copies of the same message.
`disableReplyTo`	`false`	Specifies whether Camel should ignore the `JMSReplyTo` header in any messages. Set this if you don't want Camel to send a reply back to the destination specified in the `JMSReplyTo` header.
`durableSubscriptionName`		Specifies the name of the durable topic subscription.

Option	Default value	Description
		The `clientId` option must also be set.
`maxConcurrentConsumers`	`1`	Sets the maximum number of consumer threads to use. If this value is higher than `concurrentConsumers`, new consumers are started dynamically as load demands. If load drops, these extra consumers will be freed and the number of consumers will be equal to `concurrentConsumers` again. Increasing this value isn't advisable when using topics.
`replyTo`		Sets the destination that the reply is sent to. This overrides the `JMSReplyTo` header in the message. By setting this, Camel will use a fixed reply queue. By default, Camel uses a temporary reply queue.
`replyToConcurrentConsumers`	`1`	Sets the number of threads to use for request-reply style messaging. This

Option	Default value	Description
		value is separate from `concurrentConsumers`.
`replyToMaxConcurrentConsumers`	`1`	Sets the maximum number of threads to use for request-reply style messaging. This value is separate from `maxConcurrentConsumers`.
`requestTimeout`	`20000`	Specifies the time in milliseconds before Camel will time out when sending a message in request-reply mode. You can override the endpoint value by setting the `CamelJmsRequestTimeout` header on a message.
`selector`		Sets the JMS message selector expression. Only messages passing this predicate will be consumed.
`transacted`	`false`	Enables transacted sending and receiving of messages in `InOnly` mode.

THE SJMS COMPONENT

The Camel JMS component is built on top of the Spring JMS library, so many of the options map directly to a Spring `org.springframework.jms.listener.AbstractMessageListenerContainer` used under the hood. Furthermore, the dependency on Spring brings in a whole set of other Spring JARs. For memory-sensitive deployments, or those where you aren't using Spring at all, Camel provides the SJMS component. The *S* in *SJMS* stands for *Simple* and *Spring-less*.

To use this component, you need to add camel-sjms2 to your Maven pom.xml:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-sjms2</artifactId>
  <version>2.20.1</version>
</dependency>
```

The best way to show that Camel is a great tool for JMS messaging is with an example. Let's look at how to send and receive messages over JMS.

6.3.1 Sending and receiving messages

In chapter 2, you saw how orders are processed at Rider Auto Parts. The process started out as a step-by-step procedure: orders were first sent to accounting to validate the customer standing and then to production for manufacture. This process was improved by sending orders to accounting and production at the same time, cutting out the delay involved when production waited for the okay from accounting. A multicast EIP was used to implement this scenario.

Figure 6.4 illustrates another possible solution: using a JMS topic following a publish-subscribe model. In that model, listeners such as accounting and production can subscribe to the topic, and new orders are published to the topic. In this way, both accounting and production receive a copy of the order message.

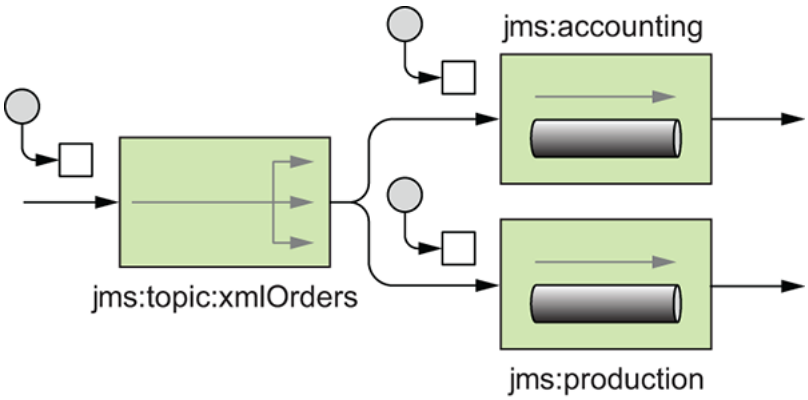

Figure 6.4 Orders are published to the xmlOrders topic, and the two subscribers (the accounting and production queues) get a copy of the order.

To implement this in Camel, you set up two consumers, which means two routes are needed:

```
from("jms:topic:xmlOrders").to("jms:accounting");
from("jms:topic:xmlOrders").to("jms:production");
```

When a message is sent (published) to the xmlOrders topic, both the accounting and production queues receive a copy.

As you saw in chapter 2, an incoming order could originate from another route (or set of routes), such as one that receives orders via a file, as shown in the following listing.

Listing 6.1 Topics allow multiple receivers to get a copy of the message

```
from("file:src/data?noop=true").to("jms:incomingOrders");
from("jms:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
        .to("jms:topic:xmlOrders")    (1)

        .when(header("CamelFileName").regex("^.*(csv|csl)$"))
            .to("jms:topic:csvOrders");
from("jms:topic:xmlOrders").to("jms:accounting");    (2)
```

(1) XML orders are routed to xmlOrders topic

(2) Both listening queues get copies

```
from("jms:topic:xmlOrders").to("jms:production");   ❷
```

To run this example, go to the chapter6/jms directory in the book's source and run this command:

```
mvn camel:run
```

This outputs the following on the command line:

```
Accounting received order: message1.xml
Production received order: message1.xml
```

Why do you get this output? Well, you have a single order file named message1.xml, and it's published to the xmlOrders topic. Both the accounting and production queues are subscribed to the topic, so each receives a copy. Testing routes consume the messages on those queues and output the messages.

To send an order to the topic by using `ProducerTemplate`, you could use the following snippet:

```
ProducerTemplate template = camelContext.createProducerTemplate();
template.sendBody("jms:topic:xmlOrders", "<?xml ...");
```

This is a useful feature for getting direct access to any endpoint in Camel.

All the JMS examples so far have been one-way only. Let's look at how to deliver a reply to the sent message.

6.3.2 Request-reply messaging

JMS messaging with Camel (and in general) is asynchronous by default. Messages are sent to a destination, and the client doesn't wait for a reply. But at times it's useful to be able to wait and get a reply after sending to a destination. One obvious application is when the JMS destination is a front end to a service—in this case, a client sending to the destination would be expecting a reply from the service.

JMS supports this type of messaging by providing a `JMSReplyTo` header, so that the receiver knows where to send the reply, and a `JMSCorrelationID`, used to match replies to requests if multiple replies are awaiting. This flow of messages is illustrated in figure 6.5.

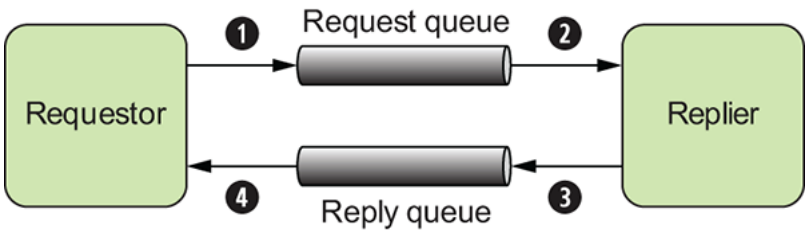

Figure 6.5 In request-reply messaging, a requestor sends a message to a request queue ❶ and then waits for a reply in the reply queue ❹. The replier waits for a new message in the request queue ❷, inspects the `JMSReplyTo` address, and then sends a reply back to that destination ❸.

Camel takes care of this style of messaging so you don't have to create special reply queues, correlate reply messages, and the like. By changing the message exchange pattern (MEP) to `InOut`, Camel will enable request-reply mode for JMS.

To demonstrate, let's take a look at an order validation service within the Rider Auto Parts back-end systems that checks orders against the company database to make sure the parts listed are actual products. This service is exposed via a queue named *validate*. The route exposing this service over JMS could be as simple as this:

```
from("jms:validate").bean(ValidatorBean.class);
```

When calling this service, you need to tell Camel to use request-reply messaging by setting the MEP to `InOut`. You can use the `exchangePattern` option to set this as follows:

```
from("jms:incomingOrders").to("jms:validate?exchangePattern=InOut")...
```

You can also specify the MEP by using the `inOut` DSL method:

```
from("jms:incomingOrders").inOut().to("jms:validate")...
```

With the `inOut` method, you can even pass in an endpoint URI as an argument, which shortens your route:

```
from("jms:incomingOrders").inOut("jms:validate")...
```

By specifying an `InOut` MEP, Camel will send the message to the validate queue and wait for a reply on a temporary queue that it creates automatically. When the `ValidatorBean` returns a result, that message is propagated back to the temporary reply queue, and the route continues on from there.

Rather than using temporary queues, you can explicitly specify a reply queue. You can do that by setting the `JMSReplyTo` header on the message or by using the `replyTo` URI option described in table 6.5.

A handy way of calling an endpoint that can return a response is by using the request methods of the `ProducerTemplate`. For example, you can send a message into the `incomingOrders` queue and get a response back with the following call:

```
Object result = template.requestBody("jms:incomingOrders",
    "<order name=\"motor\" amount=\"1\" customer=\"honda\"/>");
```

This returns the result of the `ValidatorBean`.

To try this out, go to the chapter6/jms directory in the book's source and run this command:

```
mvn test -Dtest=RequestReplyJmsTest
```

The command runs a unit test demonstrating request-reply messaging as discussed in this section.

In the JMS examples you've looked at so far, several data mappings have been happening behind the scenes—mappings that are necessary to conform to the JMS specification. Camel could be transporting any type of data, so that data needs to be converted to a type that JMS supports. We'll look into this next.

6.3.3 Message mappings

Camel hides a lot of the details when doing JMS messaging, so you don't have to worry about them. But one detail you should be aware of is that

Camel maps both bodies and headers from the arbitrary types and names allowed in Camel to JMS-specific types.

Body mapping

Although Camel poses no restrictions on the content of a message body, JMS specifies different message types based on the body type. Figure 6.6 shows the five concrete JMS message implementations.

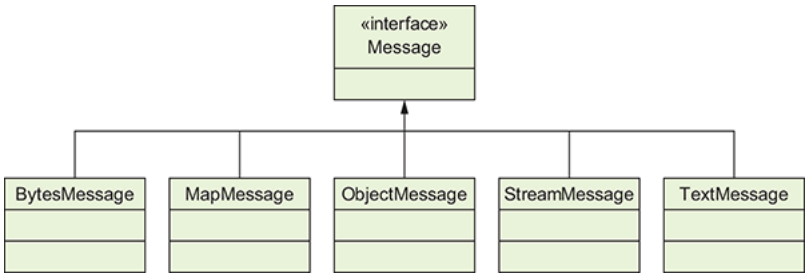

Figure 6.6 The `javax.jms.Message` interface has five implementations, each of which is built for a different body type.

The conversion to one of these five JMS message types occurs when the exchange reaches a JMS producer; said another way, it happens when the exchange reaches a route node like this:

```
to("jms:jmsDestinationName")
```

At this point, Camel examines the body type and determines which JMS message to create. This newly created JMS message is then sent to the JMS destination specified.

Table 6.6 shows what body types are mapped to JMS messages.

Table 6.6 When sending messages to a JMS destination, Camel body types are mapped to specific JMS message types

Camel body type
`String`, `org.w3c.dom.Node`
`byte[]`, `java.io.File`, `java.io.Reader`, `java.io.InputStream`, `java.nio.ByteB`
`java.util.Map`
`java.io.Serializable`

Another conversion happens when consuming a message from a JMS destination. Table 6.7 shows the mappings in this case.

Table 6.7 When receiving messages from a JMS destination, JMS message types are mapped to Camel body types

JMS message type	Camel body type
`TextMessage`	`String`
`BytesMessage`	`byte[]`
`MapMessage`	`java.util.Map`
`ObjectMessage`	`Object`
`StreamMessage`	No mapping occurs

Although this automatic message mapping allows you to fully use Camel's transformation and mediation abilities, you may sometimes need to keep the JMS message intact. An obvious reason is to increase performance—not mapping every message means it takes less time for each message to be processed. Another reason could be that you're storing an object type that doesn't exist on Camel's classpath. In this case, if Camel tried to deserialize the object, it would fail when finding the class.

TIP You can also implement your own custom Spring `org.springframework.jms.support.converter.MessageConverter` by using the `messageConverter` option.

To disable message mapping for body types, set the `mapJmsMessage` URI option to `false`.

Header mapping

Headers in JMS are even more restrictive than body types. In Camel, a header can be named anything that will fit in a Java `String`, and its value can be any Java object. This presents a few problems when sending to and receiving from JMS destinations.

These are the restrictions in JMS:

- Header names that start with `JMS` are reserved; you can't use these header names.
- Header names must be valid Java identifiers.
- Header values can be any primitive type and their corresponding object types. These include `boolean`, `byte`, `short`, `int`, `long`, `float`, and `double`. Valid object types include `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, and `String`.

To handle these restrictions, Camel does several things. First, any headers that you set starting with `JMS` are dropped before sending to a JMS destination. Camel also attempts to convert the header names to be JMS-compliant. Any period (`.`) characters are replaced by `_DOT_`, and any hyphens (`-`) are replaced with `_HYPHEN_`. For example, a header named `org.apache.camel.Test-Header` would be converted to `org_DOT_apache_DOT_camel_DOT_Test_HYPHEN_Header` before being sent to a JMS destination. If this message is consumed by a Camel route at some point down the line, the header name will be converted back.

To conform to the JMS specification, Camel drops any header that has a value not listed in the list of primitives or their corresponding object types. Camel also allows `CharSequence`, `Date`, `BigDecimal`, and `BigInteger` header values, all of which are converted to their `String` representations to conform to the JMS specification.

You should now have a good grasp of what Camel can do for your JMS messaging applications. Several types of messaging that we've looked at before, such as JMS and FTP, run on top of other protocols. Let's look at using Camel for these kinds of low-level communications.

6.4 Networking: Netty4 component

So far in this chapter, you've seen a mixture of old integration techniques (such as file-based integration) and newer technologies (such as JMS). All these can be considered essential in any integration framework. Another essential mode of integration is using low-level networking protocols, such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Even if you haven't heard of these protocols before, you've definitely used them—protocols such as email, FTP, and HTTP run on top of TCP.

To communicate over these and other protocols, Camel uses Netty and Apache MINA. Both Netty and MINA are networking frameworks that provide asynchronous event-driven APIs and communicate over various protocols including TCP and UDP. In this section, we use Netty to demonstrate low-level network communication with Camel.

The Netty4 component is located in the camel-netty4 module of the Camel distribution. You can access this by adding it as a dependency to your Maven POM like this:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-netty4</artifactId>
  <version>2.20.1</version>
</dependency>
```

The most common configuration options are listed in table 6.8.

Table 6.8 Common URI options used to configure the Netty4 component

Option	Default value	Description
`decoder`		Specifies the bean used to marshal the incoming message body. It must extend from `io.netty.channel.ChannelInboundHandlerAdapter` be loaded into the registry, and referenced using the `#beanName` style.
`decoders`		Specifies a list of Netty `io.netty.channel.ChannelInboundHandlerAdapter` beans to use to marshal the incoming message body. should be specified as a comma-separated list of bea references (for example, `"#decoder1,#decoder2"`).
`encoder`		Specifies the bean used to marshal the outgoing message body. It must extend from `io.netty.channel.ChannelOutboundHandlerAdapte` be loaded into the registry, and referenced using the `#beanName` style.
`encoders`		Specifies a list of Netty `io.netty.channel.ChannelOutboundHandlerAdapte` beans to use to marshal the outgoing message body. should be specified as a comma-separated list of bea references (for example, `"#encoder1,#encoder2"`).
`textline`	`false`	Enables the `textline` codec when you're using TCP and no other codec is specified. The `textline` codec understands bodies that have string content and enc with a line delimiter.
`delimiter`	`LINE`	Sets the delimiter used for the `textline` codec. Possible values include `LINE` and `NULL`.

Option	Default value	Description
`sync`	`true`	Sets the synchronous mode of communication. Clien will be able to get a response back from the server.
`requestTimeout`	`0`	Sets the time in milliseconds to wait for a response from a remote server. By default, there's no time-out
`encoding`	JVM default	Specifies the `java.nio.charset.Charset` used to encode the data.

In addition to the URI options, you have to specify the transport type and port you want to use. In general, a Netty4 component URI looks like this,

```
netty4:transport://hostname:port[?options]
```

where `transport` is one of `tcp` or `udp`.

Let's now see how to use the Netty4 component to solve a problem at Rider Auto Parts.

6.4.1 Using Netty for network programming

Back at Rider Auto Parts, the production group has been using automated manufacturing robots for years to assist in producing parts. What they've been lacking, though, is a way of tracking the whole plant's health from a single location. They have floor personnel manually monitoring the machines. What they'd like to have is an operations center with a single-screen view of the entire plant.

To accomplish this, they've purchased sensors that communicate machine status over TCP. The new operations center needs to consume these messages over JMS. Figure 6.7 illustrates this setup.

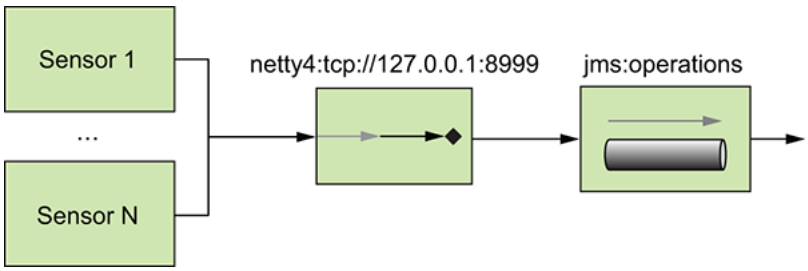

Figure 6.7 Sensors feed status messages over TCP to a server, which then forwards them to a JMS operations queue.

Hand-coding a TCP server such as this wouldn't be a trivial exercise. You'd need to spin up new threads for each incoming socket connection, as well as transform the body to a format suitable for JMS. Not to mention the pain involved in managing the low-level networking protocols.

In Camel, a possible solution is accomplished with a single line:

```
from("netty4:tcp://localhost:8999?textline=true&sync=false")
    .to("jms:operations");
```

Here you set up a TCP server on port 8999 by using Netty, and it parses messages by using the `textline` codec. The `sync` property is set to `false` to make this route `InOnly` —any clients sending a message won't get a reply back.

NOTE If you set `sync=true`, the route will become `InOut` and return a result to the caller. The `transform` DSL method is handy for setting the return value. See chapter 3 for more details about this.

You may be wondering what a `textline` codec is, and maybe even what a codec is! In TCP communications, a single message payload going out may not reach its destination in one piece. All will get there, but it may be broken up or fragmented into smaller packets. It's up to the receiver (in this case, the server) to wait for all the pieces and assemble them back into one payload.

A *codec* decodes or encodes the message data into something that the applications on either end of the communications link can understand. As figure 6.8 illustrates, the `textline` codec is responsible for grabbing packets as they come in and trying to piece together a message that's terminated by a specified character.

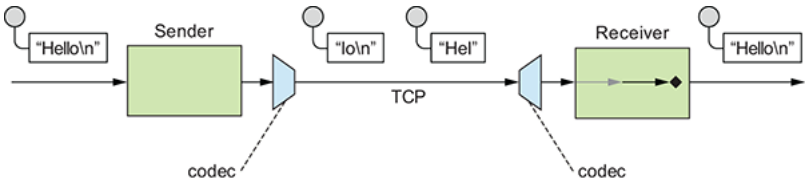

Figure 6.8 During TCP communications, a payload may be broken into multiple packets. A `textline` codec can assemble the TCP packets into a full payload by appending text until it encounters a delimiter character.

This example is provided in the book's source in the chapter6/netty directory. Try it by using the following command:

```
mvn test -Dtest=NettyTcpTest
```

Object serialization codec

If you hadn't specified the `textline` URI option in the previous example, the Netty4 component would've defaulted to using the object serialization codec. This codec takes any `Serializable` Java object and sends its bytes over TCP. This is a handy codec if you aren't sure what payload format to use. If you're using this codec, you also need to ensure that the classes are on the classpath of both the sender and the receiver.

At times your payload will have a custom format that neither `textline` nor object serialization accommodates. In that case, you need to create a custom codec.

6.4.2 Using custom codecs

The TCP server you set up for Rider Auto Parts in the previous section has worked out well. Sensors have been sending back status messages in plain text, and you used the Netty `textline` codec to successfully decode them. But one type of sensor has been causing an issue: the sensor connected to the welding machine sends its status back in a custom binary format. You need to interpret this custom format and send a status message formatted like the ones from the other sensors. You can do this with a custom Netty codec.

In Netty, a codec consists of two parts:

- `ChannelOutboundHandler`—This has the job of taking an input payload and putting bytes onto the TCP channel. In this example, the sensor

transmits the message over TCP, so you don't have to worry about this too much, except for testing that the server works.

- `ChannelInboundHandler`—This interprets the custom binary message from the sensor and returns a message that your application can understand.

You can specify a custom codec in a Camel URI by using the `encoder/decoder` options (or multiple with the `encoders/decoders` options) and specifying references to instances in the registry.

The custom binary payload that you have to interpret with your codec is 8 bytes in total; the first 7 bytes are the machine ID, and the last byte is a value indicating the status. You need to convert this to the plain-text format used by the other sensors, as illustrated in figure 6.9.

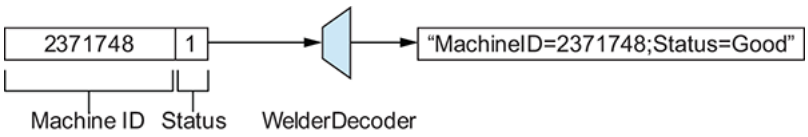

Figure 6.9 The custom welder sensor decoder is used to interpret an 8-byte binary payload and construct a plain-text message body. The first 7 bytes are the machine ID, and the last byte represents a status. In this case, a value of 1 means Good.

Your route looks similar to the previous example:

```
from("netty4:tcp://localhost:8998?encoder=#welderEncoder&decoder=#welder
    .to("jms:operations");
```

Note that you need to change the port that it listens on, so as not to conflict with your other TCP server. You also add a reference to the custom codecs loaded into the registry. In this case, the codecs are loaded into a `JndiRegistry` like this:

```
JndiRegistry jndi = ...
jndi.bind("welderDecoder", new WelderDecoder());
jndi.bind("welderEncoder", new WelderEncoder());
```

Now that the setup is complete, you can get to the real meat of the custom codec. As you may recall, decoding the custom binary format into a plain-text message was the most important task for this particular application. This decoder is shown in the following listing.

Listing 6.2 The decoder for the welder sensor

```
@ChannelHandler.Sharable
public class WelderDecoder extends MessageToMessageDecoder<ByteBuf> {
    static final int PAYLOAD_SIZE = 8;

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf msg,
        List<Object> out) throws Exception {
        if (msg.isReadable()) {
            // fill byte array with incoming message
            byte[] bytes = new byte[msg.readableBytes()];
            int readerIndex = msg.readerIndex();
            msg.getBytes(readerIndex, bytes);

            // first 7 bytes are the sensor ID, last is the status
            // and the result message will look something like
            // MachineID=2371748;Status=Good
            StringBuilder sb = new StringBuilder();
            sb.append("MachineID=")
      .append(new String(bytes, 0, PAYLOAD_SIZE - 1)).append(";")   (1)
```

(1) Gets first 7 bytes as machine ID

```
                .append("Status=");
      if (bytes[PAYLOAD_SIZE - 1] == '1') {   (2)
```

(2) Gets last byte as status

```
                sb.append("Good");
            } else {
                sb.append("Failure");
            }
            out.add(sb.toString());
        } else {
            out.add(null);
        }
```

```
        }
    }
```

This decoder may look complex, but it's doing only two main things: extracting the first 7 bytes and using that as the machine ID string ❶, and checking the last byte for a status of 1, which means Good ❷.

To try this example yourself, go to the chapter6/netty directory of the book's source and run the following unit test:

```
mvn test -Dtest=NettyCustomCodecTest
```

Now that you've tried low-level network communications, it's time to interact with one of the most common applications in the enterprise: the database.

6.5 Working with databases: JDBC and JPA components

In pretty much every enterprise-level application, you need to integrate with a database at some point, so it makes sense that Camel has first-class support for accessing databases. Camel has five components that let you access databases in various ways:

- *JDBC component*—Allows you to access JDBC APIs from a Camel route.
- *SQL component*—Allows you to write SQL statements directly into the URI of the component for using simple queries. This component can also be used for calling stored procedures.
- *JPA component*—Persists Java objects to a relational database by using the Java Persistence Architecture.
- *Hibernate component*—Persists Java objects by using the Hibernate framework. This component isn't distributed with Apache Camel because of licensing incompatibilities. You can find it at the camel-extra project (https://github.com/camel-extra/camel-extra).
- *MyBatis component*—Allows you to map Java objects to relational databases.

This section covers both the JDBC and JPA components. You can do pretty much any database-related task with them that you can do with the oth-

ers. For more information on the other components, see the relevant pages on the Camel website's components list (http://camel.apache.org/components.html).

Let's look first at the JDBC component.

6.5.1 Accessing data with the JDBC component

The Java Database Connectivity (JDBC) API defines how Java clients can interact with a particular database. It tries to abstract away details about the database being used. To use this component, you need to add the camel-jdbc module to your project:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jdbc</artifactId>
  <version>2.20.1</version>
</dependency>
```

The most common URI options are shown in table 6.9.

Table 6.9 Common URI options used to configure the JDBC component

Option	Default value	Description
`readSize`	`0`	Sets the maximum number of rows that can be returned. The default of `0` causes the `readSize` to be unbounded.
`statement.` *propertyName*		Sets the property with name `propertyName` on the underlying `java.sql.Statement`.
`useHeadersAsParameters`	`false`	Switches to using a java.sql.PreparedStatement with parameters that are replaced at runtime by message headers. This is a faster and safer alternative to executing a raw `java.sql.statement.` `PreparedStatements` are precompiled, and so are faster and aren't susceptible to SQL-injection type attacks.
`useJDBC4ColumnNameAndLabelSemantics`	`true`	Sets the column and label semantics to use. Default is to use the newer JDBC 4 style, but you can set this property to `false` to enable JDBC 3 style.

The endpoint URI for the JDBC component points Camel to a `javax.sql.DataSource` loaded into the registry, and, like other components, it allows for configuration options to be set. The URI syntax is as follows:

```
jdbc:dataSourceName[?options]
```

After this is specified, the component is ready for action. But you may be wondering where the SQL statement is specified.

The JDBC component is a dynamic component in that it doesn't merely deliver a message to a destination but takes the body of the message as a command. In this case, the command is specified using SQL. In EIP terms, this kind of message is called a *command message*. Because a JDBC endpoint accepts a command, it doesn't make sense to use it as a consumer, so you can't use it in a `from` DSL statement. You can still retrieve data by using a `select` SQL statement as the command message. In this case, the query result will be added as the outgoing message on the exchange.

To demonstrate the SQL command-message concept, let's revisit the order router at Rider Auto Parts. In the accounting department, when an order comes in on a JMS queue, the accountant's business applications can't use this data. They can only import data from a database. That means any incoming orders need to be put into the corporate database. Using Camel, a possible solution is illustrated in figure 6.10.

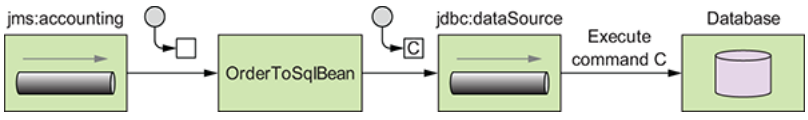

Figure 6.10 A message from the JMS accounting queue is transformed into an SQL command message by the `OrderToSqlBean` bean. The JDBC component then executes this command against its configured data source.

The main takeaway from figure 6.10 is that you're using a bean to create the SQL statement from the incoming message body. This is the most common way to prepare a command message for the JDBC component. You could use the DSL directly to create the SQL statement (by setting the body with an expression), but you have much more control when you use a custom bean.

The route for the implementation of figure 6.10 is simple on the surface:

```
from("jms:accounting")
    .to("bean:orderToSql")
    .to("jdbc:dataSource?useHeadersAsParameters=true");
```

Several things require explanation here. First, the JDBC endpoint is configured to load `javax.sql.DataSource` with the name `dataSource` in the registry. The bean endpoint here uses the bean with the name `orderToSql` to convert the incoming message to a SQL statement and to populate headers with information you'll be using in the SQL statement.

The `orderToSql` bean is shown in the following listing.

Listing 6.3 A bean that converts an incoming order to a SQL statement

```
public class OrderToSqlBean {

    public String toSql(@XPath("order/@name") String name,
                        @XPath("order/@amount") int amount,
                        @XPath("order/@customer") String customer,
                        @Headers Map<String, Object> outHeaders) {
        outHeaders.put("partName", name);
        outHeaders.put("quantity", amount);
        outHeaders.put("customer", customer);
        return "insert into incoming_orders"
            + "(part_name, quantity, customer) values"
            + " (:?partName, :?quantity, :?customer)";
    }
}
```

The `orderToSql` bean uses XPath to parse an incoming order message with a body, something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<order name="motor" amount="1" customer="honda"/>
```

The data in this order is then converted to message headers as follows:

```
partName = 'motor'
quantity = 1
customer = 'honda'
```

You return the same parameterized SQL statement every time:

```
insert into incoming_orders (part_name, quantity, customer) values (:?pa
```

The special `:?` syntax defines the parameter name in the `PreparedStatement` that will be created. The name following `:?` maps to a header name in the incoming message. In our case, the `:?partName` part of the SQL statement will be replaced with the `partName` header value `motor`. The same goes for the other parameters defined. This isn't the default behavior of the JDBC component—you enabled this behavior by setting the `useHeadersAsParameters` URI option to `true`.

This SQL statement becomes the body of a message that will be passed into the JDBC endpoint. In this case, you're updating the database by inserting a new row. You won't be expecting any result back. But Camel will set the `CamelJdbcUpdateCount` header to the number of rows updated. If there were any problems running the SQL command, an `SQLException` would be thrown.

If you were running a query against the database (using a SQL `select` command), Camel would return the rows as a `List<Map<String, Object>>`. Each entry in the `List` is a `LinkedHashMap` that maps the column name to a column value. Camel would also set the `CamelJdbcRowCount` header to the number of rows returned from the query.

To run this example, change to the chapter6/jdbc directory of the book's source and run the following command:

```
mvn test -Dtest=JdbcTest
```

Having raw access to the database through JDBC is a must-have ability in any integration framework. At times, though, you need to persist more than raw data; sometimes you need to persist whole Java objects. You can do this with the JPA component, which we'll look at next.

6.5.2 Persisting objects with the JPA component

Rider Auto Parts has a new requirement: instead of passing around XML order messages, management would like to adopt a POJO model for orders.

A first step would be to transform the incoming XML message into an equivalent POJO form. In addition, the order persistence route in the accounting department would need to be updated to handle the new POJO body type. You could manually extract the necessary information as you did for the XML message in listing 6.3, but a better solution exists for persisting objects.

The Java Persistence API (JPA) is a wrapper layer on top of object-relational mapping (ORM) products such as Hibernate, OpenJPA, EclipseLink, and the like. These products map Java objects to relational data in a database, which means you can save a Java object in your database of choice, and load it up later when you need it. This is a powerful ability, and it hides many details. Because this adds quite a bit of complexity to your application, plain JDBC should be considered first to see if it meets your requirements.

To use the JPA component, you need to add the camel-jpa module to your project:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jpa</artifactId>
  <version>2.20.1</version>
</dependency>
```

You also need to add JARs for the ORM product and database you're using. The examples in this section use OpenJPA and the Apache Derby database, so you need the following dependencies as well:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derby</artifactId>
</dependency>
<dependency>
```

```
    <groupId>org.apache.openjpa</groupId>
    <artifactId>openjpa-persistence-jdbc</artifactId>
  </dependency>
```

The JPA component has URI options, many of which can be applied only to either a consumer or producer endpoint. The URI options are shown in table 6.10.

Table 6.10 Common URI options used to configure the JPA component

Option	Consumer/producer mode	Default value	Description
`persistenceUnit`	Both	`camel`	Specifies the JPA persistence unit na used.
`transactionManager`	Both		Sets the transactio manager to be use transactions are en and this property i specified, Camel wi a `JpaTransactionMa`
`maximumResults`	Consumer	`-1`	Specifies the maxi number of objects returned from a qu The default of `-1` an unlimited numb results.
`maxMessagesPerPoll`	Consumer	`0`	Sets the maximum number of objects returned during a poll. The default of means an unlimite number of results.
`consumeLockEntity`	Consumer	`true`	Specifies whether t entities in the data will lock while the being consumed b Camel. By default, they lock.

Option	Consumer/producer mode	Default value	Description
`consumeDelete`	Consumer	`true`	Specifies whether t entity should be de in the database afte consumed.
`consumer.delay`	Consumer	`500`	Sets the delay in milliseconds betwe each poll.
`consumer.initialDelay`	Consumer	`1000`	Sets the initial dela milliseconds before first poll.
`consumer.query`	Consumer		Sets the custom SQ query to use when consuming objects.
`consumer.namedQuery`	Consumer		References a name query to consume objects.
`consumer.nativeQuery`	Consumer		Specifies a query in native SQL dialect database you're usi This isn't very port but it allows you to advantage of featu specific to a particu database.
`flushOnSend`	Producer	`true`	Causes objects that sent to a JPA produ be immediately per

Option	Consumer/producer mode	Default value	Description
			to the underlying database. Otherwis may stay in memor the ORM tool until decides to persist.

A requirement in JPA is to annotate any POJOs that need to be persisted with the `javax.persistence.Entity` annotation. The term *entity* is borrowed from relational database terminology and roughly translates to an *object* in object-oriented programming. Your new POJO order class needs to have this annotation if you want to persist it with JPA. The new order POJO is shown in the following listing.

Listing 6.4 An annotated POJO representing an incoming order

```
@Entity
```

Required annotation for objects to be persisted

```
public class PurchaseOrder implements Serializable {
    private String name;
    private double amount;
    private String customer;

    public PurchaseOrder() {
    }
    public double getAmount() {
        return amount;
    }
    public void setAmount(double amount) {
        this.amount = amount;
    }
    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }
    public void setCustomer(String customer) {
        this.customer = customer;
    }
    public String getCustomer() {
        return customer;
    }
}
```

This POJO can be created from the incoming XML order message easily with a message translator, as shown in chapter 3. For testing purposes, you can use a producer template to send a new `PurchaseOrder` to the accounting JMS queue, like so:

```
PurchaseOrder purchaseOrder = new PurchaseOrder();
purchaseOrder.setName("motor");
purchaseOrder.setAmount(1);
purchaseOrder.setCustomer("honda");

template.sendBody("jms:accounting", purchaseOrder);
```

Your route from section 6.5.1 is now a bit simpler. You send directly to the JPA endpoint after an order is received on the queue:

```
from("jms:accounting").to("jpa:camelinaction.PurchaseOrder");
```

Now that your route is in place, you have to configure the ORM tool. This is by far the most configuration you'll have to do when using JPA with Camel. As we've mentioned, ORM tools can be complex.

There are two main bits of configuration: hooking the ORM tool's entity manager up to Camel's JPA component, and configuring the ORM tool to connect to your database. For demonstration purposes here, we use Apache OpenJPA, but you could use any other JPA-compliant ORM tool.

The beans required to set up the OpenJPA entity manager are shown in the following listing.

Listing 6.5 Hooking up the Camel JPA component to OpenJPA

```
<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">
  <bean id="jpa"    (1)
```

(1) Hooks JPA component up to entity manager

```
    class="org.apache.camel.component.jpa.JpaComponent">
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>
  <bean id="entityManagerFactory"    (2)
```

(2) Creates entity manager

```
    class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="camel"/>
    <property name="jpaVendorAdapter" ref="jpaAdapter"/>
  </bean>
  <bean id="jpaAdapter"    (3)
```

(3) Uses OpenJPA and Apache Derby database

```
    class="org.springframework.orm.jpa.vendor.OpenJpaVendorAdapter">
    <property name="databasePlatform"
      value="org.apache.openjpa.jdbc.sql.DerbyDictionary"/>
    <property name="database" value="DERBY"/>
  </bean>
  <bean id="transactionTemplate"    (4)
```

❹

Allows JPA component to participate in transactions

```
    class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager">
      <bean class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"
      </bean>
    </property>
  </bean>
</beans>
```

This Spring beans file does numerous things to set up JPA. First, it creates a Camel `JpaComponent` and specifies the entity manager to be used ❶. This entity manager ❷ is then hooked up to OpenJPA and the Derby order database ❸. It also sets up the entity manager so it can participate in transactions ❹.

There's one more thing left to configure before JPA is up and running. When the entity manager was created in the preceding listing ❷, you set the `persistenceUnitName` to `camel`. This persistence unit defines what entity classes will be persisted, as well as the connection information for the underlying database. In JPA, this configuration is stored in the persistence.xml file in the META-INF directory on the classpath. The following listing shows the configuration required for your application.

Listing 6.6 Configuring the ORM tool with the persistence.xml file

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
  <persistence-unit name="camel" transaction-type="RESOURCE_LOCAL">
    <class>camelinaction.PurchaseOrder</class>    ❶
```

❶

Lists entity classes to be persisted

```
    <properties>
```

```
    <property name="openjpa.ConnectionDriverName"    ❷
```

❷

Provides database connection information

```
        value="org.apache.derby.jdbc.EmbeddedDriver"/>
        <property name="openjpa.ConnectionURL"
        value="jdbc:derby:memory:order;create=true"/>
        <property name="openjpa.ConnectionUserName" value="sa"/>
        <property name="openjpa.ConnectionPassword" value=""/>
        <property name="openjpa.jdbc.SynchronizeMappings"
        value="buildSchema"/>
      </properties>
    </persistence-unit>
</persistence>
```

You need to be aware of two main points in this listing. First, classes that you need persisted need to be defined here ❶, and there can be more than one `class` element. Also, if you need to connect to another database or otherwise change the connection information to the database, you need to do so here ❷.

Now that all the setup is complete, your JPA route is complete. To try this example, browse to the chapter6/jpa directory and run the JpaTest test case with this Maven command:

```
mvn test -Dtest=JpaTest
```

This example sends a `PurchaseOrder` to the accounting queue and then queries the database to make sure the entity class was persisted.

Manually querying the database via JPA is a useful ability, especially in testing. In `JpaTest`, the query was performed like so:

```
JpaEndpoint endpoint = context.getEndpoint("jpa:camelinaction.PurchaseOr
EntityManager em = endpoint.getEntityManagerFactory().createEntityManage

List list = em.createQuery(
    "select x from camelinaction.PurchaseOrder x").getResultList();
```

```
assertEquals(1, list.size());
assertIsInstanceOf(PurchaseOrder.class, list.get );

em.close();
```

First, you create an `EntityManager` instance by using the `EntityManagerFactory` from the `JpaEndpoint`. You then search for instances of your entity class in the database by using JPQL, which is similar to SQL but deals with JPA entity objects instead of tables. A simple check is then performed to make sure the object is the right type and that there's only one result.

Now that we've covered accessing databases, and messaging that can span the entire web, we're going to shift our attention to communication within the JVM.

6.6 In-memory messaging: Direct, Direct-VM, SEDA, and VM components

Having braved so many of Camel's messaging abilities in this chapter, you might think there couldn't be more. Yet there's still another important messaging topic to cover: in-memory messaging.

Camel provides four main components in the core to handle in-memory messaging. For synchronous messaging, there are the Direct and Direct-VM components. For asynchronous messaging, there are the SEDA and VM components. The only difference between Direct and Direct-VM is that the Direct component can be used for communication within a single `CamelContext`, whereas the Direct-VM component is a bit broader and can be used for communication within a JVM. If you have two `CamelContext`s loaded into an application server, you can send messages between them by using the Direct-VM component. Similarly, the only difference between SEDA and VM is that the VM component can be used for communication within a JVM.

NOTE For more information on staged event-driven architecture (SEDA) in general, see https://en.wikipedia.org/wiki/Staged_event-driven_architecture.

Let's look first at the Direct components.

6.6.1 Synchronous messaging with Direct and Direct-VM

The Direct component is about as simple as a component can get, but it's extremely useful. It's probably the most common Camel endpoint you'll see in a route.

A direct endpoint URI looks like this:

```
direct:endpointName
```

Table 6.11 lists the only three URI options.

Table 6.11 Common URI options used to configure the Direct and Direct-VM components

Option	Default value	Description
`block`	`false`	Causes producers sending to a direct endpoint to block until there are active consumers. Note Camel 2.21 changes this default value to `true`.
`timeout`	`30000`	The time-out in milliseconds for blocking the producer when there were no active consumers.
`failIfNoConsumers`	`true`	Specifies whether to throw an exception when a producer tries to send to an endpoint with no active consumers. Used only when `block` is set to `false`.

What does this give you? The Direct component lets you make a synchronous call to a route or, conversely, expose a route as a synchronous service.

To demonstrate, say you have a route that's exposed by a direct endpoint as follows:

```
from("direct:startOrder")
  .to("cxf:bean:orderEndpoint");
```

Sending a message to the `direct:startOrder` endpoint invokes a web service defined by the `orderEndpoint` CXF endpoint bean. Let's also say that you send a message to this endpoint by using `ProducerTemplate`:

```
String reply =
  template.requestBody("direct:startOrder", params, String.class);
```

`ProducerTemplate` creates a `Producer` under the hood that sends to the `direct:startOrder` endpoint. In most other components, some processing happens between the producer and the consumer. For instance, in a JMS component, the message could be sent to a queue on a JMS broker. With the Direct component, the producer *directly* calls the consumer. And by *directly*, we mean that in the producer there's a method invocation on the consumer. The only overhead of using the Direct component is a method call!

This simplicity and minimal overhead make the Direct component a great way of starting routes and synchronously breaking up routes into multiple pieces. But even though using the Direct component carries little overhead, its synchronous nature doesn't fit well with all applications. If you need to operate asynchronously, you need the SEDA or VM components, which we'll look at next.

6.6.2 Asynchronous messaging with SEDA and VM

As you saw in the discussion of JMS earlier in the chapter (section 6.3), using message queuing as a means of sending messages has many benefits. You also saw that a routing application can be broken into many logical pieces (routes) and connected using JMS queues as bridges. But using JMS

for this purpose in an application on a single host adds unnecessary complexity for some use cases.

If you want to reap the benefits of asynchronous messaging, but you aren't concerned with JMS specification conformance or the built-in reliability that JMS provides, you may want to consider an in-memory solution. By ditching the specification conformance and any communications with a message broker (which can be costly), an in-memory solution can be much faster. Note that there's no message persistence to disk, as in JMS, so you run the risk of losing messages in the event of a crash; your application should be tolerant of losing messages.

Camel provides two in-memory queuing components: SEDA and VM. They both share the options listed in table 6.12.

Table 6.12 Common URI options used to configure the SEDA and VM components

Option	Default value	Description
`size`	`Integer.MAX_VALUE`	Sets the maximum number of messages the queue can hold.
`concurrentConsumers`	`1`	Sets the number of threads servicing incoming exchanges. Increase this number to process more exchanges concurrently.
`waitForTaskToComplete`	`IfReplyExpected`	Specifies whether the client should wait for an asynchronous task to complete. The default is to wait only if it's an `InOut` MEP. Other values include `Always` and `Never`.
`timeout`	`30000`	Sets the time in milliseconds to wait for an asynchronous

Option	Default value	Description
		send to complete. A value less than or equal to `0` disables the time-out.
`multipleConsumers`	`false`	Specifies whether to allow the SEDA queue to have behavior like a JMS topic (a publish-subscribe style of messaging).

One of the most common uses for SEDA queues in Camel is to connect routes to form a routing application. For example, recall the example presented in section 6.3.1 in which you used a JMS topic to send copies of an incoming order to the accounting and production departments. In that case, you used JMS queues to connect your routes. Because the only parts that are hosted on separate hosts are the accounting and production queues, you can use SEDA queues for everything else. This new, faster solution is illustrated in figure 6.11.

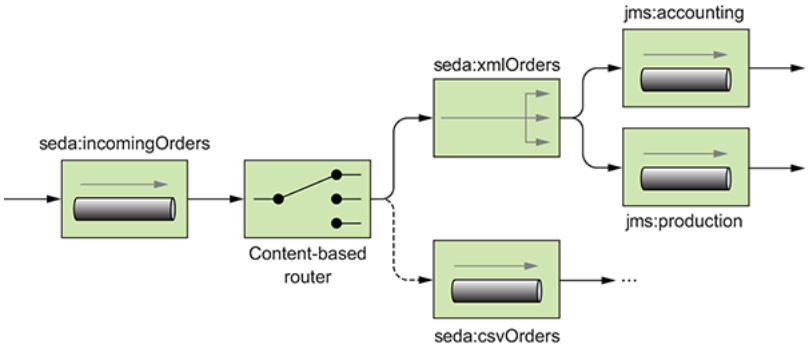

Figure 6.11 SEDA queues can be used as a low-overhead replacement for JMS when messaging is within `CamelContext`. For messages being sent to other hosts, JMS can be used. In this case, all order routing is done via SEDA until the order needs to go to the accounting and production departments.

Any JMS messaging that you were doing within `CamelContext` could be switched over to SEDA. You still need to use JMS for the accounting and production queues, because they're located in physically separate departments.

You may have noticed that the JMS xmlOrders topic has been replaced with a SEDA queue in figure 6.11. In order for this SEDA queue to behave like a JMS topic (using a publish-subscribe messaging model), you need to set the `multipleConsumers` URI option to `true`, as shown in the following listing.

Listing 6.7 A topic allows multiple receivers to get a copy of the message

```
from("file:src/data?noop=true")
    .to("seda:incomingOrders");    (1)
```

Orders enter set of routes

```
from("seda:incomingOrders")
    .choice()
        .when(header("CamelFileName").endsWith(".xml"))
            .to("seda:xmlOrders")
```

XML orders are routed to xmlOrders topic

```
        .when(header("CamelFileName").regex("^.*(csv|csl)$"))
            .to("seda:csvOrders");

from("seda:xmlOrders?multipleConsumers=true")
```

Both listening queues get copies

```
    .to("jms:accounting");

from("seda:xmlOrders?multipleConsumers=true")
    .to("jms:production");
```

This example behaves in the same way as listing 6.1, except that it uses SEDA endpoints instead of JMS. To run this example, go to the chapter6/seda directory in the book's source and run this command:

```
mvn test -Dtest=OrderRouterWithSedaTest
```

This outputs the following on the command line:

```
Accounting received order: message1.xml
Production received order: message1.xml
```

Why did you get this output? Well, you have a single order file named message1.xml, and it's published to the xmlOrders topic. Both the accounting and production queues are subscribed to the topic, so each receives a copy. The testing routes consume the messages on those queues and output the messages.

So far, you've been kicking off routes either by hand or by consuming from a filesystem directory. How can you kick off routes automatically? Or better yet, how can you schedule a route's execution to occur?

6.7 Automating tasks: Scheduler and Quartz2 components

Often in enterprise projects you need to schedule tasks to occur either at a specified time or at regular intervals. Camel supports this kind of service with the Timer, Scheduler, and Quartz2 components. The Scheduler component is useful for simple recurring tasks, but when you need more control of when things get started, the Quartz2 component is a must. The Timer component can also be used for simple recurring tasks. The difference is the Scheduler component uses the improved Java scheduler API, as opposed to the Timer component which uses the older `java.util.Timer` API.

This section first presents the Scheduler component and then moves on to the more advanced Quartz2 component.

6.7.1 Using the Scheduler component

The Scheduler component comes with Camel's core library and uses `ScheduledExecutor-Service` from the JRE to generate message exchanges at regular intervals. This component supports only consuming, because sending to a scheduler doesn't make sense.

Some common URI options are listed in table 6.13.

Table 6.13 Common URI options used to configure the Scheduler component

Option	Default value	Description
`delay`	`500`	Specifies the time in milliseconds between generated events.
`initialDelay`	`1000`	Specifies the time in milliseconds before the first event is generated.
`useFixedDelay`	`true`	If `true`, a delay occurs between the completion of one event and the generation of the next. If `false`, events are generated at a fixed rate based on the delay period without considering completion of the previous event.

As an example, let's print a message stating the time to the console every 2 seconds. The route looks like this:

```
from("scheduler:myScheduler?delay=2000")
    .setBody().simple("Current time is ${header.CamelTimerFiredTime}")
    .to("stream:out");
```

The scheduler URI configures the underlying `ScheduledExecutorService` to have the execution interval of 2,000 milliseconds.

TIP When the value of milliseconds gets large, you can opt for a shorter notation using the `s`, `m`, and `h` keywords. For example, 2,000 milliseconds can be written as `2s`, meaning 2 seconds. 90,000 milliseconds can be written as `1m30s`, and so on.

When this scheduler fires an event, Camel creates an exchange with an empty body and sends it along the route. In this case, you're setting the

body of the message by using a Simple language expression. The `CamelTimerFiredTime` header was set by the Scheduler component; for a full list of headers set, see the online documentation (http://camel.apache.org/scheduler.html).

NOTE The `scheduler` endpoint and corresponding thread can be shared between routes; you just have to use the same scheduler name.

You can run this simple example by changing to the chapter6/scheduler directory of the book's source and running this command:

```
mvn test -Dtest=SchedulerTest
```

You'll see output similar to the following:

```
Current time is Thu Apr 04 13:43:51 NST 2013
```

As you can see, an event was fired every 2 seconds. But suppose you want to schedule a route to execute on the first day of each month. You can't do that easily with the Scheduler component. You need Quartz.

6.7.2 Enterprise scheduling with Quartz

Like the Scheduler component, the Quartz2 component allows you to schedule the generation of message exchanges. But the Quartz2 component gives you much more control over how this scheduling happens. You can also take advantage of Quartz's many other enterprise features.

We don't cover all of Quartz's features here—only ones exposed directly in Camel. For a complete look at using Quartz, see the Quartz website: www.quartz-scheduler.org.

The common URI options for the Quartz2 component are listed in table 6.14.

Table 6.14 Common URI options used to configure the Quartz2 component

Option	Default value	Description
`cron`		Specifies a cron expression used to determine when the timer fires.
`trigger.repeatCount`	`0`	Specifies the number of times to repeat the trigger. A value of `-1` causes the timer to repeat indefinitely.
`trigger.repeatInterval`	`1000`	Specifies the interval in milliseconds at which to generate events.
`job.`*`propertyName`*		Sets the property with name `propertyName` on the underlying Quartz `JobDetail`.
`trigger.`*`propertyName`*		Sets the property with name `propertyName` on the underlying Quartz `Trigger`.

Before you can use the Quartz2 component, you need to add the following dependency to your Maven POM file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-quartz2</artifactId>
  <version>2.20.1</version>
</dependency>
```

Let's now reproduce the scheduler example from the previous section with Quartz. To do this, you can use the `trigger.repeatInterval` option, which is similar to the `delay` option for the Scheduler component. The route looks like this:

```
from("quartz2:myTimer?trigger.repeatInterval=2000&trigger.repeatCount=-1
    .setBody().simple("Current time is ${header.firedTime}")
    .to("stream:out");
```

Although this behaves in the same way as the scheduler example, a few things are going on under the covers.

First, `myTimer` sets the underlying `Trigger` object's name. Timers in Quartz are made up of a `Trigger` and a `JobDetail`. Triggers can also have a group name associated with them, which you can specify by adding the group name to your URI, as follows:

```
quartz2:myGroupName/myTimer?options...
```

If the group name is omitted, as in the previous route, *Camel* is used as the group name. By default, `SimpleTrigger` is created to schedule events.

The `trigger.repeatInterval` and `trigger.repeatCount` properties configured this trigger to fire every 2,000 milliseconds for as long as the application is running (a repeat count of `-1` causes the trigger to repeat indefinitely). You may be thinking that the option names are a bit long, but there's a reason for this. As stated in table 6.14, options starting with `trigger` allow you to set properties on the `Trigger` object. In the case of the `trigger.repeatInterval` URI option, this will call the `setRepeatInterval` method on the `SimpleTrigger` object.

You can similarly set options on `JobDetail` by using properties that start with *job*, followed by a valid property name. For instance, you can set the job name by using the `job.name` URI option.

You can run this simple example by changing to the chapter6/quartz directory of the book's source and running this command:

```
mvn test -Dtest=QuartzTest
```

Using cron triggers

So far, you've replaced the Scheduler component example with a functionally equivalent Quartz-based example. How would you schedule something more complex, such as kicking off a route on the first day of each month? The answer is by using cron expressions. Readers familiar with Linux or UNIX probably have heard of the cron scheduling application. Quartz allows you to use scheduling syntax similar to the venerable cron application.

A *cron* expression is a string consisting of six or seven fields separated by whitespace. Each field denotes a date or range of dates. The structure of a cron expression is as follows:

```
<Seconds> <Minutes> <Hours> <Day of Month> <Month> <Day of week> <Year>
```

These accept numeric values (and optional textual ones) for the times you want a trigger to fire. More information on cron expressions can be found on the Quartz website (www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger).

The cron expression for occurring on the first day of each month at 6:00 a.m. is the following:

```
0 0 6 1 * ?
```

In this expression, the third digit denotes the hour at which to execute, and the fourth digit is the day of the month. You can also see that a star is placed in the month column so that every month will be triggered.

Setting up Quartz to use cron triggers in Camel is easy. You just use the `cron` option and make sure to replace all whitespace with plus characters (`+`). Your URI becomes the following:

```
quartz2:firstDayOfTheMonth?cron=0+0+6+1+*+?
```

Using this URI in a route causes a message exchange to be generated (running the route) on the first day of each month.

To try an example using a cron trigger, browse to the chapter6/quartz directory and run the QuartzCronTest test case with this Maven command:

```
mvn test -Dtest=QuartzCronTest
```

You should be able to see now how the scheduling components in Camel can allow you to execute routes at specified times. This is an important ability in time-sensitive enterprise applications.

6.8 Working with email

It's hard to think of a more pervasive technology than email in the enterprise. Modern businesses, for better or worse, run on email. Compared to the other communication mechanisms mentioned thus far, email is certainly different. Whereas other components are primarily used to communicate with other automated services, email is most often used to communicate with people. A retail application may need to notify a customer that an order has been shipped, for instance. Or maybe a back-end system needs to send an alert to a system administrator about a failure. These are all ideal for email messaging. Sure, technically you can implement inter-application messaging with email if you want to, but that isn't the most efficient way to do things.

Camel provides several components to work with email:

- *Mail component*—This is the primary component for sending and receiving email in Camel.
- *AWS-SES component*—Allows you to send email by using the Amazon Simple Email Service (SES).
- *GoogleMail component*—Gives you access to Gmail via the Google Mail Web API.

This section covers the mail component. For more information on the other components, see the relevant pages on the Camel website's components list (http://camel.apache.org/components.html).

Let's first take a look at sending email.

6.8.1 Sending mail with SMTP

Whenever you send an email, you're using the Simple Mail Transfer Protocol (SMTP) under the hood. In Camel, an SMTP URI looks like this:

```
[smtp|stmps]://[username@]host[:port][?options]
```

You'll first notice that you can select to secure your mail transfer with SSL by specifying a scheme of `smtps` instead of `smtp`. The `host` is the name of the mail server that will be sending the message, and `username` is an account on that server. The value of `port` defaults to 25 for SMTP, and 465 for SMTPS, and can be overridden if needed. The most common URI options are shown in table 6.15.

Table 6.15 Common URI options used to configure the Mail component

Option	Default value	Description
`password`		The password of the user account corresponding to username in the URI.
`subject`		Sets the subject of the email being sent. You can override this value by setting a Subject message header.
`from`	`camel@localhost`	Sets what email address will be used for the From field of the email being sent.
`to`	`username@host`	The email address you're sending to. Multiple addresses must be separated by commas.
`cc`		The carbon copy (CC) email address you're sending to. Multiple addresses must be separated by commas.

To use this component, you need to add the camel-mail module to your project:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-mail</artifactId>
  <version>2.20.1</version>
</dependency>
```

Sending an email is then simple. For example, let's say Jon wants to send an email to Claus about Camel. Using `ProducerTemplate`, you could do something like this:

```
template.sendBody(
    "smtp://jon@localhost?password=secret&to=claus@localhost",
    "Yes, Camel rocks!");
```

Here, you're logging into an email server on localhost port 25 with username `jon` and password `secret`. You're sending to the email address claus@localhost. Because you didn't specify a From option, the From email field is defaulted to camel@localhost. Now the string you passed as a message body will become the body of the email. Typically, though, you'd like to have a subject for the email. You could add a subject option to the URI or add a subject header to the message. Let's try to add a subject header:

```
template.sendBodyAndHeader(
    "smtp://jon@localhost?password=secret&to=claus@localhost",
    "Yes, Camel rocks!",
    "subject", "Does Camel rock?");
```

Again, you use `ProducerTemplate` to send, but this time you pass along a message header to set the subject as well.

The claus@localhost email address could be checked in a GUI mail client by a person, or you could get Camel to consume this for you. Let's take a look at how to do this next.

6.8.2 Receiving mail with IMAP

In Camel, you can retrieve email by using either the Internet Message Access Protocol (IMAP) or POP3. IMAP is the preferred protocol to access email. It was developed as an alternative to POP3 and is more feature rich. To consume email messages by using IMAP, you need to use a URI like this:

```
[imap|imaps]://[username@]host[:port][?options]
```

As with the SMTP component, you can select to secure your mail transfer with SSL by specifying a scheme of `imaps` instead of `imap`. The `host` is the name of the mail server that has the mail you want to consume, and `username` is an account on that server. The value of `port` defaults to 143

for IMAP, and 993 for IMAPS, and can be overridden if needed. The most common URI options are shown in table 6.16.

Table 6.16 Common URI options used to configure the Mail component

Option	Default value	Description
`password`		The password of the user account corresponding to username in the URI.
`delay`	`60000`	Delay between polling the mail server for more emails.
`delete`	`false`	If true, the email will be deleted after processing.
`folderName`	`INBOX`	The mail folder to poll for messages.
`unseen`	`true`	If `true`, consumes only new messages.

As with the SMTP component, you first need to add the camel-mail module to your project.

Let's say you want to use IMAP to receive the email Jon sent via SMTP in the preceding section. The route looks like this:

```
from("imap://claus@localhost?password=secret").to("mock:result");
```

Here, you're logging into an email server on localhost port 143 with username `claus` and password `secret`. You're polling the INBOX folder for new messages by using the default polling interval of 60 seconds. To try this example, browse to the chapter6/mail directory and run the MailTest test case with this Maven command:

```
mvn test -Dtest=MailTest
```

That covers the basics of sending and receiving email using Camel. Many more options are available. You can find them all in the online documentation (http://camel.apache.org/mail.html).

6.9 Summary and best practices

Congratulations on making it through the barrage of components covered in this chapter. By now, you should have a good understanding of how to use them in your own applications.

Here are some of the key ideas you should take away from this chapter:

- *There are tons of Camel components*—One of the great things about Camel is its extensive component library. You can rest easy knowing that most things you'll ever need to connect to are covered by Camel.
- *The Camel website has documentation on all components available*—We could cover only the most widely used and important components in this book, so if you need to use one of the many other components, documentation is available at http://camel.apache.org/components.html.
- *Camel's component model allows for your own extensions*—We briefly touched on how components are resolved at runtime in Camel. Camel imposes no restrictions on where your components come from, so you can easily write your own (as described in chapter 8) and include it in your Camel application.
- *Don't write to files manually; use the File and FTP components*—Camel's File and FTP components have many options that suit most file-processing scenarios. Don't reinvent the wheel—use what Camel has to offer.
- *Use the JMS component for asynchronous messaging with JMS*—Camel makes it easy to send messages to and receive them from JMS providers. You no longer have to write dozens of lines of JMS API calls to send or receive a simple message.
- *Use the Netty4 component for network communications*—Network programming can be difficult, given the low-level concepts you need to deal with. The Netty4 component handles these details for you, making it easy to communicate over network protocols such as TCP and UDP.
- *Hook your routes into databases by using the JDBC and JPA components*—The JDBC component allows you to access databases by using tried-

and-true SQL, whereas the JPA component is all about persisting Java objects into databases.

- *Use in-memory messaging when reliability isn't a concern but speed is*—Camel provides four choices for in-memory messaging: the Direct, Direct-VM, SEDA, and VM components.
- *Kick off routes at specified intervals by using the Quartz2 or Scheduler components*—Camel routes can do useful things. Some tasks need to be executed at specified intervals, and the Quartz2 and Scheduler components come into play here.

Components in Camel fill the role of bridging out to other transports, APIs, and data formats. They're also the on- and off-ramps to Camel's routing abilities.

Next up in chapter 7, is a topic that has been a buzzword for a while now: *microservices*. Camel wasn't designed from the ground up with this in mind, but it has certainly become a useful way of building these small single-purpose services.

Table 6.2 Components in the camel-core module (continued)