protected RouteBuilder createRouteBuilder() throws Exception {  
 return new RouteBuilder() {  
 public void configure() throws Exception {  
 // START SNIPPET: e1  
 // 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");

public void testOrderERROR() throws Exception {

MockEndpoint error = getMockEndpoint("mock:error");

//mock:error get the message  
 error.expectedBodiesReceived("Order ERROR");  
 error.expectedHeaderReceived("orderid", "failed");

// mock result did not get any exchange  
 MockEndpoint result = getMockEndpoint("mock:result");  
 result.expectedMessageCount(0);  
  
 Object out = template.requestBodyAndHeader("direct:start", "Order: kaboom", "customerid", "555");

assertEquals("Order ERROR", out);  
  
 assertMockEndpointsSatisfied();  
}

C:\Users\gzhoue\Desktop\TempCode\camel\camel-core\src\test\java\org\apache\camel\processor\DeadLetterChannelHandledExampleTest.java

public 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<String, Object> 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<String, Object> in, @Body String payload, @OutHeaders Map<String, Object> out) {  
 out.put("customerid", in.get("customerid"));  
 out.put("orderid", "failed");  
 return "Order ERROR";  
 }  
}

C:\Users\gzhoue\Desktop\TempCode\camel\components\camel-spring\src\test\java\org\apache\camel\spring\processor\onexception\OrderService.java

**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(*like the example above illustrated*), 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.

<http://camel.apache.org/exception-clause.html>

**Illustration:**

public class OnExceptionContinueTest extends ContextTestSupport {  
  
 public void testContinued() throws Exception {

// mock:start get one message  
 getMockEndpoint("mock:start").expectedMessageCount(1);

// mock:result still get the original message  
 MockEndpoint mock = getMockEndpoint("mock:result");  
 mock.expectedBodiesReceived("Hello World");  
 // and we should keep the exception so we know what caused the failure  
 mock.message(0).exchangeProperty(Exchange.EXCEPTION\_CAUGHT).isInstanceOf(IllegalArgumentException.class);  
  
 template.sendBody("direct:start", "Hello World");  
  
 assertMockEndpointsSatisfied();  
 }  
  
 @Override  
 protected RouteBuilder createRouteBuilder() throws Exception {  
 return new RouteBuilder() {  
 @Override  
 // START SNIPPET: e1  
 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");  
 }  
 // END SNIPPET: e1  
 };  
 }  
}

C:\Users\gzhoue\Desktop\TempCode\camel\camel-core\src\test\java\org\apache\camel\processor\onexception\OnExceptionContinueTest.java

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:

public class OnExceptionHandleAndTransformTest extends ContextTestSupport {  
  
 public void testOnExceptionTransformConstant() throws Exception {  
 context.addRoutes(new RouteBuilder() {  
 @Override  
 public void configure() throws Exception {  
 errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(0));  
  
 // START SNIPPET: e1  
 // 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");  
 // END SNIPPET: e1  
  
 from("direct:start").process(new Processor() {  
 public void process(Exchange exchange) throws Exception {  
 throw new MyFunctionalException("Sorry you cannot do this");  
 }  
 });  
 }  
 });  
  
 Object out = template.requestBody("direct:start", "Hello World");  
 assertEquals("Sorry", out);  
 }  
  
 public void testOnExceptionTransformExceptionMessage() throws Exception {  
 context.addRoutes(new RouteBuilder() {  
 @Override  
 public void configure() throws Exception {  
 errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(0));  
  
 // START SNIPPET: e2  
 // 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());  
 // END SNIPPET: e2  
  
 from("direct:start").process(new Processor() {  
 public void process(Exchange exchange) throws Exception {  
 throw new MyFunctionalException("Sorry you cannot do this again to me");  
 }  
 });  
 }  
 });  
  
 Object out = template.requestBody("direct:start", "Hello World");  
 assertEquals("Sorry you cannot do this again to me", out);  
 }  
  
 public void testOnExceptionSimpleLangaugeExceptionMessage() throws Exception {  
 context.addRoutes(new RouteBuilder() {  
 @Override  
 public void configure() throws Exception {  
 errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(0));  
  
 // START SNIPPET: e3  
 // 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.");  
 // END SNIPPET: e3  
  
 from("direct:start").process(new Processor() {  
 public void process(Exchange exchange) throws Exception {  
 throw new MyFunctionalException("Out of order");  
 }  
 });  
 }  
 });  
  
 Object out = template.requestBody("direct:start", "Hello World");  
 assertEquals("Error reported: Out of order - cannot process this message.", out);  
 }  
  
}

### Using a Processor as a Failure Handler

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

{snippet:id=e2|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/onexception/OnExceptionProcessorInspectCausedExceptionTest.java}

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.

public class OnExceptionProcessorInspectCausedExceptionTest extends ContextTestSupport {  
  
 public void testInspectExceptionByProcessor() throws Exception {

getMockEndpoint("mock:error").expectedMessageCount(0);

getMockEndpoint("mock:myerror").expectedMessageCount(1);  
  
 try {  
 template.sendBody("direct:start", "Hello World");  
 fail("Should throw exception");  
 } catch (Exception e) {  
 // ok  
 }  
  
 assertMockEndpointsSatisfied();  
 }  
  
 @Override  
 protected RouteBuilder createRouteBuilder() throws Exception {  
 return new RouteBuilder() {  
 @Override  
 public void configure() throws Exception {

// no receive any exception  
 errorHandler(deadLetterChannel("mock:error").maximumRedeliveries(3));  
  
 // START SNIPPET: e1  
 // 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();  
 // END SNIPPET: e1  
  
 from("direct:start").process(new Processor() {  
 public void process(Exchange exchange) throws Exception {  
 throw new MyFunctionalException("Sorry you cannot do this");  
 }  
 });  
 }  
 };  
 }  
  
 // START SNIPPET: e2  
 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);  
 }  
 }  
 // END SNIPPET: e2  
}