11 Error handling

This chapter covers

- Understanding error handling
- Knowing where and when Camel's error handling applies
- Using the various error handlers in Camel
- Using redelivery policies
- Handling exceptions with `onException`
- Reusing error handlers in all your routes
- Performing fine-grained control of error handling

You've now reached the halfway mark of this book, and we've covered a lot of ground. But what you're about to embark on in this chapter is one of the toughest topics in integration: how to deal with the unexpected.

Writing applications that integrate disparate systems is a challenge when it comes to handling unexpected events. In a single system that you fully control, you can handle these events and recover. But systems that are integrated over the network have additional risks: the network connection could be broken, a remote system might not respond in a timely manner, or it might even fail for no apparent reason. Even on your local server, unexpected events can occur, such as the server's disk filling up or the server running out of memory. Regardless of which errors occur, your application should be prepared to handle them.

In these situations, log files are often the only evidence of the unexpected event, so logging is important. Camel has extensive support for logging and for handling errors to ensure that your application can continue to operate.

In this chapter, you'll discover how flexible, deep, and comprehensive Camel's error handling is and how to tailor it to deal with most situations. We'll cover all the error handlers Camel provides out of the box, and when they're best used, so you can pick the ones suited to your applications. You'll also learn how to configure and master redelivery, so Camel can try to recover from particular errors. We'll also look at exception policies, which allow you to differentiate among errors and handle specific ones, and at how scopes can help you define general rules for implementing route-scoped error handling. Finally, we'll look at what Camel offers when you need fine-grained control over error handling, so that it reacts under only certain conditions.

This chapter covers a complicated topic that Camel users may have trouble learning. Therefore, we've taken the time and space to cover all aspects of this in detail, and as a result this chapter is long and not a light read. We suggest you take a break halfway through; we'll tell you when.

11.1 Understanding error handling

Before jumping into the world of error handling with Camel, you need to take a step back and look at errors more generally. You also need to look at where and when error handling starts, because some prerequisites must happen beforehand.

11.1.1 Recoverable and irrecoverable errors

When it comes to errors, you can divide them into two main categories: recoverable and irrecoverable errors, as illustrated in figure 11.1.

An *irrecoverable error* remains an error no matter how many times you try to perform the same action again. In the integration space, that could mean trying to access a database table that doesn't exist, which would cause the JDBC driver to throw `SQLException`.

A *recoverable error*, on the other hand, is a temporary error that might not cause a problem on the next attempt. A good example of such an error is a problem with the network connection resulting in `java.io.IOException`. On a subsequent attempt, the network issue could be resolved, and your application could continue to operate.

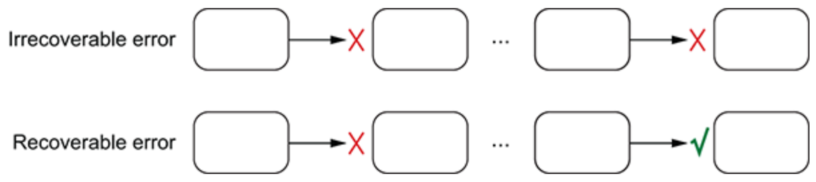

Figure 11.1 Errors can be categorized as either recoverable or irrecoverable. Irrecoverable errors continue to be errors on subsequent attempts; recoverable errors may be quickly resolved on their own.

In your daily life as a Java developer, you won't often encounter this division of errors into recoverable and irrecoverable. Generally, exception-handling code uses one of the two patterns illustrated in the following two code snippets.

The first snippet illustrates a common error-handling idiom: all kinds of exceptions are considered irrecoverable, and you give up immediately, throwing the exception back to the caller, often wrapped:

```
public void handleOrder(Order order) throws OrderFailedException {
    try {
        service.sendOrder(order);
    } catch (Exception e) {
        throw new OrderFailedException(e);
    }
}
```

The next snippet improves on this situation by adding a bit of logic to handle redelivery attempts before eventually giving up:

```
public void handleOrder(Order order) throws OrderFailedException {
    boolean done = false;
    int retries = 5;
    while (!done) {
        try {
            service.sendOrder(order);
            done = true;
        } catch (Exception e) {
            if (--retries == 0) {
                throw new OrderFailedException(e);
            }
        }
    }
}
```

Around the invocation of the service is the logic that attempts redelivery, in case an error occurs. After five attempts, it gives up and throws the exception.

What the preceding example lacks is logic to determine whether the error is recoverable or irrecoverable, and to react accordingly. In the recoverable case, you could try again, and in the irrecoverable case, you could give up immediately and rethrow the exception.

In Camel, a recoverable error is represented as a plain `Throwable` or `Exception` that can be set or accessed from `org.apache.camel.Exchange` by using one of the following methods:

```
void setException(Throwable cause);
```

or

```
Exception getException();
<T> T getException(Class<T> type);
```

NOTE The `setException` method on `Exchange` accepts a `Throwable` type, whereas the `getException` method returns an `Exception` type. `getException` also doesn't return a `Throwable` type because of backward API compatibility. The second `getException` method accepts a class type that allows you to traverse the exception hierarchy to find a matching exception. Section 11.4 covers how this works.

An irrecoverable error is represented as a message with a fault flag that can be set or accessed from `org.apache.camel.Exchange`. For example, to set `Unknown` `customer` as a fault message, you would do the following:

```
Message msg = Exchange.getOut();
msg.setFault(true);
msg.setBody("Unknown customer");
```

The fault flag must be set using the `setFault(true)` method.

Why are the two types of errors represented differently? Two reasons. First, the Camel API was designed around the Java Business Integration (JBI) specification, which includes a fault message concept. Second, Camel has error handling built into its core, so whenever an exception is thrown back to Camel, it catches it and sets the thrown exception on the exchange as a recoverable error, as illustrated here:

```
try {
    processor.process(exchange);
} catch (Throwable e) {
   exchange.setException(e);
}
```

Using this pattern allows Camel to catch and handle all exceptions that are thrown. Camel's error handling can then determine how to deal with the errors: retry, propagate the error back to the caller, or do something else. End users of Camel can set irrecoverable errors as fault messages, and Camel can react accordingly and stop routing the message.

FAULT OR EXCEPTION

In practice, you can represent `Exception` as a nonrecoverable error as well by using exception classes that by nature are nonrecoverable. For example, an `InvalidOrderIdException` exception represents a nonrecoverable error, as a given order ID is invalid. It's often more common with Camel to use this approach than to use fault messages. Fault messages are often only used with legacy components such as JBI (https://en.wikipedia.org/wiki/Java_Business_Integration) or SOAP web services.

Now that you've seen recoverable and irrecoverable errors in action, let's summarize how they're represented in Camel:

- Recoverable errors are represented as exceptions.
- Irrecoverable errors are represented as fault messages.

Next let's look at when and where Camel's error handling applies.

11.1.2 Where Camel's error handling applies

Camel's error handling doesn't apply everywhere. To understand why, take a look at figure 11.2.

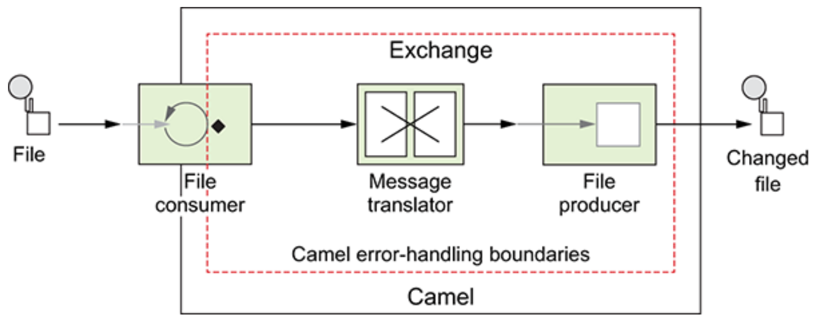

Figure 11.2 Camel's error handling applies only within the lifecycle of an exchange. The error-handler boundaries are represented by the dashed square in the figure.

Figure 11.2 shows a simple route that translates files. You have a file consumer and producer as the input and output facilities, and in between is the Camel routing engine, which routes messages encompassed in an exchange. It's during the lifecycle of this exchange that the Camel error handling applies. That leaves a little room on the input side where this error handling can't operate—the file consumer must be able to successfully read the file, instantiate the exchange, and start the routing before the error handling can function. This applies to any kind of Camel consumer.

What happens if the file consumer can't read the file? The answer is component specific, and each Camel component must deal with this in its own way. Some components will ignore and skip the message, others will retry a certain number of times, and others will gracefully recover. In the case of the file consumer, a `WARN` will be logged and the file will be skipped, and a new attempt to read the file will occur on the next polling.

But what if you want to handle this differently? What if instead of logging a `WARN`, you want to let the Camel error handler handle the exception, or what if the file keeps failing on the subsequent polls? These are all great questions that you'll learn how to deal with when you have more experience with Camel's error handler. You'll come back to these questions and get answers in section 11.4.8. What you need to understand for now is that figure 11.2 represents the default behavior for where Camel's error handler works.

That's enough background information. Let's dig into how error handling in Camel works. In the next section, you'll start by looking at the various error handlers Camel provides.

11.2 Using error handlers in Camel

In the last section, you learned that Camel regards all exceptions as recoverable and stores them on the exchange by using the `setException(Throwable cause)` method. This means error handlers in Camel will react only to exceptions set on the exchange. The rule of thumb is that error handlers in Camel trigger only when `exchange.getException() != null`.

Camel provides a range of error handlers. They're listed in table 11.1.

Table 11.1 Error handlers provided in Camel

Error Handler	Description
`DefaultErrorHandler`	This is the default error handler that's automatically enabled, in case no other has been configured.
`DeadLetterChannel`	This error handler implements the Dead Letter Channel EIP.
`TransactionErrorHandler`	This is a transaction-aware error handler extending the default error handler. Transactions are covered in the chapter 12 and are only briefly touched on in this chapter.
`NoErrorHandler`	This handler is used to disable error handling altogether.
`LoggingErrorHandler`	This error handler just logs the exception. This error handler is deprecated, in favor of using the `DeadLetterChannel` error handler, using a log endpoint as the destination.

At first glance, having five error handlers may seem overwhelming, but you'll learn that the default error handler is used in most cases.

The first three error handlers in table 11.1 all extend the `RedeliveryErrorHandler` class. That class contains the majority of the error-handling logic that the first three error handlers all use. The latter two error handlers have limited functionality and don't extend `RedeliveryErrorHandler`. We'll look at each of these error handlers in turn.

11.2.1 Using the default error handler

Camel is preconfigured to use `DefaultErrorHandler`, which covers most use-cases. To understand it, consider the following route:

```
from("direct:newOrder")
    .bean("orderService, "validate")
    .bean("orderService, "store");
```

The default error handler is preconfigured and doesn't need to be explicitly declared in the route. What happens if an exception is thrown from the `validate` method on the order service bean?

To answer this, you need to dive into Camel's inner processing, where the error handler lives. In every Camel route, a channel sits between each node in the route path, as illustrated in figure 11.3.

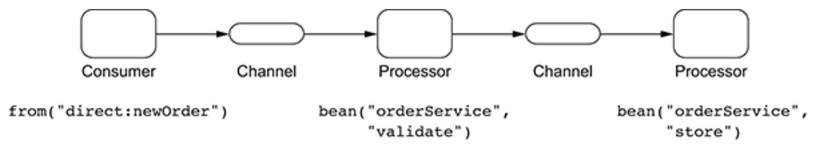

Figure 11.3 A detailed view of a route path, where channels act as controllers between the processors

A channel is between each node of the route path, which ensures that it can act as a controller to monitor and control the routing at runtime. This is the feature that allows Camel to enrich the route with error handling, message tracing, interceptors, and much more. For now, you just need to know that this is where the error handler lives.

Turning back to the example route, imagine that an exception was thrown from the order service bean during invocation of the `validate` method. In figure 11.3, the processor would throw an exception, which would be propagated back to the previous channel, where the error handler would catch it. This gives Camel the chance to react accordingly. For example, Camel could try again (redeliver), or it could route the message to another route path (detour using exception policies), or it could give up and propagate the exception back to the caller. With the default settings, Camel will propagate the exception back to the caller.

The default error handler is configured with these settings:

- No redelivery occurs.

- Exceptions are propagated back to the caller.
- The stack trace of the exception is printed to the log.
- The routing history of the exchange is printed to the log.

These settings match what happens when you're working with exceptions in Java, so Camel's behavior won't surprise Camel end users.

When an exception is thrown during routing, the default behavior is to log a route history that traces the steps back to where the exchange was routed by Camel. This allows you to quickly identify where in the route the exception occurred. In addition, Camel logs details from the current exchange, such as the message body and headers. Section 11.3.3 covers this in more detail.

Let's continue with the next error handler, the dead letter channel.

11.2.2 The dead letter channel error handler

The `DeadLetterChannel` error handler is similar to the default error handler except for the following differences:

- The dead letter channel is the only error handler that supports moving failed messages to a dedicated error queue, known as the *dead letter queue*.
- Unlike the default error handler, the dead letter channel will, by default, handle exceptions and move the failed messages to the dead letter queue.
- The dead letter channel is by default configured to not log any activity when it handles exceptions.
- The dead letter channel supports using the original input message when a message is moved to the dead letter queue.

Let's look at each of these in a bit more detail.

The dead letter channel

`DeadLetterChannel` is an error handler that implements the principles of the Dead Letter Channel EIP. This pattern states that if a message can't be processed or delivered, it should be moved to a dead letter queue. Figure 11.4 illustrates this pattern.

Figure 11.4 The Dead Letter Channel EIP moves failed messages to a dead letter queue.

As you can see, the consumer ❶ consumes a new message that's supposed to be routed to the processor ❸. The channel ❷ controls the routing between ❶ and ❸, and if the message can't be delivered to the processor ❸, the channel invokes the dead letter channel error handler, which moves the message to the dead letter queue ❹. This keeps the message safe and allows the application to continue operating.

This pattern is often used with messaging. Instead of allowing a failed message to block new messages from being picked up, the message is moved to a dead letter queue to get it out of the way.

The same idea applies to the dead letter channel error handler in Camel. This error handler has an associated dead letter queue, which is based on an endpoint, allowing you to use any Camel endpoint you choose. For example, you can use a database or a file, or just log the failed messages.

NOTE The situation gets a bit more complicated when using transactions together with a dead letter queue. Chapter 12 covers this in more detail.

When you choose to use the dead letter channel error handler, you must configure the dead letter queue as an endpoint so the handler knows where to move the failed messages. This is done a bit differently in the Java DSL and XML DSL. For example, here's how to log the message at `ERROR` level in Java DSL:

```
errorHandler(deadLetterChannel("log:dead?level=ERROR"));
```

When using Java DSL, the error handler is configured in the `RouteBuilder` classes, as shown in the following code:

```
public void configure() throws Exception {
  errorHandler(deadLetterChannel("log:dead?level=ERROR"));    ❶
```

1

Configures error handler for all routes in this RouteBuilder class

```
  from("direct:newOrder")    2
```

2

Defines the Camel route(s)

```
       .bean("orderService", "validate")
       .bean("orderService", "store");
   }
```

This configuration defines a `RouteBuilder`-scoped error handler ❶ that applies to all the following routes ❷ defined in the same class:

```
<errorHandler id="myErrorHandler" type="DeadLetterChannel"
              deadLetterUri="log:dead?level=ERROR"/>
```

Notice the difference between Java and XML DSL. In XML DSL, the `type` attribute is used to declare which error handler to use. In XML, the error handler must be configured with an ID, which would be required to enable the error handler on either the context or route scope level. The following listing is an equivalent of the prior Java DSL listing:

```
<camelContext errorHandlerRef="myErrorHandler">
```

```
  <errorHandler id="myErrorHandler" type="DeadLetterChannel    1
```

1

Configures error handler

```
                  deadLetterUri="log:dead?level=ERROR"/>
  <route>    3
```

❸

Defines the Camel route(s)

```
      <from uri="direct:newOrder"/>
      <bean ref="orderService" method="validate"/>
      <bean ref="orderService" method=store"/>
    </route>
  </camelContext>
```

Inside `<camelContext>`, you configure the error handler ❶. The error handlers in XML DSL can be used in two levels:

- Context-scoped level
- Route-scoped level

A route-scoped error handler takes precedence over a context-scoped error handler. In the preceding code, you have to configure the error handler as a context scope ❷ by referring to the error handler by its ID. The following routes ❸ then by default use the context-scoped error handler.

Now, let's look at how the dead letter channel error handler handles exceptions when it moves the message to the dead letter queue.

Handling exceptions by default

By default, Camel handles exceptions by suppressing them; it removes the exceptions from the exchange and stores them as properties on the exchange. After a message has been moved to the dead letter queue, Camel stops routing the message, and the caller regards it as processed.

When a message is moved to the dead letter queue, you can obtain the exception from the exchange by using the `Exchange.EXCEPTION_CAUGHT` property:

```
Exception e = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.
```

Now let's look at using the original message.

Using the original message with the dead letter channel

Suppose you have a route in which the message goes through a series of processing steps, each altering a bit of the message before it reaches its final destination, as in the following code:

```
errorHandler(deadLetterChannel("jms:queue:dead"));

from("jms:queue:inbox")
    .bean("orderService", "decrypt")
    .bean("orderService", "validate")
    .bean("orderService", "enrich")
    .to("jms:queue:order");
```

Now imagine that an exception occurs at the `validate` method, and the dead letter channel error handler moves the message to the dead letter queue. Suppose a new message arrives and an exception occurs at the `enrich` method, and this message is also moved to the same dead letter queue. If you want to retry those messages, can you just drop them into the inbox queue?

In theory, you could do that, but the messages that were moved to the dead letter queue no longer match the messages that originally arrived at the inbox queue—they were altered as the messages were routed. What you want instead is for the original message content to have been moved to the dead letter queue so that you have the original message to retry.

The `useOriginalMessage` option instructs Camel to use the original message when it moves messages to the dead letter queue. You configure the error handler to use the `useOriginalMessage` option as follows:

```
errorHandler(deadLetterChannel("jms:queue:dead").useOriginalMessage());
```

In XML DSL, you'd do this:

```
<errorHandler id="myErrorHandler" type="DeadLetterChannel"
              deadLetterUri="jms:queue:dead" useOriginalMessage="true"/>
```

When the original message (or any other message, for that matter) is moved to a JMS dead letter queue, the message doesn't include any information about the cause of the error, such as the exception message or its stacktrace. The cause of the exception is stored as a property on the exchange, and exchange properties aren't part of the JMS message that Camel sends. To include such information, you'd need to enrich the message with such details before being sent to the `jms:queue:dead` destination.

Enriching message with cause of error

To enrich the message with the cause of the error, you can use a Camel processor to implement the logic for enriching the message. And then you configure the error handler to use the processor. The following listing presents an example.

Listing 11.1 Using a processor to enrich the message before it's sent to the dead letter channel

```
public class FailureProcessor implements Processor {   (1)
```

(1) Processor to enrich the message with details of why it failed

```
  public void process(Exchange exchange) throws Exception {
    Exception e = exchange.getProperty(Exchange.EXCEPTION_CAUGHT,
                                       Exception.class);
    String failure = "The message failed because " + e.getMessage();
    exchange.getIn().setHeader("FailureMessage", failure);   (2)
```

(2) Stores the failure reason as a header on the exchange

```
  }
}

errorHandler(deadLetterChannel("jms:queue:dead")
  .useOriginalMessage().onPrepareFailure(new FailureProcessor());   (3)
```

3 Configures the error handler to use the processor to prepare the failed exchange

To enrich the message, you can implement the details in `org.apache.camel.Processor` ❶. Because you use a JMS message queue as the dead letter queue, you must store the failure reason on the exchange in the message payload; exchange properties aren't included in JMS messaging. Therefore, you use a message header ❷ to store the failure reason. You then need to configure the `DeadLetterChannel` error handler to use the processor before the exchange is sent to the dead letter queue ❸.

Configuring the error handler by using XML DSL is slightly different, as you need to refer to `Processor` by using the `onPrepareFailureRef` attribute on the error handler:

```
<bean id="failureProcessor" class="org.camelinaction.FailureProcessor"/>

<camelContext errorHandlerRef="myErrorHandler" ...>
   <errorHandler id="myErrorHandler" type="DeadLetterChannel"
                 deadLetterUri="jms:queue:dead" useOriginalMessage="true"
                 onPrepareFailureRef="failureProcessor"/>
   ...
</camelContext>
```

The book's source code contains this example in the chapter11/enrich directory. To try the example, use the following Maven goal for either the Java or XML version:

```
mvn test -Dtest=EnrichFailureProcessorTest
mvn test -Dtest=SpringEnrichFailureProcessorTest
```

Instead of using `Processor` to enrich the failed message, you can also use a Camel route, and let the dead letter endpoint call to the route by using the direct component. The previous example can be implemented using a route, as shown in the following listing.

Listing 11.2 Using a route with a bean to enrich the message before it's sent to the dead letter channel

```
public class FailureBean {      1
```

1

Bean to enrich the message with details of why it failed

```
  public void enrich(@Headers Map headers, Exception cause){
    String failure = "The message failed because " + e.getMessage();
    headers.put("FailureMessage", failure);      2
```

2

Stores the failure reason as a header on the message

```
  }
}

errorHandler(deadLetterChannel("direct:dead").useOriginalMessage();      3
```

3

Configures the error handler to use a route to prepare the failed exchange

```
from("direct:dead")      4
```

4

The route that enriches the exchange by calling the bean and sends the message to the dead letter queue, which is mock:dead in this example

```
    .bean(new FailureBean())
    .to("mock:dead");    ❺
```

❺ Using mock endpoint as the dead letter queue

You use a Java bean to enrich the message ❶. Using a Java bean allows you to use Camel's parameter binding in the method signature. The first parameter is the message header, which is mapped using the `@Headers` annotation. The second parameter is the cause of the exception. You then construct the error message ❷, which you add to the headers. The error handler is configured to call the `"direct:dead"` endpoint ❸, which is a route that routes to the bean ❹. After the message has been enriched from the bean, the message is then routed to its final destination, which is the dead letter queue ❺.

The book's source code contains this example in the chapter11/enrich directory. To try the example, use the following Maven goal for either the Java or XML version:

```
mvn test -Dtest=EnrichFailureBeanTest
mvn test -Dtest=SpringEnrichFailureBeanTest
```

In this example, the route is just calling a bean, but you're free to expand the route to do more work. There's only one consideration—the fact that the route is triggered by the error handler. What if the route causes a new exception to happen? Would the error handler not react again, and call the `direct:dead` route with the new exception, and if the new exception also caused yet another new exception, don't we have a potential endless loop with error handling? Yes, and that's why Camel has a built-in fatal error handler that detects this and breaks out of this situation. Section 11.4.5 covers this in more detail.

Let's move on to the transaction error handler.

11.2.3 The transaction error handler

`TransactionErrorHandler` is built on top of the default error handler and offers the same functionality, but it's tailored to support transacted routes. Chapter 12 focuses on transactions and discusses this error handler in detail, so we won't say much about it here. For now, you just need to know that it exists and it's a core part of Camel.

The remaining two error handlers are less commonly used and are much simpler.

11.2.4 The no error handler

`NoErrorHandler` is used to disable error handling. The current architecture of Camel mandates that an error handler must be configured, so if you want to disable error handling, you need to provide an error handler that's basically an empty shell with no real logic. That's `NoErrorHandler`.

11.2.5 The logging error handler

`LoggingErrorHandler` logs the failed message along with the exception. The logger uses standard log format from log kits such as log4j, commons logging, or the Java Util Logger.

Camel will, by default, log the failed message and the exception by using the log name `org.apache.camel.processor.LoggingErrorHandler` at the `ERROR` level. You can, of course, customize this.

The logging error handler can be replaced by using the dead letter channel error handler and using a log endpoint as the destination. Therefore, the logging error handler has been deprecated in favor of this approach.

That covers the five error handlers provided with Camel. Let's now look at the major features these error handlers provide.

11.2.6 Features of the error handlers

The default, dead letter channel, and transaction error handlers are all built on the same base, `org.apache.camel.processor.RedeliveryErrorHandler`, so they all have several major features in common. These features are listed in table 11.2.

Table 11.2 Noteworthy features provided by the error handlers

Feature	Description
Redelivery policies	Redelivery policies allow you to indicate whether redelivery should be attempted. The policies also define settings such as the maximum number of redelivery attempts, delays between attempts, and so on.
Scope	Camel error handlers have two possible scopes: context (high level) and route (low level). The context scope allows you to reuse the same error handler for multiple routes, whereas the route scope is used for a single route only.
Exception policies	Exception policies allow you to define special policies for specific exceptions.
Error handling	This option allows you to specify whether the error handler should handle the error. You can let the error handler deal with the error or leave it for the caller to handle.

Let's continue by covering these noteworthy features. We'll start with redelivery and scope in section 11.3. Exception policies and error handling are covered in section 11.4.

11.3 Using error handlers with redelivery

Communicating with remote servers relies on network connectivity that can be unreliable and have outages. Luckily, these disruptions cause recoverable errors—the network connection could be reestablished in a matter of seconds or minutes. Remote services can also be the source of temporary problems, such as when the service is restarted by an administrator. To help address these problems, Camel supports a redelivery mechanism that allows you to control how recoverable errors are dealt with.

This section looks at a real-life error-handling scenario and then focuses on how Camel controls redelivery and how to configure and use it. We'll also look at using error handlers with fault messages. We end this section by looking at error-handling scope and how it can be used to support multiple error handlers scoped at different levels.

11.3.1 An error-handling use case

Suppose you've developed an integration application at Rider Auto Parts that once every hour should upload files from a local directory to an HTTP server, and your boss asks why the files haven't been updated in the last few days. You're surprised, because the application has been running for the last month without a problem. This could well be a situation where neither error handling nor monitoring was in place.

Here's the Java file that contains the integration route:

```
from("file:/riders/files/upload?delay=15m")
    .to("http://riders.com/upload?user=gear&password=secret");
```

This route will periodically scan for files in the /riders/files/upload folder, and if any files exist, it will upload them to the receiver's HTTP server using the HTTP endpoint.

But no explicit error handling is configured, so if an error occurs, the default error handler is triggered. That handler doesn't handle the exception but instead propagates it back to the caller. Because the caller is the file consumer, it'll log the exception and do a file rollback, meaning that any picked-up files will be left on the filesystem, ready to be picked up in the next scheduled poll.

At this point, you need to reconsider how errors should be handled in the application. You aren't in major trouble, because you haven't lost any files. Camel will move only successfully processed files out of the upload folder; failed files will just stack up.

The error occurs when sending the files to the HTTP server, so you look into the log files and quickly determine that Camel can't connect to the remote HTTP server because of network issues. Your boss decides that the application should retry uploading the files if there's an error, so the files won't have to wait for the next hourly upload.

To implement this, you can configure the error handler to redeliver up to 5 times with 10-second delays:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5)
    .redeliveryDelay(10000));
```

Configuring redelivery can hardly get any simpler than that. But let's take a closer look at how to use redelivery with Camel.

11.3.2 Using redelivery

The first three error handlers in table 11.1 all support redelivery. This is implemented in the `RedeliveryErrorHandler` class, which they extend. `RedeliveryErrorHandler` must then know whether to attempt redelivery; that's what the redelivery policy is for.

A redelivery policy defines how and whether redelivery should be attempted. Table 11.3 outlines the most commonly used options supported by the redelivery policy and the default settings. The Camel team is working on a new website for the project that will overhaul the online documentation and include up-to-date documentation for all the redelivery options. Until then, you can find this information in the source code by looking at the `RoutePolicy` class:
https://github.com/apache/camel/blob/master/camel-core/src/main/java/org/apache/camel/processor/RedeliveryPolicy.java.

Table 11.3 Options provided in Camel for configuring redelivery

Option	Type	Default	Description
`backOffMultiplier`	`double`	`2.0`	Exponential back-off multiplier used to multiply each consequent delay. `redeliveryDelay` is the starting delay. Exponential back-off is disabled by default.
`collisionAvoidanceFactor`	`double`	`0.15`	A percentage to use when calculating a random delay offset (to avoid using the same delay at the next attempt). Will start with the `redeliveryDelay` as the starting delay. Collision avoidance is disabled by default.
`logExhausted`	`boolean`	`true`	Specifies whether the exhaustion of redelivery attempts (when all redelivery attempts have

Option	Type	Default	Description
			failed) should be logged.
`logExhaustedMessageBody`	`boolean`	`false`	Specifies whether the message history should include the message body and headers. This is turned off by default to avoid logging content from the message payload, which can carry sensitive data.
`logExhaustedMessageHistory`	`boolean`		Specifies whether the message history should be logged when logging is exhausted. This option is by default `true` for all error handlers, except the dead letter channel, where it's `false`.
`logRetryAttempted`	`boolean`	`true`	Specifies whether redelivery attempts should be logged.

Option	Type	Default	Description
`logStackTrace`	`boolean`	`true`	Specifies whether stacktraces should be logged when all redelivery attempts have failed.
`maximumRedeliveries`	`int`	`0`	Maximum number of redelivery attempts allowed. `0` is used to disable redelivery, and `-1` will attempt redelivery forever until it succeeds.
`maximumRedeliveryDelay`	`long`	`60000`	An upper bound in milliseconds for redelivery delay. This is used when you specify nonfixed delays, such as exponential back-off, to avoid the delay growing too large.
`redeliveryDelay`	`long`	`1000`	Fixed delay in milliseconds between each

Option	Type	Default	Description
			redelivery attempt.
`useExponentialBackOff`	`boolean`	`false`	Specifies whether exponential back-off is in use.

In the Java DSL, Camel has fluent builder methods for configuring the redelivery policy on the error handler. For instance, if you want to redeliver up to five times, use exponential back-off, and have Camel log at `WARN` level when it attempts a redelivery, you could use this code:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5)
    .retryAttemptedLogLevel(LoggingLevel.WARN)
    .backOffMultiplier(2)
    .useExponentialBackOff());
```

Configuring this in XML DSL is done as follows:

```
<errorHandler id="myErrorHandler" type="DefaultErrorHandler"
    <redeliveryPolicy maximumRedeliveries="5"
                      retryAttemptedLogLevel="WARN"
                      backOffMultiplier="2"
                      useExponentialBackOff="true"/>
</errorHandler>
```

Notice in XML DSL that you configure the redelivery policies by using the `<redeliveryPolicy>` element.

We've now established that Camel uses the information from the redelivery policy to determine whether and how to do redeliveries. But what happens inside Camel? As you'll recall from figure 11.4, Camel includes a channel between every processing step in a route path, and there's functionality in these channels, such as error handlers. The error handler detects every exception that occurs and acts on it, deciding what to do, such as redeliver or give up.

Now that you know a lot about `DefaultErrorHandler`, it's time to try a little example.

11.3.3 Using DefaultErrorHandler with redelivery

In the book's source code, you'll see an example in the chapter11/errorhandler directory. The example uses the following route configuration:

```
errorHandler(defaultErrorHandler()  ❶
```

❶

Configures error handler

```
     .maximumRedeliveries(2)
     .redeliveryDelay(1000)
     .retryAttemptedLogLevel(LoggingLevel.WARN));

from("seda:queue.inbox")
     .bean("orderService", "validate")
     .bean("orderService", "enrich")  ❷
```

❷

Invokes enrich method

```
     .log("Received order ${body}")
     .to("mock:queue.order");
```

This configuration first defines a context-scoped error handler ❶ that will attempt at most two redeliveries using a one-second delay. When it attempts the redelivery, it will log this at the `WARN` level (as you'll see in a few seconds). The example is constructed to fail when the message reaches the `enrich` method ❷.

And here's the example using XML DSL instead:

```
<camelContext errorHandlerRef="myErrorHandler">
  <errorHandler id="myErrorHandler" type="DefaultErrorHandler">
    <redeliveryPolicy maximumRedeliveries="2"
                      redeliveryDelay="1000"
                      retryAttemptedLogLevel="WARN"/>
  </errorHandler>

  <route>
    <from uri="seda:queue.inbox"/>
    <bean ref="orderService" method="validate"/>
    <bean ref="orderService" method="enrich"/>
    <log message="Received order ${body}"/>
    <to uri="mock:queue.order"/>
  </route>
</camelContext>
```

You can run this example by using the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=DefaultErrorHandlerTest
mvn test -Dtest=SpringDefaultErrorHandlerTest
```

When running the example, you'll see the following log entries outputted on the console. Notice how Camel logs the redelivery attempts:

```
2017-03-08 14:28:16,959 [a://queue.inbox] WARN DefaultErrorHandler - Fai
2017-03-08 14:28:17,960 [a://queue.inbox] WARN DefaultErrorHandler - Fai
```

These log entries show that Camel failed to deliver a message, which means the entry is logged after the attempt is made. `On delivery attempt: 0` identifies the first attempt; attempt 1 is the first redelivery attempt. Camel also logs the `exchangeId` (which you can use to correlate messages) and the exception that caused the problem (without the stacktrace, by default).

When Camel performs a redelivery attempt, it does so at the point of origin. In the preceding example, the error occurs when invoking the `enrich` method ❷, which means Camel will redeliver by retrying the `.bean("orderService", "enrich")` step in the route.

CAMEL REDELIVERY HAPPENS AT THE POINT OF ERROR

It's important to understand that Camel's error handler performs redelivery at the point of error, and not from the beginning of the route. When an exception happens during routing, Camel doesn't *roll back* the entire exchange and start all over again. No, Camel performs the redelivery at the *same point* where the error happened. The next chapter covers transactions, including the concept of rolling back a transaction and restarting the transaction process all over again. But you shouldn't mix up these concepts with transactions using Camel's error handler. They aren't the same. That said, tricks can allow you to let Camel's error handler redeliver an entire route in case an exception was thrown anywhere from the route. Section 11.4.7 covers such an example.

After all redelivery attempts have failed, we say it's *exhausted*, and Camel logs this at the `ERROR` level by default. (You can customize this with the options listed in table 11.3.) When the redelivery attempts are exhausted, the log entry is similar to the previous ones, but Camel explains that it's exhausted after three attempts:

```
2017-03-08 14:28:18,961 [a://queue.inbox] ERROR DefaultErrorHandler -
Failed delivery for (MessageId: ID-davsclaus-pro-local-1512...).
Exhausted after delivery attempt: 3 caught: camelinaction.OrderException
ActiveMQ in Action is out of stock
```

In addition to logging the exception message, Camel also logs a *route trace* that prints each step from all the routes the message had taken up until this error, as shown here:

```
Message History
---------------------------------------------------------------------------
RouteId  ProcessorId   Processor                                 Elapsed (ms)
[route2] [route2     ] [seda://queue.inbox                     ] [       3009]
[route2] [bean3      ] [bean[ref:orderService method: val]     ] [          1]
[route2] [bean4      ] [bean[ref:orderService method: enh]     ] [       2007]
```

Here you can see that the exception occurred at the last step, which is at ID `bean4`, which would be calling the `orderService` bean and the `en-`

`rich` method. Further below is the full stacktrace of the caused exception:

```
Stacktrace
---------------------------------------------------------------------
camelinaction.OrderException: ActiveMQ in Action is out of stock
  at camelinaction.OrderService.enrich(OrderService.java:22)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at org.apache.camel.component.bean.MethodInfo.invoke(MethodInfo.java:4
```

As you can see, detailed information from Camel is provided when an exception occurs, and the exception can't be mitigated by the error handler (all redelivery attempts have failed, and the exchange is regarded as exhausted).

The level of information can be turned down so that it's only the exception being logged by setting the option log exhausted message history to `false`:

```
errorHandler(defaultErrorHandler()
  .maximumRedeliveries(2)
  .redeliveryDelay(1000)
  .retryAttemptedLogLevel(LoggingLevel.WARN)
  .logExhaustedMessageHistory(false);
```

And here's how to configure it using XML DSL:

```
<errorHandler id="myErrorHandler" type="DefaultErrorHandler">
    <redeliveryPolicy redeliveryDelay="1000"
                      retryAttemptedLogLevel="WARN"
                      logExhaustedMessageHistory="false"/>
</errorHandler>
```

TIP The default error handler has many options, which are listed in table 11.3. We encourage you to try loading this example into your IDE and playing with it. Change the settings on the error handler and see what happens.

Unlike the default error handler, the dead letter channel won't log any activity by default. But you can configure the dead letter channel to do so, such as setting `logExhaustedMessageHistory=true` to enable the detailed logging.

The preceding log output identifies the number of redelivery attempts, but how does Camel know this? Camel stores this information on the exchange. Table 11.4 reveals where this information is stored.

Table 11.4 Headers on the exchange related to error handling

Header	Type	Description
`Exchange.REDELIVERY_COUNTER`	`int`	The current redelivery attempt
`Exchange.REDELIVERED`	`boolean`	Whether this exchange is being redelivered
`Exchange.REDELIVERY_EXHAUSTED`	`boolean`	Whether this exchange has attempted (exhausted) all redeliveries and has still failed

The information in table 11.4 is available only when Camel performs a redelivery; these headers are absent on the regular first attempt. It's only when a redelivery is triggered that these headers are set on the exchange.

11.3.4 Error handlers and scopes

Scopes can be used to define error handlers at different levels. Camel supports two scopes: a context scope and a route scope.

IMPORTANT When using Java DSL, context scope is implemented as per the `RouteBuilder` scope, but can become a global scope by letting all `RouteBuilder` classes inherit from a base class where the context-scoped error handler is configured. You'll see this in action later in this section.

Camel allows you to define a global context-scoped error handler that's used by default, and, if needed, you can also configure a route-scoped error handler that applies only to a particular route. This is illustrated in the following listing.

Listing 11.3 Using two error handlers at different scopes

```
errorHandler(defaultErrorHandler()    1
```

1

Defines context-scoped error handler

```
        .maximumRedeliveries(2)
        .redeliveryDelay(1000)
        .retryAttemptedLogLevel(LoggingLevel.WARN));

    from("file://target/orders?delay=10000")
        .bean("orderService", "toCsv")
        .to("mock:file")
        .to("seda:queue.inbox");

    from("seda:queue.inbox")
        .errorHandler(deadLetterChannel("log:DLC")    2
```

2

Defines route-scoped error handler

```
            .maximumRedeliveries(5).retryAttemptedLogLevel(LoggingLevel.INFO
            .redeliveryDelay(250).backOffMultiplier(2))
```

```
      .bean("orderService", "validate")
   .bean("orderService", "enrich") ❸
```

❸ Invokes enrich method

```
      .to("mock:queue.order");
```

Listing 11.3 is an improvement over the previous error-handling example. The default error handler is configured as in the previous example ❶, but you have a new route that picks up files, processes them, and sends them to the second route. This first route will use the default error handler ❶ because it doesn't have a route-scoped error handler configured, but the second route has a route-scoped error handler ❷. It's a `DeadLetterChannel` that will send failed messages to a log. Notice that it has different options configured than the former error handler.

The book's source code includes this example, which you can run by using the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=RouteScopeTest
```

This example should fail for some messages when the `enrich` method ❸ is invoked. This demonstrates how the route-scoped error handler is used as an error handler.

The most interesting part of this test class is the `testOrderActiveMQ` method (see the example with the source code), which will fail in the second route and therefore show the dead letter channel in action. There are a couple of things to notice about this, such as the exponential back-off, which causes Camel to double the delay between redelivery attempts, starting with 250 milliseconds and ending with 4 seconds.

The following snippets show what happens at the end when the error handler is exhausted:

```
2017-03-08 17:03:44,534 [a://queue.inbox] INFO DeadLetterChannel - Faile
delivery for (MessageId: ID-davsclaus-pro-local-1512...). On delivery
```

```
attempt: 5 caught: camelinaction.OrderException: ActiveMQ in Action is o
of stock
2017-03-08 17:03:44,541 [a://queue.inbox] INFO  DLC -
Exchange[BodyType:String, Body:amount=1,name=ActiveMQ in Action,id=123]
2017-03-08 17:03:44,542 [a://queue.inbox] ERROR DeadLetterChannel - Fail
delivery for (MessageId: ID-davsclaus-pro-local-1512...). Exhausted
after delivery attempt: 6 caught: camelinaction.OrderException: ActiveMQ
Action is out of stock. Processed by failure processor:
sendTo(Endpoint[log://DLC])
```

As you can see, the dead letter channel moves the message to its dead letter queue, which is the `log://DLC` endpoint. After this, Camel also logs an `ERROR` line indicating that this move was performed.

We encourage you to try this example and adjust the configuration settings on the error handlers to see what happens.

So far, the error-handling examples in this section have used the Java DSL. Let's look at configuring error handling with XML DSL.

Using error handling with XML DSL

Let's revise the example in listing 11.3 to use XML DSL. Here's how that's done.

Listing 11.4 Using error handling with XML DSL

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

<camelContext id="camel" errorHandlerRef="defaultEH"   1
```

1

Specifies context-scoped error handler

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

  <errorHandler id="defaultEH">   2
```

2 Sets up context-scoped error handler

```
    <redeliveryPolicy maximumRedeliveries="2" redeliveryDelay="1000"
                      retryAttemptedLogLevel="WARN"/>
  </errorHandler>

  <errorHandler id="dlc" 3
```

3 Sets up route-scoped error handler

```
                type="DeadLetterChannel" deadLetterUri="log:DLC">
    <redeliveryPolicy maximumRedeliveries="5" redeliveryDelay="250"
                      retryAttemptedLogLevel="INFO"
                      backOffMultiplier="2" useExponentialBackOff="true"
  </errorHandler>

  <route>
    <from uri="file://target/orders?delay=10000"/>
    <bean ref="orderService" method="toCsv"/>
    <to uri="mock:file"/>
    <to uri="seda:queue.inbox"/>
  </route>

  <route errorHandlerRef="dlc"> 4
```

4 Specifies route-scoped error handler

```
    <from uri="seda:queue.inbox"/>
    <bean ref="orderService" method="validate"/>
    <bean ref="orderService" method="enrich"/>
    <to uri="mock:queue.order"/>
  </route>
</camelContext>
```

To use a context-scoped error handler in XML DSL, you must configure it by using an `errorHandlerRef` attribute ❶ on the `camelContext` tag. The `errorHandlerRef` refers to an `<errorHandler>`, which in this case is the default error handler with ID `"defaultEH"` ❷. There's another error handler, a `DeadLetterChannel` error handler ❸, that's used at route scope in the second route ❹.

As you can see, the differences between the Java DSL and XML DSL mostly result from using the `errorHandlerRef` attribute to reference the error handlers in XML DSL, whereas Java DSL can have route-scoped error handlers within the routes.

You can try this example by running the following Maven goal from the chapter11/errorhandler directory:

```
mvn test -Dtest=SpringRouteScopeTest
```

The XML DSL file is located in the src/test/resources/camelinaction directory.

Now we've covered how to use error handlers in Java and XML DSL, but there's a subtle difference when using error handlers in the two DSLs. In XML DSL, context-scoped error handlers are reusable among all the routes as is. But in Java DSL, you must use an abstract base class, which your `RouteBuilder` classes extend to reuse context-scoped error handlers. The following section highlights the difference.

11.3.5 Reusing context-scoped error handlers

Reusing a context-scoped error handler in XML DSL is straightforward, as you'd expect. The following listing shows how easy that is.

Listing 11.5 Reusing context-scoped error handler in XML DSL

```
<bean id="orderService" class="camelinaction.OrderService"/>
<camelContext id="camel" errorHandlerRef="defaultEH"     ❶
```

❶

Specifies context-scoped error handler

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

  <errorHandler id="defaultEH">  ❷
```

❷ Sets up context-scoped error handler

```
      <redeliveryPolicy maximumRedeliveries="2" redeliveryDelay="1000"
                        retryAttemptedLogLevel="WARN"/>
    </errorHandler>

  <route>  ❸
```

❸ Route using context-scoped error handler out of the box

```
      <from uri="file://target/orders?delay=10000"/>
      <bean ref="orderService" method="toCsv"/>
      <to uri="mock:file"/>
      <to uri="seda:queue.inbox"/>
    </route>

  <route>  ❹
```

❹ Another route using context-scoped error handler out of the box

```
      <from uri="seda:queue.inbox"/>
      <bean ref="orderService" method="validate"/>
      <bean ref="orderService" method="enrich"/>
      <to uri="mock:queue.order"/>
    </route>
  </camelContext>
```

In XML DSL, you configure the context-scoped error handler on `<camelContext>` by using the `errorHandlerRef` attribute ❶. The error handler is then configured by using the `<errorHandler>` tag ❷. And all routes (❸ ❹) will automatically use the context-scoped error handler by default.

In Java DSL, the situation is a bit different, as the context-scoped error handler is essentially scoped to its `RouteBuilder` instance. The same example from listing 11.5 has been ported to Java DSL in the following listing.

Listing 11.6 Reusing context-scoped error handler in Java DSL

```
public abstract class BaseRouteBuilder extends RouteBuilder {  ❶
```

❶

Base class to hold the reused error handler

```
    public void configure() throws Exception {
        errorHandler(deadLetterChannel("mock:dead")  ❷
```

❷

The context-scoped error handler to reuse

```
            .maximumRedeliveries(2)
            .redeliveryDelay(1000)
            .retryAttemptedLogLevel(LoggingLevel.WARN));
    }
}

public class InboxRouteBuilder extends BaseRouteBuilder {  ❸
```

❸

RouteBuilder extending the base class

```
    public void configure() throws Exception {
        super.configure();     (4)
```

(4) Calling super.configure() to reuse the context-scoped error handler

```
        from("file://target/orders?delay=10000")     (5)
```

(5) Route that will use the context-scoped error handler

```
                  .bean("orderService", "toCsv")
                  .to("mock:file")
                  .to("seda:queue.inbox");
        }
    }

    public class OrderRouteBuilder extends BaseRouteBuilder {     (6)
```

(6) Another RouteBuilder extending the base class

```
        public void configure() throws Exception {
        super.configure();     (7)
```

(7) Calling super.configure() to reuse the context-scoped error handler

```
        from("seda:queue.inbox")     (8)
```

❽

Route that also will use the context-scoped error handler

```
                .bean("orderService", "validate")
                .bean("orderService", "enrich")
                .to("mock:queue.order");
        }
    }
```

In Java DSL, you'd need to use object-oriented principles such as inheritance to reuse the context-scoped error handler. Therefore, you create a base class ❶, where you configure the context-scoped error handler ❷ in the `configure` method. Our `RouteBuilder` classes now must extend the base class (❸ ❻) and call the `super.configure` method (❹ ❼), which calls the parent class that has the error handler to reuse. Each of the routes (❺ ❽) will now use the context-scoped error handler that you defined in the base class ❷.

As you can see, the subtle difference is that in Java DSL you must rely on inheritance and must remember to call the `super.configure` method.

We encourage you to try this example in action with the source code from the book. The example is in the chapter11/reuse directory, and you can try the example with the following Maven goal:

```
mvn test -Dtest=ReuseErrorHandlerTest
mvn test -Dtest=SpringReuseErrorHandlerTest
```

Try to experiment and see what happens if you forget to call the `super.configure` method in the `OrderRouteBuilder` class.

This concludes our discussion of scopes and redelivery.

We'll continue in the next section to look at the other two major features that error handlers provide, as listed in table 11.2: exception policies and error handling. This is a good time to take a break before continuing. Go grab a fresh cup of coffee or tea; or maybe it's time to walk the dog or empty the dishwasher.

11.4 Using exception policies

Exception policies are used to intercept and handle specific exceptions in particular ways. For example, exception policies can influence, at runtime, the redelivery policies the error handler is using. They can also handle an exception or even detour a message.

NOTE In Camel, exception policies are specified with the `onException` method in the route, so we use the term `onException` interchangeably with *exception policy*.

We'll cover exception policies piece by piece, looking at how they catch exceptions, how they work with redelivery, and how they handle exceptions. Then we'll take a look at custom error handling and put it all to work in an example.

11.4.1 Understanding how onException catches exceptions

We'll start by looking at how Camel inspects the exception hierarchy to determine how to handle the error. This will give you a better understanding of how you can use `onException` to your advantage.

Imagine you have this exception with underlying wrapped exceptions being thrown:

```
org.apache.camel.RuntimeCamelException (wrapper by Camel)
+ com.mycompany.OrderFailedException
  + java.net.ConnectException
```

The real cause is a `ConnectException`, but it's wrapped in an `OrderFailedException` and yet again in a `RuntimeCamelException`.

Camel will traverse the hierarchy from the bottom up to the root searching for an `onException` that matches the exception. In this case, Camel will start with `java.net.ConnectException`, move on to `com.mycompany.OrderFailedException`, and finally reach `RuntimeCamelException`. For each of those three exceptions, Camel will compare the exception to the defined `onExceptions` to select the best matching `onException` policy. If no suitable policy can be found, Camel relies on the configured error-handler settings. You'll drill down and look at how the matching works, but for now you can think of this as Camel doing a big `instanceof` check against the exceptions in the hierarchies, following the order in which the `onExceptions` were defined.

Suppose you have a route with the following `onException`:

```
onException(OrderFailedException.class).maximumRedeliveries(3);
```

The aforementioned `ConnectException` is being thrown, and the Camel error handler is trying to handle this exception. Because you have an exception policy defined, it will check whether the policy matches the thrown exception. The matching is done as follows:

1. Camel starts with the `java.net.ConnectException` and compares it to `onException(OrderFailedException.class)`. Camel checks whether the two exceptions are exactly the same type, and in this case they're not—`ConnectionException` and `OrderFailedException` aren't the same type.
2. Camel checks whether `ConnectException` is a subclass of `OrderFailedException`, and this isn't true either. So far, Camel hasn't found a match.
3. Camel moves up the exception tree (wrapped by) and compares again with `OrderFailedException`. This time there's an exact match, because they're both of the type `OrderFailedException`.

No more matching takes place. Camel got an exact match, and the exception policy will be used.

When an exception policy has been selected, its configured policy will be used by the error handler. In this example, the policy defines the maximum redeliveries to be `three`, so the error handler will attempt at most three redeliveries when this kind of exception is thrown.

Any value configured on the exception policy will override options configured on the error handler. For example, suppose the error handler had the `maximumRedeliveries` option configured as `5`. Because the `onException` has the same option configured, its value of `3` will be used instead.

IMPORTANT The book's source code has an example that demonstrates what you've just learned. Take a look at the `OnExceptionTest` class in chapter11/onexception. It has multiple test methods (`testOnExceptionDirectMatch` and `testOnExceptionWrappedMatch`), each showing a scenario of how `onException` works. You can run the tests by using Maven from the command shell: `mvn test -Dtest=OnExceptionTest`.

Let's make the example a bit more interesting and add a second `onException` definition:

```
onException(OrderFailedException.class).maximumRedeliveries(3);
onException(ConnectException.class).maximumRedeliveries(10);
```

If the same exception hierarchy is thrown as in the previous example, Camel would select the second `onException` because it directly matches `ConnectionException`. This allows you to define different strategies for different kinds of exceptions. In this example, it's configured to use more redelivery attempts for connection exceptions than for order failures.

TIP This example demonstrates how `onException` can influence the redelivery polices the error handler uses. If an error handler were configured to perform only three redelivery attempts, the preceding `onException` would overload this with 10 redelivery attempts in the case of connection exceptions.

But what if there are no direct matches? Let's look at another example. This time, imagine that a `java.io.IOException` exception is thrown. Camel will do its matching, and because `OrderFailedException` isn't a direct match, and `IOException` isn't a subclass of it, it's out of the game. The same applies for `ConnectException`. In this case, there are no `onException` definitions that match, and Camel will fall back to using the configuration of the current error handler.

You can see this in action by running the following Maven goal from the chapter 11/onexception directory:

```
mvn test -Dtest=OnExceptionFallbackTest
```

onException and gap detection

Can Camel do better if there isn't a direct hit? Yes, because Camel uses a gap-detection mechanism that calculates the gaps between a thrown exception and the `onException`s and then selects the `onException` with the lowest gap as the winner. An example will explain this better.

Suppose you have these three `onException` definitions, each having a different redelivery policy:

```
onException(ConnectException.class)
    .maximumRedeliveries(5);

onException(IOException.class)
    .maximumRedeliveries(3).redeliveryDelay(1000);

onException(Exception.class)
    .maximumRedeliveries(1).redeliveryDelay(5000);
```

And imagine this exception is thrown:

```
org.apache.camel.OrderFailedException
+ java.io.FileNotFoundException
```

Which of those three `onException`s would be selected?

Camel starts with `java.io.FileNotFoundException` and compares it to the `onException` definitions. Because there are no direct matches, Camel uses gap detection. In this example, only `onException(IOException.class)` and `onException(Exception.class)` partly match, because `java.io.FileNotFoundException` is a subclass of `java.io.IOException` and `java.lang.Exception`.

Here's the exception inheritance hierarchy for `FileNotFoundException`:

```
java.lang.Exception
+ java.io.IOException
```

```
    + java.io.FileNotFoundException
```

Looking at this exception hierarchy, you can see that `java.io.FileNotFoundException` is a direct subclass of `java.io.Exception`, so the gap is computed as 1. The gap between `java.lang.Exception` and `java.io.FileNotFoundException` is 2. At this point, the best candidate has a gap of 1.

Camel will then follow the same process with the next exception from the thrown exception hierarchy, which is `OrderFailedException`. This time, it's only the `onException(Exception.class)` that partly matches, and the gap between `OrderFailed-Exception` and `Exception` is also 1:

```
java.lang.Exception
+ OrderNotFoundException
```

What now? You have two gaps, both calculated as 1. In the case of a tie, Camel will always pick the first match, because the cause exception is most likely the last in the hierarchy. In this case, it's a `FileNotFoundException`, so the winner will be `onException(IOException.class)`.

This example is provided in the source code for the book in the chapter11/onexception directory. You can try it using the following Maven goal:

```
mvn test -Dtest=OnExceptionGapTest
mvn test -Dtest=SpringOnExceptionGapTest
```

Multiple exceptions per onException

So far, you've seen only examples with one exception per `onException`, but you can define multiple exceptions in the same `onException`:

```
onException(XPathException.class, TransformerException.class)
    .to("log:xml?level=WARN");
onException(IOException.class, SQLException.class, JMSException.class)
    .maximumRedeliveries(5).redeliveryDelay(3000);
```

Here's the same example using XML DSL:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <onException>
        <exception>javax.xml.xpath.XPathException</exception>
        <exception>javax.xml.transform.TransformerException</exception>
        <to uri="log:xml?level=WARN"/>
    </onException>
    <onException>
        <exception>java.io.IOException</exception>
        <exception>java.sql.SQLException</exception>
        <exception>javax.jms.JMSException</exception>
        <redeliverPolicy maximumRedeliveries="5" redeliveryDelay="3000"/
    </onException>
</camelContext>
```

Our next topic is how `onException` works with redelivery. Even though we've touched on this already in our examples, we'll go into the details in the next section.

11.4.2 Understanding how onException works with redelivery

`onException` works with redeliveries, but you need to be aware of a couple of things that might not be immediately obvious.

Suppose you have the following route:

```
from("jetty:http://0.0.0.0/orderservice")
    .to("netty4:tcp://erp.rider.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

You use the Camel Jetty component to expose an HTTP service where statuses of pending orders can be queried. The order status information is retrieved from a remote ERP system by the Netty component using low-level socket communication. You've learned how to configure this on the error handler itself, but it's also possible to configure this on `onException`.

Suppose you want Camel to retry invoking the external TCP service, in case there has been an I/O-related error, such as a lost network connection. To do this, you can add `onException` and configure the redelivery policy as you like. In the following example, the redelivery tries at most five times:

```
onException(IOException.class).maximumRedeliveries(5);
```

You've already learned that `onException(IOException.class)` will catch those I/O-related exceptions and act accordingly. But what about the delay between redeliveries?

In this example, the delay will be 1 second. Camel will use the default redelivery policy settings outlined in table 11.3 and then override those values with values defined in `onException`. Because the delay wasn't overridden in `onException`, the default value of 1 second is used.

TIP When you configure redelivery policies, they override the existing redelivery policies set in the current error handler. This is convention over configuration, because you need to configure only the differences, which is often just the number of redelivery attempts or a different redelivery delay.

Now let's make it a bit more complicated:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .redeliveryDelay(2000));

onException(IOException.class).maximumRedeliveries(5);

from("jetty:http://0.0.0.0/orderservice")
    .to("netty4:tcp://erp.rider.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

What would the redelivery delay be if `IOException` were thrown? Yes, it's 2 seconds, because `onException` will fall back and use the redelivery policies defined by the error handler, and its value is configured as `redeliveryDelay(2000)`.

Now let's remove the `maximumRedeliveries (5)` option from `onException`, so it's defined as `onException(IOException.class)`:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .redeliveryDelay(2000));

onException(IOException.class);

from("jetty:http://0.0.0.0/orderservice")
    .to("netty4:tcp://erp.rider.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

What would the maximum number of redeliveries be now, if `IOException` were thrown? You'll probably say the answer is 3—the value defined on the error handler. In this case, though, the answer is 0. Camel won't attempt any redelivery because any `onException` will override the `maximumRedeliveries` to 0 by default (redelivery is disabled by default) unless you explicitly set the `maximumRedeliveries` option.

The reason Camel implements this behavior is our next topic: using `onException` to handle exceptions.

11.4.3 Understanding how onException can handle exceptions

Suppose you have a complex route that processes a message in multiple steps. Each step does some work on the message, but any step can throw an exception to indicate that the message can't be processed and that it should be discarded. This is where handling exceptions with `onException` comes into the game.

Handling an exception with `onException` is similar to exception handling in Java itself. You can think of it as being like using a `try ... catch` block.

This is best illustrated with an example. Imagine you need to implement an ERP server-side service that serves order statuses. This is the ERP service you called from the previous section:

```
public void configure() {
    try {
            from("netty4:tcp://0.0.0.0:4444?textline=true")
                .process(new ValidateOrderId())
                .to("jms:queue:order.status")
                .process(new GenerateResponse());
        } catch (JMSException e) {
            .process(new GenerateFailureResponse());     ❶
```

❶ Rethrows caught exception

```
        }
    }
```

This snippet of pseudocode involves multiple steps in generating the response. If something goes wrong, you catch the exception and return a failure response ❶.

We call this *pseudocode* because it shows your intention but the code won't compile. This is because the Java DSL uses the fluent builder syntax, where method calls are stacked together to define the route. The regular `try ... catch` mechanism in Java works at runtime to catch exceptions that are thrown when the `configure` method is executed, but in this case the `configure` method is invoked only once, when Camel is started (when it initializes and builds up the route path to use at runtime).

Don't despair. Camel has a counterpart to the classic `try ... catch ... finally` block in its DSL: `doTry ... doCatch ... doFinally`.

Using doTry, doCatch, and doFinally

The following listing shows how to make the code compile and work at runtime as you would expect with a `try ... catch` block.

Listing 11.7 Using `doTry ... doCatch` with Camel routing

```
    public void configure() {
        from("netty4:tcp://0.0.0.0:4444?textline=true")
            .doTry()
                .process(new ValidateOrderId())
                .to("jms:queue:order.status")
                .process(new GenerateResponse())
            .doCatch(JMSException.class)
                .process(new GenerateFailureResponse())
            .end();
    }
```

The `doTry` `...` `doCatch` block is a bit of a sidetrack, but it's useful because it helps bridge the gap between thinking in regular Java code and thinking in EIPs.

Using onException to handle exceptions

The `doTry` `...` `doCatch` block has one limitation: it's only route-scoped. The blocks work only in the route in which they're defined. `onException`, on the other hand, works in both context and route scopes, so let's try revising the last listing using `onException`. That's illustrated in the following listing.

Listing 11.8 Using `onException` in context scope

```
onException(JMSException.class)
    .handled(true)    ❶
```

❶

Handles all JMSExceptions

```
    .process(new GenerateFailueResponse());

from("netty4:tcp://0.0.0.0:4444?textline=true")
    .process(new ValidateOrderId())
    .to("jms:queue:order.status")
    .process(new GenerateResponse());
```

A difference between `doCatch` and `onException` is that `doCatch` will handle the exception, whereas `onException`, by default, won't handle it. That's why you use `handled(true)` ❶ to instruct Camel to handle this exception. As a result, when `JMSException` is thrown, the application acts as if the exception were caught in a `catch` block using the regular Java `try` `...` `catch` mechanism.

In listing 11.8, you should also notice that the concerns are separated and the normal route path is laid out nicely and simply; it isn't mixed up with the exception handling.

Imagine that a message arrives on the TCP endpoint, and the Camel application routes the message. The message passes the validate processor and is about to be sent to the JMS queue, but this operation fails and `JMSException` is thrown. Figure 11.5 is a sequence diagram showing the steps that take place inside Camel in such a situation. It shows how `onException` is triggered to handle the exception.

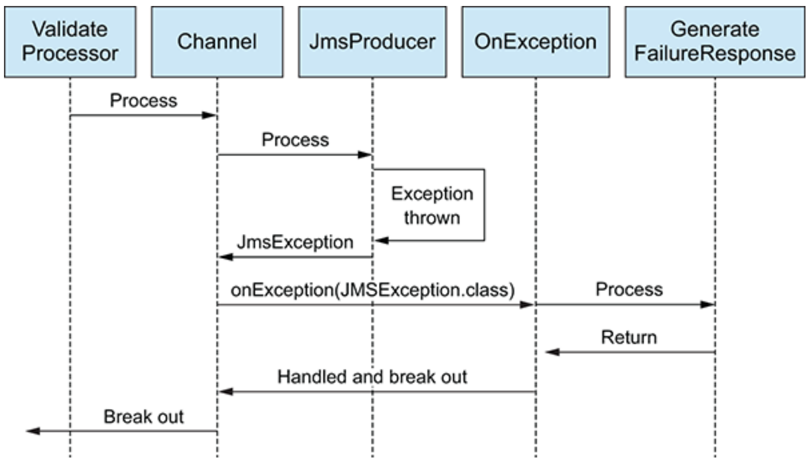

Figure 11.5 Sequence diagram of a message being routed and `JMSException` being thrown from `JmsProducer`, which is handled by `onException`. `onException` generates a failure that's returned to the caller.

Figure 11.5 shows how `JmsProducer` throws `JMSException` to `Channel`, which is where the error handler lives. The route has `onException` defined that reacts when `JMSException` is thrown, and it processes the message. The `GenerateFailureResponse` processor generates a custom failure message that's supposed to be returned to the caller. Because `onException` was configured to handle exceptions `handled(true)`, Camel will *break out* from continuing the routing and will return the failure message to the initial consumer, which in turn returns the custom reply message.

NOTE `onException` doesn't handle exceptions by default, so listing 11.8 uses `handled(true)` to indicate that `onException` should handle the exception. This is important to remember, because it must be specified when you want to handle the exception. Handling an exception won't continue routing from the point where the exception was thrown. Camel will break out of the route and continue routing on `onException`. If you want to ignore the exception and continue routing, you must use `continued(true)`, which is discussed in section 11.4.6.

Before we move on, let's take a minute to look at the example from listing 11.8 revised to use XML DSL. The syntax is a bit different, as you can see.

Listing 11.9 XML DSL revision of listing 11.8

```
<camelContext xmlns="http://camel.apache.org/schemas/spring">
    <onException>
        <exception>javax.jms.JMSException</exception>
    <handled><constant>true</constant></handled>    1
```

1

Handles all JMSExceptions

```
        <process ref="failureResponse"/>
    </onException>

    <route>
        <from uri="netty4:tcp://0.0.0.0:4444?textline=true"/>
        <process ref="validateOrder"/>
        <to uri="jms:queue:order.status"/>
        <process ref="generateResponse"/>
    </route>
</camelContext>

<bean id="failureResponse"    2
```

❷

Processor generates failure response

```
        class="camelinaction.FailureResponseProcessor"/>
  <bean id="validateOrder" class="camelinaction.ValidateProcessor"/>
  <bean id="generateResponse" class="camelinaction.ResponseProcessor"/>
```

Notice how `onException` is set up—you must define the exceptions in the `exception` tag. Also, `handled(true)` ❶ is a bit longer because you must enclose it in the `<constant>` expression. There are no other noteworthy differences in the rest of the route.

Listing 11.9 uses a custom processor to generate a failure response ❷. Let’s take a closer look at that.

11.4.4 Custom exception handling

Suppose you want to return a custom failure message, as in listing 11.9, that not only indicates what the problem was, but also includes details from the current exchange. How can you do that?

Listing 11.9 laid out how to do this by using `onException`. The following listing shows how the failure `Processor` could be implemented.

Listing 11.10 Using a processor to create a failure response to be returned to the caller

```
public class FailureResponseProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        Exception e = exchange.getProperty(Exchange.EXCEPTION_CAUGHT,
              Exception.class);    ❶
```

❶

Gets the exception

```
        StringBuilder sb = new StringBuilder();
        sb.append("ERROR: ");
        sb.append(e.getMessage());
```

```
            sb.append("\nBODY: ");
            sb.append(body);
            exchange.getIn().setBody(sb.toString());
        }
    }
```

First, you grab the information you need: the message body and the exception ❶. It may seem a bit odd that you get the exception as a property instead of `exchange.getException()`. You do that because you've marked `onException` to handle the exception; this was done at ❶ in listing 11.9. Consequently, Camel moves the exception from the `Exchange` to the `Exchange.EXCEPTION_CAUGHT` property. The rest of the processor builds the custom failure message that's to be returned to the caller.

You may wonder whether there are other properties Camel sets during error handling, and there are. They're listed in table 11.5. But from an end-user perspective, only the first two properties in table 11.5 matter. The other two properties are used internally by Camel in its error handling and routing engine.

One example of when the `FAILURE_ENDPOINT` property comes in handy is when you route messages through the Recipient List EIP, which sends a copy of the message to a dynamic number of endpoints. Without this information, you wouldn't know precisely which of those endpoints failed.

Table 11.5 Properties on the exchange related to error handling

Property	Type	Description
`Exchange.EXCEPTION_CAUGHT`	`Exception`	The exception that was caught.
`Exchange.FAILURE_ENDPOINT`	`String`	The URL of the endpoint that failed if a failure occurred when sending to an endpoint. If the failure didn't occur while sending to an endpoint, this property is `null`.
`Exchange.ERRORHANDLER_HANDLED`	`boolean`	Whether the error handler handled the exception.
`Exchange.FAILURE_HANDLED`	`boolean`	Whether `onException` handled the exception. Or `true` if the exchange was moved to a dead letter queue.

It's worth noting that in listing 11.10 you use a Camel `Processor`, which forces you to depend on the Camel API. You can use a bean instead, as shown in the following listing.

Listing 11.11 Using a bean to create a failure response to be returned to the caller

```
public class FailureResponseBean {
    public String failMessage(String body, Exception e) {    ❶
```

❶

Exception provided as parameter

```
        StringBuilder sb = new StringBuilder();
        sb.append("ERROR: ");
        sb.append(e.getMessage());
        sb.append("\nBODY: ");
        sb.append(body);
        return sb.toString();
    }
}
```

As you can see, you can use Camel's parameter binding ❶ to declare the parameter types you want to use. The first parameter is the message body, and the second is the exception.

Suppose you use a custom processor or bean to create a failure response. What would happen if a new exception were thrown from the processor or bean? That's the topic for the next section.

11.4.5 New exception while handling exception

This section looks at how Camel reacts when a new exception occurs while handling a previous exception, which has been constructed as an example in the following listing.

Listing 11.12 Using `onException` in context scope

```
onException(AuthorizationException.class)    ❶
```

❶

onException to handle AuthorizationException

```
    .handled(true)
  .process(new NotAllowedProcessor());  2
```

2

By calling the NotAllowedProcessor

```
onException(Exception.class)  3
```

3

onException to handle all other kind of Exceptions

```
    .handled(true)
  .process(new GeneralErrorProcessor());  4
```

4

By calling the GeneralErrorProcessor

```
public class NotAllowedProcessor implements Processor {

  public void process(Exchange exchange) throws Exception {
    ...
  throw new NullPointerException("null");  5
```

5

Some code causes a NullPointerException

```
  }
}

public void GeneralErrorProcessor implement Processor {

  public void process(Exchange exchange) throws Exception {
    ...
    throw new AuthorizationException("forbidden");   ❻
```

❻ Some code causes a AuthorizationException

```
    }
  }
```

The first `onException` ❶ is configured to handle all `AuthorizationException` exceptions by calling `NotAllowedProcessor` ❷. The second `onException` ❸ is set up to handle all other kinds of exceptions by calling `GeneralErrorProcessor` ❹.

Because the `onException`s are calling processors, what could happen if these processors cause new exceptions to be thrown? The code in listing 11.12 has been purposely set up to trigger such situations. Suppose at first `AuthorizationException` is thrown during routing, which leads to `NotAllowedProcessor` to be called, and unfortunately this processor throws `NullPointerException` ❺, which causes `onException` ❸ to react and call `GeneralErrorProcessor`, which also unfortunately throws an `AuthorizationException` exception ❻. This scenario would lead to an endless circular error handling.

To avoid these unfortunate scenarios, Camel doesn't allow further error handling while already handling an error. In other words, when `NullPointerException` ❺ is thrown the first time, Camel detects that another exception was thrown during error handling, and prevents any further action from taking place. This is done by the `org.apache.camel.processor.FataFallbackErrorHandler`, which catches the new exception, logs a warning, sets this as the exception on the `Exchange`, and stops any further routing.

The following code shows a snippet of the warning logged by Camel:

```
2017-03-14 12:46:54,885 [main           ] ERROR FatalFallbackErrorHandle
Exception occurred while trying to handle previously thrown exception on
exchangeId: ID-davsclaus-pro-57045-1426333614411-0-2 using
```

```
[Channel[DelegateSync[camelinaction.NotAllowedProcessor@3b938003]]].
The previous and the new exception will be logged in the following.
2017-03-14 12:46:54,886 [main           ] ERROR FatalFallbackErrorHandle
\--> Previous exception on exchangeId:
  ID-davsclaus-pro-57045-1426333614411-0-2
  camelinaction.AuthorizationException: Forbidden
  at camelinaction.NewExceptionTest$1.configure(NewExceptionTest.java:41
2017-03-14 12:46:54,894 [main           ] ERROR FatalFallbackErrorHandle
\--> New exception on exchangeId:
  ID-davsclaus-pro-57045-1426333614411-0-2
  java.lang.NullPointerException
  at camelinaction.NotAllowedProcessor.process(NotAllowedProcessor.java:
```

You can see this in action by running the following Maven goal from the chapter 11/newexception directory:

```
mvn test -Dtest=NewExceptionTest
mvn test -Dtest=SpringNewExceptionTest
```

Instead of handling exceptions, in some situations you'll want to ignore the exception and continue routing, although those situations are rare.

11.4.6 Ignoring exceptions

In section 11.4.3, you learned about how `onException` can handle exceptions. Handling an exception means that Camel will break out of the route. But sometimes all you want is to catch the exception and continue routing. You can do that in Camel using `continued`. All you have to do is to use `continued(true)` instead of `handled(true)`.

Suppose you want to ignore any `ValidationException` that may be thrown in the route, laid out in listing 11.8. The following listing shows how to do this.

Listing 11.13 Using continued to ignore `ValidationExceptions`

```
onException(JMSException.class)
    .handled(true)
    .process(new GenerateFailueResponse());
```

```
onException(ValidationException.class)
    .continued(true);     ❶
```

❶

Ignores all ValidationExceptions

```
from("netty4:tcp://0.0.0.0:4444?textline=true")
    .process(new ValidateOrderId())
    .to("jms:queue:order.status")
    .process(new GenerateResponse());
```

As you can see, all you have to do is add another `onException` that uses `continued(true)` ❶.

NOTE You can't use both `handled` and `continued` on the same `onException`.

Now imagine that a message once again arrives at the TCP endpoint, and the Camel application routes the message, but this time the validate processor throws `ValidationException`. This situation is illustrated in figure 11.6.

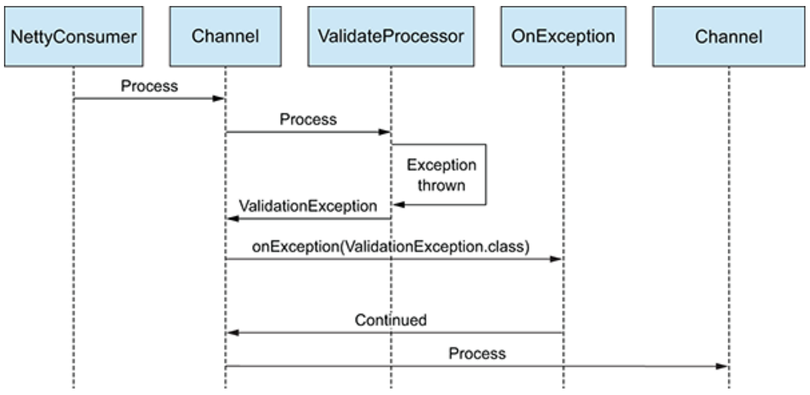

Figure 11.6 Sequence diagram of a message being routed and `ValidationException` being thrown from `ValidateProcessor`. The exception is handled and continued by the `onException` policy, causing the message to continue being routed as if the exception weren't thrown.

When `ValidateProcessor` throws `ValidationException`, it's propagated back to the channel, which lets the error handler kick in. The route has an `onException` defined that instructs the channel to continue routing the message—`continued(true)`. When the message arrives at the next channel, it's as if the exception weren't thrown. This is much different from what you saw in section 11.4.3 when using `handled(true)`, which causes the processing to break out and *not* continue routing.

You've learned a bunch of new stuff, so let's continue with the error-handler example and put your knowledge into practice.

11.4.7 Implementing an error-handler solution

Suppose your boss brings you a new problem. This time, the remote HTTP server used for uploading files is unreliable, and he wants you to implement a secondary failover to transfer the files by FTP to a remote FTP server.

You've been studying *Camel in Action*, and you've learned that Camel has extensive support for error handling and that you could use `onException` to provide this kind of feature. With great confidence, you fire up the editor and alter the route as shown in the following listing.

Listing 11.14 Route using error handling with failover to FTP

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(5).redeliveryDelay(10000));

onException(IOException.class).maximumRedeliveries(3)    ❶
```

❶

Exception policy

```
    .handled(true)
    .to("ftp://gear@ftp.rider.com?password=secret");

from("file:/rider/files/upload?delay=15m")
    .to("http://rider.com/upload?user=gear&password=secret");
```

This listing adds `onException` ❶ to the route, telling Camel that in the case of `IOException`, it should try redelivering up to three times, using a 10-second delay. If there's still an error after the redelivery attempts, Camel will handle the exception and reroute the message to the FTP endpoint instead. The power and flexibility of the Camel routing engine shines here. `onException` is just another route, and Camel will continue on this route instead of the original route.

NOTE In listing 11.14, it's only when `onException` is exhausted that it will reroute the message to the FTP endpoint ❶. The `onException` has been configured to redeliver up to three times before giving up and being exhausted.

The book's source code contains this example in the chapter11/usecase directory, and you can try it out yourself. The example contains a server and a client that you can start by using Maven:

```
mvn compile exec:java -PServer
mvn compile exec:java -PClient
```

Both the server and client output instructions on the console about what to do next, such as copying a file to the target/rider folder to get the ball rolling.

Here, toward the end of this chapter, we've saved a great example for you to try. In this example, you'll use an external database, and then try to see what happens when you cut the connection between Camel and the database. To make the example easier to run, the book's accompanying source code uses a Docker image to use Postgres as the database.

11.4.8 Bridging the consumer with Camel's error handler

At the start of this chapter, we discussed how Camel's error handler works. You learned that Camel's error handler takes effect during routing exchanges. If an error happens within the consumer prior to an exchange being created, the error is handled individually by each Camel component. Most of the components will log the error and not create any exchange to be routed.

To better understand the content of this section, take a look at figure 11.7 (which is a copy of figure 11.2).

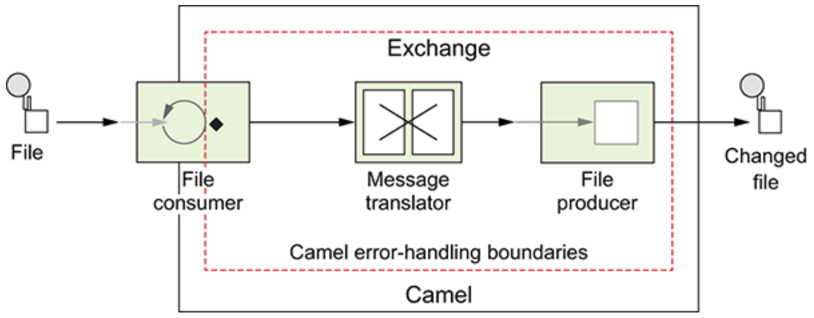

Figure 11.7 Camel's error handling applies only within the lifecycle of an exchange. The error-handler boundaries are represented by the dashed square in the figure.

Earlier in the chapter we asked, "What happens if the file consumer can't read the file?" You learned that this was component specific and that the file consumer would log a `WARN`, skip the file, and attempt to read the file again on the next poll. But what if you want to handle that error? What can you do? You can configure the consumer to bridge with Camel's error handler. When an error is thrown internally in the consumer, and the bridge is enabled, the consumer creates an empty exchange with the caused error as the exception. The exchange is then routed by Camel, and as the routing engine detects the exception immediately, it allows the regular error handler to deal with the exception. This is illustrated in figure 11.8.

Figure 11.8 Camel's error handling now includes early errors happening from within the consumer, when the bridge error-handler option has been enabled.

Notice that Camel's error-handling boundaries have expanded to include the consumer. If the consumer captures an exception earlier while attempting to detect or read incoming files, the exception is captured and sent along to Camel's error handler to deal with the exception. This functionality is known as *bridging the consumer* with Camel's error handler.

To learn how all this works, let's use an example to cover this from head to tail: a book-order system. A database is used to store incoming orders in a table. Then a Camel application polls the database table for any rows inserted into the table, and for each row a log message is printed. If an error occurs, Camel's error handler is configured to react and route a message to a dead letter channel. Figure 11.9 presents this example in a visual style.

Figure 11.9 The JPA consumer is bridged to Camel's error handler so `onException` now also triggers when an error happens internally in the JPA consumer, such as no connection to the PostgresSQL database.

The book's source code contains this example in the chapter11/bridge directory. The example uses Docker to run the Postgres database, and instructions are provided in the readme.md file from the example source code.

If you have Docker set up correctly on your computer, the database can be started in one line:

```
docker run --name my-postgres -p 5432:5432 -e POSTGRES_PASSWORD=secret -
```

When the Postgres database is up and running, you can run the Camel application using the following:

```
mvn clean install exec:java
```

The example is constructed so that at one point it waits for you to press Enter before continuing. The idea is that it gives you time to stop the Postgres database so you can see what happens in the Camel application when the consumer can't connect to the database, and therefore an error happens internally in the consumer. When you do this, the console should print something similar to this:

```
2017-05-11 20:08:39,425 [ction.BookOrder] WARN  books - We do not care C
```

The preceding error message (highlighted in bold) tells you that an exception happened, but you don't care. The cause of the error follows with `Connection to 0.0.0.0:5432 refused.` That last part of the error message is a JPA error message when a connection to the database can't be established. (The Postgres database runs on host `0.0.0.0`.)

If you take a moment to dive into the source code of this example, you'll learn that this error message was created by Camel's error handler, as shown in the following listing.

Listing 11.15 Example that bridges the consumer with Camel's error handler

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

  <endpoint id="newBookOrders" uri="jpa:camelinaction.BookOrder">    1
```

1 Configures the JPA endpoint outside the route, allowing you to configure each option using <property> elements

```
    <property key="delay" value="1000"/>
    <property key="bridgeErrorHandler" value="true"/>    2
```

2 Bridges the consumer with Camel’s error handler

```
    <property key="backoffMultiplier" value="10"/>    3
```

3 Turns on back-off to multiply the polling delay with x10 after one error

```
    <property key="backoffErrorThreshold" value="1"/>
  </endpoint>

  <onException>
    <exception>java.lang.Exception</exception>
    <redeliveryPolicy logExhaustedMessageHistory="false"    4
```

4 Tones down logging noise when Camel’s error handler is handling the exception

```
                        logExhausted="false"/>
      <handled>
        <constant>true</constant>
      </handled>
      <log message="We do not care ${exception.message}"
         loggingLevel="WARN"/>    5
```

5

Prints a logging message indicating you don't care about the exception

```
    </onException>

    <route id="books">
      <from uri="ref:newBookOrders"/>    6
```

6

The route that polls the Postgres database

```
      <log message="Order ${body.orderId} - ${body.title}"/>
    </route>
  </camelContext>
```

To configure the JPA consumer, you set up the endpoint ❶ and provide it with the ID `newBookOrders`. Each option is configured using a `<property>` tag. Notice that you can also configure the endpoint using the more common URI style. To bridge the consumer with Camel's error handler, all you have to do is set `bridgeErrorHandler` to `true` ❷. In case of an error, you don't want the consumer to continue polling as frequently as normal, so you apply the back-off multiplier option with a value of 10 ❸. Instead of polling every 1,000 ms, the consumer will poll 10 x 1,000 ms = 10,000 ms (1 sec → 10 sec). The back-off threshold tells Camel after how many subsequent errors the back-off should apply. In this example, that occurs after the first error. But if you set the threshold to 5, only after five subsequent errors will the polling be delayed by the back-off multiplier. When an exception happens, you want Camel's error handler to react and print only a brief custom message to the log ❺. To avoid having any additional information logged, you turn off the logging of the exhausted error on the error handler ❹; this ensures that only the custom message is being logged. And at the bottom of the listing is the route that uses the JPA consumer to pick up new book orders ❻.

TIP We encourage you to try this example on your own. For example, see what happens when you turn off `bridgeErrorHandler`. You could also try changing the back-off thresholds and see what happens if you remove the `onException` code.

The example uses a few options related to back-off; let's take a moment to look at these options formally in a table.

Using back-off to let the consumer be less aggressive

Table 11.6 lists all the back-off options that Camel provides out of the box.

Table 11.6 Back-off options to let a consumer be less aggressive during polling

Option	Description
`backoffMultiplier`	Lets the polling consumer back off if there's been a number of subsequent idles/errors in a row. The multiplier is then the number of polls that will be skipped before the next actual attempt happens again. When this option is in use, `backoffIdleThreshold` and/or `backoffErrorThreshold` must also be configured.
`backoffIdleThreshold`	The number of subsequent idle polls that should happen before `backoffMultiplier` kicks in.
`backoffErrorThreshold`	The number of subsequent error polls (failed due to an error) that should happen before `backoffMultiplier` kicks in.

Before finishing up this chapter, we need to take a look at a few more error-handling features. They're used rarely, but they provide power in situations where you need more fine-grained control.

11.5 Working with other error-handling features

Here are some of the other features Camel provides for error handling:

- `onWhen` —Allows you to dictate when an exception policy is in use
- `onExecutionOccurred` —Allows you to execute code *right* after an exception occurs
- `onRedeliver` —Allows you to execute code before the message is redelivered
- `retryWhile` —Allows you, at runtime, to determine whether to continue redelivery or to give up

We'll look at each in turn.

11.5.1 Using onWhen

The `onWhen` predicate filter allows more fine-grained control over when `onException` should be triggered.

Suppose a new problem has emerged with your application in listing 11.14. This time, the HTTP service rejects the data and returns an HTTP 500 response with the constant text `ILLEGAL DATA`. Your boss wants you to handle this by moving the file to a special folder where it can be manually inspected to see why it was rejected.

First, you need to determine when an HTTP error 500 occurs and whether it contains the text `ILLEGAL DATA`. You decide to create a Java method that can test this, as shown in the following listing.

Listing 11.16 A helper to determine whether an HTTP error 500 occurred

```
public final class MyHttpUtil {
    public static boolean isIllegalDataError(
                                        HttpOperationFailedException cause)
        int code = cause.getStatusCode();    ❶
```

❶

Gets HTTP status code

```
        if (code != 500) {
            return false;
        }
        return "ILLEGAL DATA".equals(cause.getResponseBody().toString())
    }
}
```

When the HTTP operation isn't successful, the Camel HTTP component will throw an `org.apache.camel.component.http.HttpOperationFailedException` exception, containing information about why it failed. The `getStatusCode` method on `HttpOperationFailedException` ❶ returns the HTTP status code. This allows you to determine whether it's an HTTP error code 500 with the `ILLEGAL DATA` body text.

Now you need to set up Camel's error handler to use the bean from listing 11.16 to determine whether it's this special error your boss told you to handle. You can do this using the `onWhen` predicate with `onException`:

```
onException(HttpOperationFailedException.class)
    .onWhen(bean(MyHttpUtil.class, "isIllegalData"))
    .handled(true)
    .to("file:/rider/files/illegal");
```

When the HTTP server fails, an `HttpOperationFailedException` exception is thrown, and `onException` is triggered. The `onWhen` predicate means that the `isIllegalData` method on the `MyHttpUtil` bean (listing 11.16) returns either `true` or `false`. If `true` was returned, the exception is handled, and Camel will store the message in the filesystem in the rider/files/illegal folder.

Here's how you use `onWhen` in XML DSL:

```
<onException>
  <exception>org.apache.camel.component.http.
             HttpOperationFailedException</exception>
  <onWhen><method ref="myHttpUtil" method="isIllegalData"/></onWhen>
  <handled><constant>true</constant></handled>
  <to uri="file:/rider/files/illegal"/>
</onException>
```

```
<bean id="myHttpUtil" class="com.rider.MyHttpUtil"/>
```

`onWhen` is a general function that also exists in other Camel features, such as interceptors and `onCompletion`, so you can use this technique in various situations.

`onExceptionOccurred`, discussed next, allows you to call a Camel processor right after an exception is thrown.

11.5.2 Using onExceptionOccurred

`onExceptionOccurred` is designed to allow code to be executed right after an exception is thrown, and before any redelivery may happen. This allows you to do some custom processing immediately after the exception is thrown, such as doing custom logging, or any need you may have.

`onExceptionOccurred` can be configured on the error handler:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .onExceptionOccurred(new MyLogExceptionProcessor());
```

Or on `onException`:

```
onException(IOException.class)
    .maximumRedeliveries(5)
    .onExceptionOccurred(new MyLogExceptionProcessor());
```

In XML DSL, `onExceptionOccurred` is configured as a reference to a bean, as follows:

```
<onException onExceptionOccurredRef="myLogException">
    <exception>java.io.IOException</exception>
</onException>

<bean id="myLogException"
       class="com.mycompany.MyLogExceptionProcessor"/>
```

Next, let's look at `onRedeliver`, which is similar to `onExceptionOccurred` but instead of triggering right after the exception, it triggers right before any redelivery is about to take place.

11.5.3 Using onRedeliver

The purpose of `onRedeliver` is to allow code to be executed before a redelivery is performed. This gives you the power to do custom processing on the exchange before Camel makes a redelivery attempt. You can, for instance, use it to add custom headers to indicate to the receiver that this is a redelivery attempt. `onRedeliver` uses `org.apache.camel.Processor`, in which you implement the code to be executed.

`onRedeliver` can be configured on the error handler:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .onRedeliver(new MyOnRedeliveryProcessor());
```

Or on `onException`:

```
onException(IOException.class)
    .maximumRedeliveries(5)
    .onRedeliver(new MyOtherOnRedeliveryProcessor());
```

In XML DSL, `onRedeliver` is configured as a reference to a bean, as follows:

```
<onException onRedeliveryRef="myOtherRedelivery">
   <exception>java.io.IOException</exception>
</onException>

<bean id="myOtherRedelivery"
      class="com.mycompany.MyOtherOnRedeliveryProcessor"/>
```

Finally, let’s look at one last feature: `retryWhile`.

11.5.4 Using retryWhile

`retryWhile` is used when you want fine-grained control over the number of redelivery attempts. It’s also a predicate that’s scoped, so you can define it on the error handler or on `onException`.

You can use `retryWhile` to implement your own generic retry rule set that determines how long it should retry. The following listing shows some skeleton code demonstrating how this can be done.

Listing 11.17 Skeleton code to illustrate principle of using `retryWhile`

```
public class MyRetryRuleset {
  public boolean shouldRetry(    ❶
```

❶ Method used as predicate to determine whether to retry

```
                          @Header(Exchange.REDELIVERY_COUNTER) Integer counter,
                          Exception causedBy) {
      ...
      // return true or false
    }
}
```

Using your own `MyRetryRuleset` class, you can implement your own logic to determine whether it should continue retrying. In listing 11.17, the method is named `shouldRetry` ❶ and returns a `boolean`. Camel has no restriction on the name of the method. For example, the method could've been named `anotherBrickInTheWall` or any other fantastic name that may spring to mind. If the method returns `true`, a redelivery attempt is conducted; if it returns `false`, it gives up.

To use your rule set, you configure `retryWhile` on `onException` as follows:

```
onException(IOException.class).retryWhile(bean(MyRetryRuletset.class));
```

In XML DSL, you configure `retryWhile` as shown:

```
<onException>
  <exception>java.io.IOException</exception>
  <retryWhile><method ref="myRetryRuleset"/></retryWhile>
</onException>
```

```
<bean id="myRetryRuleset" class="com.mycompany.MyRetryRuleset"/>
```

That gives you fine-grained control over the number of redelivery attempts performed by Camel.

That's it! We've now covered all the features Camel provides for fine-grained control over error handling.

11.6 Summary and best practices

In this chapter, you saw how recoverable and irrecoverable errors are represented in Camel. We also looked at all the provided error handlers, focusing on the most important of them. You saw that Camel can control how exceptions are dealt with, using redelivery policies to set the scene and exception policies to handle specific exceptions differently. Finally, we looked at what Camel has to offer when it comes to fine-grained control over error handling, putting you in control of error handling in Camel.

Let's revisit some of the key ideas from this chapter, which you can apply to your own Camel applications:

- *Error handling is hard*—Realize from the beginning that the unexpected can happen and that dealing with errors is hard. The challenge keeps rising when businesses have more and more of their IT portfolio integrated and operate it 24/7/365.
- *Error handling isn't an afterthought*—When IT systems are being integrated, they exchange data according to agreed-upon protocols. Those protocols should also specify how errors will be dealt with.
- *Separate routing logic from error handling*—Camel allows you to separate routing logic from error-handling logic. This avoids cluttering up your logic, which otherwise could become harder to maintain. Use Camel features such as error handlers, `onException`, and `doTry ... doCatch`.
- *Try to recover*—Some errors are recoverable, such as connection errors. You should apply strategies to recover from these errors.
- *Capture error details*—When errors happen, try to capture details by ensuring that sufficient information is logged.

- *Use monitoring tooling*—Use tooling to monitor your Camel applications so the tooling can react and alert personnel if severe errors occur. Chapter 16 covers such strategies.
- *Build unit tests*—Build unit tests that simulate errors to see whether your error-handling strategies are up to the task. Chapter 9 covered testing.
- *Bridge the consumer with Camel's error handler*—This allows you to deal with these errors in the same way as errors happened during routing.

This chapter covered the Dead Letter Channel EIP (among others), which deals with errors by moving failure messages to a dead letter queue for later inspection. The next chapter covers how Camel works with transactions and how to set up and use these transactions in various containers such as Java EE, Spring, and OSGi. The chapter also talks about what can be done in the absence of transactions by using compensation and idempotency.