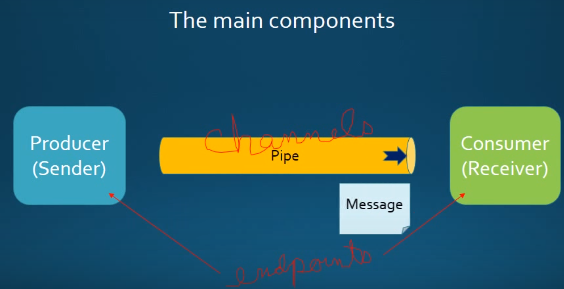
Spring Integration

# Introduction

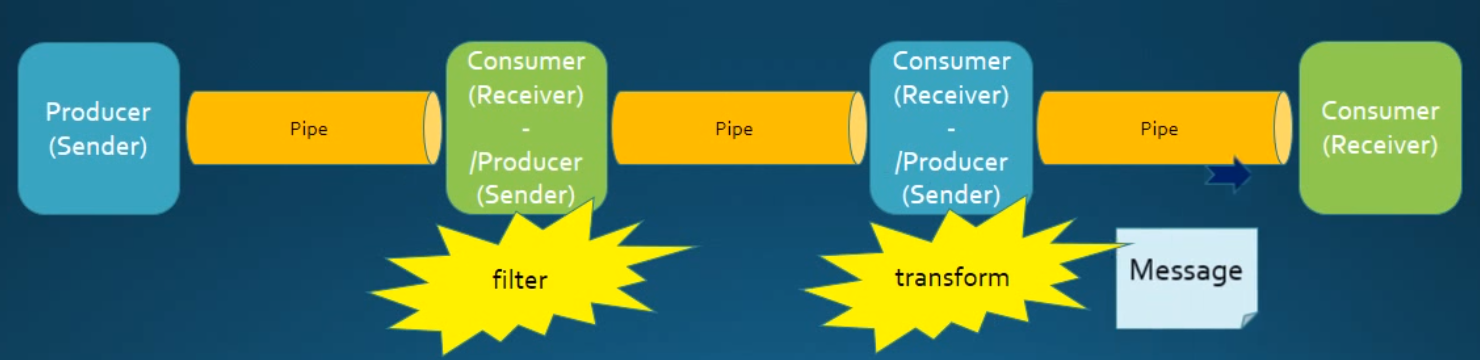
The main components: **Messages, Channels** and **Endpoints**.

From the *vertical* perspective, a layered architecture facilitates separation of concerns, and interface-based contracts between layers promote loose coupling. Message-driven architectures add a *horizontal* perspective, yet these same goals are still relevant.

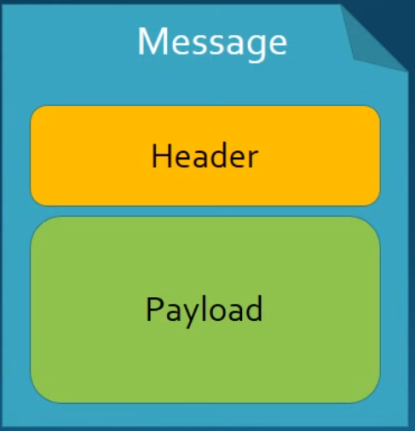
Messaging systems typically follow the similarly abstract *"pipes-and-filters"* model. The "filters" represent any component that is capable of producing and/or consuming messages, and the "pipes" transport the messages between filters so that the components themselves remain loosely-coupled. It is important to note that these two high-level paradigms are not mutually exclusive. The underlying messaging infrastructure that supports the "pipes" should still be encapsulated in a layer whose contracts are defined as interfaces. Likewise, the "filters" themselves would typically be managed within a layer that is logically above the application’s service layer, interacting with those services through interfaces much in the same way that a web-tier would.



Chains and pointing channels together



## Messages

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

**Payload**: contains the data

## Message Endpoints

One of the primary goals of Spring Integration is to simplify the development of enterprise integration solutions through *inversion of control*. This means that you should not have to implement consumers and producers directly, and you should not even have to build Messages and invoke send or receive operations on a Message Channel. Instead, you should be able to focus on your specific domain model with an implementation based on plain Objects. Then, by providing declarative configuration, you can "connect" your domain-specific code to the messaging infrastructure provided by Spring Integration. The components responsible for these connections are Message Endpoints. This does not mean that you will necessarily connect your existing application code directly. The important thing is to achieve separation of concerns between such integration logic and business logic. The goal should be to provide a thin but dedicated layer that translates inbound requests into service layer invocations, and then translates service layer return values into outbound replies.

A Message Endpoint represents the "filter" of a pipes-and-filters architecture. As mentioned above, the endpoint’s primary role is to connect application code to the messaging framework and to do so in a noninvasive manner. In other words, the application code should ideally have no awareness of the Message objects or the Message Channels. This is similar to the role of a Controller in the MVC paradigm. Just as a Controller handles HTTP requests, the Message Endpoint handles Messages. Just as Controllers are mapped to URL patterns, Message Endpoints are mapped to Message Channels. The goal is the same in both cases: isolate application code from the infrastructure.

* **Channel 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.
* **Splitter**: accept a Message from input channel, split it into multiple Messages.
* **Aggregator**: receives multiple Messages and combines them into a single Message.

## Message Channels

**publicinterface** MessageChannel {

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

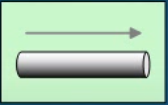
**boolean** send(Message<?> message);

*//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)

**publicinterface** 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);

}

In Spring integration, *Pollable Channels* are capable of buffering Messages within a queue.

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

**Advantage**: it allows for throttling the inbound Messages and thereby prevents overloading a consumer.

**Disadvantage**: it adds some complexity, since a consumer can only receive the Messages from such a channel if a *poller* is configured.

Waits for the consumer to get the messages, consumers actively 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.

#### QueueChannel

To create a QueueChannel, use the <queue/> sub-element. If you do not provide a value for the capacity attribute on this <queue/> sub-element, the resulting queue will be unbounded.

**<int:channel id**=**"queueChannel">**

**<queue capacity**=**"25"/>**

**</int:channel>**

#### PriorityChannel

To create a PriorityChannel, use the <priority-queue/> sub-element.

<**int:channel id="priorityChannel"**>  
 <**int:priority-queue capacity="20"**/>  
</**int:channel**>

By default, the channel will consult the priority header of the message. However, a custom Comparator reference may be provided instead. Also, note that the PriorityChannel (like the other types) does support the datatype attribute. As subtype of the QueueChannel, it also supports a capacity attribute.

<**int:channel id="priorityChannel" datatype="example.Widget"**>  
 <**int:priority-queue comparator="widgetComparator" capacity="10"**/>  
</**int:channel**>

### Subscribable Channel(non-buffering)

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

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

**publicinterface** 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.

#### PublishSubscribeChannel

The PublishSubscribeChannel implementation broadcasts any Message sent to it to all of its subscribed handlers. This is most often used for sending *Event Messages* whose primary role is notification. Note that the PublishSubscribeChannel is intended for sending only. Since it broadcasts to its subscribers directly when its` send(Message)` method is invoked, consumers cannot poll for Messages. Any subscriber must be a MessageHandler itself, and the subscriber’s handleMessage(Message) method will be invoked in turn.

Starting with version 3.0, the behavior has changed such that a send is always considered successful if at least the minimum subscribers are present (and successfully handle the message). This behavior can be modified by setting the minSubscribers property, which defaults to 0.

If a TaskExecutor is used, only the presence of the correct number of subscribers is used for this determination, because the actual handling of the message is performed asynchronously.

#### RendezvousChannel

The RendezvousChannel enables a "direct-handoff" scenario where a sender will block until another party invokes the channel’s receive() method or vice-versa. The RendezvousChannel is also useful for implementing request-reply operations. The sender can create a temporary, anonymous instance of RendezvousChannel which it then sets as the replyChannel header when building a Message. After sending that Message, the sender can immediately call receive (optionally providing a timeout value) in order to block while waiting for a reply Message. This is very similar to the implementation used internally by many of Spring Integration’s request-reply components.

A RendezvousChannel is created when the queue sub-element is a <rendezvous-queue>. It does not provide any additional configuration options to those described above, and its queue does not accept any capacity value since it is a 0-capacity direct handoff queue.

**<int:channel id**=**"rendezvousChannel"/>**

**<int:rendezvous-queue/>**

**</int:channel>**

#### DirectChannel

The DirectChannel has point-to-point semantics, it implements the SubscribableChannel interface instead of the PollableChannel interface, so it dispatches Messages directly to a subscriber. However, it differs from the PublishSubscribeChannel in that it will only send each Message to a single subscribed MessageHandler.

DirectChannel is the default type by the configuration. A default channel will have a round-robin load-balancer and will also have failover enabled.

**<int:channel id**=**"directChannel"/>**

To disable one or both of these, add a <dispatcher/> sub-element and configure the attributes.

**<int:channel id**=**"failFastChannel">**

**<int:dispatcher failover**=**"false"/>**

**</channel>**

**<int:channel id**=**"channelWithFixedOrderSequenceFailover">**

**<int:dispatcher load-balancer**=**"none"/>**

**</int:channel>**

#### ExecutorChannel

The ExecutorChannel is a point-to-point channel that supports the same dispatcher configuration as DirectChannel. The key difference between these two dispatching channel types is that the ExecutorChannel delegates to an instance of TaskExecutor to perform the dispatch. This means that the send method typically will not block, but it also means that the handler invocation may not occur in the sender’s thread. It therefore does not support transactions spanning the sender and receiving handler.

**<int:channel id**=**"executorChannelWithoutFailover">**

**<int:dispatcher task-executor**=**"someExecutor" failover**=**"false"/>**

**</int:channel>**

The *load-balancer* and *failover* options are also both available on the <dispatcher/> subelement. The channel will by default have a round-robin load-balancing strategy with failover enabled unless explicit configuration is provided for one or both of those attributes.

#### Scoped Channel

## Channel Interceptor

The ChannelInterceptor strategy interface provides methods for each of those operations:

**public interface** ChannelInterceptor {

Message<?> preSend(Message<?> message, MessageChannel channel);

**void** postSend(Message<?> message, MessageChannel channel, **boolean** sent);

**void** afterSendCompletion(Message<?> message, MessageChannel channel, **boolean** sent, Exception ex);

**boolean** preReceive(MessageChannel channel);

Message<?> postReceive(Message<?> message, MessageChannel channel);

**void** afterReceiveCompletion(Message<?> message, MessageChannel channel, Exception ex);

}

Registering the interceptor with a channel:

channel.addInterceptor(someChannelInterceptor);

## Wire Tap

It is a simple interceptor that sends the Message to another channel without otherwise altering the existing flow. It can be very useful for debugging and monitoring.

**<int:channel id**=**"in">**

**<int:interceptors>**

**<int:wire-tap channel**=**"logger"/>**

**</int:interceptors>**

**</int:channel>**

**<int:logging-channel-adapter id**=**"logger" level**=**"DEBUG"/>**

Wire-tap as a component is not invoked asynchronously be default. Instead, Spring Integration focuses on a single unified approach to configuring asynchronous behavior: the Message Channel. What makes certain parts of the message flow *sync* or *async* is the type of Message Channel that has been configured within that flow.

Wire-tap component is ONLY responsible for performing the following 3 tasks:

* intercept a message flow by tapping into a channel (e.g., channelA)
* grab each message
* send the message to another channel (e.g., channelB)

It is essentially a variation of the Bridge, but it is encapsulated within a channel definition (and hence easier to enable and disable without disrupting a flow). Also, unlike the bridge, it basically forks another message flow. *Synchronous* or *asynchronous* depends on the type of Message Channel that channelB is.

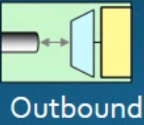
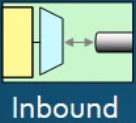
# Adapters

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 integration 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.



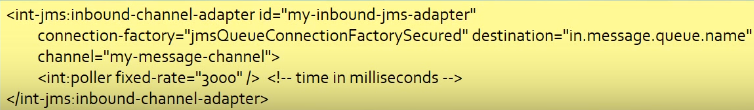
Built-in adapters:

* 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.

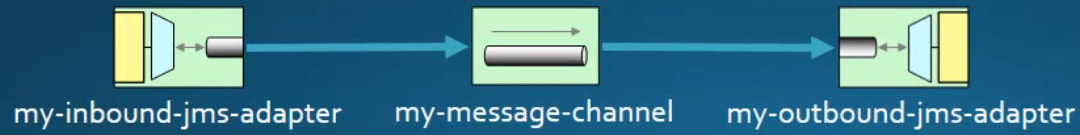


## A JMS Outbound Adapter

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:



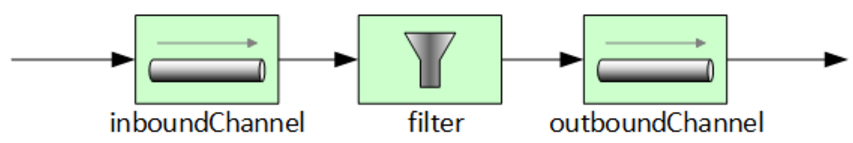
# Filters

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.

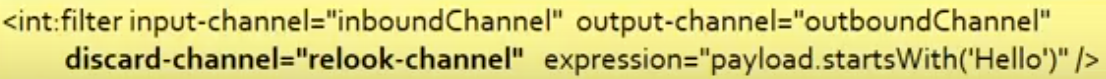
## Built-in filters

* 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*:

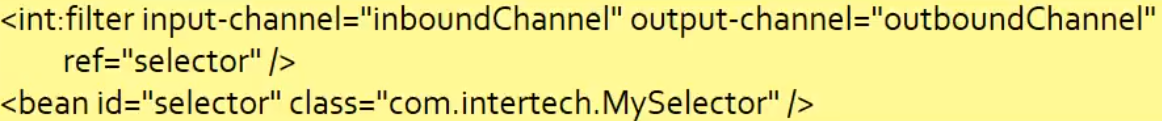


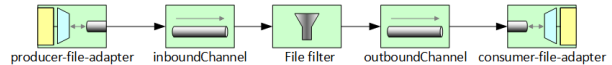
## Custom Filters

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

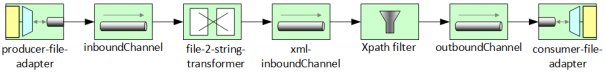
**public interface** MessageSelector **extends** GenericSelector<Message<?>>{  
**boolean** accept(Message<?> message);  
}

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

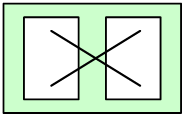




Or



# 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



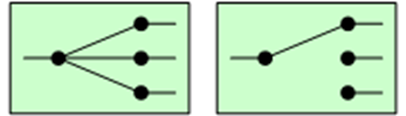
It takes a message with an object payload from the *inboundChannel*, calls toString on the object and puts the result string into the *outboundChannel*.

String to String transformer using Spring Expression Language(SpEL)



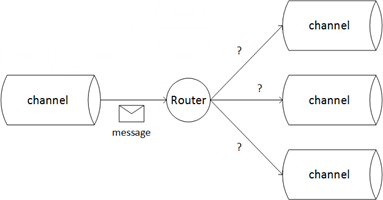


# Routers



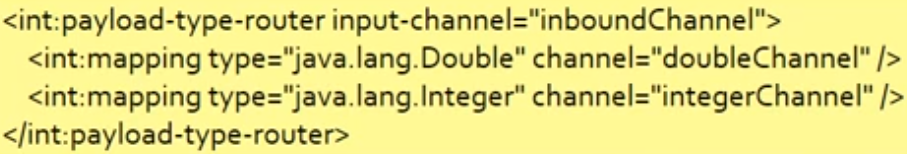
Routers distribute messages to one or more channels.

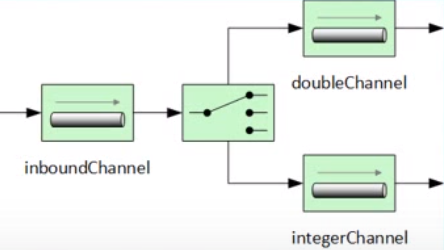
* Redipient list Routers: simply distribute the message to all listed message channels.
* Content Routers: examine the message payload or headers in order to select a particular destination message channel.



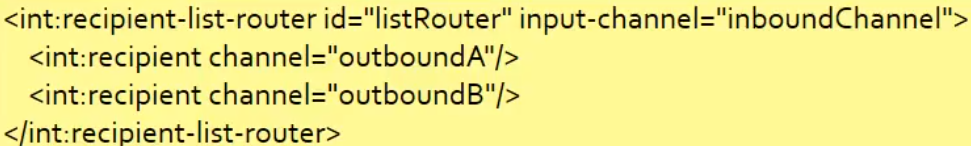
## Example

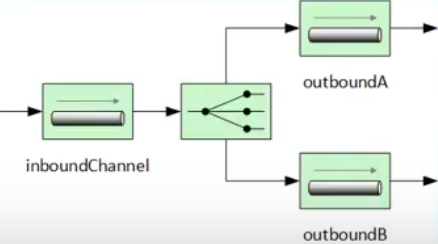
* Below is the configuration for a simple payload type content router:



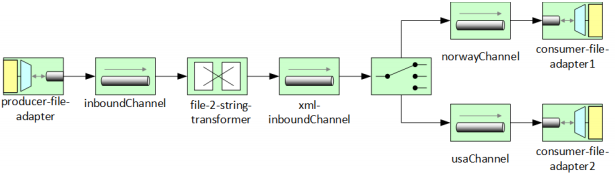


* Recipient list routers:

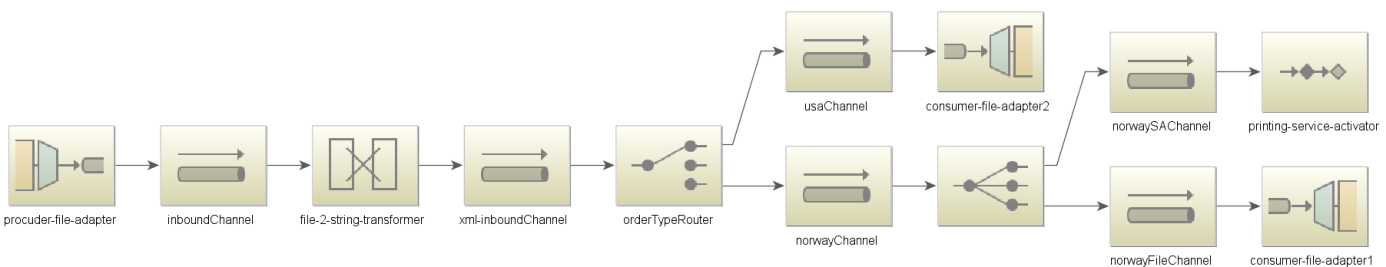




<**int-xml:xpath-router id="orderTypeRouter" input-channel="xml-inboundChannel"**>  
<**int-xml:xpath-expression expression="/shiporder/shipto/country"** />  
<**int-xml:mapping value="Norway" channel="norwayChannel"** />  
<**int-xml:mapping value="USA" channel="usaChannel"** />  
</**int-xml:xpath-router**>



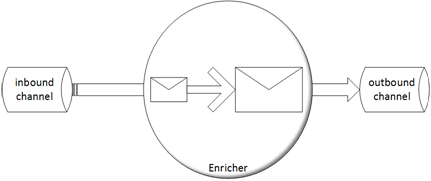
<**int:recipient-list-router input-channel="norwayChannel"**>  
<**int:recipient channel="norwayFileChannel"**/>  
<**int:recipient channel="norwaySAChannel"**/>  
</**int:recipient-list-router**>



# Enrichers

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

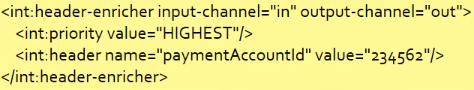
Enrichers is a special type of transformer. Enrichers take a message and enhance it by adding information to its header or payload.



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

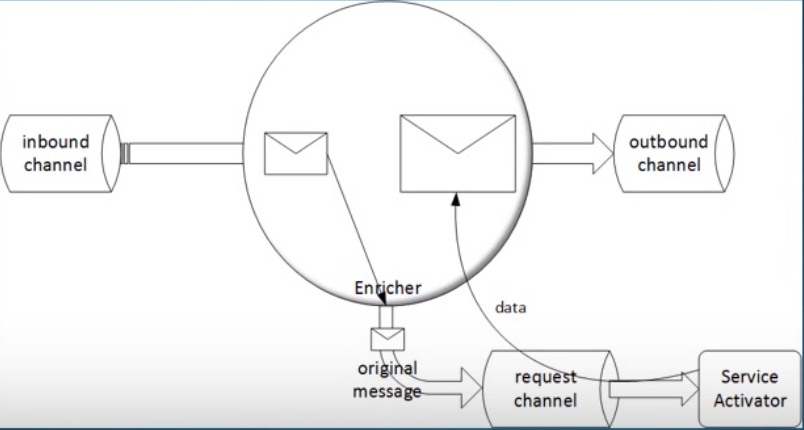
This example takes a message and adds a priority Header as well as a custom header called paymentAccountId.



Payloader enricher, add data to the message payload.

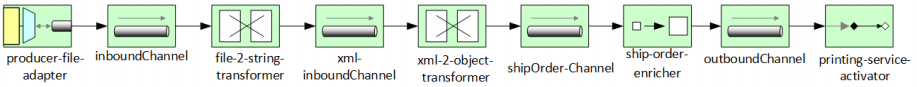


Custom enricher: that uses another Spring integration component. In this case is simple service activator to provide the data backed in the channel.



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

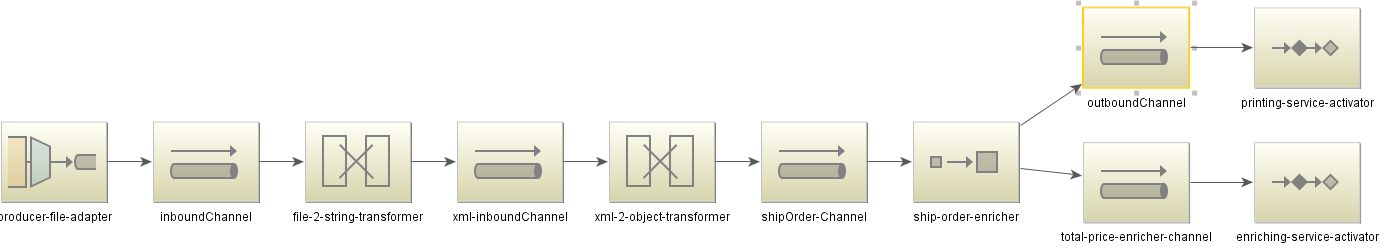
**output-channel="outboundChannel"**>  
<**int:property name="shipped" expression="true"**/>  
</**int:enricher**>



<**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"**/>



# Service Activators

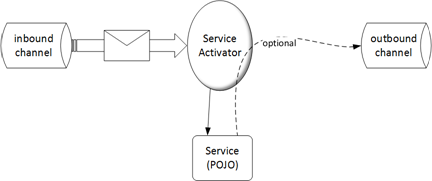
http://cdn.intertech.com/Blog/wp-content/uploads/2014/09/serviceactivator.png

The name of this endpoint aptly defines what it does. A service activator is an spring integration component that triggers (or activates) a Spring-managed service object or bean.

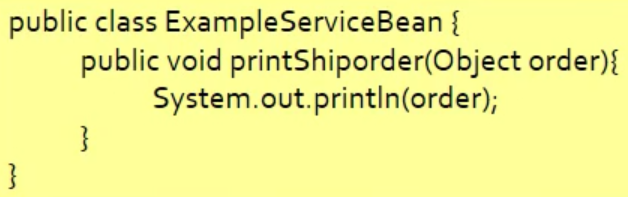
A service activator polls a message channel looking for messages. On the arrival of a message, it calls the processing method of the service bean(which is typically just a POJO).

The service activator's method is passed the message or the payload of the message base on the parameter type to the processing method. In fact, the service's processing method can be passed no data. In this case, the service activator is considered an event-style component that triggers processing just on the mere arrival of the message.

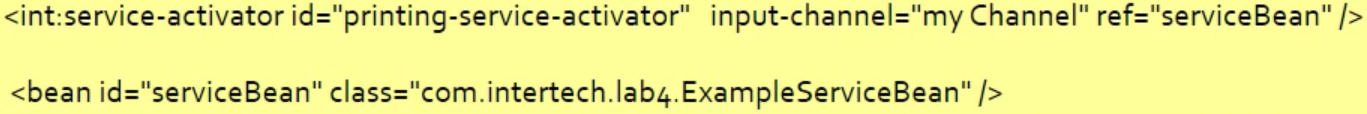
The service’s processing method may also optionally return a value (bundled up into a message) or message. The output, when returned, can be sent to an Spring Integration output channel.



## Example

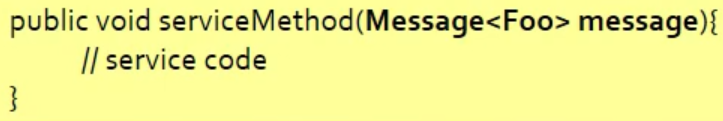


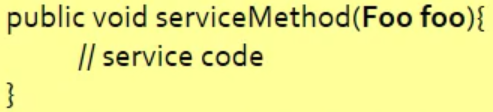
The service activator configuration must specify the message channel that it polls for messages and the class of the service bean.



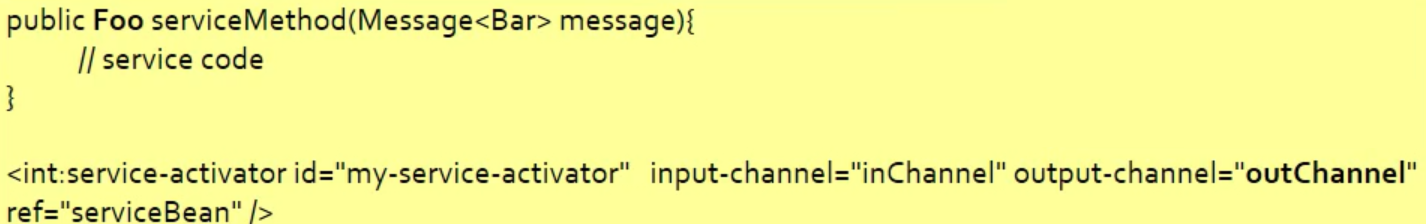
If there were more public methods in the service, the service activator would need a *method* attribute.

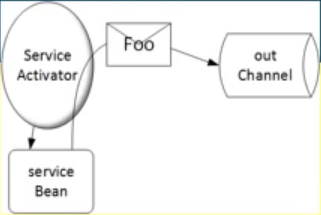
The parameter of the process method can be message (the incoming message self) or an arbitrary type(the payload of the message).

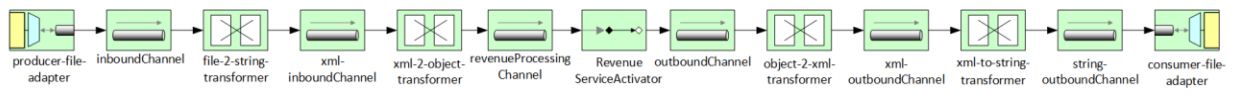




Process method with return value:



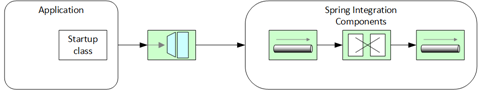




# Gateway

http://cdn.intertech.com/Blog/wp-content/uploads/2014/09/gateway.png

Gateways are a means of loosely coupling other application components from the Spring Integration API or other messaging API. The gateway serves as a facade to a spring integration system.



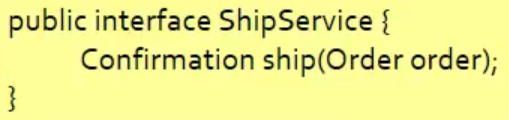
Gateways are defined by an interface.

* synchronous: causing the application to block and wait for the spring integration to respond.
* asynchronous: allowing the application to do other work while a long running spring integration process.

## How to use

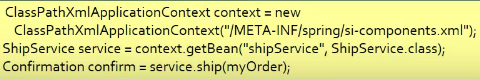
Applications must provide an interface to make requests of the spring integration system.

* The interface should be devoid of spring integration API to keep the application decoupled from spring integration.
* Spring Integration will implement the interface with a org.springframework.integration.gateway.GatewayProxyFactoryBean.



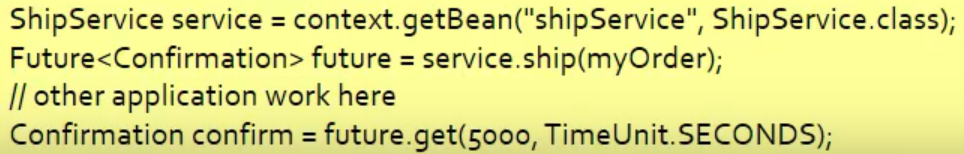
Next Configure the synchronous Gateway in the spring integration configuration.





Configuration asynchronous Gateway,





## Example

