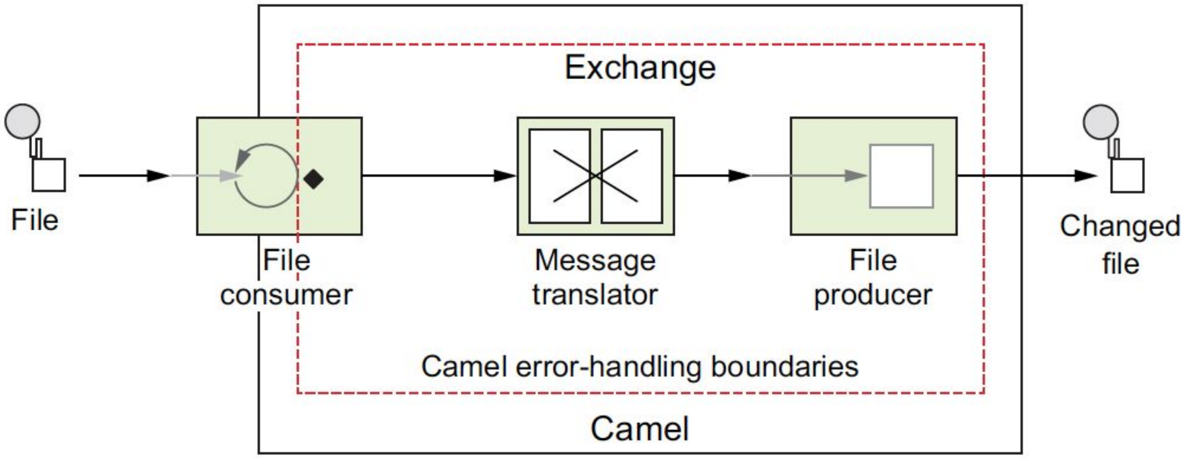
Camel Error Handling Workshop

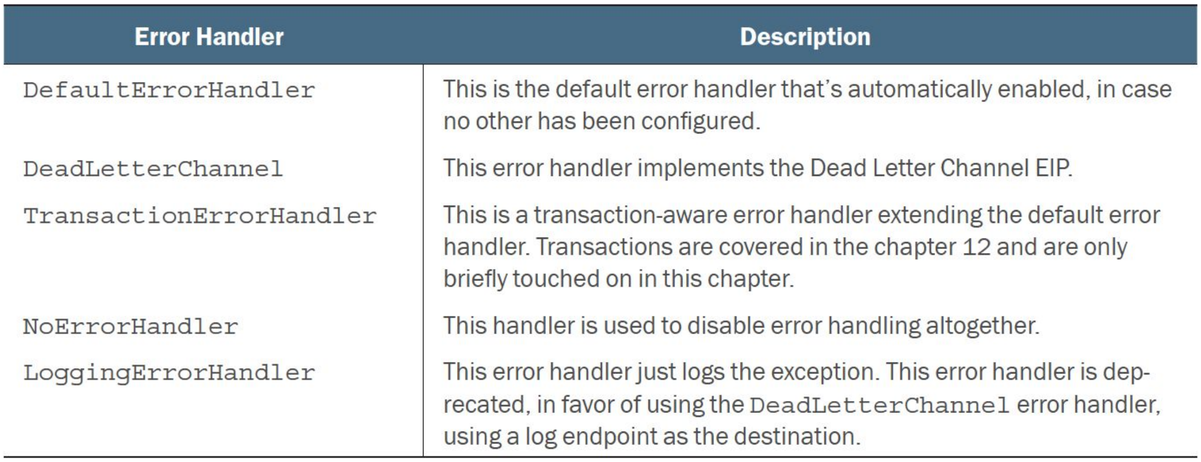
Where Camel’s error handling applies



Error handlers provided in Camel

Error handlers in Camel will react only to exceptions set on the exchange:

exchange.getException() != null

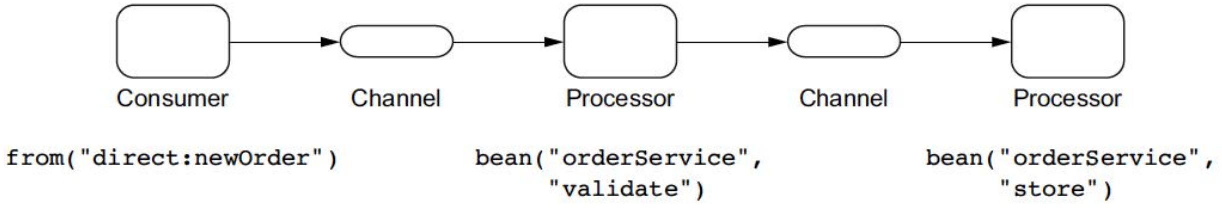


Error Handler	Description
DefaultErrorHandler	This is the default error handler that’s automatically enabled, in case no other has been configured.
DeadLetterChannel	This error handler implements the Dead Letter Channel EIP.
TransactionErrorHandler	This is a transaction-aware error handler extending 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.

The first three error handlers all extend the RedeliveryErrorHandler class. That class contains the majority of the error-handling logic that the first three error handlers all use.

Using the default error handler

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

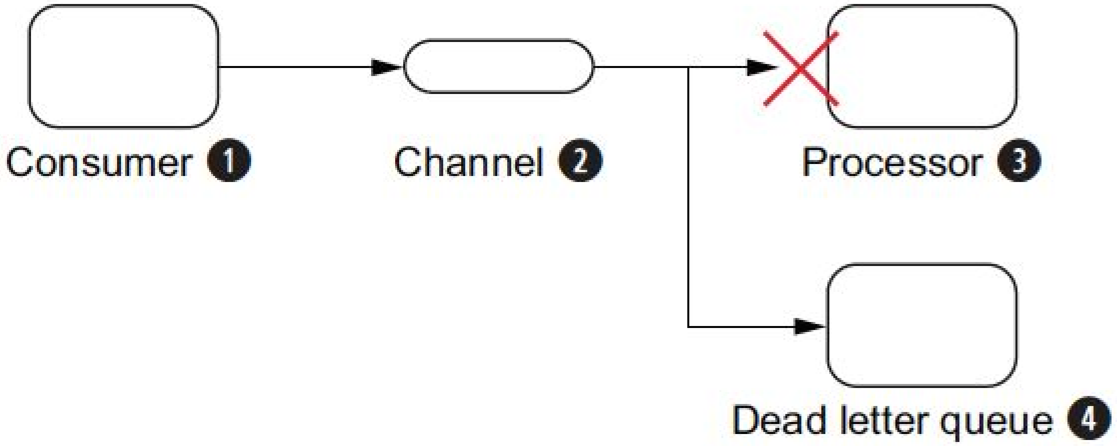


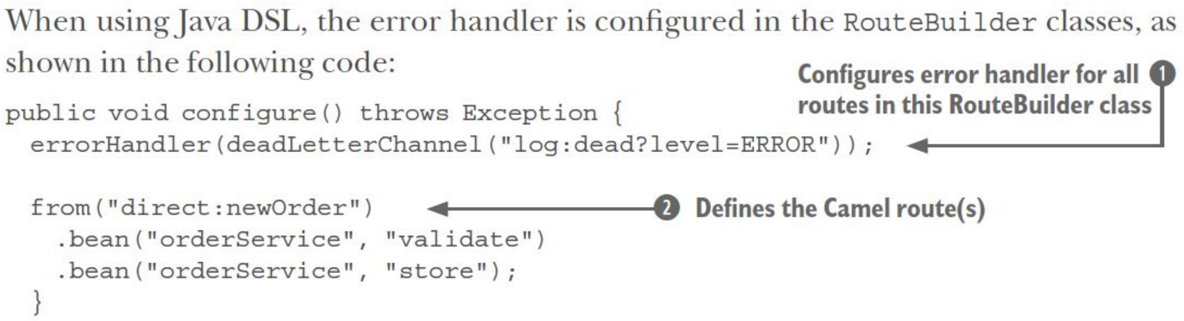
The default error handler is configured with these settings:

- 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.

- The transaction error handler
- The no error handler
- The logging error handler

The dead letter channel handler



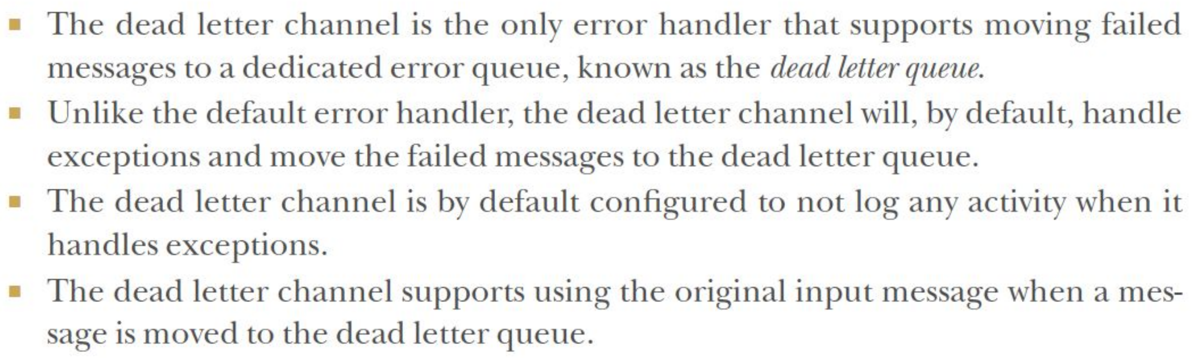


When using Java DSL, the error handler is configured in the RouteBuilder classes, as shown in the following code:

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

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

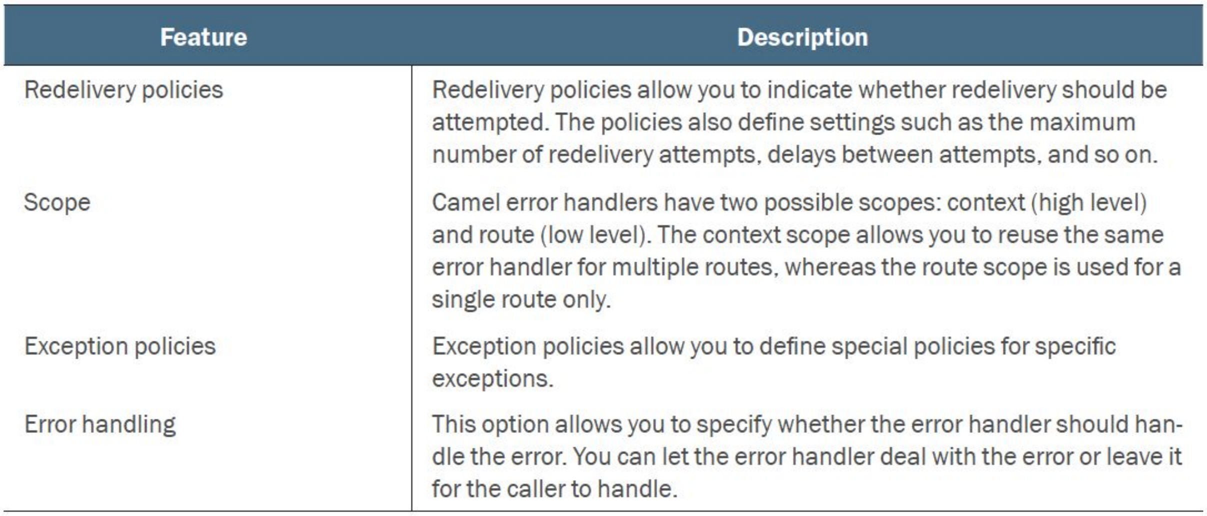
1 Configures error handler for all routes in this RouteBuilder class

2 Defines the Camel route(s)



- 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.

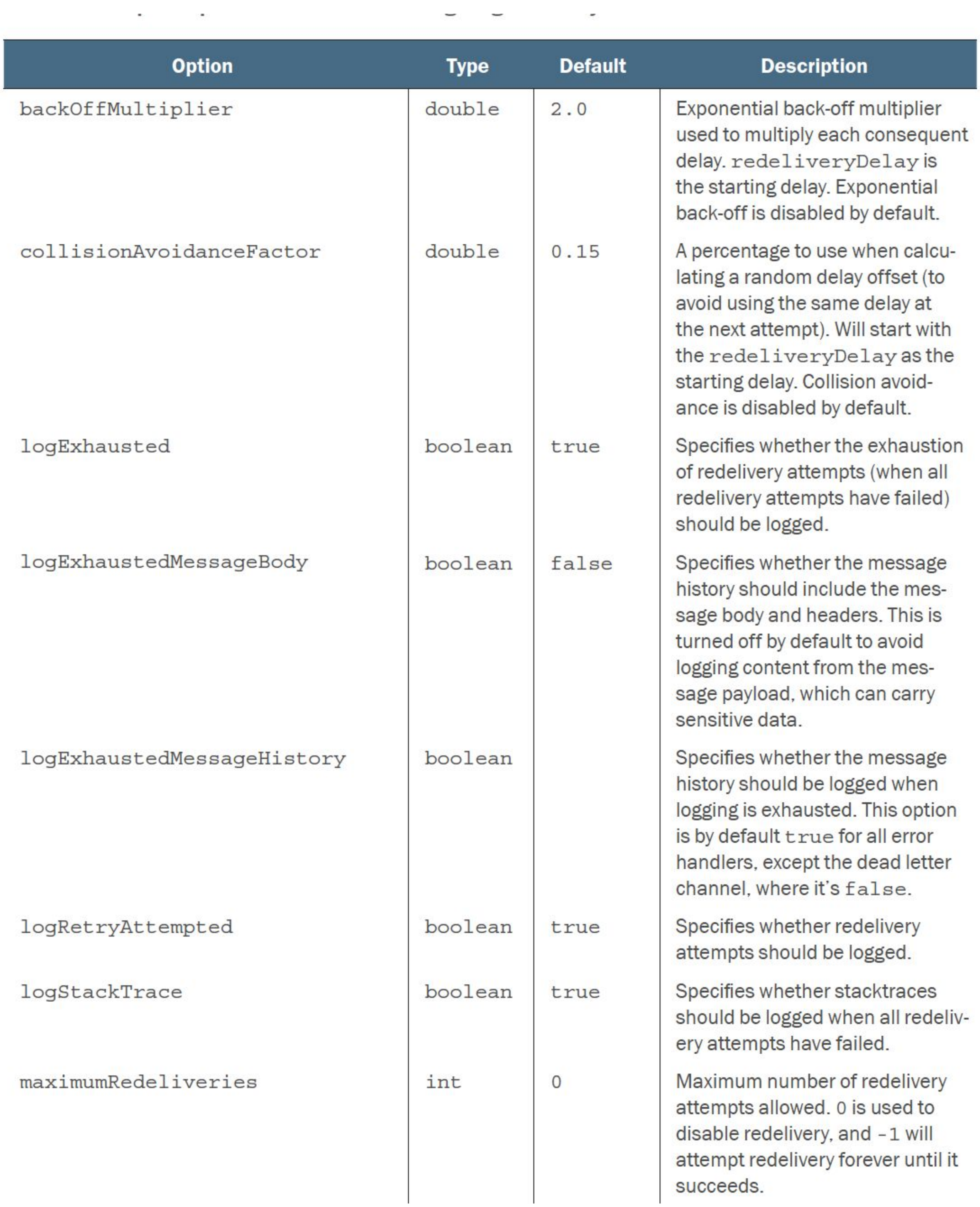
Features of the error handlers



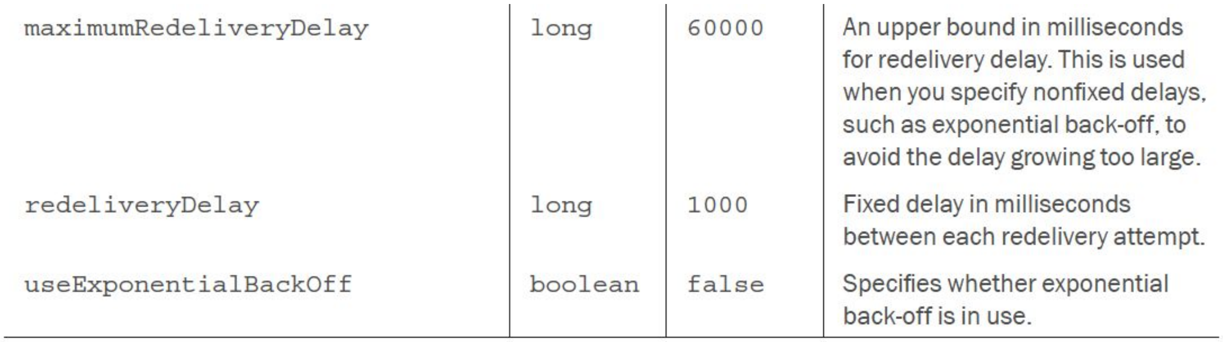
Feature	Description
Redelivery policies	Redelivery policies allow you to indicate whether redelivery should be attempted. The policies also define settings such as the maximum number of redelivery attempts, delays between attempts, and so on.
Scope	Camel error handlers have two possible scopes: context (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.

```
Exception e = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Exception.class);
```

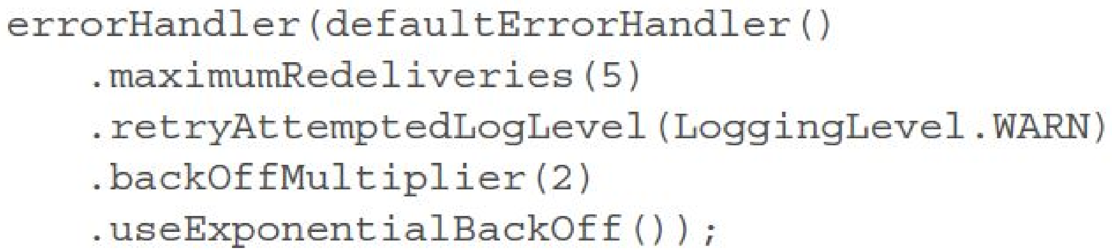
Using redelivery



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 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.
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.

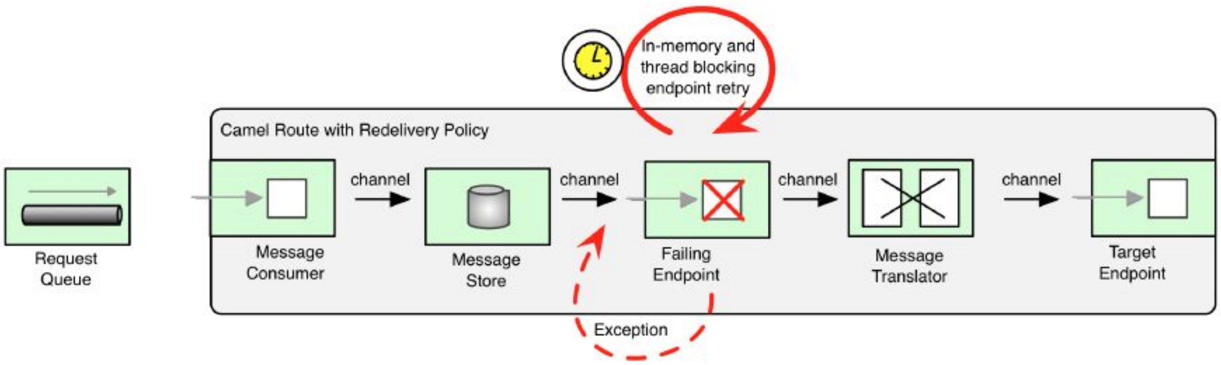


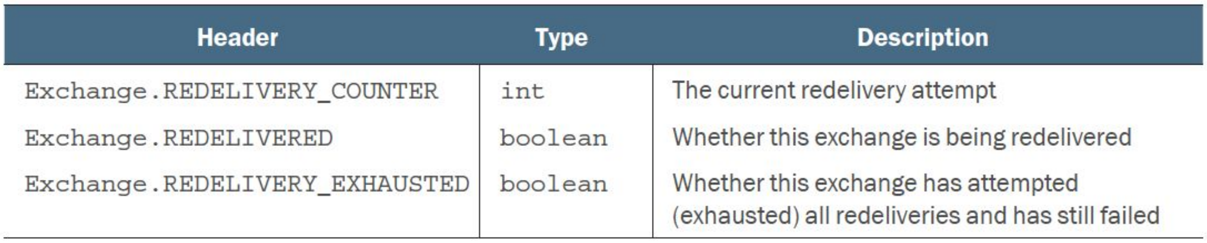
maximumRedeliveryDelay	long	60000	An upper bound in milliseconds for redelivery delay. This is used when you specify nonfixed delays, such as exponential back-off, to avoid the delay growing too large.
redeliveryDelay	long	1000	Fixed delay in milliseconds between each redelivery attempt.
useExponentialBackOff	boolean	false	Specifies whether exponential back-off is in use.



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

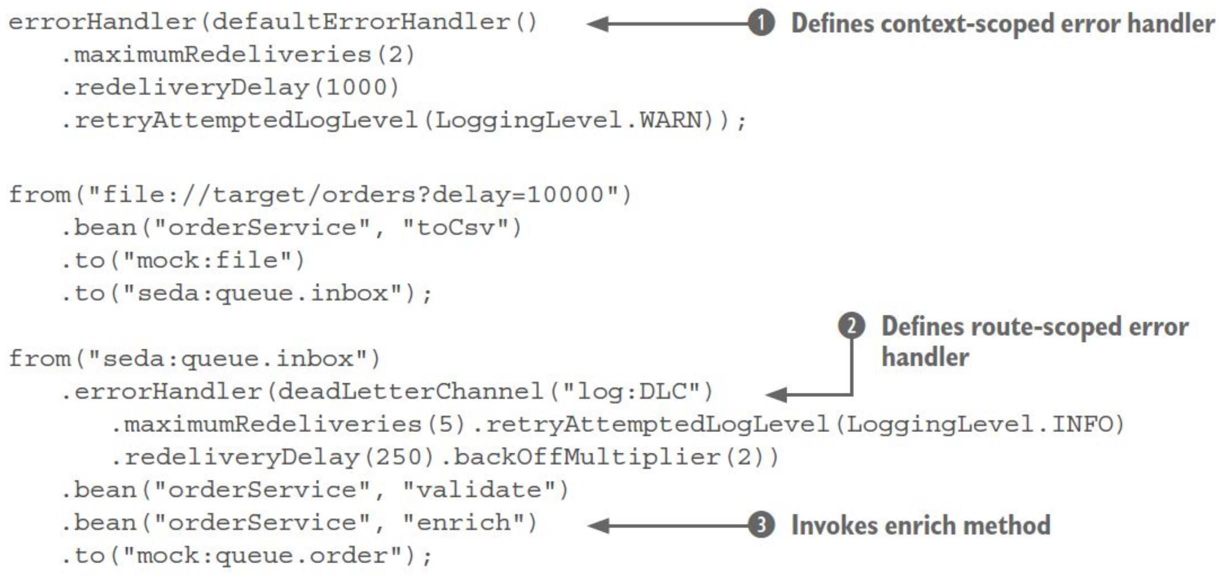
Camel redelivery happens at the point of error





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

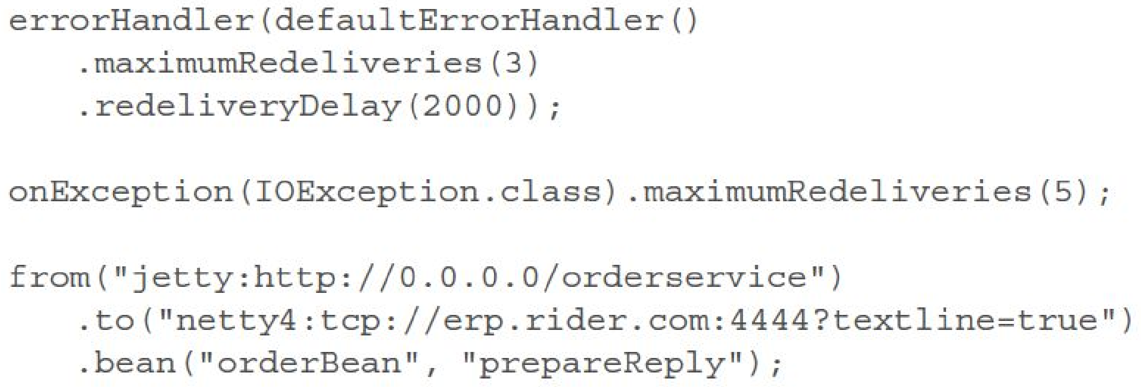
Error handler scopes



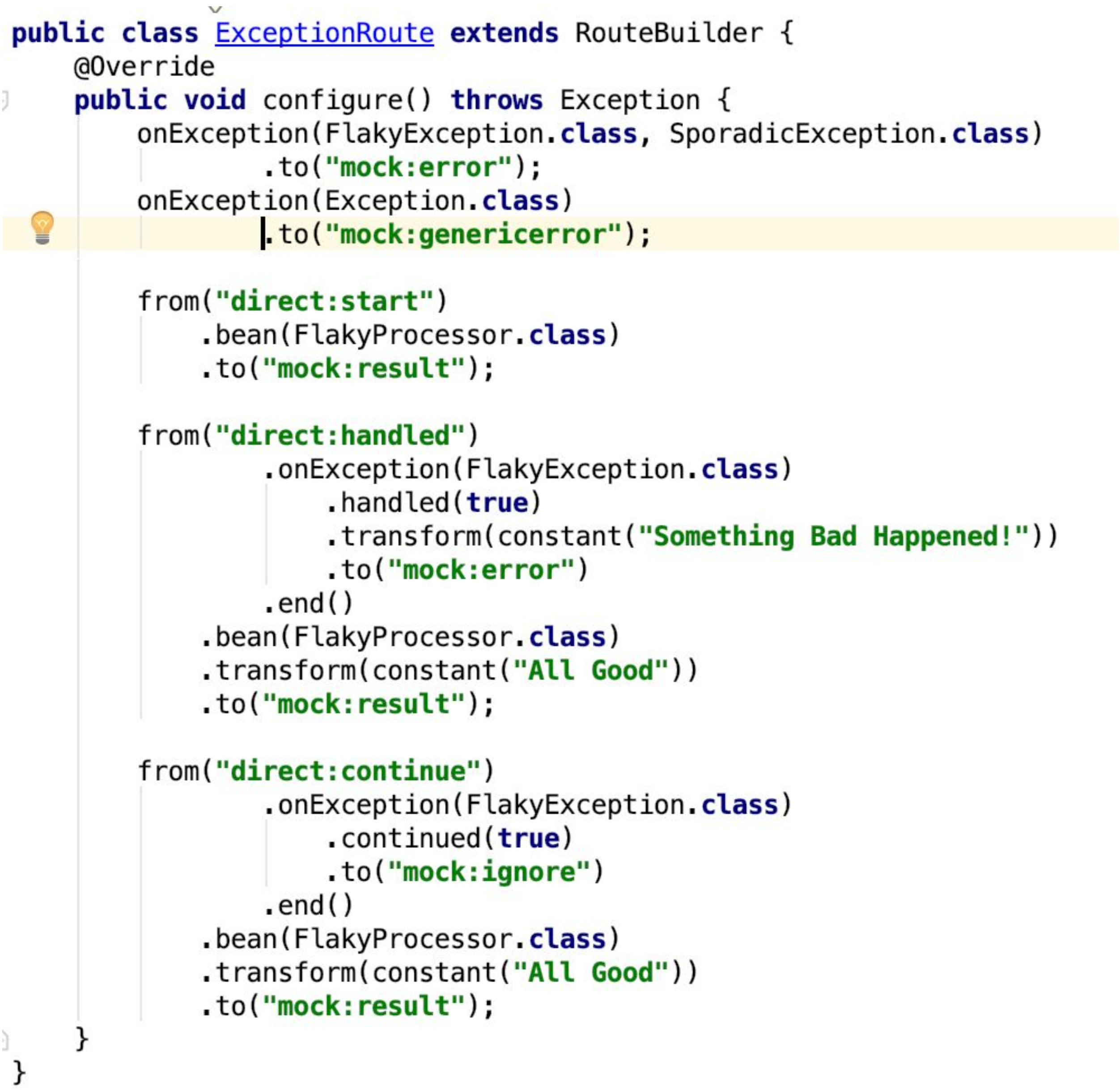
```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(2)
    .redeliveryDelay(1000)
    .retryAttemptedLogLevel(LoggingLevel.WARN));

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

from("seda:queue.inbox")
    .errorHandler(deadLetterChannel("log:DLC")
        .maximumRedeliveries(5).retryAttemptedLogLevel(LoggingLevel.INFO)
        .redeliveryDelay(250).backOffMultiplier(2))
    .bean("orderService", "validate")
    .bean("orderService", "enrich")
    .to("mock:queue.order");
```

1 Defines context-scoped error handler
2 Defines route-scoped error handler
3 Invokes enrich method

Convention over configuration/Using exception policies



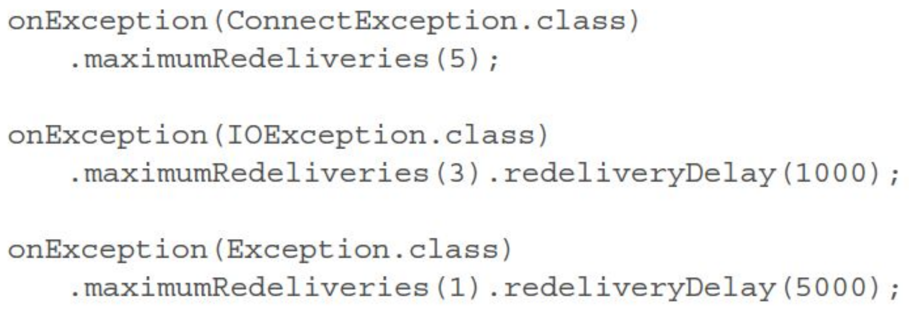
```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .redeliveryDelay(2000));

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");
```



```java
public class ExceptionRoute extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        onException(FlakyException.class, SporadicException.class)
                .to("mock:error");
        onException(Exception.class)
                .to("mock:genericerror");

        from("direct:start")
            .bean(FlakyProcessor.class)
            .to("mock:result");

        from("direct:handled")
                .onException(FlakyException.class)
                    .handled(true)
                    .transform(constant("Something Bad Happened!"))
                    .to("mock:error")
                .end()
            .bean(FlakyProcessor.class)
            .transform(constant("All Good"))
            .to("mock:result");

        from("direct:continue")
                .onException(FlakyException.class)
                    .continued(true)
                    .to("mock:ignore")
                .end()
            .bean(FlakyProcessor.class)
            .transform(constant("All Good"))
            .to("mock:result");
    }
}
```

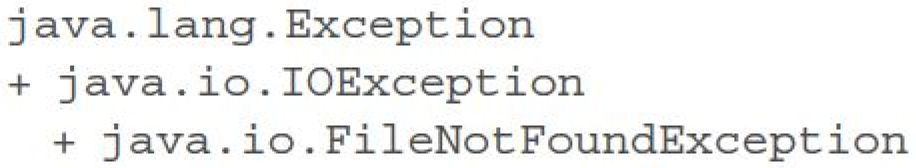
Exception gap detection

Given these policies:



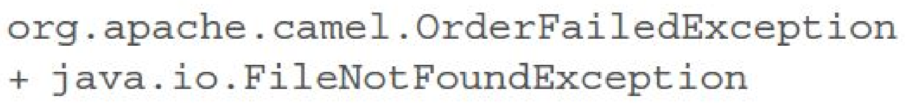
```
onException(ConnectException.class)
    .maximumRedeliveries(5);

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

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

And imagine this exception is thrown:



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

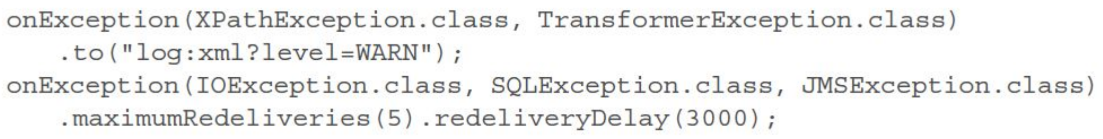
And imagine this exception is thrown:



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

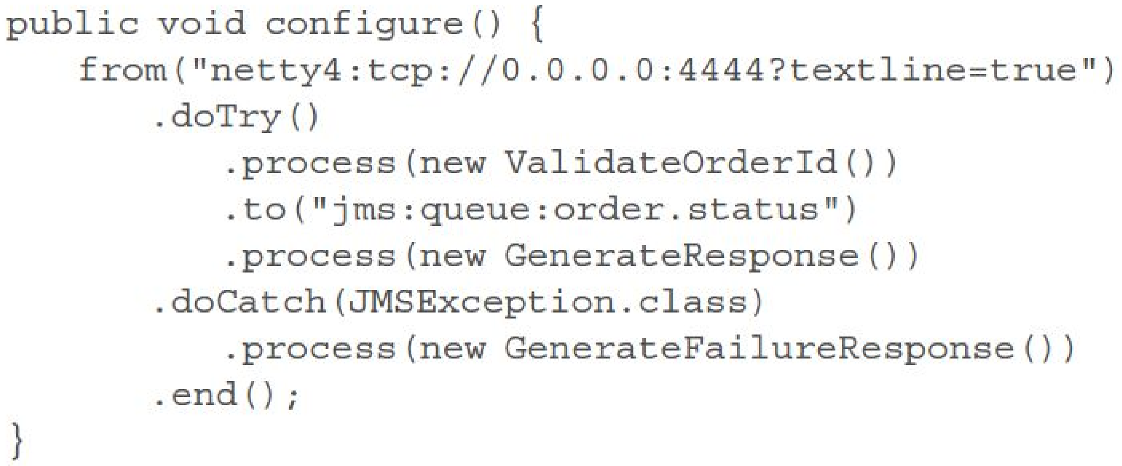
Which of those three onExceptions would be selected?

Multiple exceptions per onException



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

Using doTry, doCatch, and doFinally

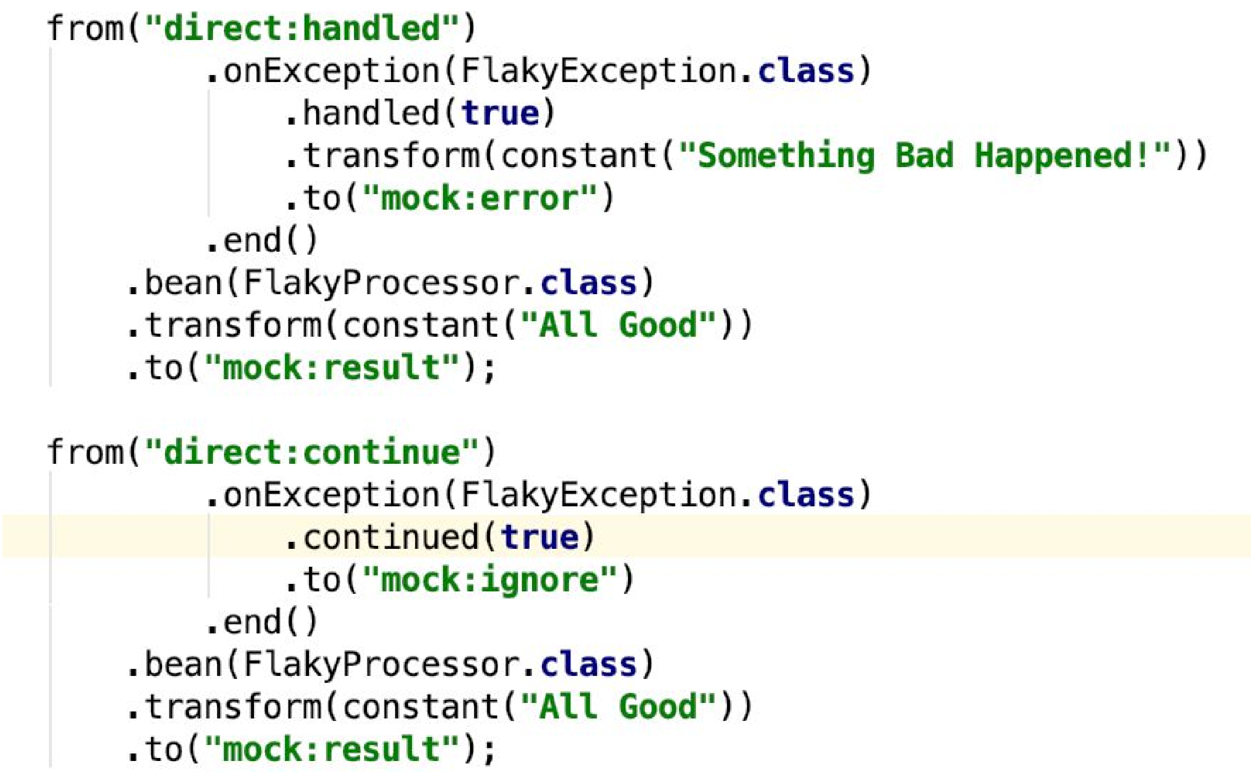


```
public void configure() {
   from("netty4:tcp://0.0.0.0:4444?textline=true")
      .doTry()
         .process(new ValidateOrderId())
         .to("jms:queue:order.status")
         .process(new GenerateResponse())
      .doCatch(JMSException.class)
         .process(new GenerateFailureResponse())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

Limitation:

- it’s only route-scoped.
- 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) 1 to instruct Camel to handle this exception.
- Not declarative
- Embedded
- Mixed happy and unhappy processing in one route

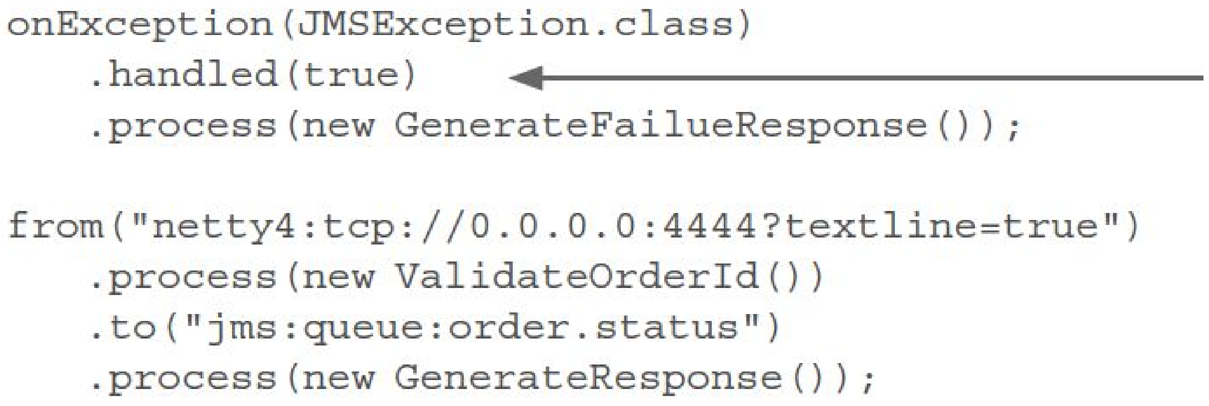
Handled/Continued



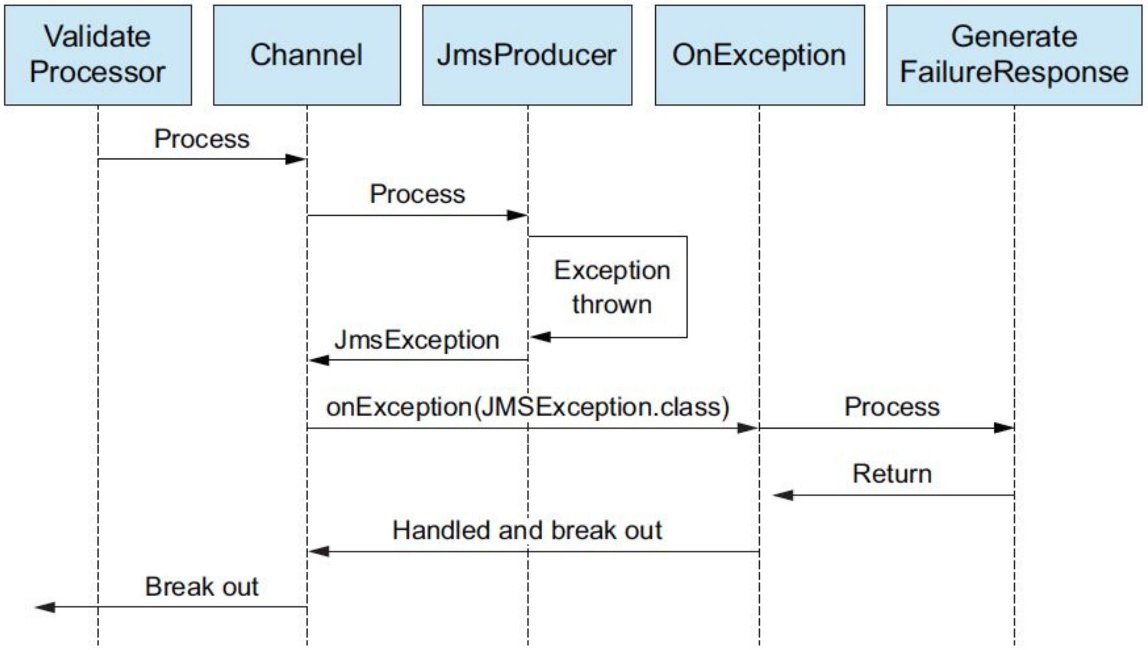
```
from("direct:handled")
        .onException(FlakyException.class)
            .handled(true)
            .transform(constant("Something Bad Happened!"))
            .to("mock:error")
        .end()
    .bean(FlakyProcessor.class)
    .transform(constant("All Good"))
    .to("mock:result");

from("direct:continue")
        .onException(FlakyException.class)
            .continued(true)
            .to("mock:ignore")
        .end()
    .bean(FlakyProcessor.class)
    .transform(constant("All Good"))
    .to("mock:result");
```

- **handle(true)** = the exception is handled and removed from the exchange + break out routing
- **handled(false)** = the exception is not handled, so it will be stored as an exception on the exchange + break out routing
- **continue(true)** = handled(true) + continue routing

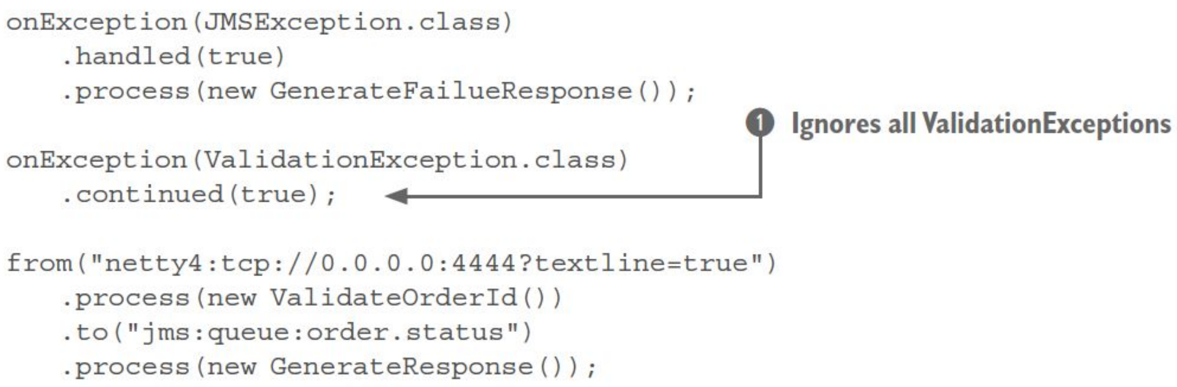
Example



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

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

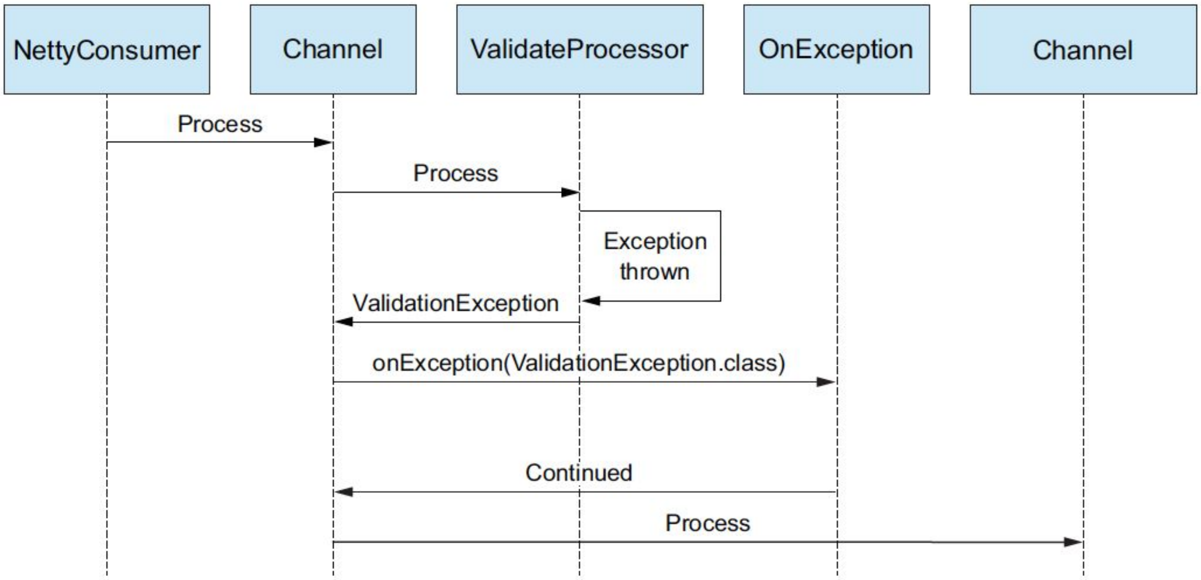


Ignoring exceptions

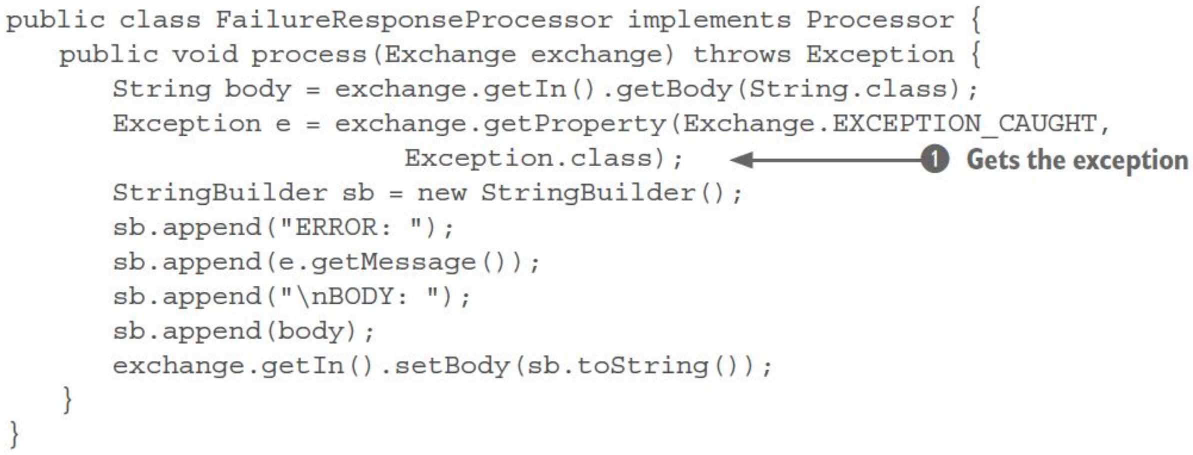


```
onException(JMSException.class)
    .handled(true)
    .process(new GenerateFailueResponse());

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

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

1 Ignores all ValidationExceptions

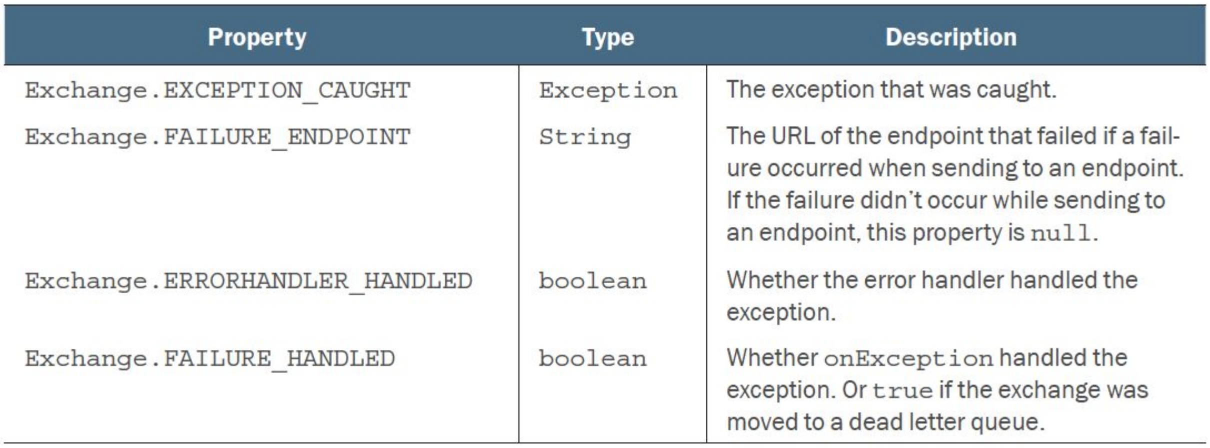


Custom exception handling



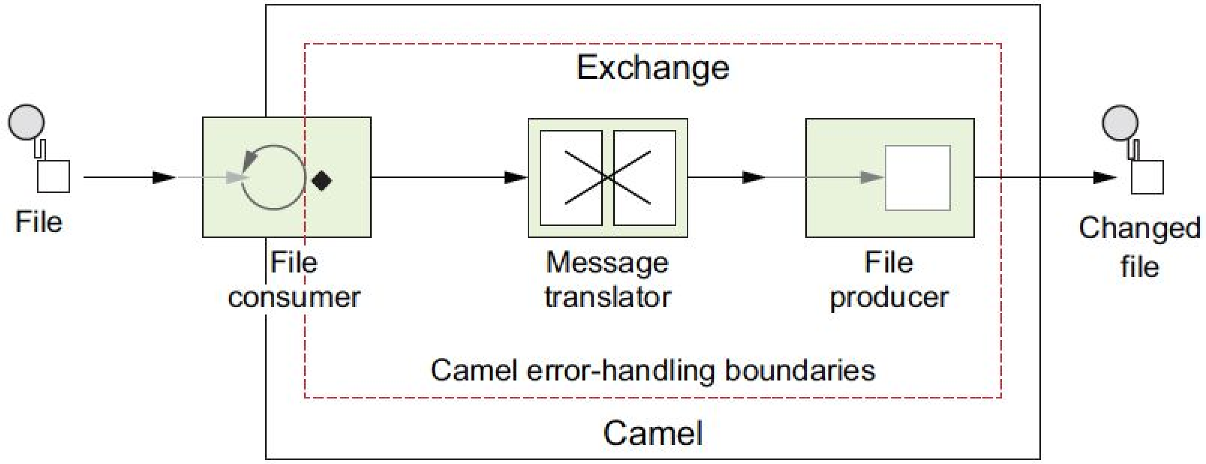
```
public class FailureResponseProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        String body = exchange.getIn().getBody(String.class);
        Exception e = exchange.getProperty(Exchange.EXCEPTION_CAUGHT,
                           Exception.class);
        StringBuilder sb = new StringBuilder();
        sb.append("ERROR: ");
        sb.append(e.getMessage());
        sb.append("\nBODY: ");
        sb.append(body);
        exchange.getIn().setBody(sb.toString());
    }
}
```

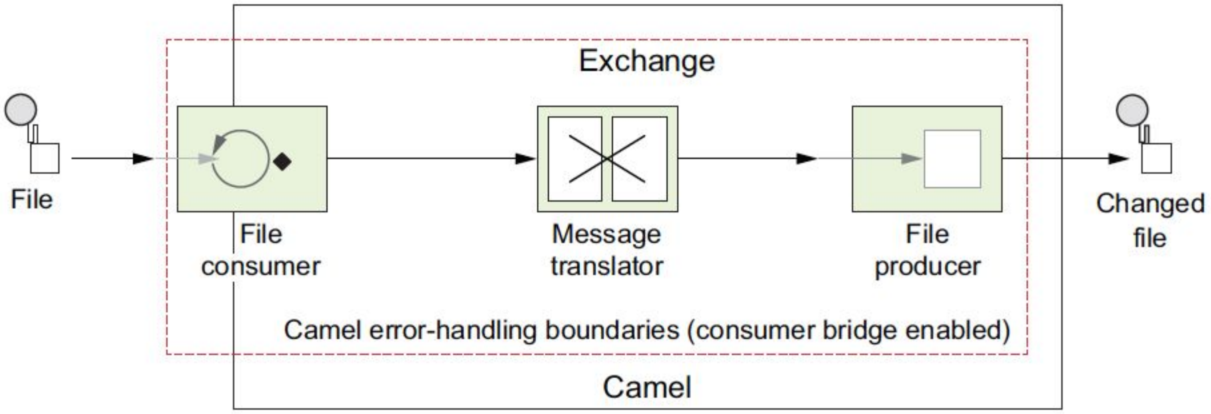
1 Gets the exception

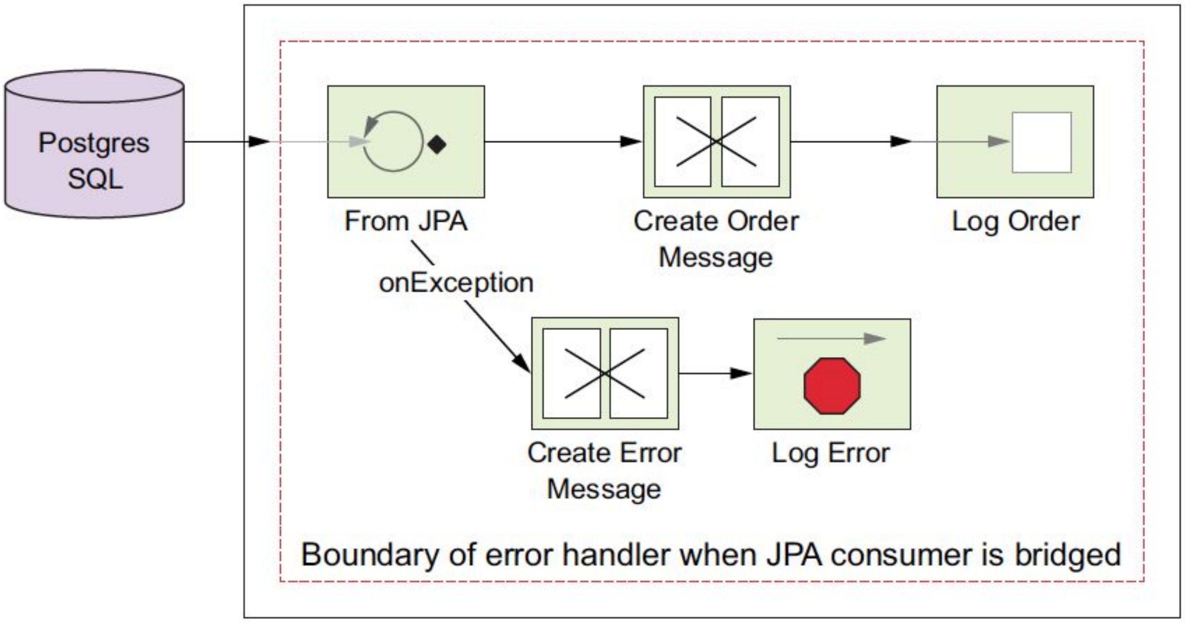


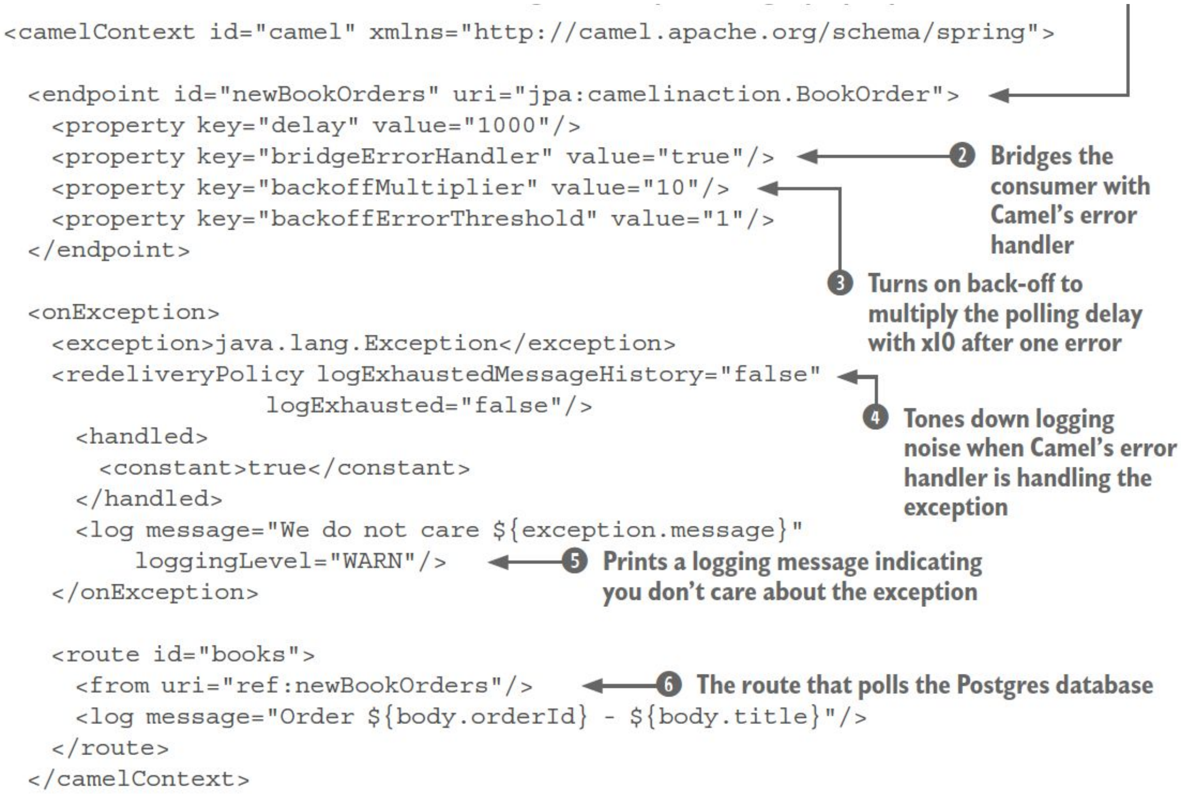
Property	Type	Description
`Exchange.EXCEPTION_CAUGHT`	`Exception`	The exception that was caught.
`Exchange.FAILURE_ENDPOINT`	`String`	The URL of the endpoint that failed if a failure occurred when sending to an endpoint. If the failure didn't occur while sending to an endpoint, this property is `null`.
`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.

Bridging the consumer with Camel’s error handler









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

  <endpoint id="newBookOrders" uri="jpa:camelinaction.BookOrder">
    <property key="delay" value="1000"/>
    <property key="bridgeErrorHandler" value="true"/>
    <property key="backoffMultiplier" value="10"/>
    <property key="backoffErrorThreshold" value="1"/>
  </endpoint>

  <onException>
    <exception>java.lang.Exception</exception>
    <redeliveryPolicy logExhaustedMessageHistory="false"
                      logExhausted="false"/>
      <handled>
        <constant>true</constant>
      </handled>
      <log message="We do not care ${exception.message}"
           loggingLevel="WARN"/>
    </onException>

    <route id="books">
      <from uri="ref:newBookOrders"/>
      <log message="Order ${body.orderId} - ${body.title}"/>
    </route>
  </camelContext>
```

❷ Bridges the consumer with Camel's error handler

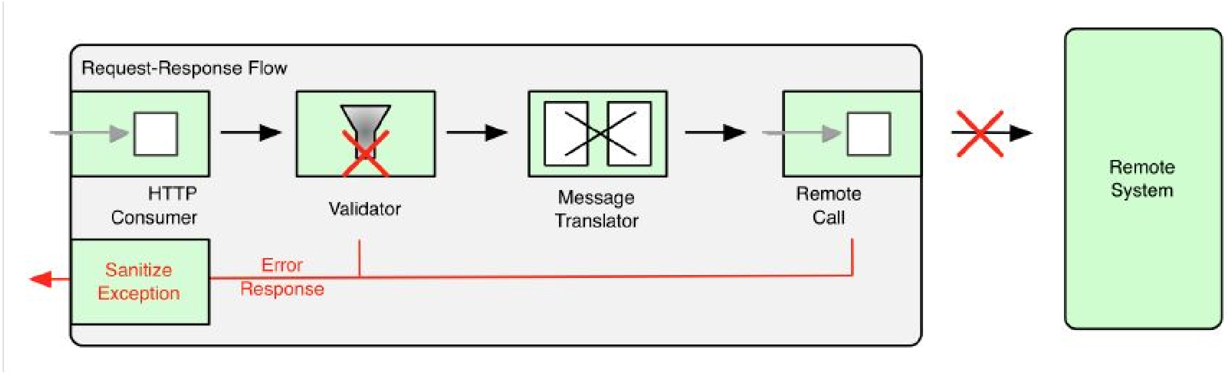
❸ Turns on back-off to multiply the polling delay with x10 after one error

❹ Tones down logging noise when Camel's error handler is handling the exception

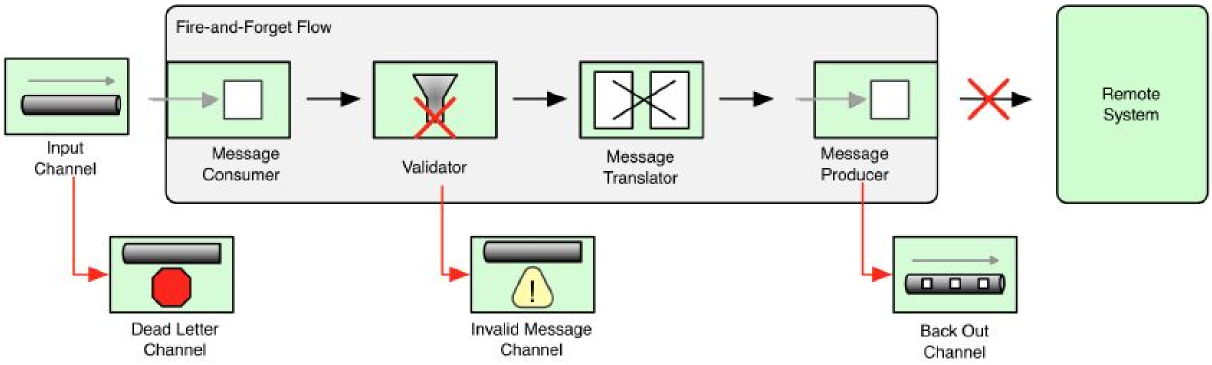
❺ Prints a logging message indicating you don't care about the exception

❻ The route that polls the Postgres database

Error Handling Patterns



Synchronous Error Channel



Asynchronous Error Channels

Summary and best practices

- **Error handling is hard** - realize from the beginning that the unexpected will happen and that dealing with errors is hard.
- **Error handling isn’t an afterthought** - when IT systems are being integrated, they exchange data according to agreed-upon protocols.
- **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.
- **Try to recover** - Some errors are recoverable, such as connection errors. You should try to recover from these errors (retry).
- **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.
- **Build unit tests** - Build unit tests that simulate errors to see whether your error-handling strategies are up to the task.
- **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.
- **Keep doTry-doCatch-doFinally usage to minimum** - for localized error hanging and not general purpose Camel route wide error handling.