Enterprise Integration Patterns with Camel

Iulian Velea

LUXOFT

LXFT LISTED NYSE

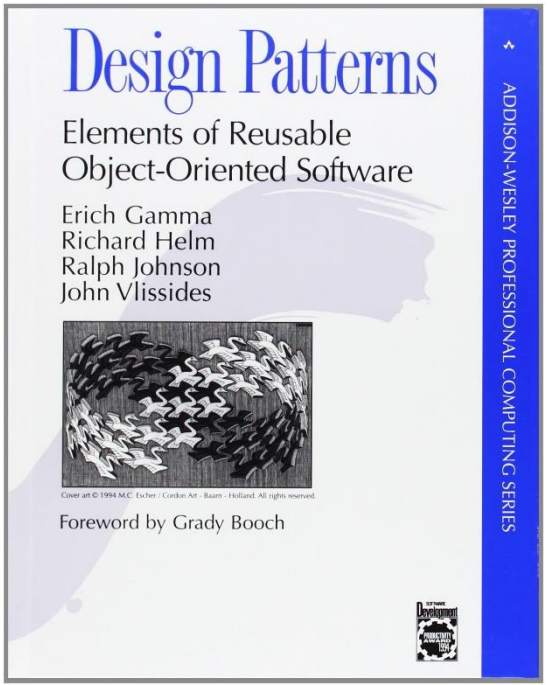
Presentation

▶ Who am I?

- Technical Leader Luxoft
- 12 years of Java
- 2+ years of Apache Camel

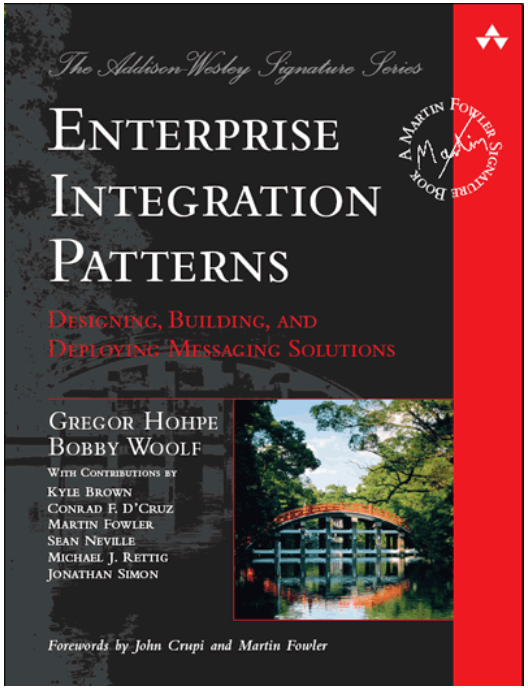
What EIPs?

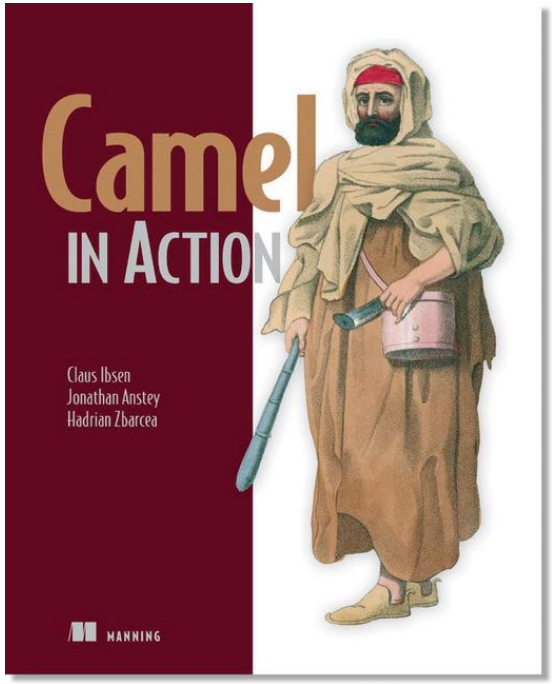
- **Patterns…**



What EIPs?

- Oh no! More patterns...





Goals

▶ Checkpoints

- Introduction to Apache Camel – Message model
- Architecture overview - CamelContext, Routes, Domain Specific Language, Processors, Components
- EIPs examples – **CBR, Filter, Multicast, WireTap, Deadletter channel, Translator, Content enricher, Aggregator, Splitter** etc.
- Routing with Camel
- Transforming data
- Invoking beans
- Error handling

Introduction. Architecture

LUXOFT

LXFT LISTED NYSE

Introduction

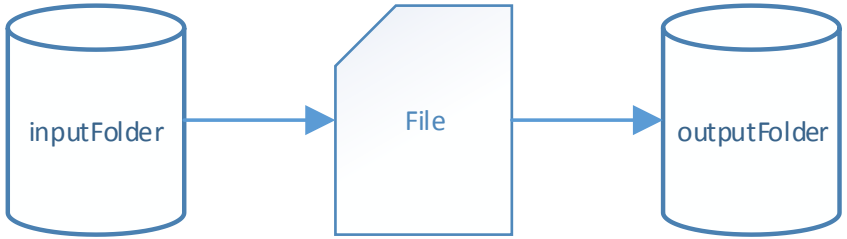
▶ What is Apache Camel?

- **Integration Framework**
- Routing engine builder
- Payload agnostic
- EIPs; DSL
- Comes with a large number of components, multiple languages, data formats, type converters
- Not an Enterprise Service Bus
- Apache Camel website: http://camel.apache.org/download.html

Introduction

▶ Getting started

- First example in system integration: routing files



Introduction

▶ Routing files comparison

```
public class RoutingFiles {

  public static void main(String[] args)
         throws Exception {
    File inputFolder = new File("data/inputFolder");
    File outputFolder = new File("data/outputFolder");

    outputFolder.mkdir();

    File[] files = inputFolder.listFiles();
    for (File source : files) {
      if (source.isFile()) {
        File dest = new File(outputFolder.getPath()
                  + File.separator + source.getName());
        copyFile(source, dest);
      }
    }
  }
```

```
  private static void copyFile(File source, File dest)
          throws IOException {
    OutputStream out = new FileOutputStream(dest);
    byte[] buffer = new byte[(int) source.length()];
    FileInputStream in = new FileInputStream(source);
    in.read(buffer);
    try {
      out.write(buffer);
    } finally {
      out.close();
      in.close();
    }
  }
}
```

Introduction

▶ Routing files comparison

```
public class RoutingFilesWithCamel {
    public static void main(String[] args) throws Exception {
        // create CamelContext
        CamelContext context = new DefaultCamelContext();

        // add our route to the CamelContext
        context.addRoutes(new RouteBuilder() {
                public void configure() {

         from("file:data/inputFolder?noop=true")
                 .to("file:data/outputFolder");

                }
            });

        // start the route
        context.start();
        Thread.sleep(10000);

        // stop the CamelContext
        context.stop();
    }
}
```

```
/pom.xml
  <dependencies>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-core</artifactId>
            <version>${camel-version}</version>
        </dependency>
  </dependencies>
```

Message model

Abstractions

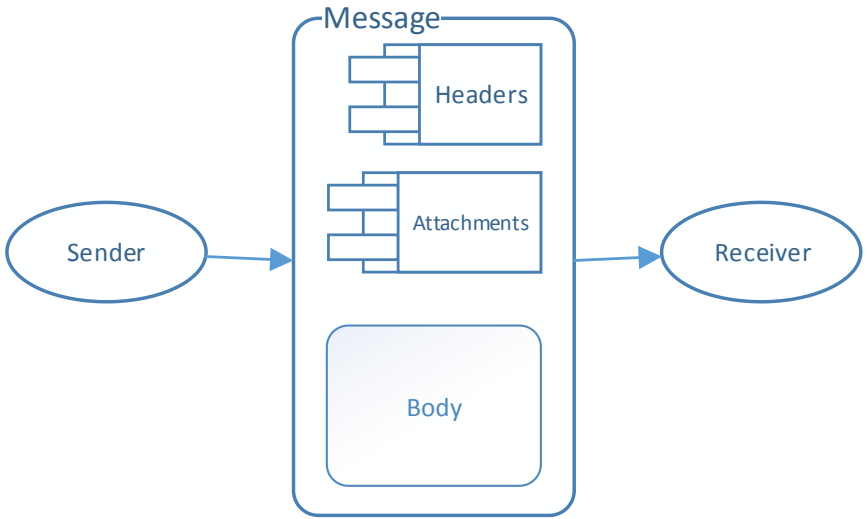
- **`org.apache.camel.Message`** – data being routed
- **`org.apache.camel.Exchange`** – exchange of message, in/out

Message model

▶ Message

- **Message EIP**
  - Used for inter-system communication
  - Message parts: body (payload); headers; [attachments]
  - Unique identifier `<java.lang.String>`
  - Headers

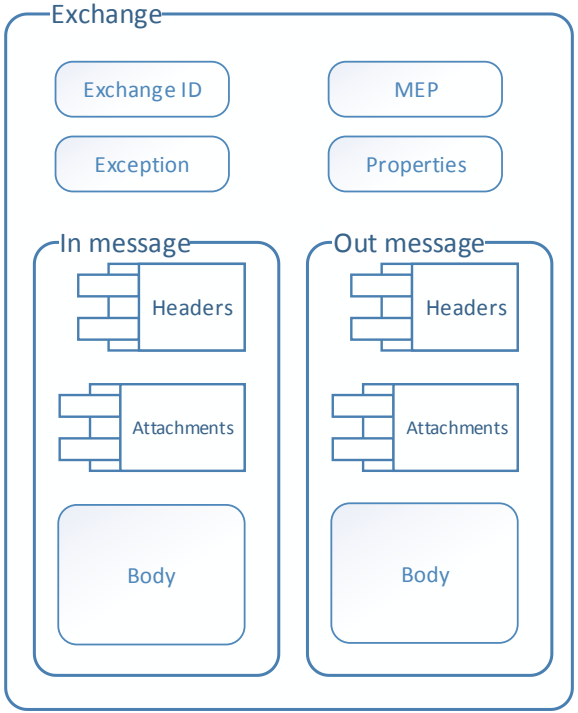
    `[name<String>, value<Object>]`
  - Body `<java.lang.Object>`



Message model

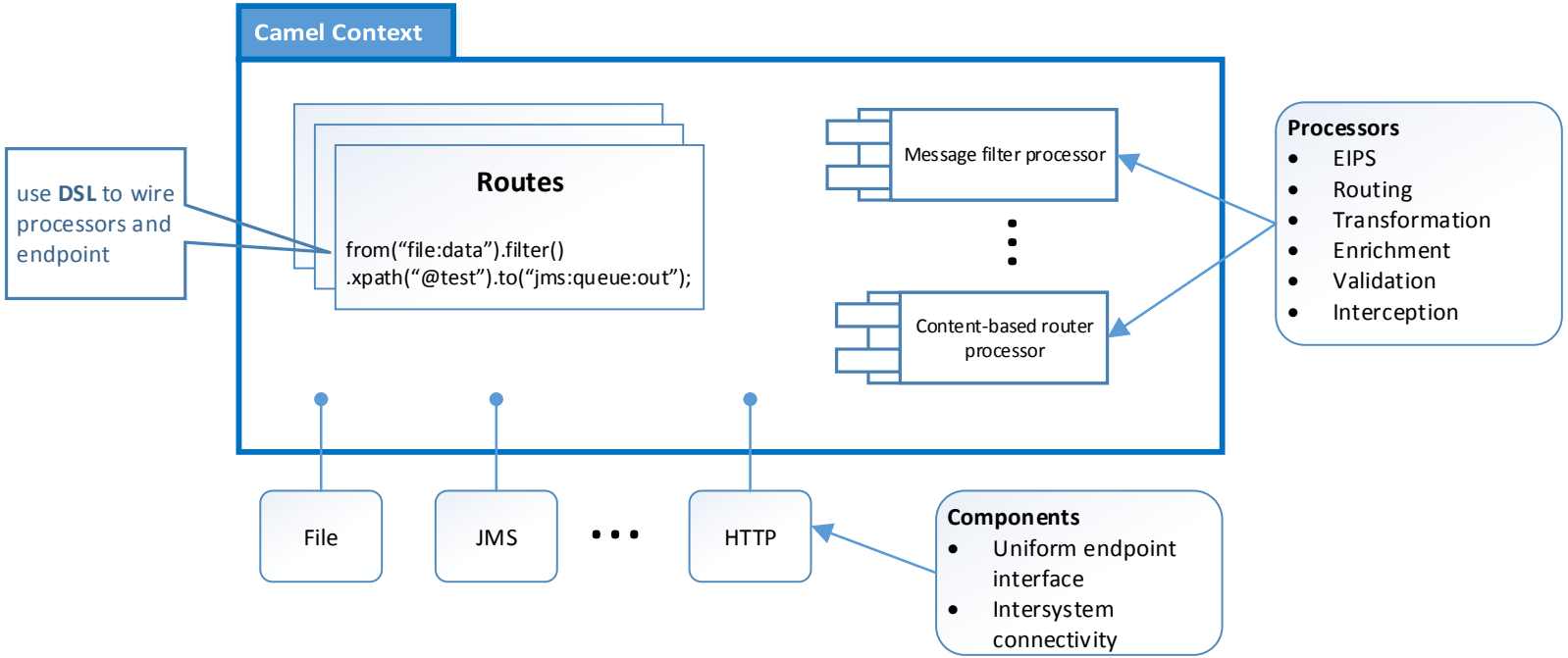
▶ Exchange

- Message's container
- Exchange parts:
  - In message / Out message
  - ExchangeID – unique ID
  - MEP (message exchange pattern)
    - **`InOnly` (Event Message EIP)**
    - **`InOut` (Request-Reply EIP)**
  - Exception
  - Properties ~ message Headers



Architecture

▶ Overview



Architecture

▶ Routes. DSL

- Routes – chain of processors
  - **Channel EIP**
  - Dynamic client – server relation
  - Extra processing can be added
  - Uniquely identified – starting/stopping, logging, debugging, monitoring
  - Defined using a DSL
  - Easy to test
- **`org.apache.camel.Processor`** – use, modify, create an exchange

- Domain Specific Language – wire processors and endpoints to create routes
  - JAVA Fluent API

```
from("file:data/inputFolder?noop=true")
.filter().xpath("@test")
.to("file:data/outputFolder");
```

  - XML DSL

```
<route>
    <from uri="file:data/inputFolder"/>
    <filter>
        <xpath>@test</xpath>
        <to uri="jms:queue:out"/>
    </filter>
</route>
```

Architecture

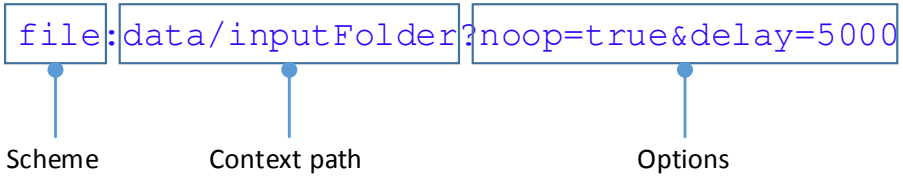
▶ Component

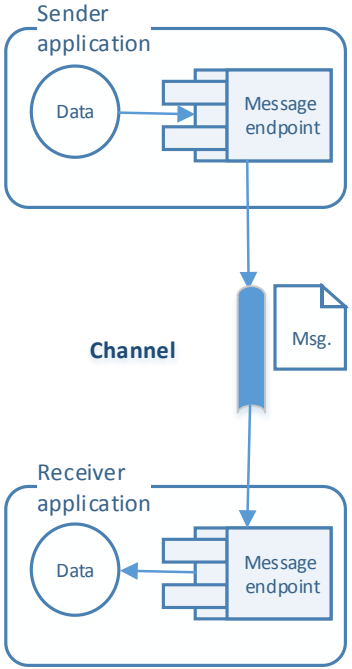
- `org.apache.camel.Component`
- Main extension points
- Used for data transports, DSLs, data formats
- Endpoints factories
- Custom components can be defined
- Associated to the endpoint URI schema

Architecture

▶ Endpoint

- **Endpoint EIP**
- **`org.apache.camel.Endpoint`**
- The end of a channel for sending or receiving messages
- Configured and refered using URIs

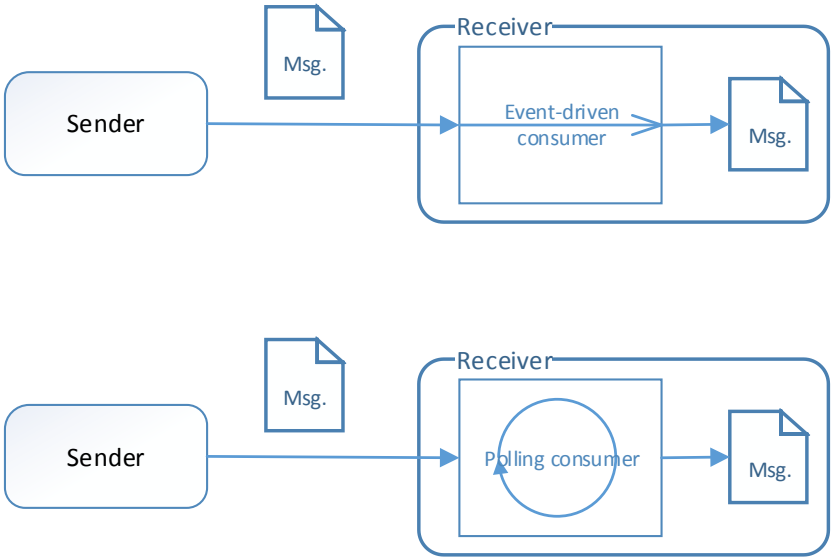




Architecture

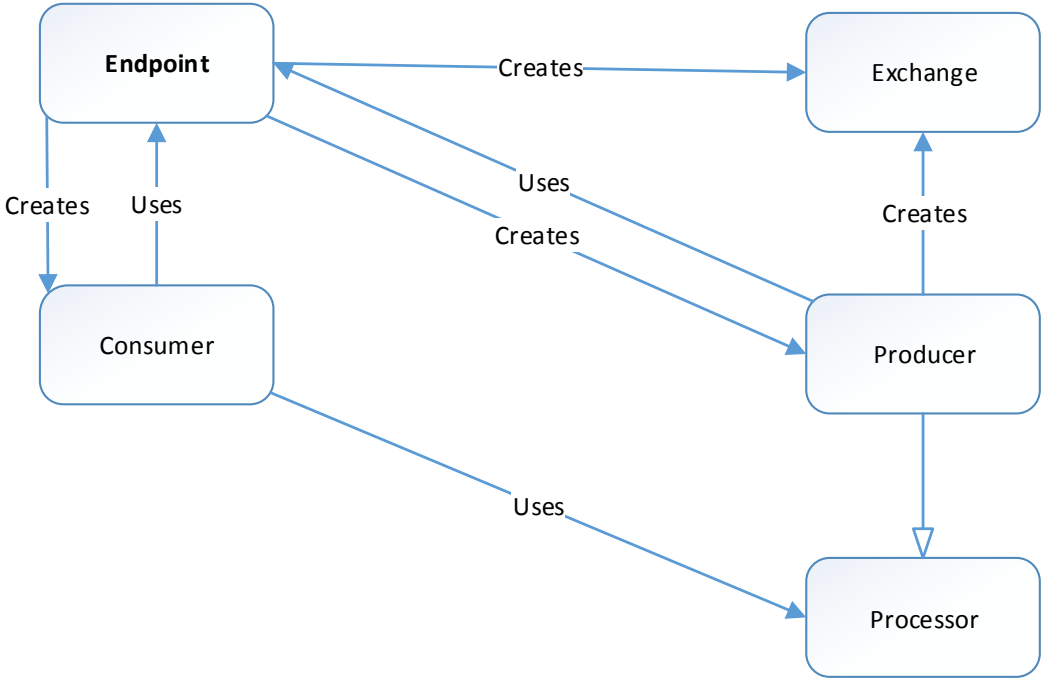
▶ Producer. Consumer

- **`Producer`** – **creates** and sends a message to an endpoint
  - Populates the message with endpoint specific data
  - **`FileProducer<File>;`**
    `JmsProducer<javax.jms.Message>`
- **`Consumer`** – **receives** message sending it for processing
  - **Event-driven consumer (EIP)** = *asynchronous receiver*
    `JmsConsumer<javax.jms.Message>`
  - **Polling consumer (EIP)** = *synchronous receiver*
    `FileConsumer<File>`



Architecture

▶ Model diagram



Routing with Camel

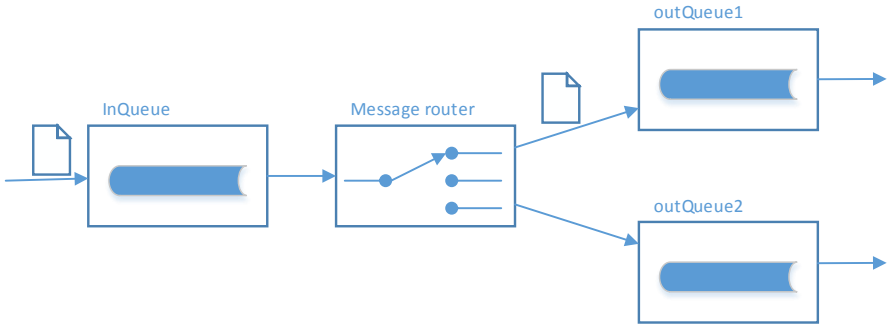
LUXOFT

LXFT LISTED NYSE

Routing with Camel

Routing overview

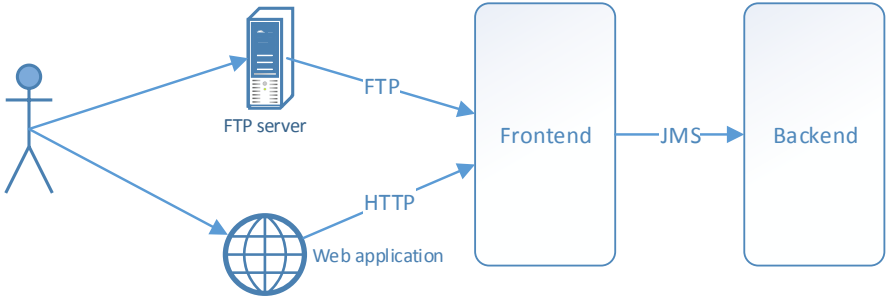
- Taking a message from an input queue and based on a set of conditions send it to one of several output queues
- Input and output systems are unaware of the conditions between
- Camel routing = step by step movement of a message
- Steps = EIP, processor, interceptor etc.



Routing with Camel

▶ Investment Bank application

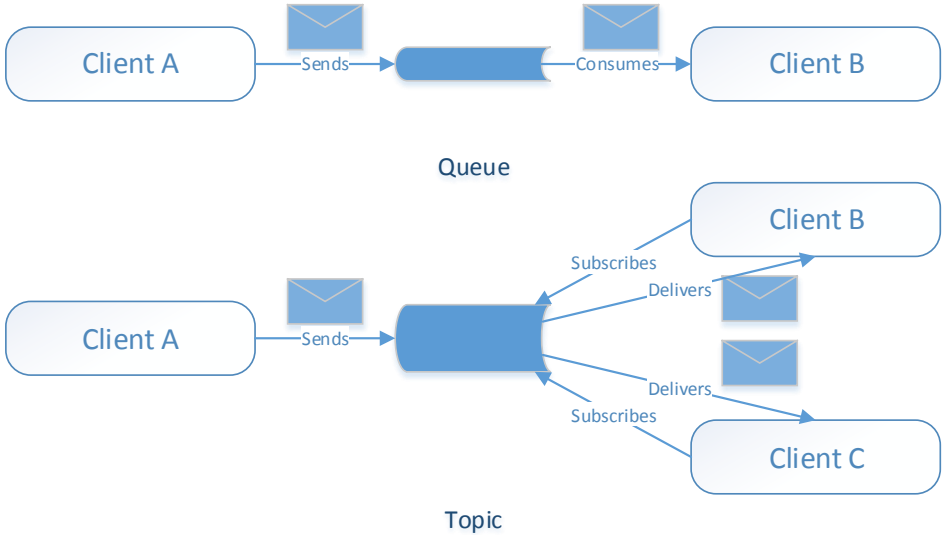
- Accepts investment orders from clients
- CSV format from an FTP server
- XML format from a Web application
- Orders are converted to POJOs for processing
- From the Frontend application they are sent to the Backend for checking and approval



Routing with Camel

▶ Sending to JMS

- JMS (Java Message Service) – API for creating, sending, receiving, reading messages
- Queue vs. Topic
- JMS providers/brokers – *ActiveMQ*



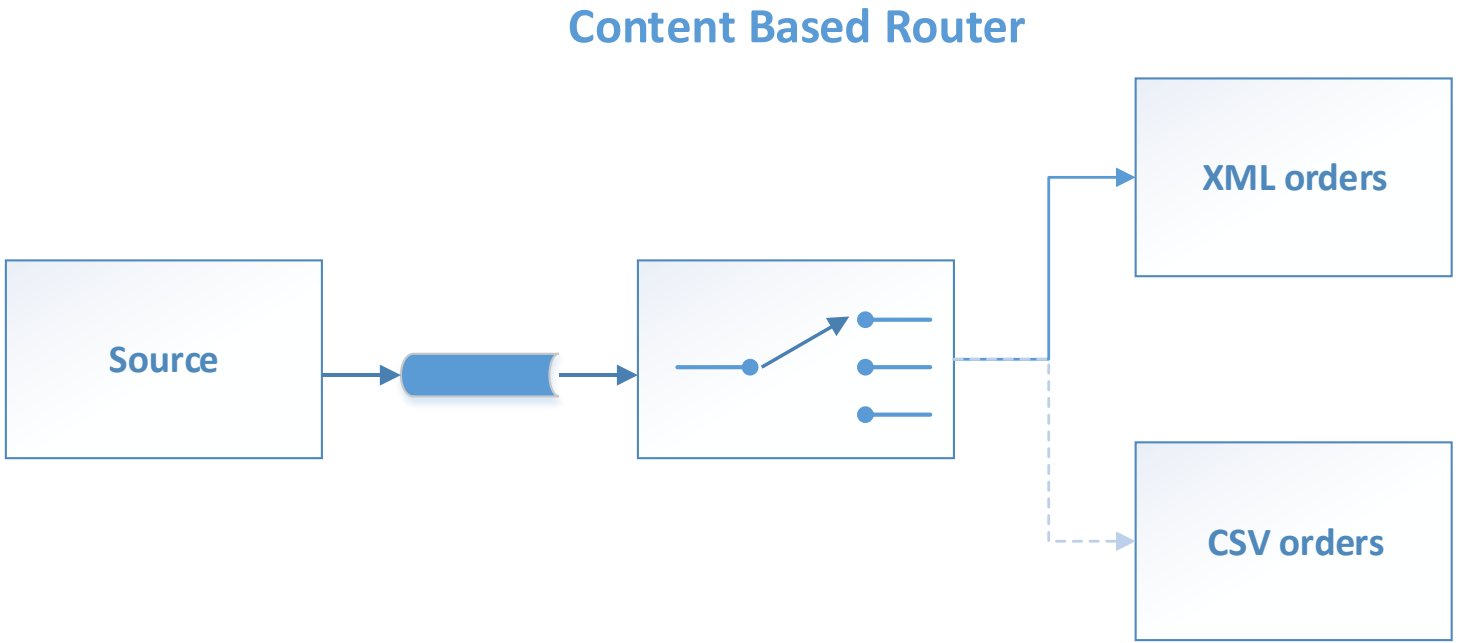
Java DSL

▶ RouteBuilder – used to create routes, access the CamelContext

```
// create CamelContext
CamelContext context = new DefaultCamelContext();

// connect to embedded ActiveMQ JMS broker
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("vm://localhost");
context.addComponent("jms", JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));

// add route to the CamelContext
context.addRoutes(
    new RouteBuilder() {
        @Override
        public void configure() {
           from("ftp://bank.com/orders?username=client&password=secret").to("jms:incomingOrders");
        }
});

// start the routes, otherwise nothing happens
context.start();
```

EIPs

▶ Content Based Router



EIPs

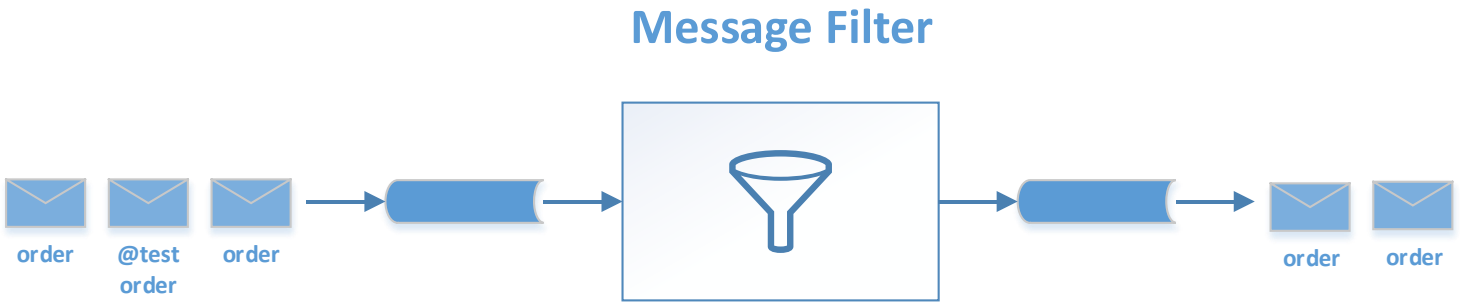
▶ Content Based Router

```
// content-based router
// dispatch messages based on extension
from("jms:incomingOrders")
.choice()
  .when(header("CamelFileName")
        .endsWith(".xml")).to("jms:xmlOrders")
  .endChoice()
  .when(header("CamelFileName")
        .regex("^.*(csv)$")).to("jms:csvOrders")
  .otherwise().to("jms:badOrders").stop()
.end()
.to("jms:continuedProcessing");
```

- Routes messages based on their content
- Uses **`org.apache.camel.Predicate`**
- Predicates can be built using expressions *(Simple, Xpath, Groovy, Python, XQuery etc.)*
- **`end()`** – to continue processing after CBR
- **`stop()`** – to stop processing

EIPs

▶ Message Filter



EIPs

▶ Message Filter

```
<order id="" clientID="" test="true">
    ...
</order>
```

```
// filter "non-test" messages
from("jms:xmlOrders")
.filter(xpath("/order[not(@test)]"))
  .process(new Processor() {
    public void process(Exchange exchange)
        throws Exception {
        System.out.println(
        "Received XML order: " +
        exchange.getIn().getHeader("CamelFileName"));
    }
});
```

- Messages pass through only if a condition is met
- Uses **`org.apache.camel.Predicate;`** **`org.apache.camel.Expression`**
- **`end()`** – to continue processing after Filter

Expressions

▶ Using expressions

- A Camel expression is evaluated at runtime on the current Exchange
- Interface **org.apache.camel.Expression**

```
public interface Expression {
    <T> T evaluate(Exchange exchange, Class<T> type);
}
```

- Can be used in the fluent builder style

```
from("direct:hey").transform(body().prepend("Hello "));
```

- Custom expressions usage

```
from("direct:hey").transform(new MyExpression());
```

Predicates

▶ Using predicates

- A Camel predicate is a specialized expressions returning a **Boolean** value
- Interface **org.apache.camel.Predicate**

```
public interface Predicate {
    boolean matches(Exchange exchange);
}
```

- Can be used in the fluent builder style

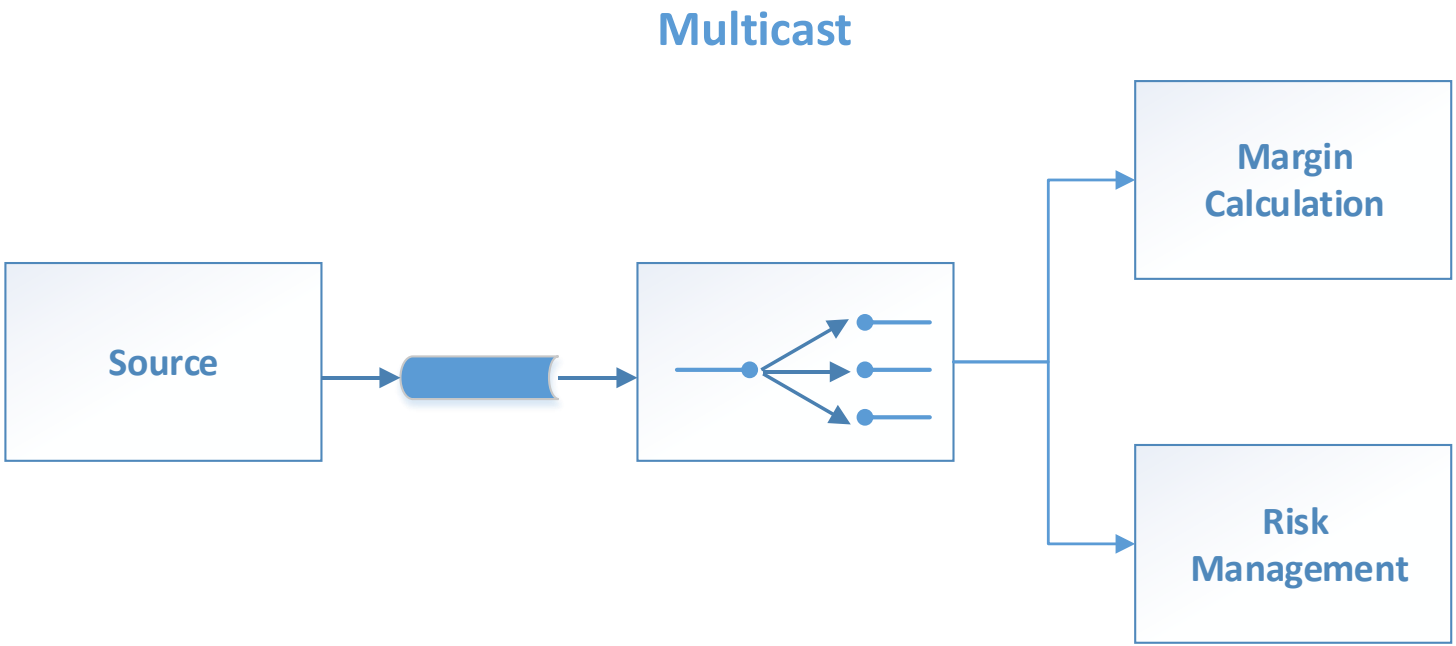
```
from("direct:quotes")
  .filter(body().contains("Camel")).to("direct:camelQuotes");
```

- Custom predicate usage

```
from("direct:quotes")
  .filter(new MyPredicate()).to("direct:camelQuotes");
```

EIPs

▶ Multicast



EIPs

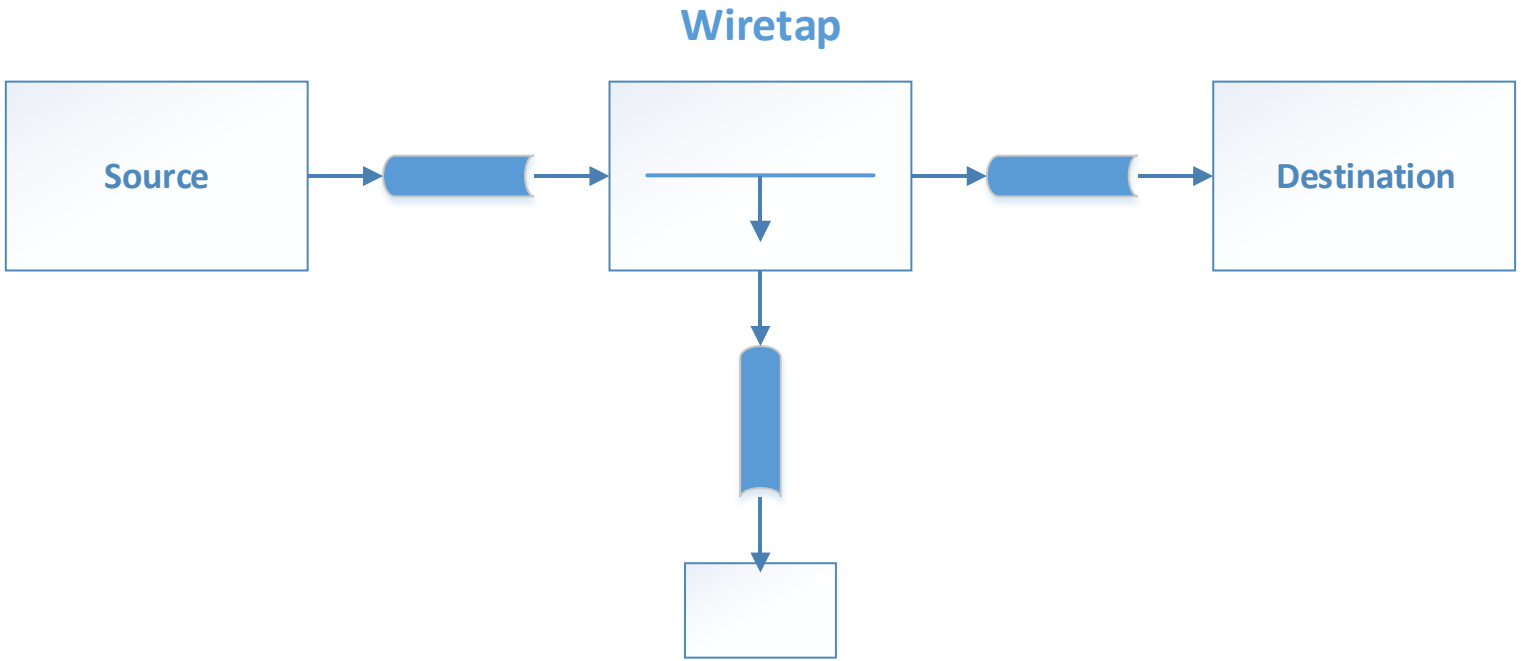
▶ Multicast

```
// send in parallel
// to margin calculation and risk management
from("jms:xmlOrders")
.multicast()
.parallelProcessing().executorService(executor)
.stopOnException()
.to("jms:marginCompQueue","jms:riskMgmtQueue");
```

- Send copies to several destinations
- Default processing is sequential
- **`parallelProcessing()`** – sending messages in parallel
- Default thread pool size is 10 – customizable using **`executorService()`**
- **`stopOnException()`** – stop multicast on first exception

EIPs

▶ Wiretap



EIPs

▶ Wiretap

```
// send in parallel to audit
// the result doesn't influences
// the main processing
from("jms:incomingOrders")
.wireTap("jms:orderAudit")
.choice()
    .when(header("CamelFileName").endsWith(".xml"))
        .to("jms:xmlOrders")
    .when(header("CamelFileName").regex("^.*(csv)$"))
        .to("jms:csvOrders")
    .otherwise()
        .to("jms:badOrders");
```

- Copy/Send messages to another channel
- Async processing

Transforming data

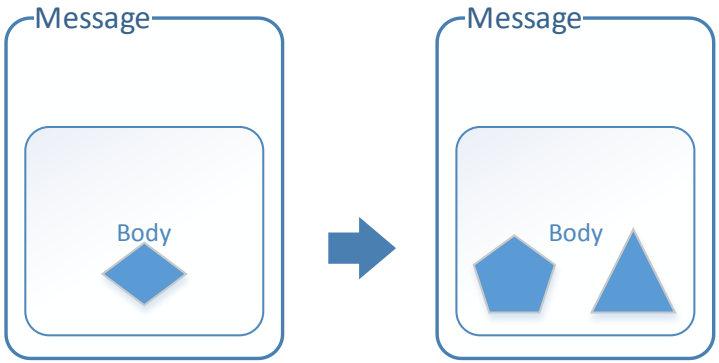
LUXOFT

LXFT
LISTED
NYSE

Transforming Data

▶ Overview

- Data mapping - the process of mapping between two distinct data models – key factor for data integration
- Camel offers Java code mapping instead of a custom mapping tool
- Data format transformation
  - from one form to another, e.g. *CSV to XML*
- Data type transformation
  - from one type to another, e.g. **`java.lang.String`** to **`javax.jms.TextMessage`**



Transforming Data

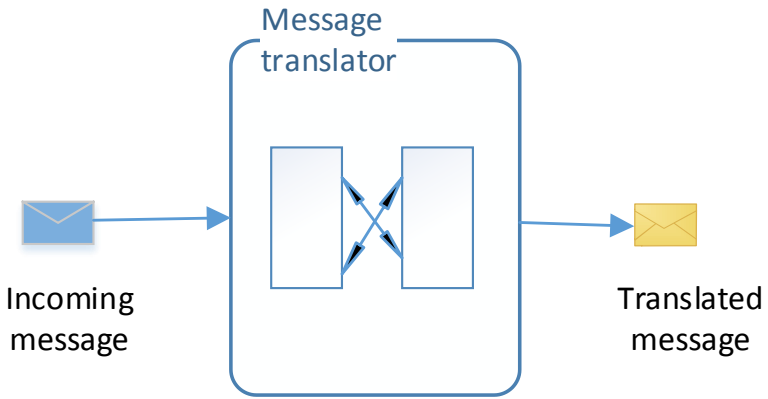
▶ Overview

- Six types of data transformation in Camel
  - Transformation in routes – **Message Translator** or **Content Enricher** EIPs
  - Using components – XSLT for XML
  - Using data formats – CSV, JSON, XML etc.
  - Using templates – Apache Velocity
  - Using type converters – **`String`** to **`File`**
  - *Transformation in component adapters – components transform the received data to and from the corresponding protocols

Transforming using EIPs and Java

▶ Message Translator

- Translates a message from one format to another
- Similar with the **Adapter** pattern
- Implementations
  - `Processor`
  - `Beans`
  - `transform()`



Transforming using EIPs and Java

▶ Message Translator using a Processor

- `org.apache.camel.Processor`
- `public void process(Exchange exchange) throws Exception;`

```
from("quartz://report?cron=0+0+6+*+*")
.to("http://bank.com/orders/cmd=received&date=yesterday")
        .process(new OrdersToCsvProcessor())
.to("file://bank/orders/received?fileName=report-${header.Date}.csv");
```

```
public class OrderToCsvProcessor implements Processor {

public void process(Exchange exchange) throws Exception {
  String custom = exchange.getIn().getBody(String.class);
        ...
  StringBuilder csv = new StringBuilder();
        ...
  exchange.getIn().setBody(csv.toString());
  }
}
```

Transforming using EIPs and Java

▶ Message Translator using a bean

```
public class OrderToCsvBean {

    public static String map(String custom) {
        String id = custom.substring(0, 9);
        ...

        StringBuilder csv = new StringBuilder();
        csv.append(id.trim());
        ...

        return csv.toString();
    }

}
```

```
from("quartz://report?cron=0+0+5+*+*")
.to("http://bank.com/orders/cmd=received&date=yesterday")
        .bean(new OrdersToCsvBean())
.to("file://bank/orders/received?fileName=report-${header.Date}.csv");
```

Transforming using EIPs and Java

▶ Tranforming using transform()

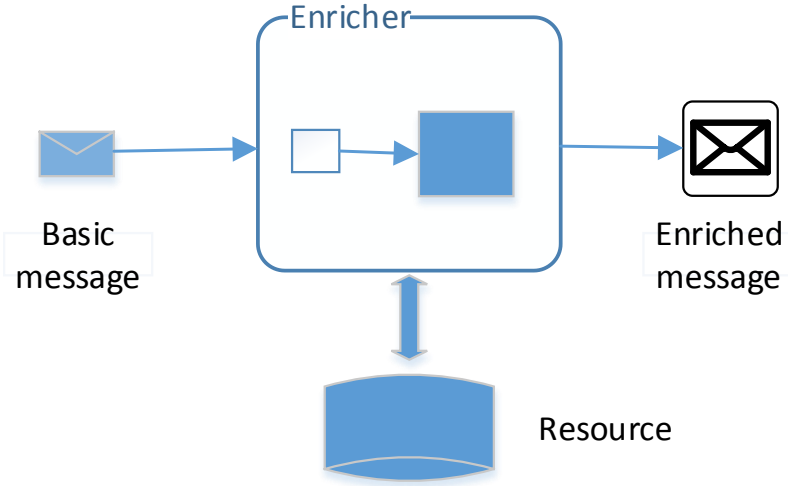
- Uses expressions
  - built from other individual expressions
  - custom expressions

```
from("direct:start")
.transform(body().regexReplaceAll("\n", "<br/>"))
.to("direct:result");


from("direct:start").transform(
new Expression() {
  @Override
  public <T> T evaluate(Exchange exchange, Class<T> type) {
    String body = exchange.getIn().getBody(String.class);
    body = body.replaceAll("\n", "<br/>");
    body = "<body>" + body + "</body>";
    return (T) body;
}}).to("direct:result");
```

Transforming using EIPs and Java

▶ Content Enricher

- The message is enriched with data obtained from another source
- **`pollEnrich()`** – merges data using a **consumer**
- **`enrich()`** – merges data using a **producer**




```
org.apache.camel.processor.AggregationStrategy
{
   Exchange aggregate(
        Exchange oldExchange,
        Exchange newExchange);
}
```

Transforming using EIPs and Java

▶ Enrich with pollEnrich()

- Use case scenario – orders report
  - map data received over HTTP to CSV
  - merge with orders from FTP

```
from("quartz://report?cron=0+0+3+*+*")
.to("http://bank.com/orders/cmd=received&date=yesterday")
.process(new OrdersToCsvProcessor())
.pollEnrich("ftp://bank.com/orders/?username=client&passowrd=secret",
    new AggregationStrategy() {
        @Override
        public Exchange aggregate(
                    Exchange oldExchange, Exchange newExchange) {
            if (newExchange == null) {
                return oldExchange;
            }
            String http = oldExchange.getIn().getBody(String.class);
            String ftp = newExchange.getIn().getBody(String.class);
            String body = http + "\n" + ftp;

            oldExchange.getIn().setBody(body);
            return oldExchange;
        }})
.to("file://bank/orders/received?fileName=report-${header.Date}.csv");
```

Transforming using EIPs and Java

▶ Enrich with pollEnrich()

- **`pollEnrich`** uses a polling consumer with three timeout modes
  - **`pollEnrich(timeout = -1)`** waits until a message arrives. It will block.
  - **`pollEnrich(timeout = 0)`** polls immediately if any message exists, otherwise returns **null**. It will never block.
  - **`pollEnrich(timeout > 0)`** polls the message, but if no message exists, it will wait for one at most until timeout expires.

Beans

Using beans

Introduction

- Component models (CORBA, EJB, JBI, SCA, OSGi) impose a lot of restrictions dictating what you can and cannot do
- POJO model – evolved in the Spring Framework – is a more simple and pragmatic model
- Camel is a lightweight container favoring POJO model for beans, working perfectly with already existing ones

Introduction

- Invoking beans in two ways

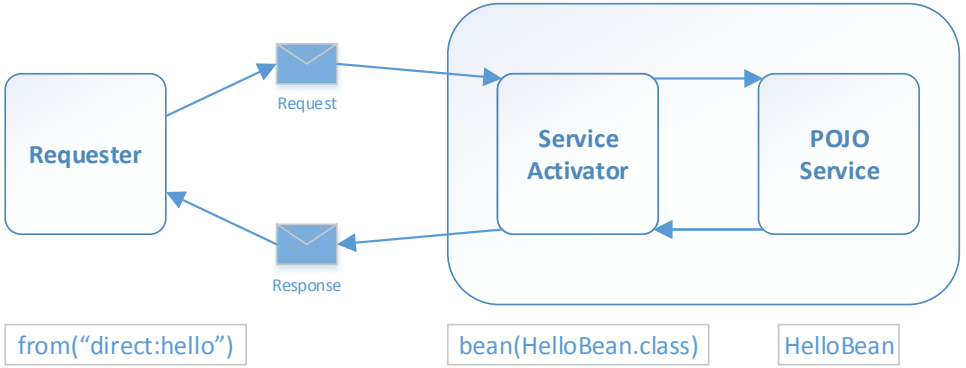
```
from("direct:hello")
        .process(new Processor() {
@Override
public void process(Exchange exchange) throws Exception {
    String name = exchange.getIn().getBody(String.class);

    HelloBean helloBean = new HelloBean();
     String salutation = helloBean.hello(name);

    exchange.getOut().setBody(salutation);
  }
});
```

```
public class HelloBean {
        public String hello(String name) {
                return "Hello" + name;
        }
}

…

from("direct:hello")
        .bean(HelloBean.class);
```

Introduction

▶ Service Activator EIP

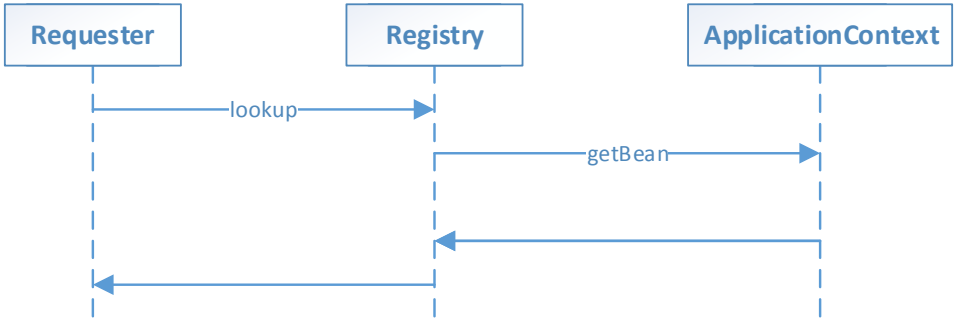
- It acts as a mediator between the requester and the POJO service
- The requester sends a request to the service activator
- The activator adapts the request to a format understood by the service
- The service replies to the activator which passes the response back to the requester



Bean registries

▶ Camel Registry

- The Camel Registry uses the real bean registry to look-up beans
- The Registry is a Service Provider Interface



`org.apache.camel.spi.Registry`

```
HelloBean helloBean =
  getContext().getRegistry()
  .lookupByNameAndType("helloBean",HelloBean.class);
```

Bean registries

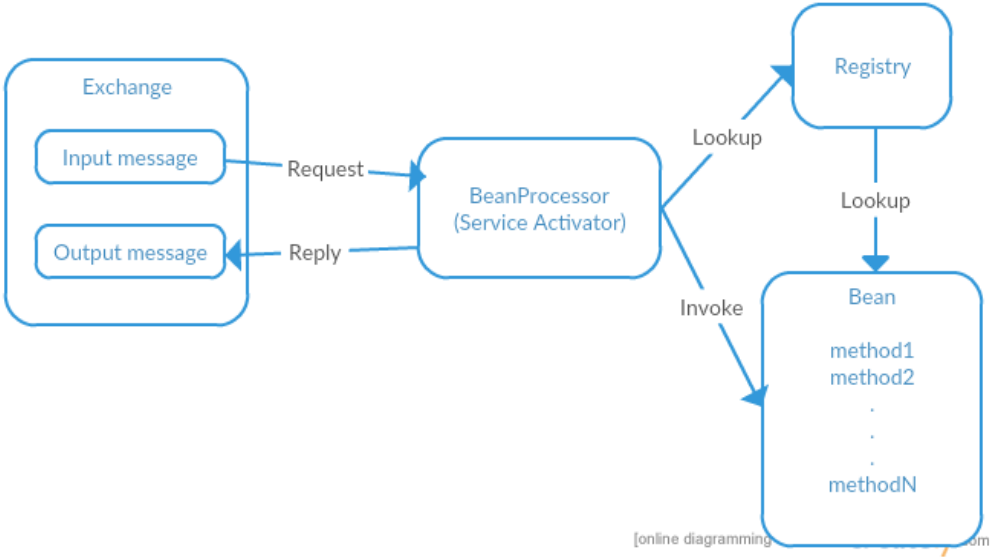
▶ Registries implementations

- **`SimpleRegistry`** – may be used when testing for instance
- **`JndiRegistry`** – uses an existing Java Naming and Directory Interface registry
- **`ApplicationContextRegistry`** – looks up beans in Spring ApplicationContext
- **`OsgiServiceRegistry`** – looks up beans in the OSGi service reference registry in an OSGi environment

Selecting bean methods

General steps

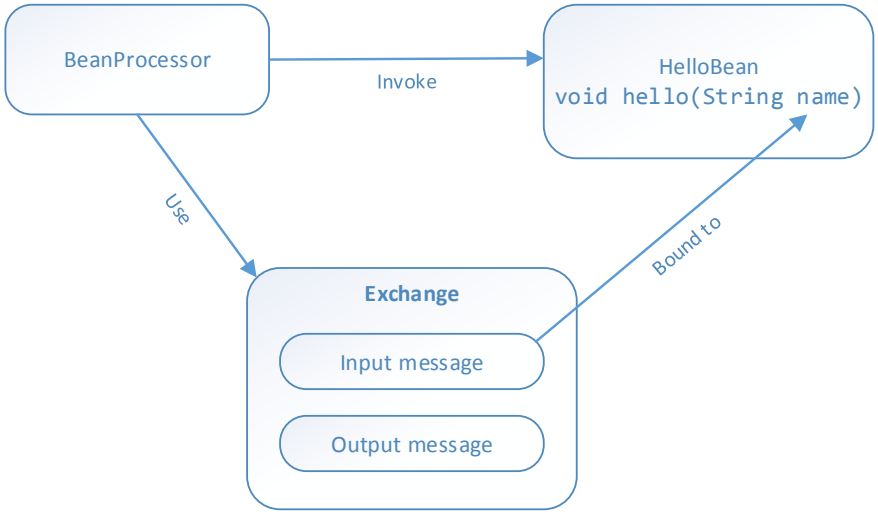
- At compile time there are no direct bindings
  - Camel resolves them at runtime
- Steps performed by the **`BeanProcessor`** when processing an **Exchange**:
  1. looks up the bean in the registry
  2. selects the method to invoke on the bean
  3. binds to the parameters of the selected method
  4. invokes the method
  5. handles any invocation errors (exceptions thrown are set on the Exchange for further handling)
  6. sets the method return value if any, as the body on the output message



Bean parameter binding

▶ Example

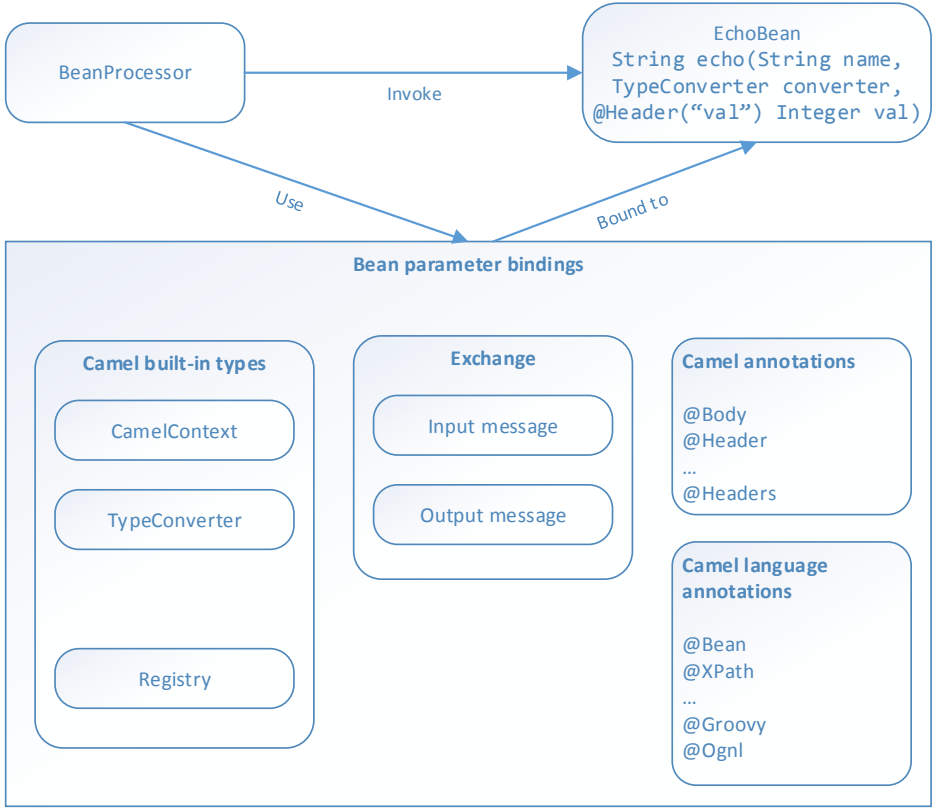
- **`BeanProcessor`** uses the input message to bind its body to the first parameter of the method
- Camel creates an expression that converts the input message to a **`String`**



Parameter binding

▶ Binding with multiple parameters

- Parameter binding uses four areas:
  - Camel built-in types
  - Exchange – allows binding the input message (body, headers)
  - Camel annotations
  - Camel language annotations – allows binding parameters to languages



Parameter binding

▶ Binding using built-in types

Type	Description
`Exchange`	The Camel Exchange
`Message`	The input message. It contains the body that is often bound to the first parameter.
`CamelContext`	It can be used to access Camel internal parts
`TypeConverter`	The type converter mechanism. It can be used when type conversion is needed.
`Registry`	The bean registry. Used for bean look-up.
`Exception`	An exception, if one was thrown. Bound only if the exchange has failed.

Parameter binding

▶ Binding using Camel annotations

Annotation	Description
`@Attachments`	Bind the parameters to the message attachments. The parameter must be a `java.util.Map`
`@Body`	Bind the parameter to the message body.
`@Header(name)`	Binds the parameter to the given message header.
`@Headers`	Binds the parameter to all input headers. The type of the parameter must be `java.util.Map`
`@OutHeaders`	Binds the parameter to the output message headers.
`@Property(name)`	Binds the parameter to the given property
`@Properties`	Binds all the Exchange properties. The parameter must be a `java.util.Map`

Parameter binding

▶ Binding using Camel language annotations

Annotation	Description	Dependency
`@Bean`	Invokes a method on a bean	`camel-core`
`@BeanShell`	Evaluates a bean shell script	`camel-script`
`@EL`	Evaluates an EL script (unified JSP and JSF scripts)	`camel-juel`
`@Groovy`	Evaluates a Groovy script	`camel-script`
`@JavaScript`	Evaluates a JavaScript script	`camel-script`
`@MVEL`	Evaluates a MVEL script	`camel-mvel`
`@OGNL`	Evaluates an OGNL script	`camel-ognl`
`@PHP`	Evaluates a PHP script	`camel-script`
`@Python`	Evaluates a Python script	`camel-script`

Parameter binding

- Binding using Camel language annotations

Annotation	Description	Dependency
`@Ruby`	Evaluates a **Ruby** script	`camel-script`
`@Simple`	Evaluates a **Simple** expression	`camel-core`
`@XPath`	Evaluates an **Xpath** expression	`camel-core`
`@XQuery`	Evaluates an **Xquery** expression	`camel-saxon`

```
public void updateStatus(
       @XPath("/order/status/text()") String status,
       @XPath( value = "/c:order/@customerId",
              namespaces = @NamespacePrefix(prefix = "c",
              uri = "http://camelintroduction/order")) Integer customerId)
{...}
```

Error handling

Error handling

▶ Where it applies

- It applies during the lifecycle of the **Exchange**
- *It means that there is a bit of room where it doesn't apply – on the **`Consumer`** where the **Exchange** it is created.
- Each Camel component must deal with these Exceptions in it's own way
- Components providing minor error handling features: File, FTP, Mail, iBatis, RSS, Atom, JPA and SNMP

Error handlers

▶ Built-in error handlers

- Error handlers will react only to Exceptions set on the **Exchange**

```
Exchange.getException() != null
```

- First three error handlers extend the **`RedeliveryErrorHandler`**
- The latter two error handlers have limited functionality and don't extend **`RedeliveryErrorHandler`**

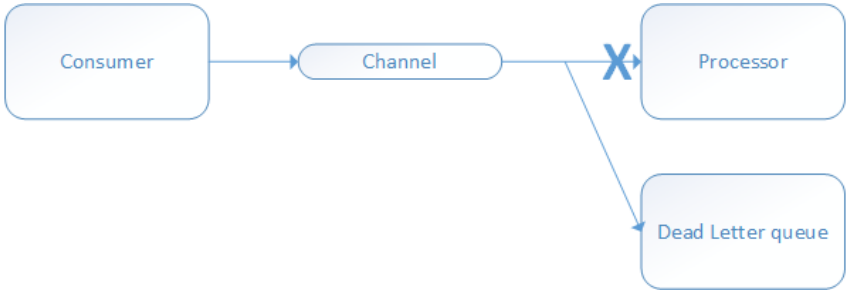
Error handler	Description
`DefaultErrorHandler`	Automatically enabled
`DeadLetterChannel`	Implements the Dead Letter Chanel EIP
`TransactionErrorHandler`	Transaction aware error handler
`NoErrorHandler`	Used to disable the error handling
`LoggingErrorHandler`	Used to log the exceptions

Error handlers

▶ Dead letter channel error handler

- Implements the **Dead Letter Channel EIP** – if a message can't be processed or delivered it should be moved to a **dead letter queue**
- The only error handler which supports the moving of failed messages to a dedicated error queue
- The dead letter queue is configured as an endpoint

```
errorHandler(

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



Error handlers

▶ Dead letter channel error handler

- Handling exceptions by default
  - By default the exceptions are suppressed, removed from the Exchange and set as the **`Exchange.CAUSED_EXCEPTION`** property
  - After the message has been moved to the dead letter queue, Camel stops routing it and the caller regards it as being processed
- Using the original message
  - When moving the messages to the dead letter queue they are already altered
  - To use the original message

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

Composed message processor

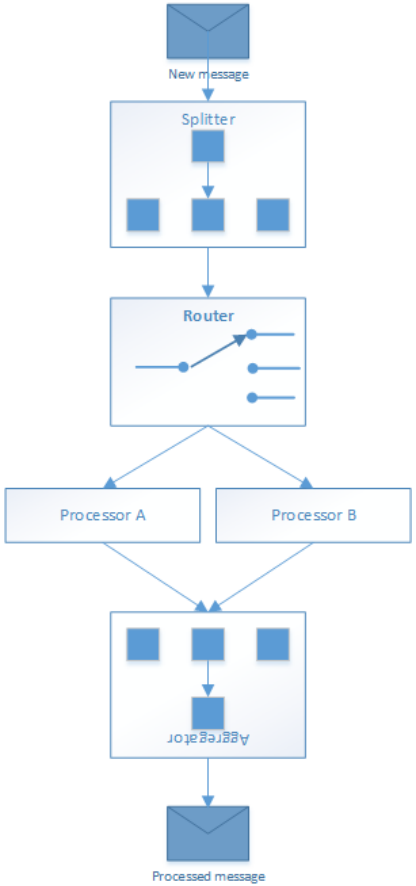
LUXOFT

LXFT
LISTED
NYSE

Introduction

▶ Aggregator and Splitter

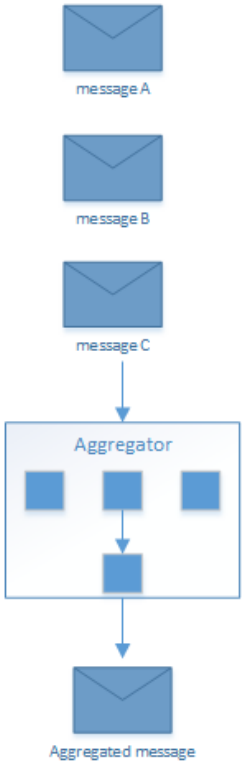
- **Aggregator** – used to combine results of individual messages into a single outgoing message
- **Splitter** – used to split a message into pieces that are routed separately
- Combining Splitter and Aggregator => **Composed Message Processor**



Aggregator

Introduction

- Combines many incoming messages into a single aggregated message
- It needs to identify the messages which are related
- Once a completion condition occurs the **aggregated message** is sent to the output
- Three configuration settings are needed
  - **Correlation identifier** – an **`Expression`** that determines which messages belong together
  - **Completion condition** – a **`Predicate`** or time-based condition that determines when the result is ready
  - **Aggregation strategy** – an **`AggregationStrategy`** implementation specifying how the messages are combined



Aggregator

▶ Use case

- Given three messages A, B and C the aggregator should output a single message containing ABC
  - when first message with correlation ID 1 arrives, a new aggregate is created and the message is stored in it
  - the completion occurs when we aggregated three messages
  - when the second message with correlation ID 1 arrives, it is added to the existing aggregate
  - the third message specifies a different correlation ID, so the aggregator starts a new aggregate
  - the fourth message relates to the first aggregate with identifier 1, we now have aggregated three messages and the completion condition is fulfilled
  - the aggregate is marked as complete and published

```
from("direct:start")
  .log("Sending ${body} with correlation key ${header.myId}")
  .aggregate(header("myId"),
  new MyAggregationStrategy())
  .completionSize(3)...
```

Corellation ID	Message	Aggregate
1	A	A
1	B	AB
2	X	AB X
1	C	ABC ✓ X

Aggregator

AggregationStrategy

```
import org.apache.camel.Exchange;
import org.apache.camel.processor.aggregate.AggregationStrategy;
public class MyAggregationStrategy implements AggregationStrategy {
  public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    if (oldExchange == null) {
      return newExchange;
    }

    String oldBody = oldExchange.getIn().getBody(String.class);
    String newBody = newExchange.getIn().getBody(String.class);
    String body = oldBody + newBody;
    oldExchange.getIn().setBody(body);

    return oldExchange;
  }
}
```

Aggregator

▶ Completion conditions

- Five conditions supported
- Aggregator supports multiple conditions - e.g **`completionSize`** and **`completionTimeout`**
- The condition that completes first will result in the aggregated message being published

Condition	Description
`completionSize`	Condition based on the number of messages aggregated together – **int** value or an **Expression**
`completionTimeout`	Condition based on an inactivity timeout. It triggers if a correlation group has been inactive longer than specified – **int** value or an **Expression**.
`completionInterval`	Condition based on a scheduled interval. Triggers periodically, there is a single timeout for all corellation groups, so they are all completed at once – **long** (miliseconds)
`completionPredicate`	Condition based on matching a **Predicate**.
`completionFromBatchConsumer`	Condition applicable when **Exchanges** are coming from a **BatchConsumer** (File, FTP, Mail, JPA)

Aggregator

▶ Persistence. Recovery

- Aggregator is a stateful EIP - agregates are kept in memory by default
- **AggregationRepository** and **RecoverableAggregationRepository**

```
aggregate(constant(true), new AggregationStrategy())
        .aggregationRepository(repo)...
```

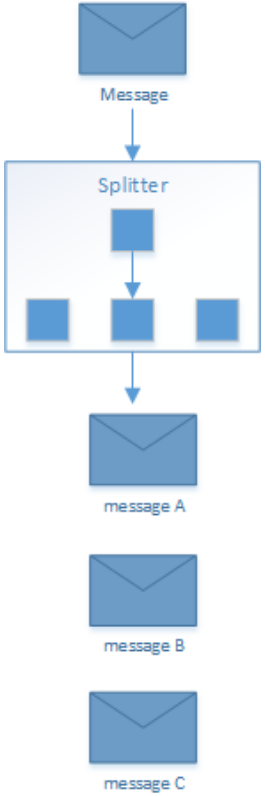
- Published aggregated messages can also be lost. Solutions:
  - Camel error handlers
  - **HawtDBAggregationRepository** – recovery, redelivery, dead letter channel and transactions

Splitter

Introduction

- Messages may consist of multiple elements

```
List<String> body = new ArrayList<String>();
body.add("A");
body.add("B");
body.add("C");
exchange.getIn().setBody(body);
...
from("direct:start")
  .split(body())
  .log("Split line ${body}")
  .end()
```

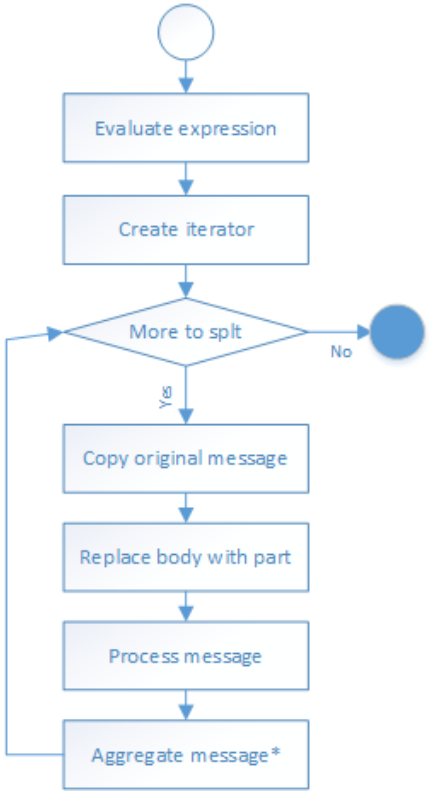


Splitter

▶ How it works

- You specify an **`Expression`** which is evaluated when a message arrives to split it
- The result is used to create a **`java.util.Iterator`**
- Camel knows by default to iterate over **`Collection`**, **`Iterator`**, **`Array`**, **`NodeList`**, **`String`**

Property	Type	Description
`Exchange.SPLIT_INDEX`	**`Integer`**	The index of the current exchange
`Exchange.SPLIT_SIZE`	**`Integer`**	The number of messages the original message was split into
`Exchange.SPLIT_COMPLETE`	**`Boolean`**	Is this the last message being processed



Splitter

Splitting messages

- Complex payloads can be split using beans

```
public class Customer {
        private int id;
        private String name;
        private List<Department> departments;
    }

public class CustomerService {
   public List<Department> splitDepartments(
         Customer customer) {
       return customer.getDepartments();
   }
}
```

```
.split().method(CustomerService.class,
"splitDepartments")

.split().simple("${body.departments}")
```

Splitter

▶ Aggregating split messages

- Known as Composed Message Pattern
- Splitter has a built-in aggregator
- The logic to recombine the split messages needs to be provided using an `AggregationStrategy`

```
.split(body(), new MyAggregationStrategy())
```

Integration frameworks

▶ Which one?

Spring Integration

- consistent model and messaging architecture to integrate several technologies
- EIPS implementation
- XML DSL – difficult to understand
- rudimentary support for technologies – just „basic stuff“ such as File, FTP, JMS, TCP, HTTP or Web Services
- the visual designer for Eclipse is ok

Mule ESB

- consistent model and messaging architecture
- EIPS implementation
- XML DSL – easy to understand
- full ESB
- very interesting connectors to important proprietary interfaces such as SAP, Tibco Rendevous, Oracle Siebel CRM, Paypal
- Mule Studio offers a very good and intuitive visual designer

Apache Camel

- consistent model and messaging architecture
- EIPS implementation
- XML DSL – easy to understand
- Java DSL, Groovy and Scala DSL – **fluent DSLs**
- many components (even more than Mule) for almost every technology
- custom components
- a very good (but commercial) visual designer, Fuse IDE is available by FuseSource – generates XML DSL code
- Talend offers a visual designer generating Java DSL code

Thank you

LUXOFT

LXFT
LISTED
NYSE