Walk through another example

Walk through another example

Introduction

Continuing the walk from our first example, we take a closer look at the routing and explain a few pointers - so you won't walk into a bear trap, but can enjoy an after-hours walk to the local pub for a large beer 🍔

First we take a moment to look at the Enterprise Integration Patterns - the base pattern catalog for integration scenarios. In particular we focus on Pipes and filters - a central pattern. This is used to route messages through a sequence of processing steps, each performing a specific function - much like the Java Servlet Filters.

Pipes and filters

In this sample we want to process a message in a sequence of steps where each steps can perform their specific function. In our example we have a JMS queue for receiving new orders. When an order is received we need to process it in several steps:

- validate
- register
- send confirm email

This can be created in a route like this:

```
<route>
   <from uri="jms:queue:order"/>
   <pipeline>
      <bean ref="validateOrder"/>
      <bean ref="registerOrder"/>
      <bean ref="sendConfirmEmail"/>
   </pipeline>
</route>
```

Pipeline is default

In the route above we specify `pipeline` but it can be omitted as its default, so you can write the route as:

```
<route>
   <from uri="jms:queue:order"/>
   <bean ref="validateOrder"/>
   <bean ref="registerOrder"/>
   <bean ref="sendConfirmEmail"/>
</route>
```

This is commonly used not to state the pipeline.

An example where the pipeline needs to be used, is when using a multicast and "one" of the endpoints to send to (as a logical group) is a pipeline of other endpoints. For example.

```
<route>
   <from uri="jms:queue:order"/>
   <multicast>
     <to uri="log:org.company.log.Category"/>
     <pipeline>
       <bean ref="validateOrder"/>
       <bean ref="registerOrder"/>
       <bean ref="sendConfirmEmail"/>
     </pipeline>
   </multicast>
</route>
```

The above sends the order (from `jms:queue:order`) to two locations at the same time, our log component, and to the "pipeline" of beans which goes one to the other. If you consider the opposite, sans the `<pipeline>`

```
<route>
   <from uri="jms:queue:order"/>
   <multicast>
     <to uri="log:org.company.log.Category"/>
     <bean ref="validateOrder"/>
     <bean ref="registerOrder"/>
     <bean ref="sendConfirmEmail"/>
   </multicast>
</route>
```

you would see that multicast would not "flow" the message from one bean to the next, but rather send the order to all 4 endpoints (1x log, 3x bean) in parallel, which is not (for this example) what we want. We need the message to flow to the validateOrder, then to the registerOrder, then the sendConfirmEmail so adding the pipeline, provides this facility.

Where as the `bean ref` is a reference for a spring bean id, so we define our beans using regular Spring XML as:

```
   <bean id="validateOrder" class="com.mycompany.MyOrderValidator"/>
```

Our validator bean is a plain POJO that has no dependencies to Camel what so ever. So you can implement this POJO as you like. Camel uses rather intelligent Bean Binding to invoke your POJO with the payload of the received message. In this example we will **not** dig into this how this happens. You should return to this topic later when you got some hands on experience with Camel how it can easily bind routing using your existing POJO beans.

So what happens in the route above. Well when an order is received from the JMS queue the message is routed like Pipes and filters:
1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as input to the registerOrder bean
3. the output from registerOrder bean is sent as input to the sendConfirmEmail bean

Using Camel Components

In the route lets imagine that the registration of the order has to be done by sending data to a TCP socket that could be a big mainframe. As Camel has many Components we will use the camel-mina component that supports TCP connectivity. So we change the route to:

```
<route>
   <from uri="jms:queue:order"/>
   <bean ref="validateOrder"/>
   <to uri="mina:tcp://mainframeip:4444?textline=true"/>
   <bean ref="sendConfirmEmail"/>
</route>
```

What we now have in the route is a `to` type that can be used as a direct replacement for the bean type. The steps is now:
1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as text to the mainframe using TCP
3. the output from mainframe is sent back as input to the sendConfirmEmai bean

What to notice here is that the `to` is not the end of the route (the world 😓) in this example it's used in the middle of the Pipes and filters. In fact we can change the `bean` types to `to` as well:

```
<route>
   <from uri="jms:queue:order"/>
   <to uri="bean:validateOrder"/>
   <to uri="mina:tcp://mainframeip:4444?textline=true"/>
   <to uri="bean:sendConfirmEmail"/>
</route>
```

As the `to` is a generic type we must state in the uri scheme which component it is. So we must write **bean:** for the Bean component that we are using.

Conclusion

This example was provided to demonstrate the Spring DSL (XML based) as opposed to the pure Java DSL from the first example. And as well to point about that the `to` doesn't have to be the last node in a route graph.

This example is also based on the **in-only** message exchange pattern. What you must understand as well is the **in-out** message exchange pattern, where the caller expects a response. We will look into this in another example.

See also

- Examples
- Tutorials
- User Guide