INTRODUCTION TO
APACHE CAMEL

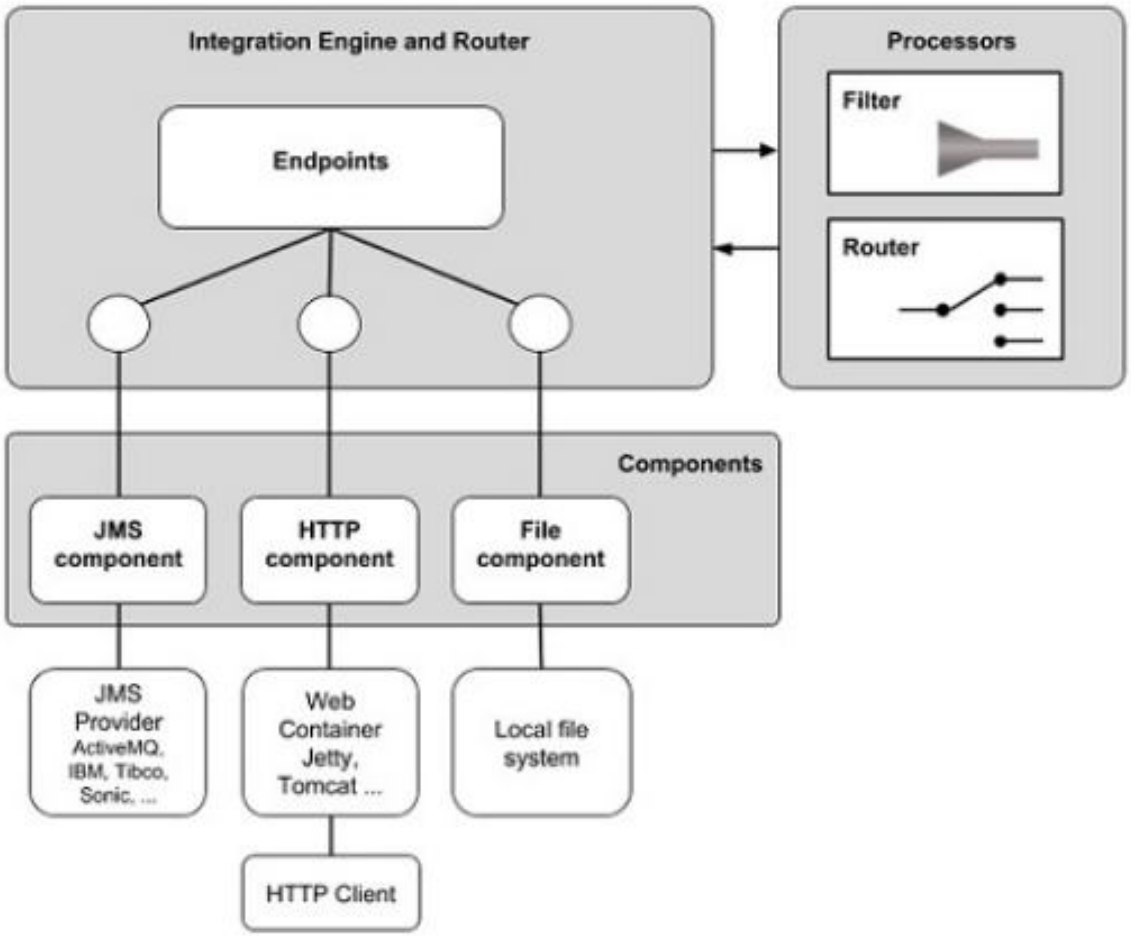
Outline

- What is Camel
- Camel Architecture
- Camel Context
- Components URI
- Domain Specific Language (DSL)
- Camel Routes and Route Builder
- In-Memory Messaging (SEDA queue)
- Camel Predicates and Expression Language
- Camel Processor
- Bean Component, Binding and Integration
- Camel Error Handling
- Concurrency in Camel
- Transaction Management in Camel
- References

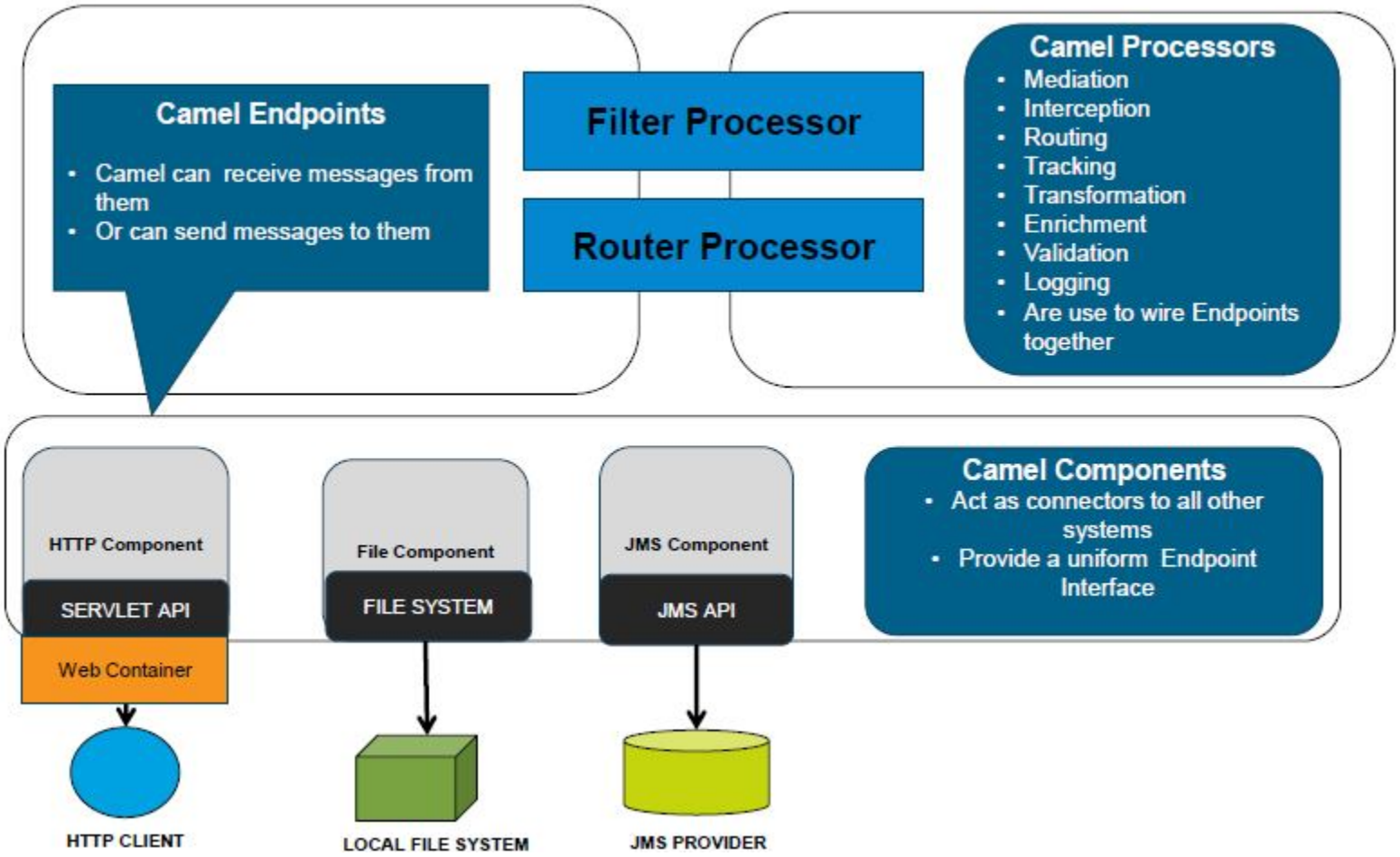
What is Apache Camel?

Apache Camel is a
powerful Open Source
Integration Framework
based on
Enterprise Integration Patterns

Camel Architecture



Camel Architecture contd.

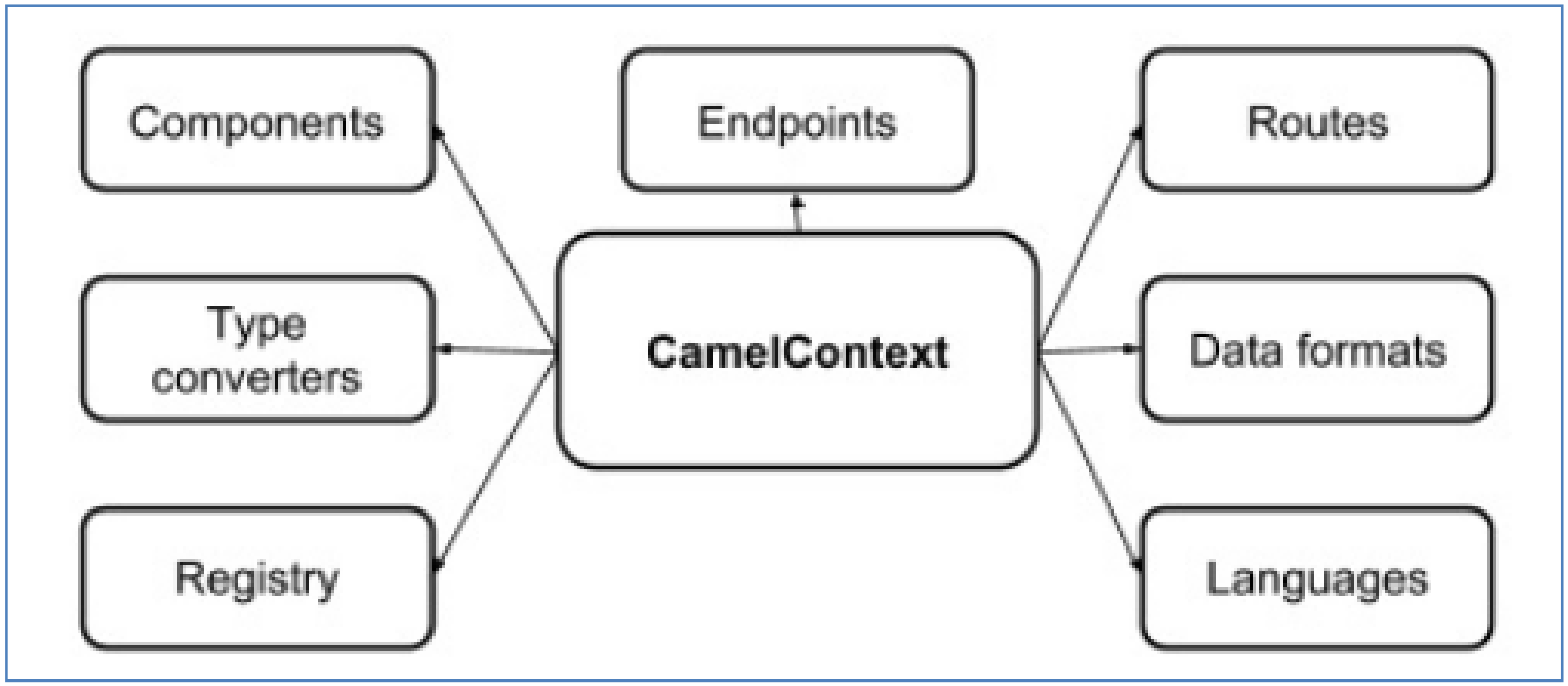


Camel Context

Camel context object represents the camel runtime system. An application typically have one camel context object.

- Camel application executes the following steps –
- Create a **camel context object**
- Add **endpoints and components** to the camel context object.
- Connect the endpoints using routes(Route Builder) and add it to the **camel context object**.
- Invoke **start()** method on the camel context object which in turn starts the camel-internal threads that are used to process the sending, receiving and processing of messages in the endpoints.
- Eventually invoke the **stop()** method on the camel context object which gracefully stops all the endpoints and camel-internal threads. If we neglect to call stop() method before terminating your application then application may terminate in an inconsistent state.
- **Sample camel code** will be shown on how to create routes and add it to the camel runtime(camel context object) in the coming slides after defining components, routes and URIs.

Camel Context



Camel Components and Endpoints

Camel - Component

- A Component is essentially a factory of Endpoint instances.
- We can explicitly configure Component instances and add them to a CamelContext in an IOC container like Spring or they can be auto-discovered using URIs.

Camel - Endpoints

- A Endpoint is usually created by a Component through which a system can produce or consume messages.
- For Example : FILE,SMTP,FTP, JMS etc.
- Endpoints are usually referred in DSL via their URIs.

Components URIs

- Components can be configured by using URIs.
- Camel makes extensive use of URIs to allow you to refer to endpoints which are lazily created by a Component if you refer to them within Routes.
- Components URIs are divided into three parts
    - Schema: Specifies the name of a component
    - A location path : Specific to component
    - Options : Is a list of name value pairs

FOR EXAMPLE

- smtp://admin@mymailserver.com?password=secret
- file:target/reports/?doneFileName=done

Components URIs

- CamelContext object maintains a mapping from component names to Component objects.
- There are two ways of populating the map.
  - Register your component explicitly.
  - Auto-discovery feature of Camel ,where Camel will automatically add a Component when an endpoint URI is used.

Components URIs

- The first way is for application-level code to invoke
  - **For Example:**

```
CamelContext context = new DefaultCamelContext();
context.addComponent("foo", new
FooComponent(context));
```

- The second way to populate the map of named Component objects in the CamelContext object is to let the CamelContext object perform lazy initialization. Developers when they write a class(com.example.myproject.FooComponent) that implements component interface and you want Camel to automatically recognize this by the name "foo" then
  - **For Example:**

```
To do this you need to create a file called foo in
META-INF/services/org/apache/camel/component/
which contains
class=com.example.myproject.FooComponent
```

Domain Specific Language (DSL)

- Domain-specific languages (DSLs) are computer languages that target a specific problem domain, rather than a general purpose domain like most programming languages.
- Camel uses a Java Domain Specific Language or DSL for creating Enterprise Integration Patterns or Routes in a variety of domain-specific languages .
- Some of the DSL are :
  - Java DSL - A Java based DSL using the fluent builder style.
  - Spring XML - A XML based DSL in Spring XML files
  - Blueprint XML - A XML based DSL in OSGi Blueprint XML files
  - Rest DSL - A DSL to define REST services using a REST style in either Java or XML.
  - Groovy DSL - A Groovy based DSL using Groovy programming language
  - Scala DSL - A Scala based DSL using Scala programming language
  - Annotation DSL - Use annotations in Java beans.
- Out of these Spring xml and Java are the most commonly used DSLs for camel

Java DSL

- It is a set of fluent interfaces that contain methods named after terms from the EIP
- The Java DSL is available by extending the RouteBuilder class, and implement the configure method.

```
RouteBuilder builder = new RouteBuilder() {
public void configure() {
from("queue:a").filter(header("foo").isEqualTo("bar")).to("queue:b");
from("queue:c").choice()
.when(header("foo").isEqualTo("bar")).to("queue:d")
.when(header("foo").isEqualTo("cheese")).to("queue:e")
.otherwise().to("queue:f");
}
};
CamelContext myCamelContext = new DefaultCamelContext();
myCamelContext.addRoutes(builder);
```

- The from() method tells Camel to consume messages from an endpoint, and the to() method instructs Camel to send messages to another endpoint.

CREATE ROUTES AND ADD TO CAMEL CONTEXT USING DSL

In the **previous slide example**, an object is created which is an instance of anonymous subclass of **RouteBuilder** with the overridden method configure().

CamelContext.addRoutes(RouteBuilder builder) method invokes builder.setContext(this) so the RouteBuilder object knows which camelcontext object it is associated with and then invokes builder.configure().

The **RouteBuilder.from(String Uri)** method invokes **getEndpoint(uri)** on camelcontext associated with the RouteBuilder object to get the specified Endpoint and then puts a FromBuilder "wrapper" around this Endpoint.

The **FromBuilder.filter(Predicate predicate)** method creates a **FilterProcessor** object for the Predicate (that is, condition) object built from the header("foo").isEqualTo("bar") expression.

In this way, these operations incrementally build up a Route object (with a RouteBuilder wrapper around it) and add it to the CamelContext object associated with the RouteBuilder.

Spring DSL

- You can configure a CamelContext inside any spring.xml using the CamelContextFactoryBean. This will automatically start the CamelContext along with any referenced Routes along any referenced Component and Endpoint instances.
- We require to add the following schemas in our xml file.

```
http://activemq.apache.org/camel/schema/spring/camel-spring.xsd
http://activemq.apache.org/camel/schema/spring
```

- We need to add the following schema location declaration

```
http://camel.apache.org/schema/spring/camel-spring.xsd
```

- The bean XML should look like below:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans.xsd
      http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd">
```

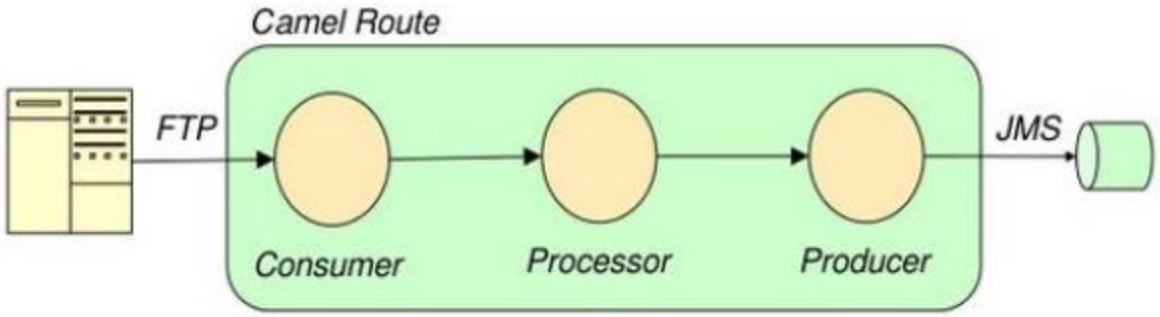
Camel Route

- A Route is the step by step moment of message:
  - From a listening endpoint in the role of consumer
  - Through a processing component(Optional)
  - To a target endpoint in the role of producer
- May involve any number of processing components that modified the original message and redirect it.
- An Application Developer specific route using:
  - Configuration file
  - Java Domain specific language (DSL)

Simple Route

Typical Scenario: Read the xml file from an FTP server and process it using Transformations and then send it to a JMS queue

- FTP consumer that consume the message
- Then sent to a processor for transformation
- Then send to JMS Queue Producer



Route Builder

- To create route you need to extend **RouteBuilder** class and override and implement **configure()** method.

```
public class MyRouteBuilder extends RouteBuilder {
        public void configure() throws Exception {
        From("ftp://demo@localhost/ftp?password==pass")
                .to(xslt:MyTransform.xslt)
            .to("activemq:queue:MyQueue");
                }
        }
```

In-Memory Messaging

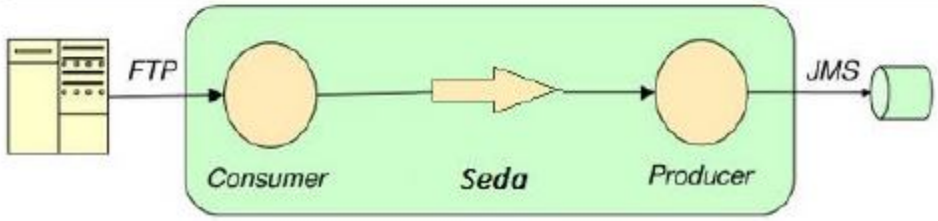
Camel provides three main components in the core to handle in-memory messaging.

- **Direct component**
  - Direct component is for synchronous messaging
- **SEDA component**
  - SEDA component is For asynchronous messaging
  - SEDA component can be used for communication within a single CamelContext
- **VM component**
  - VM component For asynchronous messaging
  - VM component is a bit broader and can be used for communication within a JVM. If you have two CamelContexts loaded into an application server.

SEDA

- The SEDA: Component provides asynchronous SEDA behavior, so that messages are exchanged on a BlockingQueue and consumers are invoked in a separate thread from the producer.
- One of the most common uses for SEDA queues in Camel is to connect routes together form a routing application.

```
public class MyRouteBuilder extends RouteBuilder {
        public void configure() throws Exception {
            From("ftp://demo@localhost/ftp?password==pass")
           . to("seda:MySeda");
            From("seda:MySeda")
           .to("activemq:queue:MyQueue");
        }
}
```



Message Routing - CBR

Content Based Router - CBR can be thought as a service routing mechanism that determines a service route by analyzing the message content at runtime. To do the conditional routing required by CBR camel introduces few key words in DSL such as Choice and When

- **from("jms:incomingOrders").choice().when(predicate).to("jms:xmlOrders")**
  .when(predicate).to("jms:csvOrders");
- Predicate used in above route is just like a If condition in Java, predicates are often built from expressions and expressions are used to extract the result from exchange based on the expression content.
- So, to check whether the filename extension is equal to .xml, you can use the following predicate: **header("CamelFileName").endsWith(".xml")**

CBR ROUTING EXAMPLE USING CHOICE AND WHEN

Consider the below code snippet –

<choice> is either one or the other, so in the below example you can only reach one destination <When> clause

```
context.addRoutes(new RouteBuilder() {Public void configure() {
from("file:src/data?noop=true").to("jms:incomingTradeOrders");
from("jms:incomingTradeOrders").choice().when(header("CamelFileName").endsWith(".xml")).to("jms:xmlTradeOrders").when(header("CamelFileName").endsWith(".csv")).to("jms:csvTradeOrders");
from("jms:xmlTradeOrders").process(new Processor() {
public void process(Exchange exchange) throws Exception {
System.out.println("Received XML order: "
+ exchange.getIn().getHeader("CamelFileName"));
}});
from("jms:csvTradeOrders").process(new Processor() {
public void process(Exchange exchange) throws Exception {
System.out.println("Received CSV order: "
+ exchange.getIn().getHeader("CamelFileName"));}};
```

USING THE OTHERWISE CLAUSE

Previous route only handles .csv and .xml files and will drop all trade orders that have a different extension.

This does not look like a comprehensive solution , One way to handle the extra extension is to use a **regular expression** as a predicate instead of the simple **endsWith** call.

The following route can handle the extra file extension:

```
from("jms:incomingTradeOrders")
.choice()
.when(header("CamelFileName").endsWith(".xml"))
.to("jms:xmlTradeOrders")
.when(header("CamelFileName").regex("^.*(csv|csl)$"))
.to("jms:csvTradeOrders");
```

The above solution still has the problem, any trade orders file extension not compatible with file extension scheme will be dropped, we should be handling bad trade orders in more efficient way, for this you can use the **otherwise** Clause, **code snippet** shown in next slide

USING THE OTHERWISE CLAUSE – CODE SNIPPET

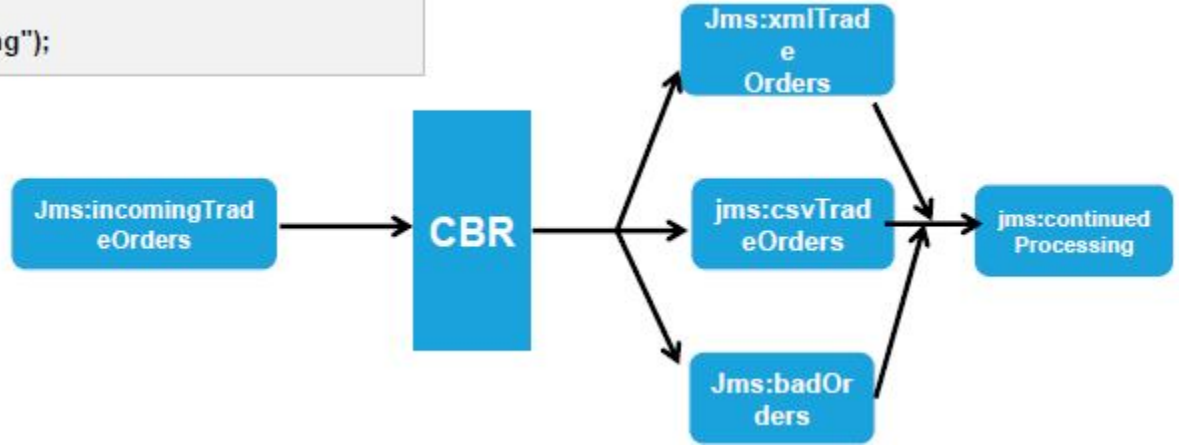
```
from("jms:incomingTradeOrders")
.choice()
.when(header("CamelFileName").endsWith(".xml"))
.to("jms:xmlTradeOrders")
.when(header("CamelFileName").regex("^.*(csv|csl)$")).to("jms:csvOrders")
.otherwise()
.to("jms:badTradeOrders");
```

All the files that do not have .xml or .csv or .csl extension will go into the badTradeOrders queue.

ROUTING AFTER A CBR

The CBR may seem like it's the end of the route; messages are routed to one of several destinations, and that's it. Continuing the flow means you need another route.

```
from("jms:incomingTradeOrders")
.choice()
.when(header("CamelFileName").endsWith(".xml"))
.to("jms:xmlTradeOrders")
.when(header("CamelFileName").regex("^.*(csv|csl)$"))
.to("jms:csvTradeOrders")
.otherwise()
.to("jms:badOrders")
.end()
.to("jms:continuedProcessing");
```



MESSAGE ROUTING - MULTICAST

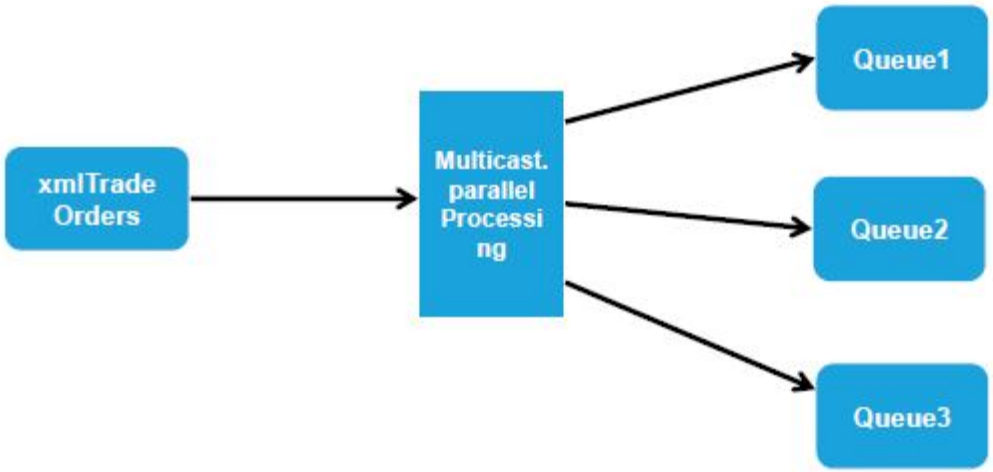
The Multicast EIP allows concurrency when sending a copy of the same message to multiple recipients. By default, the multicast sends message copies **sequentially**.

```
from("jms:xmlTradeOrders").multicast().to("jms:queue1", "jms:queue2", "jms:queue3");
```

Sending messages in parallel using the multicast involves only one extra DSL method: parallelProcessing ()

```
from("jms:xmlTradeOrders").multicast().parallelProcessing() .to("jms:queue1", "jms:queue2", "jms:queue3");
```

The above code will instruct multicast to distribute the messages to the destination queues in parallel. By default multicast will continue to send messages to destination even if one among them fails. Consider a scenario where you need to consider the whole process is failed if one of the destination fails, code sample in the next slide.



STOPPING THE MULTICAST ON EXCEPTION

Consider a scenario where you need to consider the whole process is failed if one of the destination fails

stopOnException() will stop the multicast on the first exception caught and necessary action can be taken.

```
from("jms:xmlTradeOrders").multicast().stopOnException()
.parallelProcessing().executorService(executor)
.to("jms:Queue", "jms:queue2");
```

When using the Spring DSL, this route looks a little bit different:

```
<route>
<from uri="jms:xmlTradeOrders"/>
<multicast stopOnException="true" parallelProcessing="true"
➥ executorServiceRef="executor">
<to uri="jms:queue1"/>
<to uri="jms:queue2"/>
</multicast>
</route>
```

To handle the exception coming back from the above route different error handling techniques have be discussed in the coming slides.

Splitter EIP

Splitter EIP is Used to split a message into pieces that are routed separately .

The below example show cases the usage of the splitter in camel. It splits one message into 3 messages

```
public void testSplitABC() throws Exception { MockEndpoint mock = getMockEndpoint("mock:split");
mock.expectedBodiesReceived("A", "B", "C"); List<String> body = new ArrayList<String>();
body.add("A"); body.add("B"); body.add("C"); template.sendBody("direct:start", body);
assertMockEndpointsSatisfied();
}

protected RouteBuilder createRouteBuilder() throws Exception {
return new RouteBuilder() {
public void configure() throws Exception {
from("direct:start")
.split(body())
.log("Split line ${body}")
.to("mock:split");
} }; } }
```

USING BEANS FOR SPLITTING

Consider the below code snippet -

```
public class EmployeeService {
public List<Department> splitDepartments(Employee employee) {
return employee.getDepartments();
} }
```

Assume, splitDepartments method returns a List of Department objects, which is what you want to split by.

In the Java DSL, you can use the EmployeeService bean for splitting by telling Camel to invoke the splitDepartments method. This is done by using the method call expression as shown in bold:

```
public void configure() throws Exception {
from("direct:start")
.split().method(EmployeeService.class, "splitDepartments")
.to("log:split")
.to("mock:split");
}
```

SPLITTER - STOP PROCESSING IN CASE OF EXCEPTION

Splitter will continue to process the remaining messages in a exchange in case it hits an exception and this is default behavior.

For instance if you have an Exchange with 100 rows that you split and route each sub message. During processing of these sub messages an exception is thrown at the 10th. What Camel does by default is to process the remainder 90 messages.
But sometimes you just want Camel to stop and let the exception be propagated back, and let the Camel error handler handle it. You can do this in Camel 2.1 by specifying that it should stop in case of an exception occurred. This is done by the **stopOnException** option as shown below:

Code Sample :

```
from("direct:start")
    .split(body().tokenize(",")).stopOnException()
        .process(new MyProcessor())
        .to("mock:split");
```

SPLITTER – SHARE UNIT OF WORK

By default splitter don`t share the unit of work between parent exchange and sub exchanges and each sub exchanges will have its own individual unit of work

USE CASE: Your application receives large files which need to be split and processed and when an exception occurs on any of the split sub exchanges you don`t want individual messages to be sent to the dead letter queue and want camel to move the original parent message to be sent to the queue. In this scenario we can use shareUnitOfwork() method to achieve the desired output.

```
errorHandler(deadLetterChannel("mock:dead").useOriginalMessage().
maximumRedeliveries(5).redeliveryDelay(1000));
from("direct:start").to("mock:abc") .
split(body().tokenize(",")).shareUnitOfWork().to("mock:bcd").
to("direct:line").end().
to("mock:result");
```

Even though for each sub message error handler gets invoked it will not be moved to deadletterqueue after all the redeliveries failed to send the message successfully as individual messages because of the shareUnitOfWork() method used in the above route. It will set the exception at the parent exchange level and mark it as rollback and hence no more redeliveries are performed on the exchange and it will moved into the dead letter queue.

CAMEL PREDICATES

- Camel supports a pluggable interface called Predicate which can be used to integrate a dynamic predicate into Enterprise Integration Patterns such as when using
    - **Message Filter :** Allows you to filter messages
    - **Content Based Router** : Allows you to route messages to the correct destination based on the contents of the message exchanges.
- A Predicate is being evaluated to a Boolean value so the result is either true or false.
    - This makes Predicate so powerful as it is often used to control the routing of message in which path they should be routed.

TYPES OF CAMEL PREDICATES

- **Negating a Predicate** – We use the not method on the PredicateBuilder to negate a predicate by importing the not static to make our route easy to read.

```
import static org.apache.camel.builder.PredicateBuilder.not
from("direct:start") .choice()
        .when(not(header("username").regex("goofy|pluto"))).to("mock:people")
        .otherwise().to("mock:animals")
        .end();
```

- **Compound Predicates** – We can also create compound predicates using boolean operators such as and, or, not and many others.
  - Using the PredicateBuilder class, we can combine predicates from different Expression Languages based on logical operators and comparison operators as

```
Predicate admin = and(user, header("admin").isEqualTo("true"));
```

  - Expresses that the admin user must be a User AND have admin header as true.
- **Extensible Predicates** - Camel supports extensible Predicates using multiple languages such as Bean Language, Json path, JX path, Spring Expression language, SQL and Scripting languages

EXPRESSION LANGUAGE

- Expression language evaluates an expression at runtime on the current instance of Exchange that is under processing which makes it easy to define predicates in routes, such as those needed when using the Content-Based Router.
- It is easy with expression language to access information from the Exchange message by using the built-in variables.
- This can be done by using Message Translator EIP, which leverages an expression to transform a message which expresses that we want to prepend 'Hello' to the message body.

```
from("direct:hey").transform(simple("Hello ${body}"));
simple("${body.address}")
```

LANGUAGES WHICH ARE SUPPORTED IN CAMEL

- Bean Language
- JSon Path
- JX Path
- Exchange Property

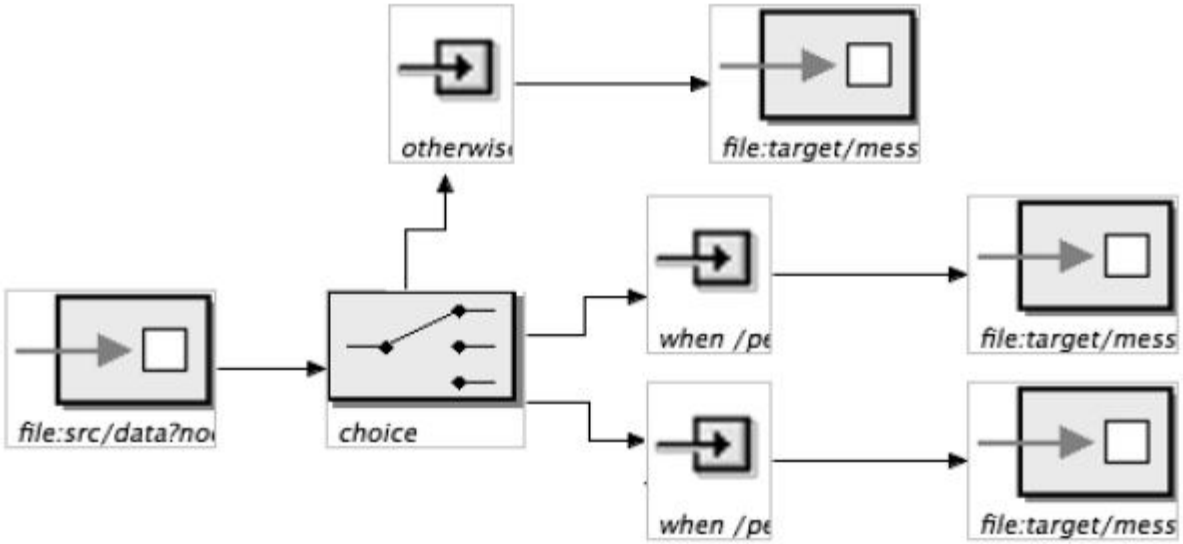
- Spring Expression Language
- SQL
- Xpath

- Scripting languages
  - Bean Shell
  - Java Script
  - Python
  - Ruby

USE OF PREDICATES AND EXPRESSION LANGUAGE

- Content Based Router uses 'When' And 'Otherwise' to route a request from an input endpoint to other endpoints depending on evaluation of expressions.

```
from("file:src/data") .choice()
                .when().simple("${body} contains 'CamelDemo'")).to("file:target/message/camel")
                 .when().simple("${body} contains 'SpringDemo'")).to("file:target/message /spring")
                .otherwise().to(" file:target/message/ other");
```



CAMEL PROCESSOR

- The processor is a core Camel concept that represents a node capable of using, creating or modifying an incoming exchange.
- During routing, exchanges flow from one processor to another; as such, you can think of a route as a graph having specialized processors as the nodes, and lines that connect the output of one processor to the input of another.
- Many of the processors are implementations of EIPs, but one could easily implement their own custom processor and insert it into a route.
- The Processor is a low-level API where you work directly on the Camel Exchange instance.
- Configuring a processor:

```
public class CustomProcessor implements Processor {
public void process(Exchange exchange) throws Exception {
        String custom = exchange.getIn()
        .getBody(String.class);
        }
}
```

CAMEL PROCESSOR(EXAMPLE)

- Once the processor has been configured It can be easily used like this in a route.

```
from("activemq:myQueue").processRef("customprocessor");
```

- "CustomProcessor" is nothing but the processor class which implements the Processor interface.
- You will override the process method in your processor class which will take the exchange as your parameter.

```
public class CustomProcessor implements Processor {
public void process(Exchange exchange) throws Exception {
String custom = exchange.getIn()
.getBody(String.class);
```

- Above you can get the message from the exchange and transform it according to the requirements.

BEAN COMPONENT

- This component is used to bind messages of camel to bean components
- The format to configure bean using camel is :

```
bean:beanID[?options]
```

- Here bean id is the configured Java class to which the Camel message is binded.
- In the options you can provide the method name which should be invoked if this bean component is called
- Bean component configuration for Java DSL is more simpler than others

```
from("direct:start").beanRef("beanName", "methodName");
```

- The above example directly takes the bean and the method name which it invokes.

BEAN BINDING

- This defines which methods are invoked while the bean component is called
- The following are the ways in which camel recognizes as to which method must be invoked:
- If the message contains the header CamelBeanMethodName then that method is invoked, converting the body to the type of the method's argument.
- One can explicitly specify the method name in the DSL

```
from("direct:start").beanRef("beanName", "methodName");
```

- If the bean has a method marked with the @Handler annotation, then that method is selected

```
@Handler
  public String process(String body) { }
```

- For the method to be invoked camel will bind the IN body to the body parameter and convert it to a String.

```
public String process(String body)
```

BEAN INTEGRATION

Camel supports the integration of beans and POJOs in a number of ways. The following are some of the ways in which java beans can be integrated:

- **@Consume :** to mark a particular method of a bean as being a consumer method. The uri of the annotation defines the Camel Endpoint to consume from

```
@Consume(uri="activemq:consumer")
 public void onCheese(String name) {}
```

- **@Produce**: Similarly to mark a method as a producer

```
@Produce(uri = "activemq:foo")
 protected MyListener producer;
```

- **@EndpointInject or @BeanInject**: These annotations are used to inject a particular bean from a spring xml file. These can be used to achieve DI from spring. Here foo bean would be instantiated in spring xml file.

```
@BeanInject("foo")
  MyFooBean foo;
```

DEFAULT ERROR HANDLER

- This is the default error handler which is automatically enabled when no other handler has been configured.
- The default error handler doesn't need to be explicitly declared in the route.
- In every Camel route, there is a Channel that sits between each node in the route graph
- This is the feature that allows Camel to enrich the route with error handling, message tracing, interceptors, and much more.
- In this channel the default Error handler is configured

- This is same as the Dead Letter Channel, however it does not support a dead letter queue, which is the only difference between the two of them.
- The default error handler is configured with these settings:
  - No redelivery
  - Exceptions are propagated back to the caller

DEFAULT ERROR HANDLER(EXAMPLE)

- The below code snippet is an example

```
from("bean:consumer").to("bean:validateOrder").to("jms:queue:order");
```

- We have configured a consumer bean which would be validated and sent to a JMS queue order

```
onException(ValidationException.class).handled(true).transform(body(constant("INV
ALID ORDER")));
```

- We have added an onException to catch exceptions and route them differently, for instance to catch a ValidationException and return a fixed response to the caller.
- When the ValidationException is thrown from the validateOrder bean, it is intercepted by the DefaultErrorHandler that lets the onException(ValidationException.class) handle it.
- The Exchange is routed to this onException route, and since we use handled(true) the original exception is cleared and we transform the message into a fixed response that is returned to consumer endpoint.
- Hence the response is channeled back to its caller which in this case is consumer bean

DEAD LETTER CHANNEL

- The DeadLetterChannel error handler is similar to the default error handler except for the following differences:
  - The dead letter channel is the only error handler that supports moving failed messages to a dedicated error queue, which is 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 supports using the original input message when a message is moved to the dead letter queue.
  - The Dead Letter Channel lets you control behaviors including redelivery, whether to propagate the thrown Exception to the caller (the **handled** option), and where the (failed) Exchange should now be routed to.
- The DeadLetterChannel is an error handler that implements the principles of the Dead Letter Channel EIP. This pattern states that if a message can't be processed or delivered, it should be moved to a dead letter queue
- This pattern is often used with messaging. Instead of allowing a failed message to block new messages from being picked up, the message is moved to a dead letter queue to get it out of the way.
- Configuring Dead Letter channel using fluent builder

```
errorHandler(deadLetterChannel("jms:queue:dead")
    .maximumRedeliveries(3).redeliveryDelay(5000));
```

DEAD LETTER CHANNEL(EXAMPLE)

- Take the following route for example

```
from("bean:consumer").to("bean:validateOrder").to("jms:queue:order");
```

- We will set the error handler as dead letter channel

```
errorHandler(deadLetterChannel("jms:queue:dead")
  .useOriginalMessage().maximumRedeliveries(5).redeliverDelay(5000);
```

- If any error message is caused then the message is passed to the jms:queue:dead.
- The route listen for consumer bean messages , validates, transforms and handles it. During this the Exchange payload/the consumer message is transformed/modified. So in case something goes wrong and we want to move the message to another destination the message might be transformed
- Hence we use useOriginalMessage option which saves the original input message which we received from our consumer bean.
- maximumRedeliveries sets the property of number of time the message is redelivered and redeliverdelay is the time it waits for the next delivery

TRANSACTION ERROR HANDLER

- This is a transaction-aware error which uses the same base as the Default Error Handler
- By default any exception thrown during routing will be propagated back to the caller and the exchange ends immediately.
- Take the following route for example

```
from("bean:consumer").transacted().to("bean:validateOrder").to("jms:queue:order“);
```

- Setting the error handler as Transaction Error Handler

```
onException(ValidationException.class).handled(true).transform(body(constant("INVALID ORDER")));
from("jms:queue:foo").transacted().to("bean:handleFoo");
```

- When the ValidationException is thrown from the validate order bean it is intercepted by the transaction error handler and it let the onException(ValidationException.class) handle it so the exchange is routed to this route and since we use handled(true) then the original exception is cleared and we transform the message into a fixed response that are returned to jms queue endpoint that returns it to the original caller.

LOGGING ERROR HANDLER

- The LoggingErrorHandler logs the failed message along with the exception.
- The logger uses standard log format from log kits such as log4j, commons logging, or the Java Util Logger.
- Camel will, by default, log the failed message and the exception using the log name org.apache.camel.processor.LoggingErrorHandler at ERROR level. One can also customize this
- Configuring Logging error handler

```
errorHandler(loggingErrorHandler("mylogger.name").level(LoggingLevel.INFO));
```

- The logging category, logger and level may all be defined in the builder.

NOERROR HANDLING

- The no error handler is used for disabling error handling.
- The current architecture of Camel mandates that an error handler must be configured, so if one wants to disable error handling one needs to provide an error handler that's basically an empty shell with no real logic. The NoErrorHandler is just that.
- Configuring a noErrorHandler

```
errorHandler(noErrorHandler());
```

- Or we can do in Spring DSL as follows,

```
<bean id="noErrorHandler" class="org.apache.camel.builder.NoErrorHandlerBuilder"/>
 <camelContext errorHandlerRef="noErrorHandler" xmlns="http://camel.apache.org/schema/spring">
</camelContext>
```

ERROR HANDLER SCOPES

Camel supports 2 types of scopes: context scope and route scope.

Context Scope :

```
errorHandler(defaultErrorHandler() .maximumRedeliveries(4) .redeliveryDelay(2000)
.retryAttemptedLogLevel(LoggingLevel.WARN));
from("file://target/tradeOrders?delay=20000") .beanRef("tradeService", "toCsv")
.to("mock:file") .to("seda:queue.inbox");
```

Route Scope :

```
from("seda:queue.inbox").errorHandler(deadLetterChannel("log:DLC")
.maximumRedeliveries(6).retryAttemptedLogLevel(LoggingLevel.INFO)
.redeliveryDelay350).backOffMultiplier(3)) .beanRef("tradeService", "validate")
.beanRef("tradeService", "enrich")
.to("mock:queue.tradeorder");
```

HANDLING FAULTS

By default the Camel error handlers will only react to exceptions. Because a fault isn't represented as an exception but as a message that has the fault flag enabled, faults will not be recognized and handled by Camel error handlers.

There may be times when you want the Camel error handlers handle faults as well. Suppose a Camel route invokes a remote web service that returns a fault message, and you want this fault message to be treated like an exception and moved to a dead letter queue.

```
errorHandler(deadLetterChannel("mock:dead"));
from("seda:queue.inbox")
.beanRef("orderService", "toSoap")
.to("mock:queue.order");
```

Now if the orderService bean returns SOAP fault – Under normal scenarios Camel error handler wont react, to make it so you need to instruct the camel to do so

getContext().setHandleFault(true); // this will enable the fault handling on the camel context object, the scope is context scope.

```
If you want to enable at route basis then -
from("seda:queue.inbox").handleFault() .beanRef("orderService", "toSoap")
.to("mock:queue.order"); // route basis handling faults
```

Once handling faults is enabled, camel error handlers will recognize the soap faults and react

Under the hood the soap fault is converted into exception with the help of an interceptor.

USING ONREDELIVER: ONEXCEPTION()

USE CASE: The purpose of onRedeliver is to allow some code to be executed before a **redelivery** is **performed**. This gives you the power to do custom processing on the Exchange before Camel makes a redelivery attempt. You can, for instance, use it to add custom headers to indicate to the receiver that this is a redelivery attempt. OnRedeliver uses

an org.apache.camel.Processor, in which you implement the code to be executed

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

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

OnRedeliver is also scoped, so if an onRedeliver is set on an onException, it **overrules** any onRedeliver set on the error handler.

In Spring DSL, onRedeliver is configured as a reference to a spring bean, as follows:

```
<onException onRedeliveryRef="myOtherRedelivery">
<exception>java.io.IOException</exception>
</onException>
<bean id="myOtherRedelivery"
class="com.mycompany.MyOtherOnRedeliveryProceossor"/>
```

CONCURRENCY USING CAMEL

Consider a scenario where we need to split messages in a large file

The below route picks up the files and then splits the file content line by line . This is done by using the Splitter EIP in streaming mode. The streaming mode ensures that the entire file isn't loaded into memory; instead it's loaded piece by piece on demand, which ensures low memory usage. Now suppose you're testing the application by letting it process a big file with 10,000 lines. If each line takes a tenth of a second to process, processing the file would take 1000 seconds which is approximately 16.7 minutes then it may end up in a situation where it cannot process all the files in a given time frame

```
public void configure() throws Exception {
from("file:test/trade")
.log("Starting to process file: ${header.CamelFileName}") //prints the file name in the log
.split(body().tokenize("\n")).streaming() // splits the message with new line as delimiter
.bean(TradeService.class, "csvToObject") // bean class with method csvToObject() to convert csv format to pojo.
.to("direct:update")
.end()
.log("Done processing file: ${header.CamelFileName}");
```

CONCURRENCY USING CAMEL *CONTD.*

How do we make splitting of the files faster and in acceptable time frame ? The obvious choice would be using multithreading if the order of the messages is not important for the application.

USING PARALLELPROCESSING

The Splitter EIP offers an option to switch on parallel processing, as shown here:

```
.split(body().tokenize("\n")).streaming().parallelProcessing()
.bean(InventoryService.class, "csvToObject")
.to("direct:update")
.end()
```

When parallel processing is enabled splitter EIP uses a thread pool to process messages concurrently. The default configuration of thread pool is to use 10 threads to execute the tasks. Which means effectively the processing rate increases 10 times faster approximately.

MANAGING TRANSACTIONS IN CAMEL

- Local Transactions(single resource)
- Global Transactions(Multiple resource)

How to make camel use transactions for a particular route –

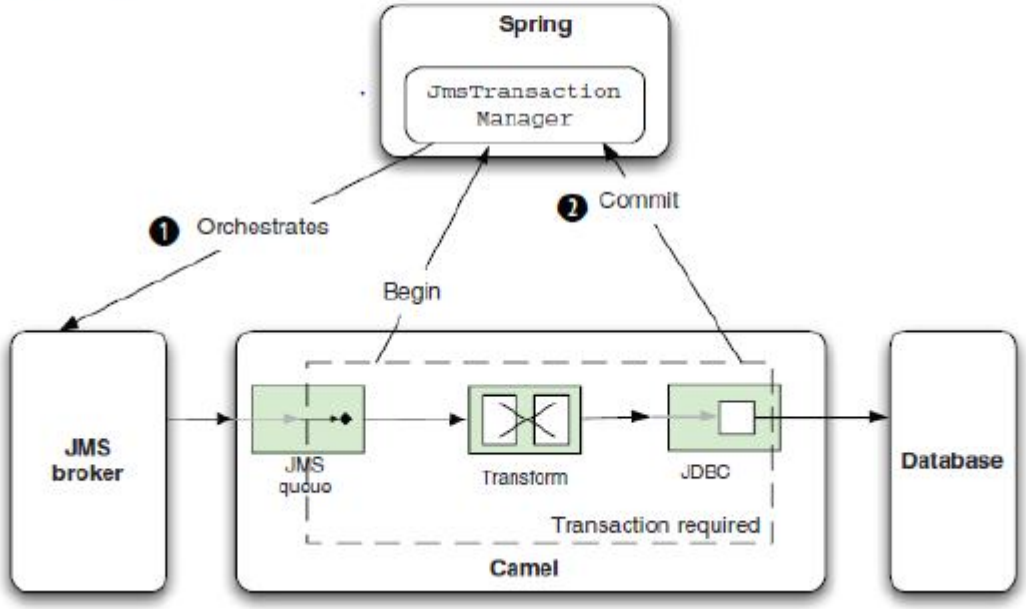
```
from("activemq:queue:trade")
.transacted()
.beanRef("trade", "toSql")
.to("jdbc:myDataSource");
```

Under the hood camel hooks up the Spring transaction manager and leverages it.

The convention over configuration only applies when you have a single transaction manager configured. In more complex scenarios with multiple transaction managers we need to add additional configuration to set up transactions

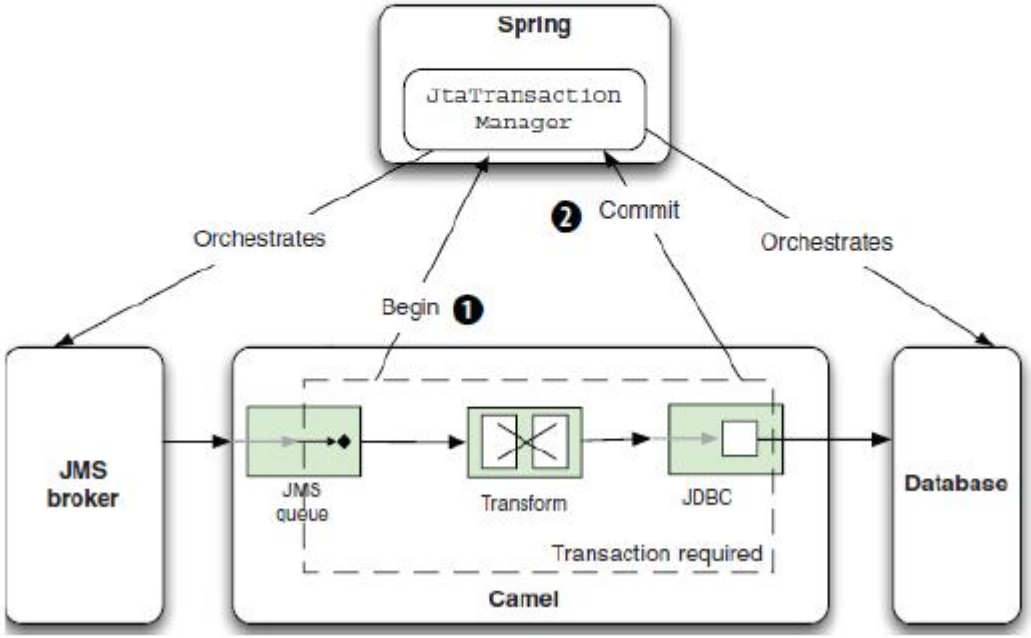
LOCAL TRANSACTIONS

Below picture depicts using of the single resource(JMS Queue) and JMS broker.JmsTransactionManager orchestrates the transaction with single participating resource. If the database decides to rollback the transaction it will throw an exception that camel TransactionErrorHandler propagates back to the JMSTransactionManager which takes appropriate action. This isn`t exactly equal to enrolling the DB in the transaction, and it can still have failure scenarios that could leave the system in an inconsistent state.



GLOBAL TRANSACTIONS

In this figure, we've switched to using the JtaTransactionManager, which handles multiple resources. Camel consumes a message from the queue, and a begin is issued. The message is processed, updating the database, and it completes successfully. JTA is an implementation of the XA standard protocol, which is a global transaction protocol. To be able to leverage XA, the resource drivers must be XA-compliant, which some JDBC and most JMS drivers are.



XA CAPABLE DRIVERS FOR RESOURCES USED IN GLOBAL TRANSACTIONS

When using JTA we need to use **XA capable drivers**, which means we need to use **ActiveMQ-XAConnectionFactory** to let it participate in the global transaction-

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

Similarly for JDBC driver-

```
<bean id="myDataSource"
class="com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean"> <property
name="uniqueResourceName" value="hsqldb"/> <property name="driverClassName"
value="org.hsqldb.jdbcDriver"/><property name="url" value="jdbc:hsqldb:mem:hs"/>
<property name="user" value="hs"/> <property name="password" value=""/>
<property name="poolSize" value="3"/>
</bean>
```

JTA TRANSACTION MANAGER

Having configured the XA drivers, you also need to use the Spring JtaTransactionManager. It should refer to the real XA transaction manager, which is Atomikos in this example

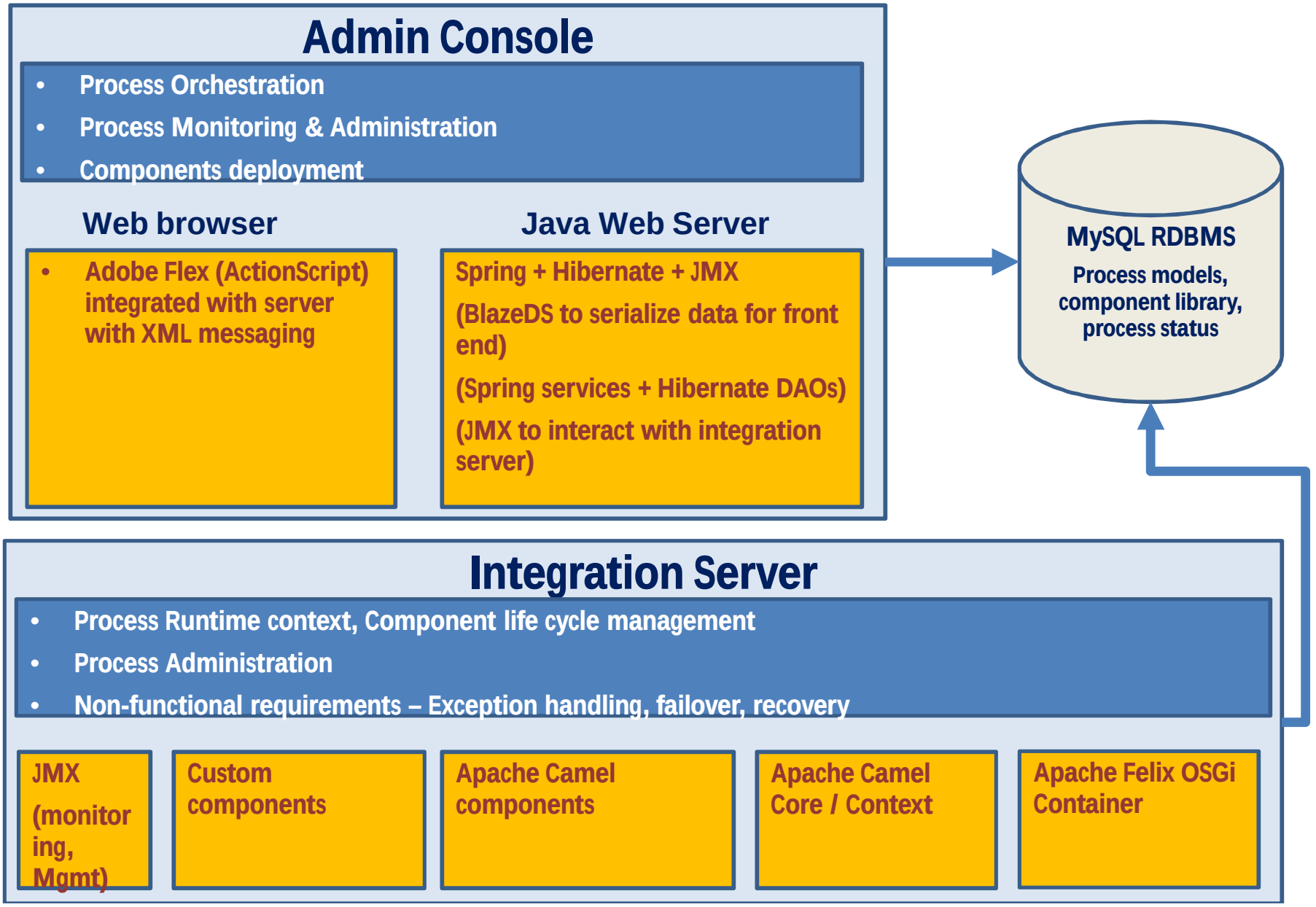
```
<bean id="jtaTransactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
<property name="transactionManager"
ref="atomikosTransactionManager"/>
<property name="userTransaction" ref="atomikosUserTransaction"/>
</bean>
```

References

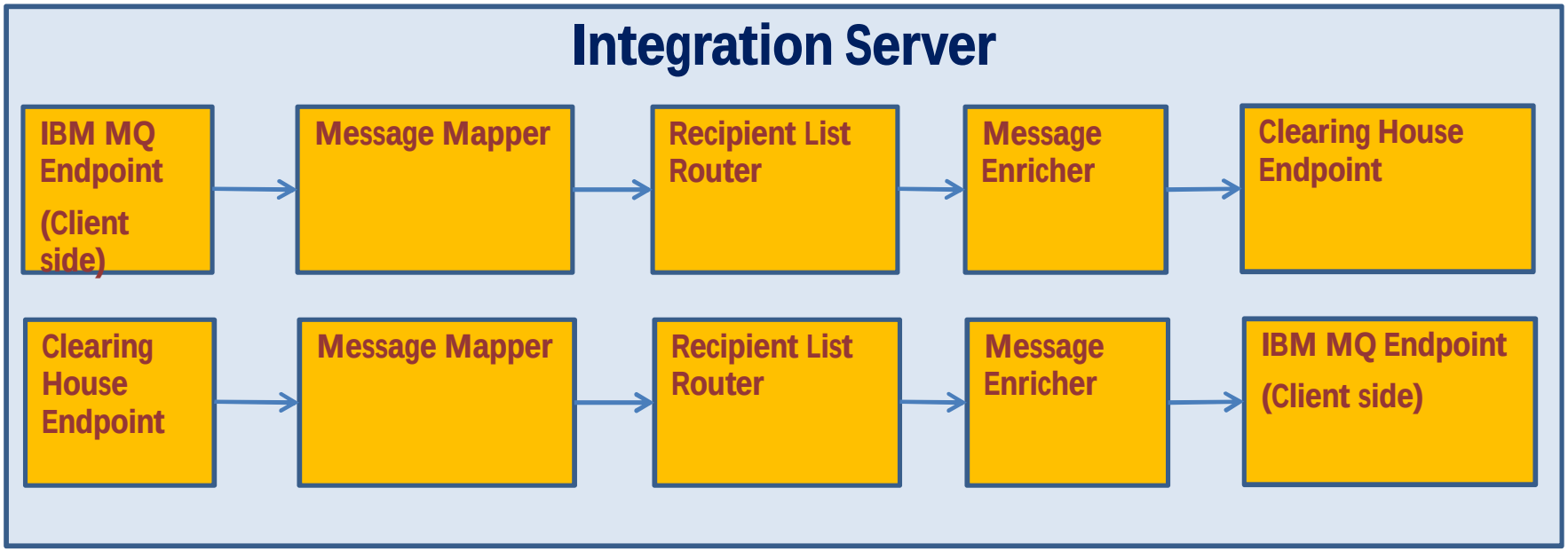
- http://camel.apache.org

Appendix

CaseStudy: TEEVRA - EAI Platform with domain specific out-of-the-box components



CaseStudy: EAI Platform with domain specific out-of-the-box components Message flows



Above pair is repeated for connectivity to each clearing house

Camel Usecase – Consuming Difference Message Formats

1. Consuming different message formats and converting into a standard format(Teevra Format).

Teevra provides different types of message consuming endpoints such as File endpoint, DB endpoint FTP end point, JMS endpoint, TCP endpoint, Seda endpoint by leveraging the camel endpoints.

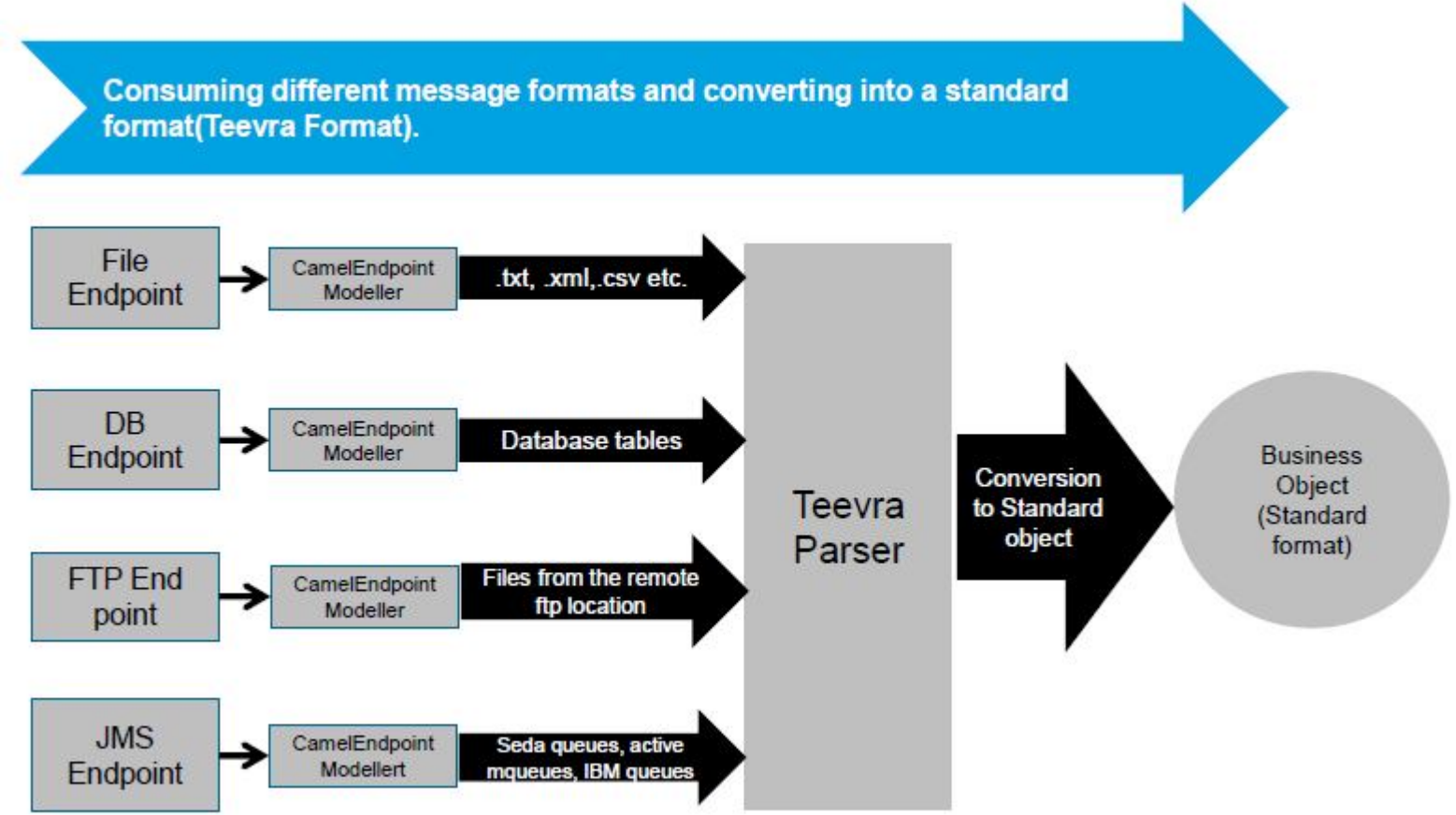
- Sample code to instantiate the camel FTP endpoint -

Code Sample :

```
FtpEndpoint<FTPFile> endPoint = new FtpEndpoint("ftp", component, config);

// will set the consumer properties specific to Teevra ftp endpoint provided in UI.

// create the route using Route Builder
routeBuilder.from(endPoint).convertBodyTo(String.class);

//Assign the route Definition to ProcessorDefiniton

ProcessorDefiniton processorType =  // routeDefinition
```

- // set the parameters in the header if required
- //add the error handler to the route to handle exceptions
- // Once the message passes the endpoint, Teevra have custom parsers written to convert into standard POJO.
- Picture in the next slide depicting the same

Camel endpoint modeler utilized camel component to create endpoint



Camel Usecase – Static Throttling

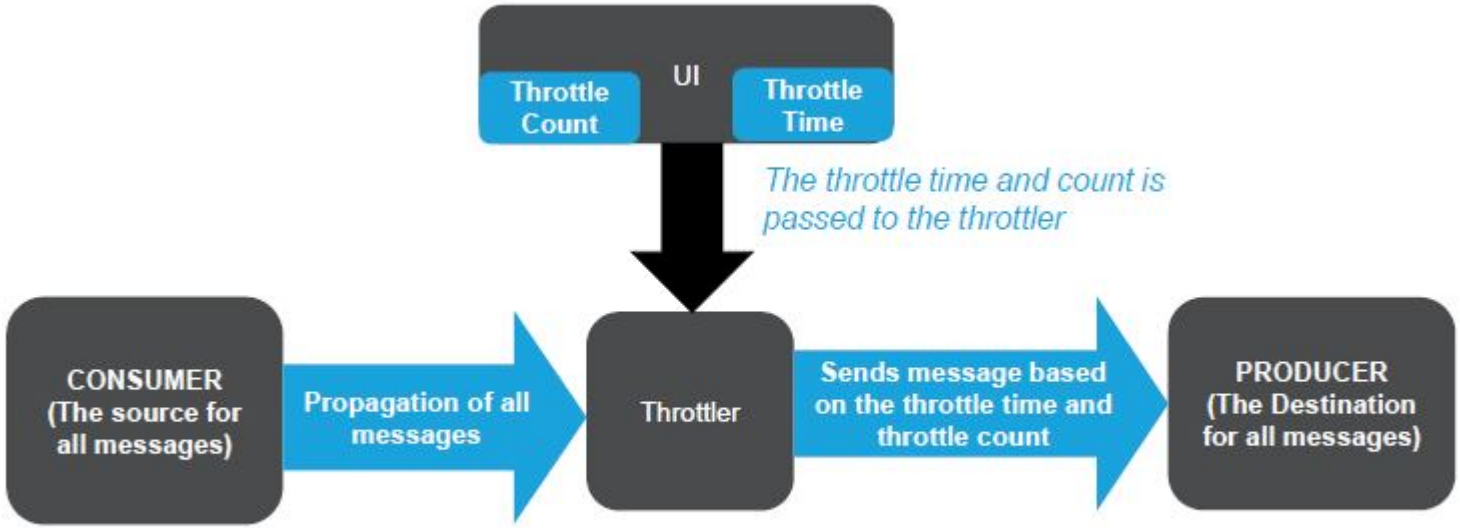
2. Implementation of static throttling inside Teevra

- Consider a business scenario where business wants Teevra to process only particular number of messages in a given time frame for example -
- Business user wants Teevra processing rate fixed as 5 msgs/sec.
- // For example throttleCount=5; throttleTime=1000 ms;

Code Sample :

```
// create ProcessorDefinition reference
ProcessorDefinition newProcessorType = null;
// Get the processorType from camel seda flow "from" route
processorType = // get the processorType from "From route"
// create the "To route " using the throttle method and timePeriodMillis
newProcessorType = processorType.throttle(throttleCount)
.timePeriodMillis(throttleTime).to(sedaEndpoint);
```

Static Throttling in Teevra



Camel Usecase - SEDA

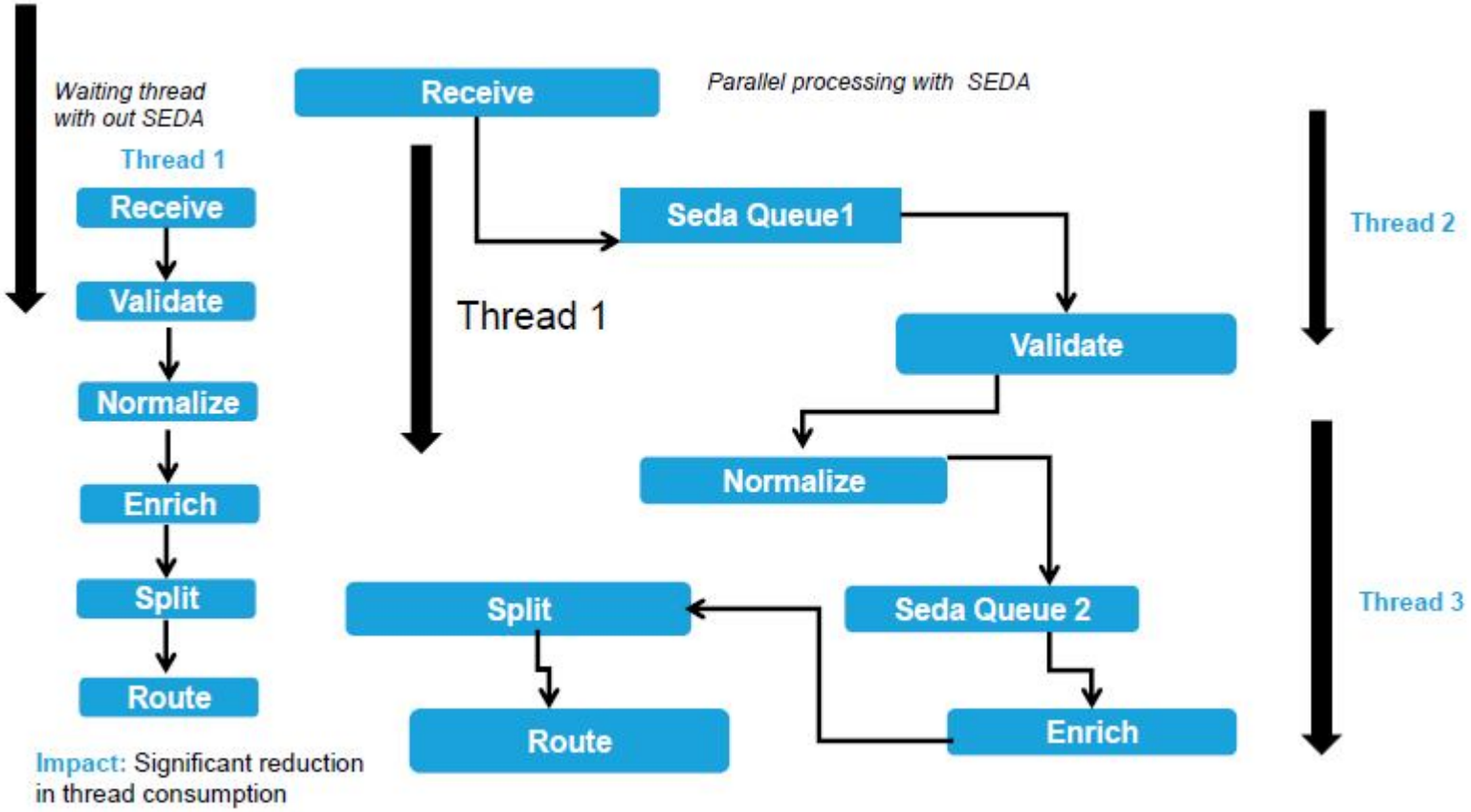
3. USING SEDA to achieve parallel processing/concurrent consumers inside Teevra

- Setting concurrent consumers for an endpoint in Teevra

Code Sample :

```
// Create an endpoint modeler instance  -
BindingEndPointModeler bindingEndPointModeler = (BindingEndPointModeler) endPointModeler;

//Get the number of concurrent threads from user through GUI
threadCount = // from some properties map mapped to GUI field

((SedaEndpoint) sedaEndpoint) .setConcurrentConsumers(threadCount);
// From the above SedaEndpoint set it to the "toURI" and assign to ProcessorDefinition reference
newProcessorType = bindingEndPointModeler.modelTo(this
.getRouteBuilder(), this.getRouteBuilder().from(
sedaEndpoint));

// add the error route to handle any exception thrown on this route
this.addConnectorErrorRoute(newProcessorType, ....);
```

USING SEDA TO ACHIEVE PARALLEL PROCESSING/ CONCURRENT CONSUMERS



Camel Usecase - CBR

4. Creating a Content based Router for Teevra Processing

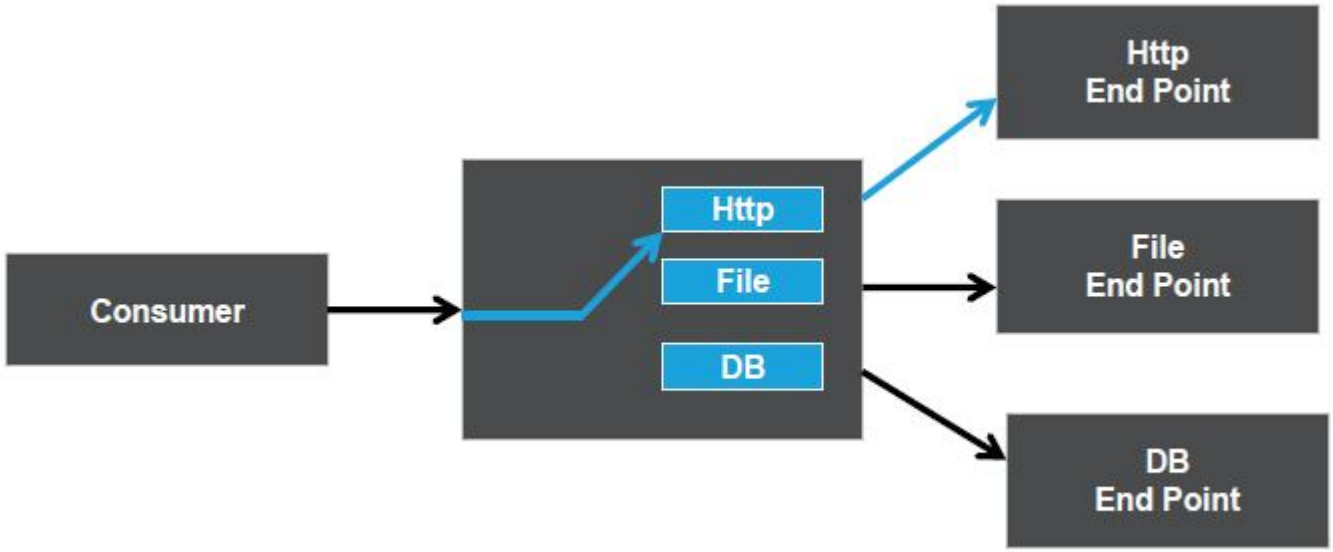
Teevra requires a conditional router which would send requests to some specific components based on some conditions.

This is implemented in Teevra using camel predicates.

Code Sample :

```
ProcessorDefinition processorType= //route definition

//creating a route definition
choiceType = processorType.choice();

//checking for condition
choiceType =choiceType.when().xpath("Some value to check");

//if condition satisfies pass it to endpoint
choiceType.to(endpoint);
```

CONTENT BASED ROUTER



Camel Usecase - Splitter

5. Teevra is expected split the files that are very large in size(more than 1GB) from interface system- Memory constraint.

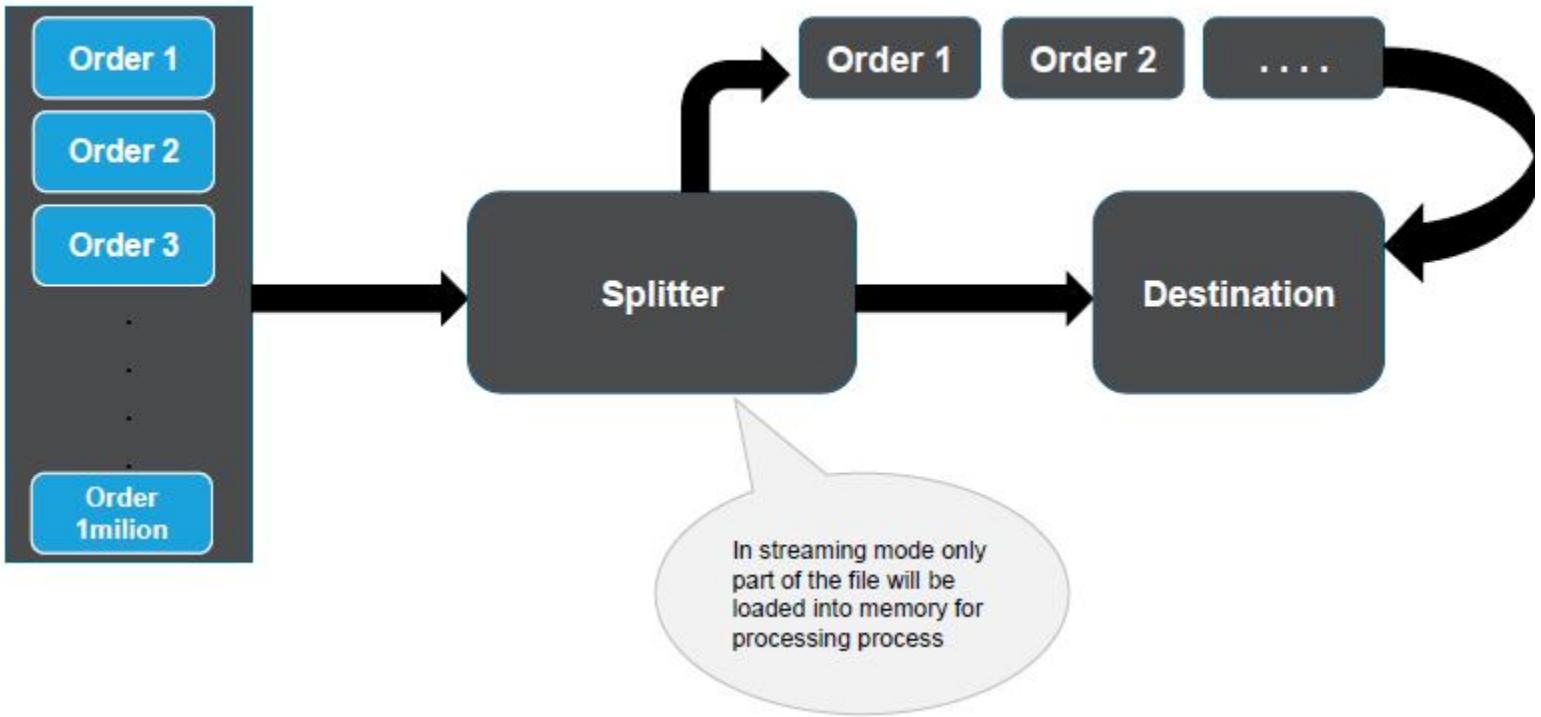
To do this Teevra have implemented **splitter** component in streaming mode, in this mode entire file will not be loaded into memory rather loads only a part of the file into memory while processing the file.

Below is the sample code snippet -

Code Sample :

```
public void configure() throws Exception {
from("file:target/trade")

.log("Starting to process big file: ${header.CamelFileName}")

.split(body().tokenize("\n")).streaming()

.bean(TradeService.class, "csvToObject")

.to("direct:update")

.end()

.log("Done processing big file: ${header.CamelFileName}");

from("direct:update")
.bean(TradeService.class, "updateTrade()");
}
```

SPLIT FILES IN STREAMING MODE



Camel Usecase – Error Handling

6. Error handling implementation in Teevra

When an exception occurs Teevra provides an option for user to configure Error Handler tab in UI according to the business requirement

- Report and abort job
- Report and continue job

Report and abort job

Code Sample :

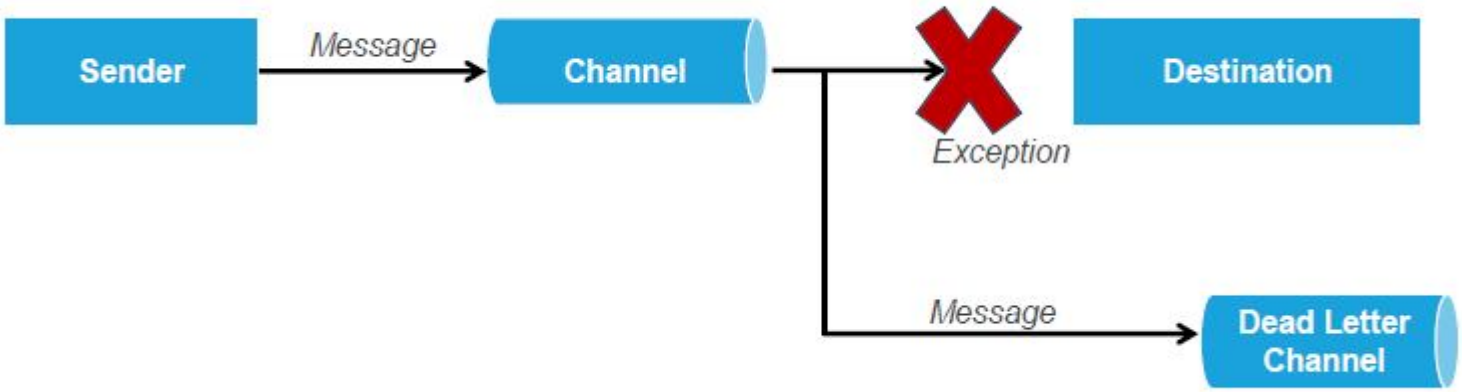
```
this.getRouteBuilder().onException(HandledException.class).handled(true).stop();
```

Report and continue job

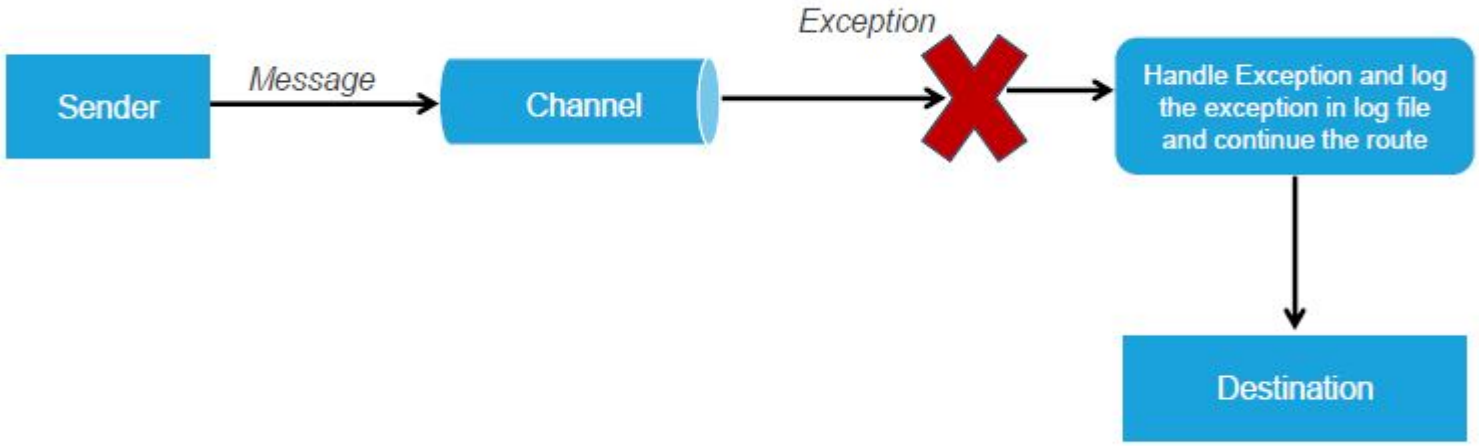
Code Sample :

```
ProcessorDefinition processorType = this.getRouteBuilder().onException(
Throwable.class).handled(true).process(
new ErrorExchangeProcessor()); // Error Exchange processor sample code shown below
public class ErrorExchangeProcessor implements Processor { Throwable error = (Throwable)
exchange.getException();
if ( error == null) {error = exchange.getProperty(Exchange.EXCEPTION_CAUGHT, Throwable.class); }
Throwable t = error;StringBuffer sb = new StringBuffer(); StackTraceElement[] trace = t.getStackTrace();
// assign the error to the String Buffer object and log into the file
logger.error("Error processing the message." + sb); // log the error in log file }
```

REPORT AND ABORT JOB



REPORT AND CONTINUE JOB



Camel Usecase – Advanced Error Handling

More use cases for error handling –

A . Using doTry – doCatch-doFinally

Code Sample :

```
From("direct:start").
doTry().process(new MyProcessor())
.doCatch(Exception.class).to("mock:error").
doFinally().to("mock:end");
```

B. Error Handle with Redeliver option

Code Sample :

```
From("direct:start").
doTry().process(new MyProcessor())
.doCatch(Exception.class).to("mock:error").
doFinally().to("mock:end");
```

- The above error handler will attempt at most three deliveries using a 2 second delay when it attempts to redelivery it will log this as warn level.
- from("seda:queue.inbox").beanRef("orderTradeService", "validate").beanRef("orderTradeService", "enrich") .log("Received Trade order ${body}") .to("mock:queue.order");
- The above example is constructed in such as way that it fails when messages reaches the enrich method
- After 3 redeliveries have completed, error handler will log the message in error mode and exits.

Camel Usecase – Advanced Error Handling contd.

More use cases for error handling –

D. Ignoring Exceptions

- An exception means that Camel will break out of the route. But there are times when all you want is to catch the exception and continue routing. This is possible to do in Camel using **continued**. All you have to do is to use **continued(true)** instead of handled(true).
- Suppose we want to ignore any **ValidationException** which may be thrown in the

Route shown below-

Code Sample :

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

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

from("mina:tcp://0.0.0.0:4444?textline=true") .process(new ValidateTradeOrderId())
.to("jms:queue:order.status")

.process(new GenerateResponse());
```

Camel Usecase – Advanced Error Handling contd.

More use cases for error handling –

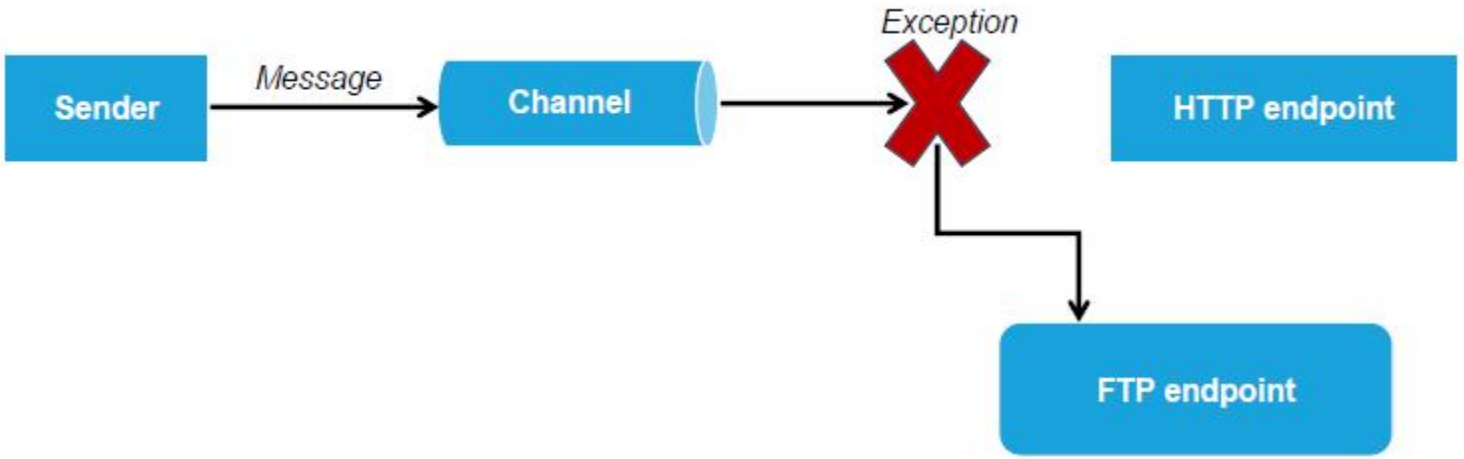
E. Implementing an error handler solution

The onException to the below route tells camel that incase of an IO exception it should try redelivering upto 3 times with 10 second delay and if still the error persists camel should handle the exception and reroute the message to FTP endpoint instead.

Code Sample :

```
errorHandler(defaultErrorHandler().

maximumRedeliveries(5).redeliveryDelay(10000));
onException(IOException.class).

maximumRedeliveries(3).handled(true)

.to("ftp://gear@ftp.rider.com?password=secret"); // onException route rerouting to ftp endpoint incase of an Ioexception

from("file:/rider/files/upload?delay=3600000"). to("http://rider.com?user=gear&password=secret"); // actual route that
move the file from ftp endpoint to http endpoint
```

RE-ROUTING IN CASE OF AN EXCEPTION



Camel Usecase – Advanced Error Handling contd.

F . Using onWhen – onException()

- It would be better if you had more **fine-grained control over onException**, in the previous example we have seen onException rerouting based on a particular exception such as IO, but we may have to handle a scenario where based on description of the exception error handler should route or take action accordingly
- Consider a scenario where in the previous slide example the HTTP service rejects the data and returns an HTTP 500 response with the constant text "ILLEGAL DATA.
- Now, whenever an HttpOperationFailedException is thrown, Camel moves the message to the illegal folder.
- All you need to do is insert the onWhen into the onException, as shown here:

Code Sample :

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

Helper class code whether an HTTP error 500 has occurred-

Code Sample :

```
public final class MyHttpUtil { public static boolean isIllegalDataError(
HttpOperationFailedException cause) {int code = cause.getStatusCode();
if (code != 500) { return false;
}return "ILLEGAL DATA".equals(cause.getResponseBody().toString());}}
```

Camel Usecase – Transactions

Using transactions in camel with multiple routes –

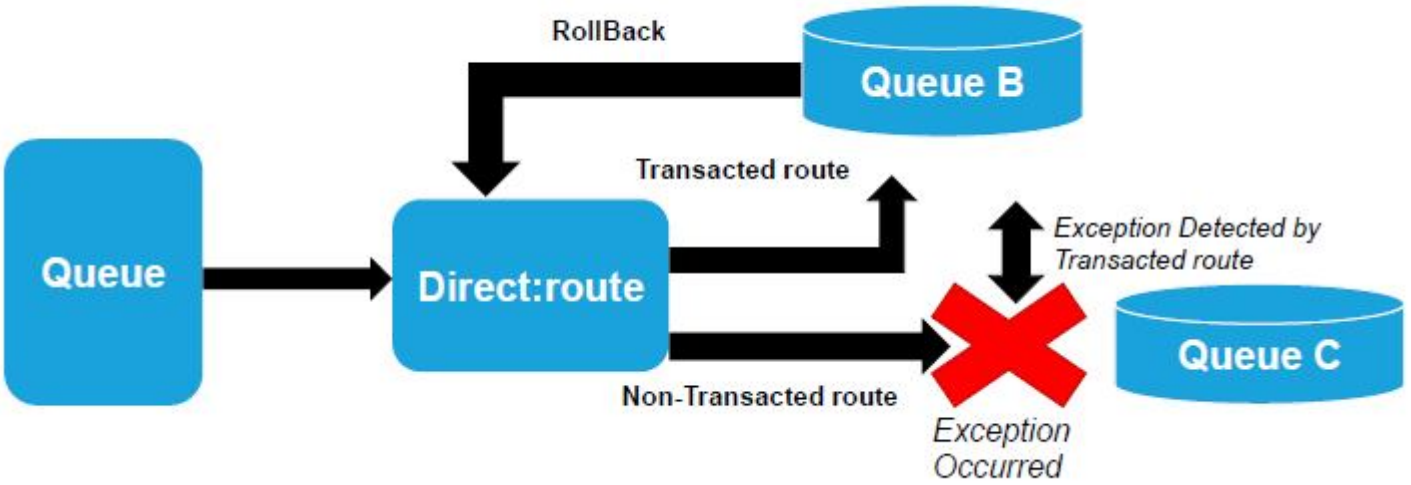
USE CASE: You have 2 routes, transacted and non transacted- scenario where you use non transacted route from a transacted route

Code Sample :

```
protected RouteBuilder createRouteBuilder() throws Exception { return new SpringRouteBuilder() {

public void configure() throws Exception {

from("activemq:queue:a") .transacted() .to("direct:quote") .to("activemq:queue:b"); // transacted route // transacted route

from("direct:quote") .choice() .when(body().contains("options")) .transform(constant("options trade"))

.when(body().contains("futures")) .throwException(new IllegalArgumentException(

"futures not allowed")) .otherwise() .transform(body().prepend("Hello ")); } };
```

In the above scenario if the transacted route fails it **rollbacks** the transaction and it is expected behavior, even when the non-transacted route fails by throwing an exception, the transacted route detects this and issues a rollback.

TRANSACTIONAL ROUTES



RETURNING A CUSTOM RESPONSE WHEN A TRANSACTION FAILS

USE CASE : Camel application exposes a webservice to interface system to execute trade orders and returns code success or failure

The camel application must deal with any thrown exceptions and return a custom failure message instead of propagating the exception back to the caller

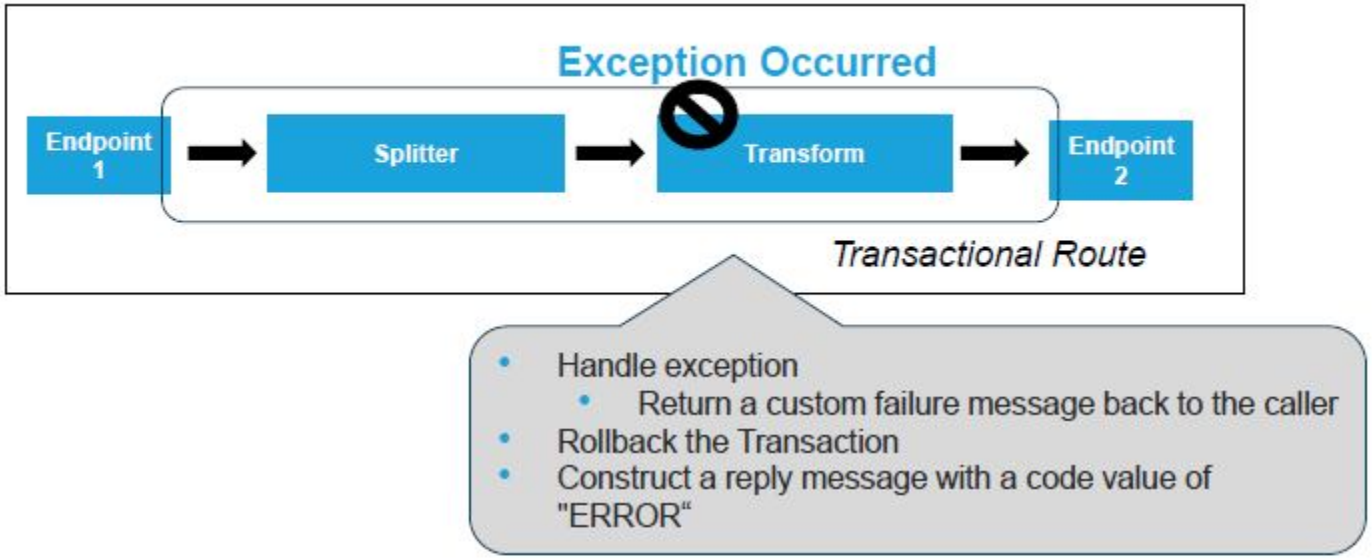
- Catch the exception and handle it to prevent it propagating back
- Mark the transaction to roll back
- Construct a reply message with a code value of "ERROR"

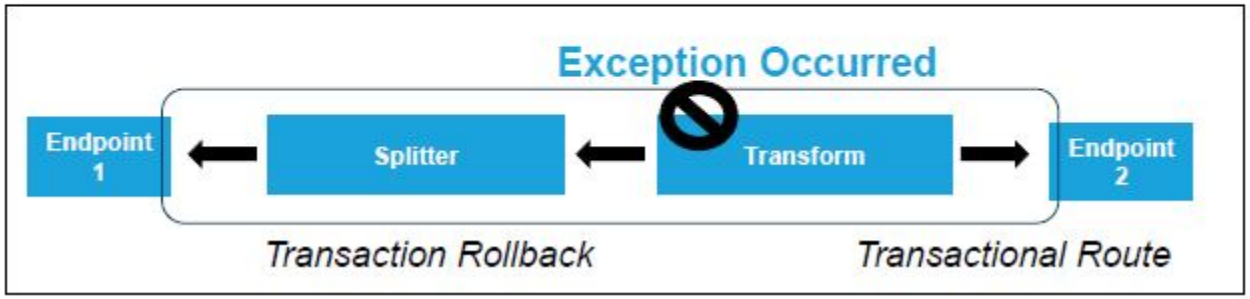
Camel can support such complex use cases because you can leverage onException,

Code Sample :

```
<onException>
<exception>java.lang.Exception</exception>
<handled><constant>true</constant></handled>
<transform><method bean="order" method="replyError"/></transform>
<rollback markRollbackOnly="true"/>
</onException>
```

WHEN A TRANSACTION FAILS





RETURNING A CUSTOM RESPONSE WHEN A TRANSACTION FAILS

Previous slide code snippet, We should tell Camel that this onException should trigger for any kind of exception that's thrown. You then mark the exception as handled, which removes the exception from the exchange, because you want to use a custom reply message instead of the thrown exception. The <rollback/> definition must always be at the end of the onException because it stops the message from being further routed. That means you must have prepared the reply message before you issue the <rollback/>. To construct the reply message, you use the order bean, invoking its replyError

Code snippet :

Code Sample :

```
public OutputOrder replyError(Exception cause) {
OutputOrder error = new OutputOrder();
error.setCode("ERROR: " + cause.getMessage());
return error;
}
```

Thank You!