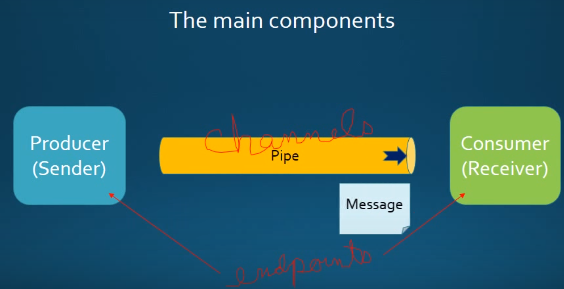
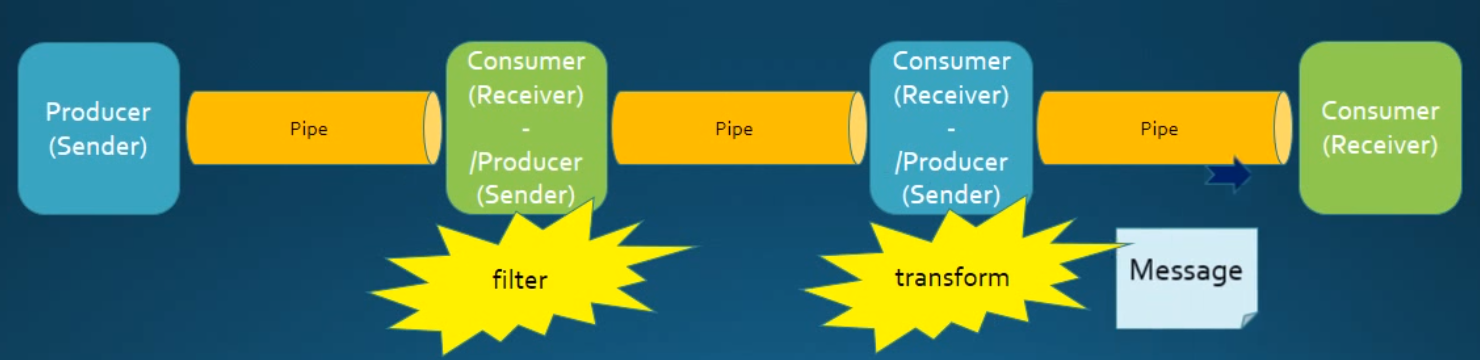
Spring Integration

# Introduction

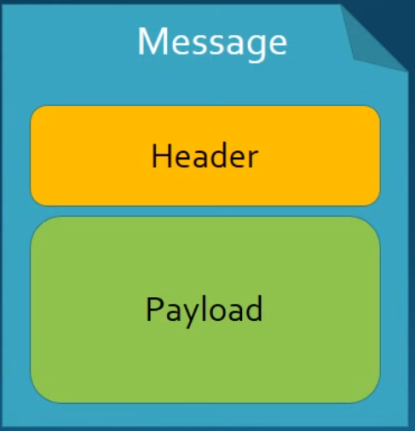
The main components: **Messages, Channels** and **Endpoints**.



Chains and pointing channels together



## Messages

**Header**: contains system information, e.g. creation timestamp

**Payload**: contains the data

## Message Endpoints

* **Adapters**: Connect your channel to some other system
* **Filter**: remove some messages from channels based on header, content, etc)
* **Transformer**: convert a message content or structure
* **Enricher**: add conent to the message header or payload
* **Service** activator: invoke service operations based on the arrival of a message.
* **Gateway**: connect your channels without spring integration coupling.

## Message Channels

**public** **interface** MessageChannel {

*//Send a Message to this channel. If successfully, returns true.*

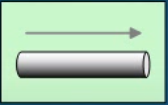
**boolean** send(Message<?> message);

*//Send a Message, blocking until either accepted or timeout elapses.*

**boolean** send(Message<?> message, **long** timeout);

}

The icon in Enterprise Integration Patterns(EIP) diagrams:



Two general classifications of message channels.

### Pollable Channel(buffering)

**public** **interface** PollableChannel **extends** MessageChannel {

*// Receive a message from this channel, blocking indefinitely if*

*// necessary.*

Message<?> receive();

*// Receive a message from this channel, blocking until either a message*

*// is available or the specified timeout period elapses and return null*

Message<?> receive(**long** timeout);

}

Pollable Channel may buffer its messages.

* Requires a queue to hold the messages
* The queue has a designated capacity

Waits for the consumer to get the messages, consumers activately poll to receive messages.

It is typically a point-to-point channel, only one receiver of a message in the channel. Usually used for Document or information.

### Subscribable Channel(non-buffering)

*// A MessageChannel that maintains a registry of subscribers and invokes*

*// them to handle messages sent through this channel.*

**public** **interface** SubscribableChannel **extends** MessageChannel {

*//Register a message handler.*

**boolean** subscribe(MessageHandler handler);

*//Un-register a message handler.*

**boolean** unsubscribe(MessageHandler handler);

}

Subscribable channel allows multiple subscribers (or consumers) to register for its messages.

* Messages are delivered to all registered subscribers on message arrival
* It has to manage a list or registry of subscribers

It doesn’t buffer its messages. Usually used for “event” messages.

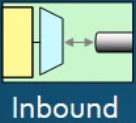
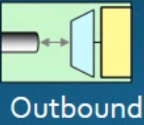
# Adapters

Adapters: are the endpoints in spring integration that connects channel to an actual system. It provides the bridge between integration framework and the external systems and services(bootk: pro spring integration.)

It provides separation of concerns, that helps to separate the messaging API from what is to transport and protocol used entire spring integration system. You don’t want your code have to know a lot about JMS or JDBC, the spring intgration adapters help provide for those capabilities.

Adapters are classified as either inbound or outbound adapters

* **Inbound Adapter**: bring messages into the spring integration channels
* **Outbound Adapter**: Get the messages out at the Spring Integration Channels and into outside the applications databases etc.

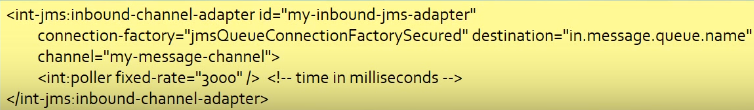
Built-in adapters:

* Stream adapters
* File Adapters
* JDBC & JPA Adapters
* FTP and Secure FTP (SFTP) Adapters
* Feed(RSS, Atom, etc.) Adapters
* Mail Adapters
* MongoDB Adapters
* UDP Adapters
* Tweeter Adapter

## A JMS Inbound Adapter

The following configuration takes messages from a message Queue (via JMS under the covers) and gets it into a spring integration channel.

This inbound adapter requires a poll of the messages into the channel. It means how often should this adapter poll messages into one of the channels.

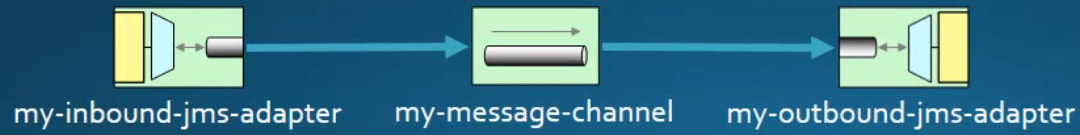


## A JMS Outbound Adapter

The following configuration takes messages from a message channel and delivers it to a message Queue (via JMS unter the covers).



Diagram for the two examples together:



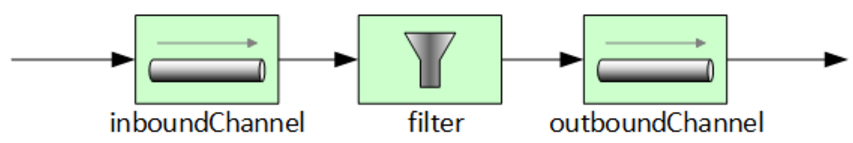
# Filters

http://cdn.intertech.com/Blog/wp-content/uploads/2014/07/filter.png

Filters are endpoints that sit between channels and allow, on the basis of a message’s content or metadata(message header), a message to

* Pass from one channel to the next or
* Reject and discard the message from the system.

Messages that are rejected are simply removed from the spring integration system, spring integration channel and other components. However optionally you can also provide a *discard channel* with your filter and send those messages which have been rejected by the filter.



Spring integration provides many filters out of the box. But you create your own custom filters.

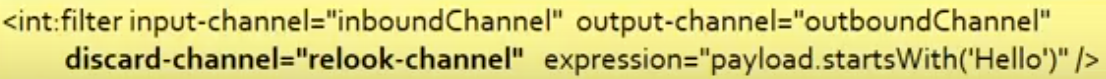
## Built-in filters

* Expression Filter: work on the basis of spring expression language
* Xpath Filter: use Xpath expression against the XML message
* XML Validation Filter: validate XML message agasinst a given schema.

Example:



Example with *discard channel*:

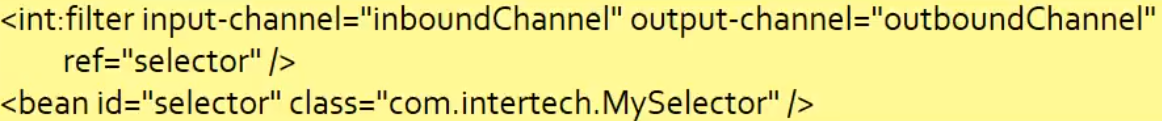


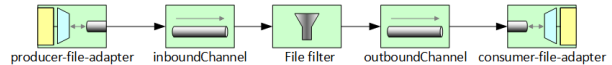
## Custom Filters

To create a custom filter, your must implement the MessageSelector Interface.

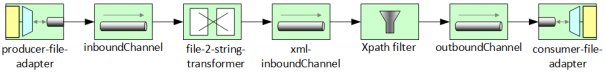
**public interface** MessageSelector **extends** GenericSelector<Message<?>> {  
 **boolean** accept(Message<?> message);  
}

**public class** MySelector **implements** MessageSelector{  
 **public boolean** accept(Message<?> message) {  
 Object payload = message.getPayload();  
 **return** payload **instanceof** String && ((String)payload).startsWith(**"Hello"**);  
 }  
}

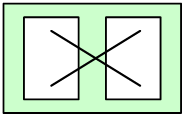




Or



# Transformers



Transformers take a message from a channel and creates a new message containing converted payload or message structure. XML can be transformed to JSON, JSON transformed to Java Objects, etc.

## Built-in Transformers

* XML <-> Object
* Object <-> String
* File <--> String
* Object Serializer/Deserializer
* Object <-> Map
* Object <-> JSON
* Claim Check (Implementing the Claim check design pattern)

You can use simple POJOs to create your own custom transformers.

## Examples



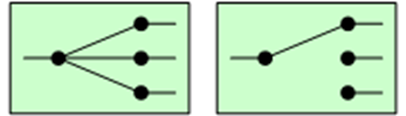
It takes a message with an object payload from the *inboundChannel*, calls toString on the object and puts the result string into the *outboundChannel*.

String to String transformer using Spring Expression Language(SpEL)



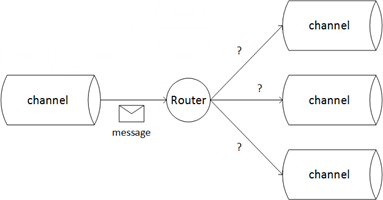


# Routers



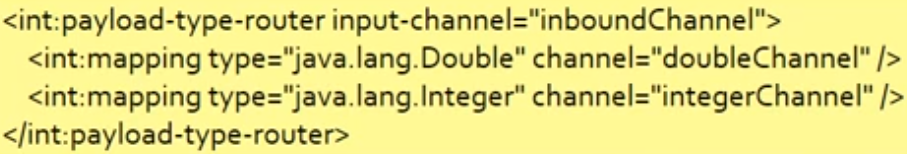
Routers distribute messages to one or more channels.

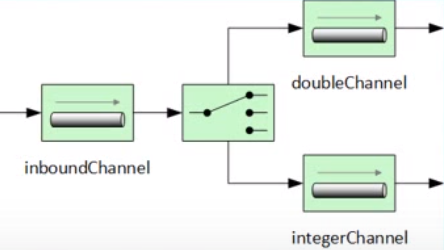
* Redipient list Routers: simply distribute the message to all listed message channels.
* Content Routers: examine the message payload or headers in order to select a particular destination message channel.



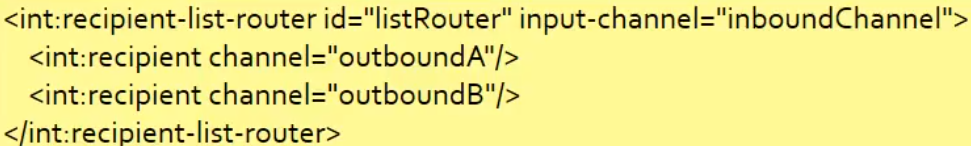
## Example

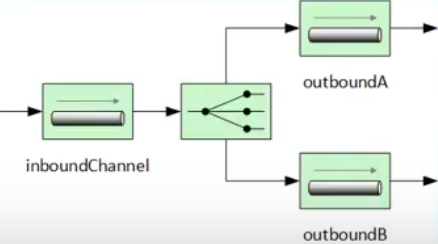
* Below is the configuration for a simple payload type content router:



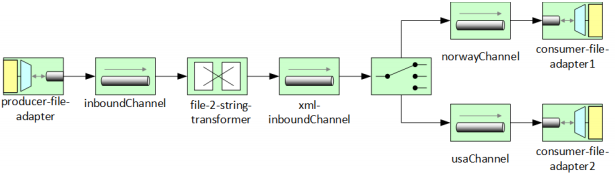


* Recipient list routers:

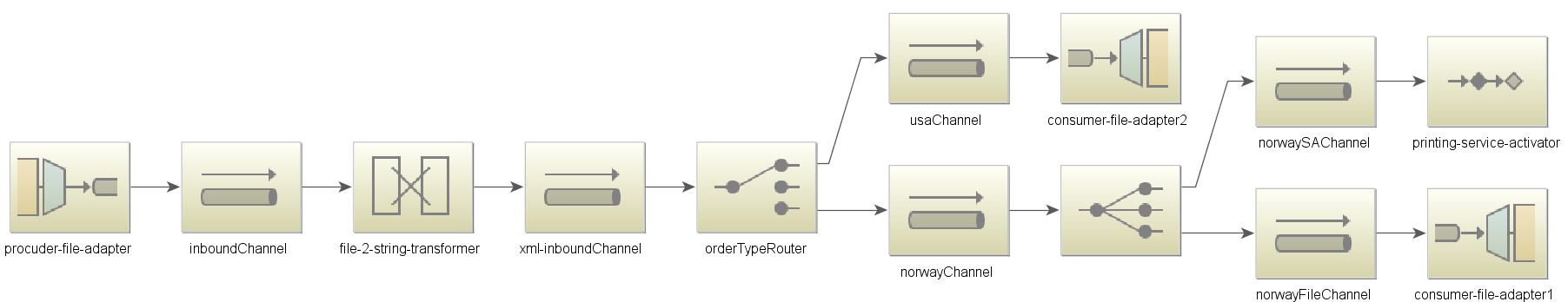




<**int-xml:xpath-router id="orderTypeRouter" input-channel="xml-inboundChannel"**>  
 <**int-xml:xpath-expression expression="/shiporder/shipto/country"** />  
 <**int-xml:mapping value="Norway" channel="norwayChannel"** />  
 <**int-xml:mapping value="USA" channel="usaChannel"** />  
</**int-xml:xpath-router**>



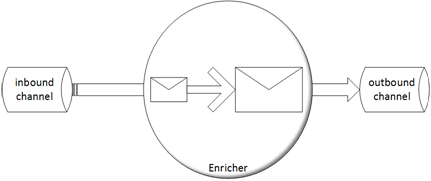
<**int:recipient-list-router input-channel="norwayChannel"**>  
 <**int:recipient channel="norwayFileChannel"**/>  
 <**int:recipient channel="norwaySAChannel"**/>  
</**int:recipient-list-router**>



# Enrichers

http://cdn.intertech.com/Blog/wp-content/uploads/2014/08/enricher.png

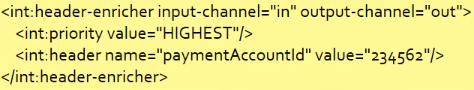
Enrichers is a special type of transformer. Enrichers take a message and enhance it by adding information to its header or payload.



Spring provides a number of enrichers out of box. You can create your own enrichers. You do that by adding specific spring integration components to the enrichment configuration. Typly it is done through the service activator.

## Example

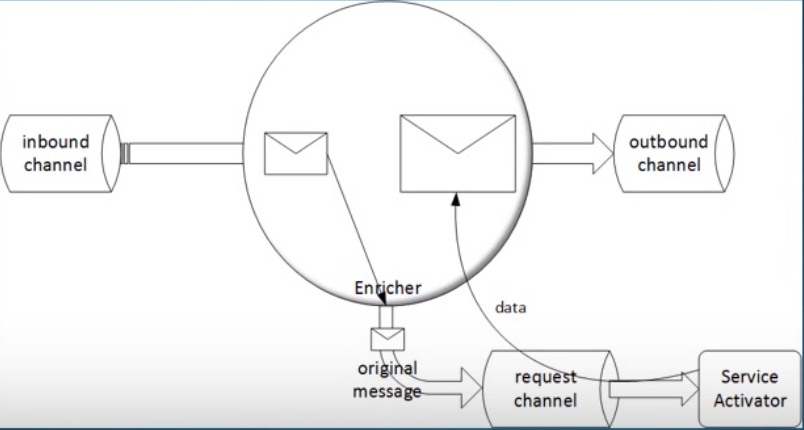
This example takes a message and adds a priority Header as well as a custom header called paymentAccountId.



Payloader enricher, add data to the message payload.

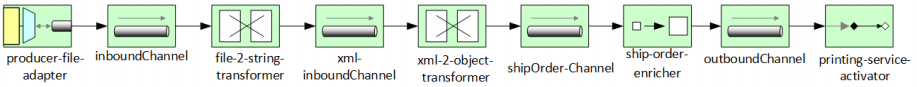


Custom enricher: that uses another Spring integration component. In this case is simple service activator to provide the data backed in the channel.



<**int:enricher id="ship-order-enricher" input-channel="shipOrder-Channel"**

**output-channel="outboundChannel"**>  
 <**int:property name="shipped" expression="true"**/>  
</**int:enricher**>



<**int:enricher id="ship-order-enricher" input-channel="shipOrder-Channel"**

**output-channel="outboundChannel" request-channel="total-price-enricher-channel"**>  
 <**int:property name="orderTotal" expression="payload"** />  
 <**int:property name="shipped" expression="true"**/>  
</**int:enricher**>

<**bean id="shipOrderEnricher" class="de.swm.integration.lab6.ShipOrderEnricher"**/>  
<**int:service-activator id="enriching-service-activator" ref="shipOrderEnricher"  
 input-channel="total-price-enricher-channel"**/>

