## Tuesday, August 17, 2010

**[Camel exception handling overview](http://www.consulting-notes.com/2010/08/camel-exception-handling-overview.html)**

## Here are some notes on adding Camel (v2.3) exception handling to a JavaDSL route.  There are various approaches/options available.  These notes cover the important distinctions between approaches... default handling The default mode uses the DefaultErrorHandler strategy which simply propagates any exception back to the caller and ends the route immediately.  This is rarely the desired behavior, at the very least, you should define a generic/global exception handler to log the errors and put them on a queue for further analysis (during development, testing, etc). onException(Exception)     .to("log:GeneralError?level=ERROR")     .to("activemq:GeneralErrorQueue"); try-catch-finally This approach mimics the Java for exception handling and is designed to be very readable and easy to implement.  It inlines the try/catch/finally blocks directly in the route and is useful for route specific error handling. from("direct:start")     .doTry()         .process(new MyProcessor())     .doCatch(Exception.class)         .to("mock:error");     .doFinally()         .to("mock:end"); onException This approach defines the exception clause separately from the route.  This makes the route and exception handling code more readable and reusable.  Also, the exception handling will apply to any routes defined in its CamelContext. from("direct:start")     .process(new MyProcessor())     .to("mock:end"); onException(Exception.class)     .to("mock:error"); handled/continued These APIs provide valuable control over the flow.   Adding handled(true) tells Camel to not propagate the error back to the caller (should almost always be used).  The continued(true) tells Camel to resume the route where it left off (rarely used, but powerful).  These can both be used to control the flow of the route in interesting ways, for example... from("direct:start")     .process(new MyProcessor())     .to("mock:end"); //send the exception back to the client (rarely used, clients need a meaningful response) onException(ClientException.class)     .handled(false)    //default     .log("error sent back to the client"); //send a readable error message back to the client and handle the error internally onException(HandledException.class)     .handled(true)     .setBody(constant("error"))     .to("mock:error"); //ignore the exception and continue the route (can be dangerous, use wisely) onException(ContinuedException.class)     .continued(true); using a processor for more control If you need more control of the handler code, you can use an inline Processor to get a handle to the exception that was thrown and write your own handle code... onException(Exception.class)     .process(new Processor() {          public void process(Exchange exchange) throws Exception {                Exception exception = (Exception) exchange.getProperty(Exchange.EXCEPTION\_CAUGHT);                //log, email, reroute, etc.          }     }); Summary Overall, the exception handling is very flexible and can meet almost any scenario you can come up with.  For the sake of focusing on the basics, many advanced features haven't been covered here.    Exception Clause

You can use the Exception Clause in the Java [DSL](http://camel.apache.org/dsl.html) to specify the error handling you require on a per exception type basis using the **onException()** method.

To get started we give quick sample before digging into how it works.  
For example if you want to perform a specific piece of processing if a certain exception is raised you can do this simply via:

|  |
| --- |
| onException(ValidationException.class).    to("activemq:validationFailed");    from("seda:inputA").    to("validation:foo/bar.xsd", "activemq:someQueue");    from("seda:inputB").to("direct:foo").    to("rnc:mySchema.rnc", "activemq:anotherQueue"); |

Here if the processing of **seda:inputA** or **seda:inputB** cause a ValidationException to be thrown (such as due to the XSD validation of the [Validation](http://camel.apache.org/validation.html) component or the Relax NG Compact syntax validation of the [Jing](http://camel.apache.org/jing.html) component), then the message will be sent to **activemq:validationFailed** queue.

You can define multiple onException clauses for different behavior

|  |
| --- |
| onException(ValidationException.class).    to("activemq:validationFailed");    onException(ShipOrderException.class).    to("activemq:shipFailed");    from("seda:order").to("bean:processOrder"); |

### Scopes

Exception clauses is scoped as either:

* global (for Java DSL that is per RouteBuilder instances, to reuse, see note below)
* or route specific

Where the **global** are the simplest and most easy to understand. In the advanced section we dig into the route specific and even combining them. However

Global scope for Java DSL is per RouteBuilder instance, so if you want to share among multiple RouteBuilder classes, then create a base abstract RouteBuilder class and put the error handling logic in its configure method. And then extend this class, and make sure to class super.configure(). We are just using the Java inheritance technique.

### How does Camel select which clause should handle a given thrown Exception

Camel uses DefaultExceptionPolicyStrategy to determine a strategy how an exception being thrown should be handled by which onException clause. The strategy is:

* the order in which the onException is configured takes precedence. Camel will test from first...last defined.
* Camel will start from the bottom (nested caused by) and recursive up in the exception hierarchy to find the first matching onException clause
* instanceof test is used for testing the given exception with the onException clause defined exception list. An exact instanceof match will always be used, otherwise the onException clause that has an exception that is the closets super of the thrown exception is selected (recurring up the exception hierarchy)

This is best illustrated with an exception:

|  |
| --- |
| onException(IOException.class).maximumRedeliveries(3);    onException(OrderFailedException.class).maximumRedeliveries(2); |

In the sample above we have defined two exceptions in which IOException is first, so Camel will pickup this exception if there is a match. IOException that is more general is selected then.

So if an exception is thrown with this hierarchy:

|  |
| --- |
| + RuntimeCamelException (wrapper exception by Camel)     + OrderFailedException         + IOException              + FileNotFoundException |

Then Camel will try testing the exception in this order: FileNotFoundException, IOException, OrderFailedException and RuntimeCamelException.  
As we have defined a onException(IOException.class) Camel will select this as it's the **closest** match.

If we add a third onException clause with the FileNotFoundException

|  |
| --- |
| onException(IOException.class).maximumRedeliveries(3);    onException(OrderFailedException.class).maximumRedeliveries(2);    onException(FileNotFoundException.class).handled(true).to("log:nofile"); |

Then with the previous example Camel will now use the last onException(FileNotFoundException.class) as its an **exact** match. Since this is an exact match it will override the general IOException that was used before to handle the same exception thrown.

Now a new situation if this exception was thrown instead:

|  |
| --- |
| + RuntimeCamelException (wrapper exception by Camel)     + OrderFailedException         + OrderNotFoundException |

Then the onException(OrderFailedException.class) will be selected - no surprise here.

And this last sample demonstrates the instanceof test aspect in which Camel will select an exception if it's an instance of the defined exception in the onException clause. Illustrated as:

|  |
| --- |
| + RuntimeCamelException (wrapper exception by Camel)     + SocketException |

Since SocketException is an instanceof IOException, Camel will select the onException(IOException.class) clause.

### Configuring RedeliveryPolicy (redeliver options)

[RedeliveryPolicy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/RedeliveryPolicy.html) requires to use the [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) as the [Error Handler](http://camel.apache.org/error-handler.html). Dead Letter Channel supports attempting to redeliver the message exchange a number of times before sending it to a dead letter endpoint. See [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) for further information about redeliver and which redeliver options exists.

No redelivery is default for onException

By default any [Exception Clause](http://camel.apache.org/exception-clause.html) will **not** redeliver! (as it sets the maximumRedeliveries option to 0).

Sometimes you want to configure the redelivery policy on a per exception type basis. By default in the top examples, if a ValidationException occurs then the message will not be redelivered; however if some other exception occurs (IOException or whatelse) the route will be retried according to the settings from the [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html).

However if you want to customize any methods on the [RedeliveryPolicy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/RedeliveryPolicy.html) object, you can do this via the fluent API. So lets retry in case of ValidationException up till two times.

|  |
| --- |
| onException(ValidationException.class).    maximumRedeliveries(2); |

And the spring DSL:

|  |
| --- |
| <onException>     <exception>com.mycompany.ValidationException</exception>     <redeliveryPolicy maximumRedeliveries="2"/>  </onException> |

You can customize any of the [RedeliveryPolicy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/RedeliveryPolicy.html) so we can for instance set a different delay of 5000 millis:

|  |
| --- |
| <onException>     <exception>com.mycompany.ValidationException</exception>     <redeliveryPolicy maximumRedeliveries="2" delay="5000"/>  </onException> |

#### Point of entry for redelivery attempts

All redelivery attempts start at the point of the failure. So the route:

|  |
| --- |
| .onException(ConnectException.class)  .from("direct:start")   .process("processor1")   .process("processor2") // <--- throws a ConnectException  .to("mock:theEnd") |

Will retry from **processor2** - not the complete route.

#### Reusing RedeliveryPolicy

**Available as of Camel 1.5.1 or later**  
You can reference a RedeliveryPolicy so you can reuse existing configurations and use standard spring bean style configuration that supports property placeholders.

|  |
| --- |
| <bean id="myRedeliveryPolicy" class="org.apache.camel.processor.RedeliveryPolicy">      <property name="maximumRedeliveries" value="${myprop.max}"/>  </bean>     <!-- here we reference our redelivery policy defined above -->   <onException redeliveryPolicyRef="myRedeliveryPolicy">       <!-- you can define multiple exceptions just adding more exception elements as show below -->       <exception>com.mycompany.MyFirstException</exception>       <exception>com.mycompany.MySecondException</exception>   </onException> |

### Asynchronous delayed redelivery

**Available as of Camel 2.4**

From **Camel 2.4** onwards Camel has a feature to **not block** while waiting for a delayed redelivery to occur. However if you use transacted routes then Camel will block as its mandated by the transaction manager to execute all the work in the same thread context. You can enable the non blocking asynchronous behavior by the asyncDelayedRedelivery option. This option can be set on the errorHandler, onException or the redelivery policies.

By default the error handler will create and use a scheduled thread pool to trigger redelivery in the future. From **Camel 2.8** onwards you can configure the executorServiceRef on the [Error Handler](http://camel.apache.org/error-handler.html) to indicate a reference to either a shared thread pool you can enlist in the registry, or a thread pool profile in case you want to be able to control pool settings.

### Catching multiple exceptions

**Available as of Camel 1.5**

In Camel 1.5 the **exception** clauses has been renamed to **onException** and it also supports multiple exception classes:

|  |
| --- |
| onException(MyBusinessException.class, MyOtherBusinessException.class)    .maximumRedeliveries(2)    .to("activemq:businessFailed"); |

And in Spring DSL you just add another exception element:

|  |
| --- |
| <onException>     <exception>com.mycompany.MyBusinessException</exception>     <exception>com.mycompany.MyOtherBusinessException</exception>     <redeliveryPolicy maximumRedeliveries="2"/>     <to uri="activemq:businessFailed"/>  </onException> |

### Using a processor as failure handler

We want to handle certain exceptions specially so we add a **onException** clause for that exception.

|  |
| --- |
| // here we register exception cause for MyFunctionException  // when this exception occur we want it to be processed by our processor  onException(MyFunctionalException.class).process(new MyFunctionFailureHandler()).stop(); |

So what happens is that whenever a MyFunctionalException is thrown it is being routed to our processor MyFunctionFailureHandler. So you can say that the exchange is diverted when a MyFunctionalException is thrown during processing. It's important to distinct this as perfect valid. The default redelivery policy from the [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) will not kick in, so our processor receives the Exchange directly, without any redeliver attempted. In our processor we need to determine what to do. Camel regards the Exchange as **failure handled**. So our processor is the end of the route. So lets look the code for our processor.

|  |
| --- |
| public static class MyFunctionFailureHandler implements Processor {        public void process(Exchange exchange) throws Exception {          // the caused by exception is stored in a property on the exchange          Throwable caused = exchange.getProperty(Exchange.EXCEPTION\_CAUGHT, Throwable.class);          assertNotNull(caused);          // here you can do what you want, but Camel regard this exception as handled, and          // this processor as a failurehandler, so it wont do redeliveries. So this is the          // end of this route. But if we want to route it somewhere we can just get a          // producer template and send it.            // send it to our mock endpoint          exchange.getContext().createProducerTemplate().send("mock:myerror", exchange);      }  } |

Notice how we get the **caused by** exception using a property on the Exchange. This is where Camel stores any caught exception during processing. So you can fetch this property and check what the exception message and do what you want. In the code above we just route it to a mock endpoint using a producer template from Exchange.

## Marking exceptions as being handled

**Available as of Camel 1.5**

Continued

See also the section Handle and continue exceptions below

Using **onException** to handle known exceptions is a very powerful feature in Camel. However prior to Camel 1.5 you could not mark the exception as being handled, so the caller would still receive the caused exception as a response. In Camel 1.5 you can now change this behavior with the new **handle** DSL. The handle is a [Predicate](http://camel.apache.org/predicate.html) that is overloaded to accept three types of parameters:

* Boolean
* [Predicate](http://camel.apache.org/predicate.html)
* [Expression](http://camel.apache.org/expression.html) that will be evaluates as a [Predicate](http://camel.apache.org/predicate.html) using this rule set: If the expressions returns a Boolean its used directly. For any other response its regarded as true if the response is not null.

For instance to mark all ValidationException as being handled we can do this:

|  |
| --- |
| onException(ValidationException).handled(true); |

### Example using handled

In this route below we want to do special handling of all OrderFailedException as we want to return a customized response to the caller. First we setup our routing as:

|  |
| --- |
| // we do special error handling for when OrderFailedException is thrown  onException(OrderFailedException.class)      // we mark the exchange as handled so the caller doesn't receive the      // OrderFailedException but whatever we want to return instead      .handled(true)      // this bean handles the error handling where we can customize the error      // response using java code      .bean(OrderService.class, "orderFailed")      // and since this is an unit test we use mocks for testing      .to("mock:error");    // this is just the generic error handler where we set the destination  // and the number of redeliveries we want to try  errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(1));    // this is our route where we handle orders  from("direct:start")      // this bean is our order service      .bean(OrderService.class, "handleOrder")      // this is the destination if the order is OK      .to("mock:result"); |

Then we have our service beans that is just plain POJO demonstrating how you can use [Bean Integration](http://camel.apache.org/bean-integration.html) in Camel to avoid being tied to the Camel API:

|  |
| --- |
| /\*\*   \* Order service as a plain POJO class   \*/  public static class OrderService {        /\*\*       \* This method handle our order input and return the order       \*       \* @param in      the in headers       \* @param payload the in payload       \* @param out     the out headers       \* @return the out payload       \* @throws OrderFailedException is thrown if the order cannot be processed       \*/      public Object handleOrder(@Headers Map<?, ?> in, @Body String payload, @OutHeaders Map<String, Object> out)          throws OrderFailedException {          out.put("customerid", in.get("customerid"));          if ("Order: kaboom".equals(payload)) {              throw new OrderFailedException("Cannot order: kaboom");          } else {              out.put("orderid", "123");              return "Order OK";          }      }        /\*\*       \* This method creates the response to the caller if the order could not be processed       \* @param in      the in headers       \* @param payload the in payload       \* @param out     the out headers       \* @return the out payload       \*/      public Object orderFailed(@Headers Map<?, ?> in, @Body String payload, @OutHeaders Map<String, Object> out) {          out.put("customerid", in.get("customerid"));          out.put("orderid", "failed");          return "Order ERROR";      }  } |

And finally the exception that is being thrown is just a regular exception:

|  |
| --- |
| /\*\*   \* Exception thrown if the order cannot be processed   \*/  public static class OrderFailedException extends Exception {        private static final long serialVersionUID = 1L;        public OrderFailedException(String message) {          super(message);      }    } |

So what happens?

If we sent an order that is being processed OK then the caller will receive an Exchange as reply containing Order OK as the payload and orderid=123 in a header.

If the order could **not** be processed and thus an OrderFailedException was thrown the caller will **not** receive this exception (as opposed to in Camel 1.4, where the caller received the OrderFailedException) but our customized response that we have fabricated in the orderFailed method in our OrderService. So the caller receives an Exchange with the payload Order ERROR and a orderid=failed in a header.

### Using handled with Spring DSL

The same route as above in Spring DSL:

|  |
| --- |
| <!-- setup our error handler as the deal letter channel -->  <bean id="errorHandler" class="org.apache.camel.builder.DeadLetterChannelBuilder">      <property name="deadLetterUri" value="mock:error"/>  </bean>    <!-- this is our POJO bean with our business logic defined as a plain spring bean -->  <bean id="orderService" class="org.apache.camel.spring.processor.onexception.OrderService" />    <!-- this is the camel context where we define the routes -->  <!-- define our error handler as a global error handler -->  <camelContext errorHandlerRef="errorHandler" xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>      <onException>      <!-- the exception is full qualified names as plain strings -->      <!-- there can be more just add a 2nd, 3rd exception element (unbounded) -->      <exception>org.apache.camel.spring.processor.onexception.OrderFailedException</exception>      <!-- we can set the redelivery policy here as well -->      <redeliveryPolicy maximumRedeliveries="1" />      <!-- mark this as handled -->      <handled>        <constant>true</constant>      </handled>      <!-- let our order service handle this exception, call the orderFailed method -->      <bean ref="orderService" method="orderFailed" />      <!-- and since this is a unit test we use mock for assertions -->      <to uri="mock:error" />    </onException>      <route>      <!-- the route -->      <from uri="direct:start" />      <!-- in the normal route then route to our order service and call handleOrder method -->      <bean ref="orderService" method="handleOrder" />      <!-- and since this is a unit test we use mock for assertions -->      <to uri="mock:result" />    </route>    </camelContext> |

### Handling and sending a fixed response back to the client

In the route above we handled the exception but routed it to a different endpoint. What if you need to alter the response and send a fixed response back to the original caller (the client). No secret here just do as you do in normal Camel routing, use [transform](http://camel.apache.org/message-translator.html) to set the response, as shown in the sample below:

|  |
| --- |
| // we catch MyFunctionalException and want to mark it as handled (= no failure returned to client)  // but we want to return a fixed text response, so we transform OUT body as Sorry.  onException(MyFunctionalException.class)          .handled(true)          .transform().constant("Sorry"); |

We modify the sample slightly to return the original caused exception message instead of the fixed text Sorry:

|  |
| --- |
| // we catch MyFunctionalException and want to mark it as handled (= no failure returned to client)  // but we want to return a fixed text response, so we transform OUT body and return the exception message  onException(MyFunctionalException.class)          .handled(true)          .transform(exceptionMessage()); |

And we can use the [Simple](http://camel.apache.org/simple.html) language to set a readable error message with the caused excepetion message:

|  |
| --- |
| // we catch MyFunctionalException and want to mark it as handled (= no failure returned to client)  // but we want to return a fixed text response, so we transform OUT body and return a nice message  // using the simple language where we want insert the exception message  onException(MyFunctionalException.class)          .handled(true)          .transform().simple("Error reported: ${exception.message} - cannot process this message."); |

## Handle and continue exceptions

**Available as of Camel 2.3**

In Camel 2.3 we introduced a new option continued which allows you to both **handle** and **continue** routing in the original route as if the exception did not occur.

For instance to just ignore and continue if the IDontCareException was thrown we can do this:

|  |
| --- |
| onException(IDontCareException).continued(true); |

You can maybe compare continued with a having a try ... catch block around each step and then just ignore the exception.  
Using continued makes it easier in Camel as you otherwise had to use [Try Catch Finally](http://camel.apache.org/try-catch-finally.html) style for this kind of use case.

### Example using continued

In this route below we want to do special handling of all IllegalArgumentException as we just want to continue routing.

|  |
| --- |
| public void configure() throws Exception {      // tell Camel to handle and continue when this exception is thrown      onException(IllegalArgumentException.class).continued(true);        from("direct:start")          .to("mock:start")          .throwException(new IllegalArgumentException("Forced"))          .to("mock:result");  } |

And the same example in Spring XML:

|  |
| --- |
| <camelContext xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>        <onException>          <exception>java.lang.IllegalArgumentException</exception>          <!-- tell Camel to handle and continue when this exception was thrown -->          <continued><constant>true</constant></continued>      </onException>        <route>          <from uri="direct:start"/>          <to uri="mock:start"/>          <throwException ref="forced"/>          <to uri="mock:result"/>      </route>    </camelContext> |

### What is the difference between handled and continued?

If handled is true, then the thrown exception will be handled and Camel will **not** continue routing in the original route, but break out. However you can configure a route in the onException which will be used instead. You use this route if you need to create some custom response message back to the caller, or do any other processing because that exception was thrown.

If continued is true, then Camel will catch the exception and in fact just ignore it and continue routing in the original route. However if you have a route configured in the onException it will route that route first, before it will continue routing in the original route.

### Using useOriginalMessage

**Available as of Camel 2.0**  
The option **useOriginalMessage** is used for routing the original input body instead of the current body that potential is modified during routing.

For instance if you have this route:

|  |
| --- |
| from("jms:queue:order:input")      .to("bean:validateOrder");      .to("bean:transformOrder")      .to("bean:handleOrder"); |

The route listen for JMS messages and validates, transforms and handle it. During this the [Exchange](http://camel.apache.org/exchange.html) payload is transformed/modified. So in case something goes wrong and we want to move the message to another JMS destination, then we can add an **onException**. But when we move the [Exchange](http://camel.apache.org/exchange.html) to this destination we do not know in which state the message is in. Did the error happen in before the transformOrder or after? So to be sure we want to move the original input message we received from jms:queue:order:input. So we can do this by enabling the **useOriginalMessage** option as shown below:

|  |
| --- |
| // will use original input body  onException(MyOrderException.class)     .useOriginalMessage().handled(true)     .to("jms:queue:order:failed"); |

Then the messages routed to the jms:queue:order:failed is the original input. If we want to manually retry we can move the JMS message from the failed to the input queue, with no problem as the message is the same as the original we received.

#### useOriginalMessage with Spring DSL

The **useOriginalMessage** option is defined as a boolean attribute on the <onException> XML tag in Spring DSL. So the definition above would be:

|  |
| --- |
| <onException useOriginalMessage="true">      <exception>com.mycompany.MyOrderException</exception>      <handled><constant>true</constant></handled>      <to uri="jms:queue:order:failed"/>  </onException> |

## Advanced Usage of [Exception Clause](http://camel.apache.org/exception-clause.html)

### Using global and per route exception clauses

Camel supports quite advanced configuration of exception clauses.

You can define exception clauses either as:

* global
* or route specific

We start off with the sample sample that we change over time. First off we use only global exception clauses:

|  |
| --- |
| // default should errors go to mock:error  errorHandler(deadLetterChannel("mock:error").redeliveryDelay(0));    // if a MyTechnicalException is thrown we will not try to redeliver and we mark it as handled  // so the caller does not get a failure  // since we have no to then the exchange will continue to be routed to the normal error handler  // destination that is mock:error as defined above  onException(MyTechnicalException.class).maximumRedeliveries(0).handled(true);    // if a MyFunctionalException is thrown we do not want Camel to redelivery but handle it our self using  // our bean myOwnHandler, then the exchange is not routed to the default error (mock:error)  onException(MyFunctionalException.class).maximumRedeliveries(0).handled(true).to("bean:myOwnHandler");    // here we route message to our service bean  from("direct:start")      .choice()          .when().xpath("//type = 'myType'").to("bean:myServiceBean")      .end()      .to("mock:result"); |

In the next sample we change the global exception policies to be pure route specific.

Must use .end() for route specific exception policies

**Important:** This requires to end the **onException** route with .end() to indicate where it stops and when the regular route continues.

|  |
| --- |
| // default should errors go to mock:error  errorHandler(deadLetterChannel("mock:error"));    // here we start the routing with the consumer  from("direct:start")        // if a MyTechnicalException is thrown we will not try to redeliver and we mark it as handled      // so the caller does not get a failure      // since we have no to then the exchange will continue to be routed to the normal error handler      // destination that is mock:error as defined above      // we MUST use .end() to indicate that this sub block is ended      .onException(MyTechnicalException.class).maximumRedeliveries(0).handled(true).end()        // if a MyFunctionalException is thrown we do not want Camel to redelivery but handle it our self using      // our bean myOwnHandler, then the exchange is not routed to the default error (mock:error)      // we MUST use .end() to indicate that this sub block is ended      .onException(MyFunctionalException.class).maximumRedeliveries(0).handled(true).to("bean:myOwnHandler").end()        // here we have the regular routing      .choice()          .when().xpath("//type = 'myType'").to("bean:myServiceBean")      .end()      .to("mock:result"); |

And now it gets complex as we combine global and route specific exception policies as we introduce a 2nd route in the sample:

|  |
| --- |
| // global error handler  // as its based on a unit test we do not have any delays between and do not log the stack trace  errorHandler(deadLetterChannel("mock:error").redeliveryDelay(0).logStackTrace(false));    // shared for both routes  onException(MyTechnicalException.class).handled(true).maximumRedeliveries(2).to("mock:tech.error");    from("direct:start")      // route specific on exception for MyFunctionalException      // we MUST use .end() to indicate that this sub block is ended      .onException(MyFunctionalException.class).maximumRedeliveries(0).end()      .to("bean:myServiceBean")      .to("mock:result");    from("direct:start2")      // route specific on exception for MyFunctionalException that is different than the previous route      // here we marked it as handled and send it to a different destination mock:handled      // we MUST use .end() to indicate that this sub block is ended      .onException(MyFunctionalException.class).handled(true).maximumRedeliveries(0).to("mock:handled").end()      .to("bean:myServiceBean")      .to("mock:result"); |

Notice that we can define the same exception MyFunctionalException in both routes, but they are configured differently and thus is handled different depending on the route. You can of course also add a new onException to one of the routes so it has an additional exception policy.

And finally we top this by throwing in a nested error handler as well, as we add the 3rd route shown below:

|  |
| --- |
| from("direct:start3")      // route specific error handler that is different than the global error handler      // here we do not redeliver and send errors to mock:error3 instead of the global endpoint      .errorHandler(deadLetterChannel("mock:error3")              .maximumRedeliveries(0))        // route specific on exception to mark MyFunctionalException as being handled      .onException(MyFunctionalException.class).handled(true).end()      // however we want the IO exceptions to redeliver at most 3 times      .onException(IOException.class).maximumRedeliveries(3).end()      .to("bean:myServiceBean")      .to("mock:result"); |

Global exception policies and nested error handlers

The sample above with both nested error handlers and both global and per route exception clauses is a bit advanced. It's important to get the fact straight that the **global** exception clauses is really global so they also applies for nested error handlers. So if a MyTechnicalException is thrown then it's the global exception policy that is selected.

### Using fine grained selection using onWhen predicate

**Available as of Camel 1.5.1 or later**

You can attach an [Expression](http://camel.apache.org/expression.html) to the exception clause to have fine grained control when a clause should be selected or not. As it's an [Expression](http://camel.apache.org/expression.html) you can use any kind of code to perform the test. Here is a sample:

|  |
| --- |
| public void configure() throws Exception {      errorHandler(deadLetterChannel("mock:error").redeliveryDelay(0).maximumRedeliveries(3));        // here we define our onException to catch MyUserException when      // there is a header[user] on the exchange that is not null      onException(MyUserException.class).onWhen(header("user").isNotNull())          .maximumRedeliveries(1)          .to(ERROR\_USER\_QUEUE);        // here we define onException to catch MyUserException as a kind      // of fallback when the above did not match.      // Notice: The order how we have defined these onException is      // important as Camel will resolve in the same order as they      // have been defined      onException(MyUserException.class)          .maximumRedeliveries(2)          .to(ERROR\_QUEUE); |

In the sample above we have two onException's defined. The first has an **onWhen** expression attached to only trigger if the message has a header with the key user that is not null. If so this clause is selected and is handling the thrown exception. The 2nd clause is a for coarse gained selection to select the same exception being thrown but when the expression is evaluated to false. **Notice:** this is not required, if the 2nd clause is omitted, then the default error handler will kick in.

### Using onRedelivery processor

**Available as of Camel 2.0**

[Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) has support for **onRedelivery** to allow custom processing of a Message before its being redelivered. It can be used to add some customer header or whatnot. In Camel 2.0 we have added this feature to [Exception Clause](http://camel.apache.org/exception-clause.html) as well, so you can use per exception scoped on redelivery. Camel will fallback to use the one defined on [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) if any, if none exists on the [Exception Clause](http://camel.apache.org/exception-clause.html). See [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) for more details on **onRedelivery**.

In the code below we want to do some custom code before redelivering any IOException. So we configure an **onException** for the IOException and set the **onRedelivery** to use our custom processor:

|  |
| --- |
| // when we redeliver caused by an IOException we want to do some special  // code before the redeliver attempt  onException(IOException.class)          // try to redeliver at most 3 times          .maximumRedeliveries(3)          .onRedelivery(new MyIORedeliverProcessor()); |

And in our custom processor we set a special timeout header to the message. You can of course do anything what you like in your code.

|  |
| --- |
| // This is our processor that is executed before IOException redeliver attempt  // here we can do what we want in the java code, such as altering the message    public static class MyIORedeliverProcessor implements Processor {        public void process(Exchange exchange) throws Exception {          // just for show and tell, here we set a special header to instruct          // the receive a given timeout value          exchange.getIn().setHeader("Timeout", 5000);      }  } |

#### Using onRedelivery in Spring DSL

In Spring DSL you need to use the **onRedeliveryRef** attribute to refer to a spring bean id that is your custom processor:

|  |
| --- |
| <onException onRedeliveryRef="myIORedeliverProcessor">      <exception>java.io.IOException</exception>  </onException> |

And our processor is just a regular spring bean (we use $ for the inner class as this code is based on unit testing)

|  |
| --- |
| <bean id="myRedeliveryProcessor"        class="org.apache.camel.processor.DeadLetterChannelOnExceptionOnRedeliveryTest$MyRedeliverProcessor"/> |

### Using fine grained retry using retryWhile predicate

**Available as of Camel 2.0**

In Camel 2.0 to 2.3 its called **retryUntil**. From Camel 2.4 onwards its named **retryWhile** because Camel will continue doing retries while the predicate returns true.

When you need fine grained control for determining if an exchange should be retried or not you can use the **retryWhile** predicate. Camel will redeliver until the predicate returns false.  
This is demonstrated in the sample below:

|  |
| --- |
| // we want to use a predicate for retries so we can determine in our bean  // when retry should stop, notice it will overrule the global error handler  // where we defined at most 1 redelivery attempt. Here we will continue until  // the predicate returns false  onException(MyFunctionalException.class)          .retryWhile(method("myRetryHandler"))          .handled(true)          .transform().constant("Sorry"); |

Where the bean myRetryHandler is computing if we should retry or not:

|  |
| --- |
| public class MyRetryBean {        // using bean binding we can bind the information from the exchange to the types we have in our method signature      public boolean retry(@Header(Exchange.REDELIVERY\_COUNTER) Integer counter, @Body String body, @ExchangeException Exception causedBy) {          // NOTE: counter is the redelivery attempt, will start from 1          invoked++;            assertEquals("Hello World", body);          assertTrue(causedBy instanceof MyFunctionalException);            // we can of course do what ever we want to determine the result but this is a unit test so we end after 3 attempts          return counter < 3;      }  } |

### Using custom ExceptionPolicyStrategy

**Available in Camel 1.4**

The default [ExceptionPolicyStrategy](http://camel.apache.org/maven/current/camel-core/apidocs/org/apache/camel/processor/exceptionpolicy/ExceptionPolicyStrategy.html) in Camel should be sufficient in nearly all use-cases (see section How does Camel select which clause should handle a given thrown Exception). However if you need to use your own this can be configued as the sample below illustrates:

|  |
| --- |
| public void configure() throws Exception {      // configure the error handler to use my policy instead of the default from Camel      errorHandler(deadLetterChannel("mock:error").exceptionPolicyStrategy(new MyPolicy()));        onException(MyPolicyException.class)          .maximumRedeliveries(1)          .setHeader(MESSAGE\_INFO, constant("Damm my policy exception"))          .to(ERROR\_QUEUE);        onException(CamelException.class)          .maximumRedeliveries(3)          .setHeader(MESSAGE\_INFO, constant("Damm a Camel exception"))          .to(ERROR\_QUEUE); |

Using our own strategy **MyPolicy** we can change the default behavior of Camel with our own code to resolve which [ExceptionType](http://camel.apache.org/maven/camel-core/apidocs/org/apache/camel/model/ExceptionType.html) from above should be handling the given thrown exception.

|  |
| --- |
| public static class MyPolicy implements ExceptionPolicyStrategy {        public OnExceptionDefinition getExceptionPolicy(Map<ExceptionPolicyKey, OnExceptionDefinition> exceptionPolicices,                                              Exchange exchange,                                              Throwable exception) {          // This is just an example that always forces the exception type configured          // with MyPolicyException to win.          return exceptionPolicices.get(new ExceptionPolicyKey(null, MyPolicyException.class, null));      }  } |

### Using the exception clause in Spring DSL

You can use all of the above mentioned exception clause features in the Spring DSL as well. Here are a few examples:

* Global scoped - **Available in Camel 2.0**

|  |
| --- |
| <!-- setup our error handler as the deal letter channel -->  <bean id="errorHandler" class="org.apache.camel.builder.DeadLetterChannelBuilder">      <property name="deadLetterUri" value="mock:error"/>  </bean>    <!-- this is our POJO bean with our business logic defined as a plain spring bean -->  <bean id="orderService" class="org.apache.camel.spring.processor.onexception.OrderService" />    <!-- this is the camel context where we define the routes -->  <!-- define our error handler as a global error handler -->  <camelContext errorHandlerRef="errorHandler" xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>      <onException>      <!-- the exception is full qualified names as plain strings -->      <!-- there can be more just add a 2nd, 3rd exception element (unbounded) -->      <exception>org.apache.camel.spring.processor.onexception.OrderFailedException</exception>      <!-- we can set the redelivery policy here as well -->      <redeliveryPolicy maximumRedeliveries="1" />      <!-- mark this as handled -->      <handled>        <constant>true</constant>      </handled>      <!-- let our order service handle this exception, call the orderFailed method -->      <bean ref="orderService" method="orderFailed" />      <!-- and since this is a unit test we use mock for assertions -->      <to uri="mock:error" />    </onException>      <route>      <!-- the route -->      <from uri="direct:start" />      <!-- in the normal route then route to our order service and call handleOrder method -->      <bean ref="orderService" method="handleOrder" />      <!-- and since this is a unit test we use mock for assertions -->      <to uri="mock:result" />    </route>    </camelContext> |

* Route specific scoped

|  |
| --- |
| <!-- setup our error handler as the deal letter channel -->  <bean id="deadLetter" class="org.apache.camel.builder.DeadLetterChannelBuilder">      <property name="deadLetterUri" value="mock:dead"/>  </bean>    <!-- the default error handler used in the 2nd route -->  <bean id="defaultErrorHandler" class="org.apache.camel.builder.DefaultErrorHandlerBuilder"/>    <!-- this is our POJO bean with our business logic defined as a plain spring bean -->  <bean id="orderService" class="org.apache.camel.spring.processor.onexception.OrderService"/>    <!-- this is the camel context where we define the routes -->  <camelContext xmlns="[http://camel.apache.org/schema/spring"](http://camel.apache.org/schema/spring%22)>        <route errorHandlerRef="deadLetter">          <from uri="direct:start"/>          <onException>              <exception>org.apache.camel.spring.processor.onexception.OrderFailedException</exception>              <redeliveryPolicy maximumRedeliveries="1"/>              <handled>                  <constant>true</constant>              </handled>              <bean ref="orderService" method="orderFailed"/>              <to uri="mock:error"/>          </onException>          <bean ref="orderService" method="handleOrder"/>          <to uri="mock:result"/>      </route>        <!-- The exception clause specified in the first route will not be used in this route -->      <route errorHandlerRef="defaultErrorHandler">          <from uri="direct:start\_with\_no\_handler"/>          <bean ref="orderService" method="handleOrder"/>          <to uri="mock:result"/>      </route>    </camelContext> |

### See also

The [Error Handler](http://camel.apache.org/error-handler.html) for the general error handling documentation  
The [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) for further details.  
The [Transactional Client](http://camel.apache.org/transactional-client.html) for transactional behavior

## Error handling in Camel

Error handling in Camel can roughly be separated into two distinct types:

* non transactional
* transactional

Where non transactional is the most common type that is enabled out-of-the-box and handled by Camel itself. The transaction type is handled by a backing system such as a J2EE application server.

Using try ... catch ... finally

Related to error handling is the [Try Catch Finally](http://camel.apache.org/try-catch-finally.html) feature in Camel.

### When does an error happen

An error happens when

* any uncaught exception is thrown during routing and processing of messages within Camel

So think of this as a big exception interceptor that catches all exceptions and handles what to do.

### Non transactional

By default Camel uses the non transaction type and orchestrates the error handling during processing and routing.

As there isn't a single error handling configuration that suites all uses cases, you should consider altering the default configurations to better suit you needs.

#### Camel 1.x default error handler

In Camel 1.x a global [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) is setup as the [Error Handler](http://camel.apache.org/error-handler.html) by default. It's configured as:

* redeliver up to 6 times
* pause 1 second between each redelivery attempt
* if all redelivery attempts failed then move exchange into the dead letter queue
* the default dead letter queue is a logger that logs the exchange at ERROR level (star)

Dead Letter Queue (\*)

A dead letter queue is like a black hole, it will consume the [Exchange](http://camel.apache.org/exchange.html) and the [Exchange](http://camel.apache.org/exchange.html) routing is ended with no indication that it failed.  
This works great in the [JMS](http://camel.apache.org/jms.html) Messaging world where we don't want a bad message to cause endless retries and causing the system to exhaust. The message is said to be poison and thus we want to move it to a dead letter queue so the system can continue to operate and work with the next message.

This default does not go well with other transports using in a request/reply messaging style. If the [Exchange](http://camel.apache.org/exchange.html) failed then the original caller want to be alter it failed.

So the bottom line is that you **must** configure and setup the error handling strategies that suits your business needs.

#### Camel 2.0 onwards default error handler

In Camel 2.0 onwards a global [DefaultErrorHandler](http://camel.apache.org/defaulterrorhandler.html) is set up as the [Error Handler](http://camel.apache.org/error-handler.html) by default. It's configured as:

* no redeliveries
* no dead letter queue
* if the exchange failed an exception is thrown and propagated back to the original caller wrapped in a RuntimeCamelException.

### Scopes

Camel supports 2 scopes that is determined by the DSL in use:

|  |  |  |  |
| --- | --- | --- | --- |
| **DSL** | **Scope 1** | **Scope 2** | **Note** |
| XML DSL | CamelContext | route | Scope 2 takes precedence over scope 1 |
| Java/Scala DSL | RouteBuilder | route | Scope 2 takes precedence over scope 1 |

When using XML DSL then scope 1 applies for all routes. Where as when using Java DSL then route 1 only applies for the given RouteBuilder instance. So if you have multiple RouteBuilder's then each route builder has its own scope 1.

If you want to share scope among RouteBuilder's you can use class inheritance and create a base class, and then extend this class for your RouteBuilder's and invoke the super.configure() method.

Mind that there was a bug in Camel that affected the scopes when using multiple RouteBuilder classes. See more details at [CAMEL-5456](https://issues.apache.org/jira/browse/CAMEL-5456).

### How does the [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) error handler work

When Camel is started it will inspect the routes and weave in the error handling into the routing. With up to 3 supported scopes, the error handling can be quite complex. And on top of that you have inherited error handling and you can even configure [Exception Clause](http://camel.apache.org/exception-clause.html)s to handle specific exception types differently. So yes it's advanced but very powerful when you get the grip of it.

To keep things simple we first look at the basic concept how Camel orchestrates the redelivery attempt. At any given node in the route graph Camel intercepts the current Exchange being routed and wraps it with the [Error Handler](http://camel.apache.org/error-handler.html). This ensures that the [Error Handler](http://camel.apache.org/error-handler.html) can kick in, just as the AOP around concept. If the exchange can be routed without any problems then it's forwarded to the next node in the route graph, **But** if there was an exception thrown, then the [Error Handler](http://camel.apache.org/error-handler.html) kicks in and decides what to do.

An example illustrating this:

|  |
| --- |
| errorHandler(deadLetterChannel("jms:queue:dead"));    from("seda:newOrder")     .to("bean:validateOrder")     .to("bean:storeOrder")     .to("bean:confirmOrder"); |

In this route we have 3 nodes (the dots) where the [Error Handler](http://camel.apache.org/error-handler.html) is watching us (The AOP around stuff). So when an order arrives on the seda queue we consume it and send it to the validateOrder bean. In case the validation bean processed ok, we move on to the next node. In case the storeOrder bean failed and throws an exception it's caught by the [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) that decides what to do next. Either it does a:

* redeliver
* or move it to dead letter queue

It will continue to do redeliveries based on the policy configured. By default [Dead Letter Channel](http://camel.apache.org/dead-letter-channel.html) will attempt at most 6 redeliveries with 1 second delay. So if the storeOrder bean did succeed at the 3rd attempt the routing will continue to the next node the confirmOrder bean. In case all redeliveries failed the Exchange is regarded as failed and is moved to the dead letter queue and the processing of this exchange stops. By default the dead letter queue is just a ERROR logger.

This applies to all kind of [Components](http://camel.apache.org/components.html) in Camel. The sample above only uses [Bean](http://camel.apache.org/bean.html) but it's the same for [File](http://camel.apache.org/file2.html), [Mail](http://camel.apache.org/mail.html), [Velocity](http://camel.apache.org/velocity.html) or whatever component you use.

### Transactional

Camel leverages Spring transactions. Usually you can only use this with a limited number of transport types such as JMS or JDBC based, that yet again requires a transaction manager such as a Spring transaction, a J2EE server or a Message Broker.

#### How does it work

**Camel 1.x**  
Camel does the same weaving as for the non-transactional type. The difference is that for transactional exchanges the [Error Handler](http://camel.apache.org/error-handler.html) does **not** kick in. You can say the AOP around does not apply. Camel relies solely on the backing system to orchestrate the error handling. And as such the when the backing system does redeliver it will start all over again. For instance if the exchange was started by a JMS consumer then it's started again as the JMS message is rolled back on the JMS queue and Camel will re consume the JMS message again.

**Camel 2.0**  
In Camel 2.0 we have empowered the [TransactionErrorHandler](http://camel.apache.org/transactionerrorhandler.html) to build on top of the same base that [DefaultErrorHandler](http://camel.apache.org/defaulterrorhandler.html) does. This allows you to use Camel redelivery with transactional routes as well. The Spring transaction manager is still in charge and have the last say. But you can use Camel to do some local redelivery, for instance to upload a file to a FTP server, in which Camel can do local redelivery. So this gives you the power from both worlds. In case Camel cannot redeliver the exchange will be failed and rolled back. By default the [TransactionErrorHandler](http://camel.apache.org/transactionerrorhandler.html) does **not** attempt any local redeliveries. You have to configure it to do so, for instance to set a maximum redelivers to a number > 0.

See [Transactional Client](http://camel.apache.org/transactional-client.html) for more.

**Try ... Catch ... Finally**

Camel supports the Java equivalent of try .. catch and finally directly in the DSL.  
It aims to work like its Java sisters but with more power. Especially in Camel 2.0 where we gave this feature an overhaul.

In Camel we prefix the keywords with do to avoid having same keyword as Java. So we have:

* doTry
* doCatch
* doFinally
* end to end the block in Java DSL

Notice this document is based on how it works in Camel 2.0. In Camel 1.x this feature isn't as powerful and it uses a slight different keyword names.

Camel error handling is disabled

When using doTry .. doCatch .. doFinally then the regular Camel [Error Handler](http://camel.apache.org/error-handler.html) does not apply. That means any onException or the likes does not trigger. The reason is that doTry .. doCatch .. doFinally is in fact its own error handler and that it aims to mimic and work like how try/catch/finally works in Java.

**About doCatch and its power over Java**

The doCatch in Camel is empowered over its Java sister.

First of all you can define multiple exceptions to catch in a single block.

And second of all an important aspect over the regular Java counter parts is that Camel will check in the exception hierarchy when it matches a thrown exception against the doCatch blocks. The reasons is that many times the original caused exceptions is wrapped by other wrapper exceptions, typically transposing the exception from a checked to a runtime exception.  
Camel for instance does this by wrapped it in a CamelRuntimeException. So if the original caused exception is an java.io.IOException then Camel will still match a doCatch block defined with an java.io.IOException. And just like Java the order in which you have multiple doCatch blocks matter. Camel will iterate from the top going down and use the first doCatch that matches the exception. The reason is to keep it similar to the regular java and how it selects a catch block. This differers from the [Exception Clause](http://camel.apache.org/exception-clause.html) that has a more intelligent exception selection strategy among multiple onException definitions, where it also consider the delta in the exception hierarchy to select the best definition.

A third feature is that you can attach a onWhen predicate to signal if the catch should trigger or not at runtime.

And to simulate *rethrowing* an exception from a doCatch you should use the handled predicate. If its evaluated to false Camel will reattach the exception on the [Exchange](http://camel.apache.org/exchange.html).

**Using try .. catch .. finally in Java DSL**

In the route below we have all keywords in action. As the code is based on a unit test we route using [Mock](http://camel.apache.org/mock.html).

|  |
| --- |
| from("direct:start")      .doTry()          .process(new ProcessorFail())          .to("mock:result")      .doCatch(IOException.class, IllegalStateException.class)          .to("mock:catch")      .doFinally()          .to("mock:finally")      .end(); |

And in the route below we want to indicate if an IOException occured we want to route it elsewhere and at the same time keep the exception so the original caller is notified about this exception. To do this we need to not *rethrow* the exception and this is why we use **handled** and set it to false to indicate, no we did not handle it so please keep the exception.  
The 2nd exception block can be omitted but as the code is based on an unit test we want to test the behavior non IOException as well.

|  |
| --- |
| from("direct:start")      // here is our try where we try processing the exchange in the route below if it fails      // we can catch it below, just like regular try .. catch .. finally in Java      .doTry()          .process(new ProcessorFail())          .to("mock:result")      // catch IOExcption that we do not want to handle, eg the caller should get the error back      .doCatch(IOException.class)          // mark this as NOT handled, eg the caller will also get the exception          .handled(false)          .to("mock:io")      .doCatch(Exception.class)          // and catch all other exceptions          // they are handled by default (ie handled = true)          .to("mock:error")      // here the try block ends      .end(); |

And finally we have an example of the onWhen predicate in action. We can attach it to a doCatch block and at runtime determine if the block should be triggered or not.  
In our case we only want to trigger if the caused exception message contains the **damn** word.

|  |
| --- |
| from("direct:start")      // here is our try where we try processing the exchange in the route below if it fails      // we can catch it below, just like regular try .. catch .. finally in Java      .doTry()          .process(new ProcessorFail())          .to("mock:result")      // here we catch the following 2 exceptions but only if      // the onWhen predicate matches, eg if the exception messsage      // conatins the string word Damn      .doCatch(IOException.class, IllegalStateException.class)          .onWhen(exceptionMessage().contains("Damn"))          .to("mock:catch")      // another catch for CamelExchangeException that does not have any onWhen predicate      .doCatch(CamelExchangeException.class)          .to("mock:catchCamel")      // and the finally that is always processed      .doFinally()          .to("mock:finally")      // here the try block ends      .end(); |

Use end() to end the block

Notice when using Java DSL we must use end() to indicate where the try .. catch .. finally block ends. As the example above has a finally, then the end() should be at the end of the finally block. If we are not using a finally, then the end() should be at the end of the doCatch to indicate the end there.

**Using try .. catch .. finally in Spring DSL**

We show the three sample samples using Spring DSL instead.

In the route below we have all keywords in action. As the code is based on a unit test we route using [Mock](http://camel.apache.org/mock.html).

|  |
| --- |
| <route>      <from uri="direct:start"/>      <!-- here the try starts. its a try .. catch .. finally just as regular java code -->      <doTry>          <process ref="processorFail"/>          <to uri="mock:result"/>          <doCatch>              <!-- catch multiple exceptions -->              <exception>java.io.IOException</exception>              <exception>java.lang.IllegalStateException</exception>              <to uri="mock:catch"/>          </doCatch>          <doFinally>              <to uri="mock:finally"/>          </doFinally>      </doTry>  </route> |

And in the route below we want to indicate if an IOException occured we want to route it elsewhere and at the same time keep the exception so the original caller is notified about this exception. To do this we need to not *rethrow* the exception and this is why we use **handled** and set it to false to indicate, no we did not handle it so please keep the exception.  
The 2nd exception block can be omitted but as the code is based on an unit test we want to test the behavior non IOException as well.

|  |
| --- |
| <route>      <from uri="direct:start"/>      <!-- here the try starts. its a try .. catch .. finally just as regular java code -->      <doTry>          <process ref="processorFail"/>          <to uri="mock:result"/>          <doCatch>              <!-- catch IOExcption that we do not want to handle, eg the caller should get the error back -->              <exception>java.io.IOException</exception>              <!-- mark this as NOT handled, eg the caller will also get the exception -->              <handled>                  <constant>false</constant>              </handled>              <to uri="mock:io"/>          </doCatch>          <doCatch>              <!-- and catch all other exceptions they are handled by default (ie handled = true) -->              <exception>java.lang.Exception</exception>              <to uri="mock:error"/>          </doCatch>      </doTry>  </route> |

And finally we have an example of the onWhen predicate in action. We can attach it to a doCatch block and at runtime determine if the block should be triggered or not.  
In our case we only want to trigger if the caused exception message contains the **damn** word.

|  |
| --- |
| <route>      <from uri="direct:start"/>      <!-- here the try starts. its a try .. catch .. finally just as regular java code -->      <doTry>          <process ref="processorFail"/>          <to uri="mock:result"/>          <!-- here we catch the below 2 kind of exceptions but ONLY if the onWhen predicate matches               that means that the exception message should contain the string word 'Damn' -->          <doCatch>              <exception>java.io.IOException</exception>              <exception>java.lang.IllegalStateException</exception>              <onWhen>                  <simple>${exception.message} contains 'Damn'</simple>              </onWhen>              <to uri="mock:catch"/>          </doCatch>          <!-- we can have multiple catch blocks for different exception and with their own onWhen -->          <doCatch>              <exception>org.apache.camel.CamelExchangeException</exception>              <to uri="mock:catchCamel"/>          </doCatch>          <!-- the finally is always processed -->          <doFinally>              <to uri="mock:finally"/>          </doFinally>      </doTry>  </route> |