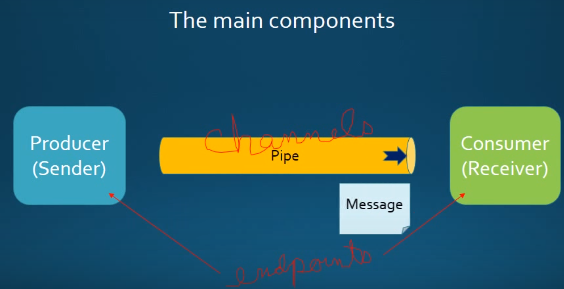
Spring Integration

# Introduction

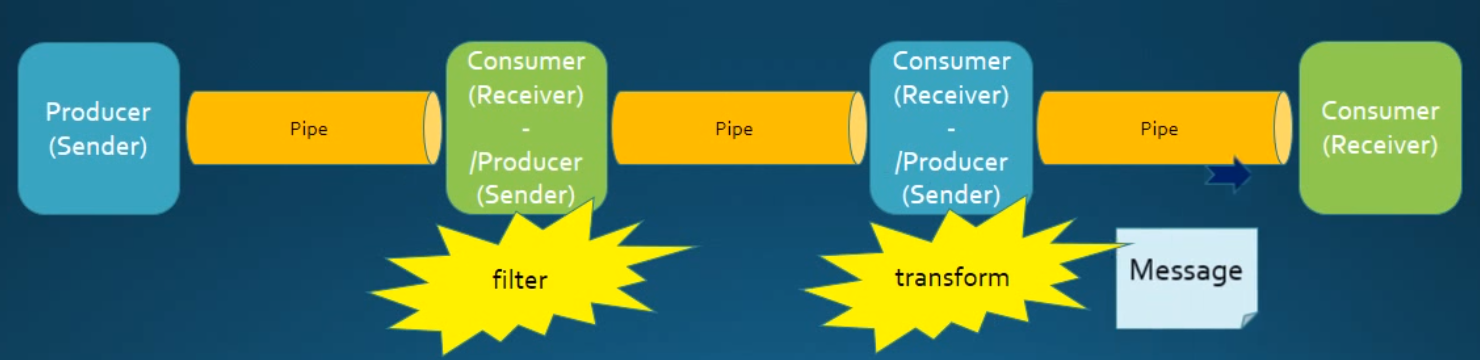
The main components: **Messages, Channels** and **Endpoints**.

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

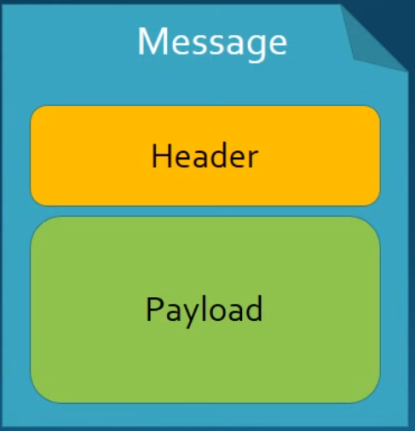
Messaging systems typically follow the similarly abstract*"pipes-and-filters"* model. The "filters" represent any component that is capable of producing and/orconsuming messages, and the "pipes" transport the messages between filters so that the componentsthemselves remain loosely-coupled. It is important to note that these two high-level paradigms are notmutually exclusive. The underlying messaging infrastructure that supports the "pipes" should still beencapsulated in a layer whose contracts are defined as interfaces. Likewise, the "filters" themselveswould 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 integrationsolutions through *inversion of control*. This means that you should not have to implement consumersand producers directly, and you should not even have to build Messages and invoke send or receiveoperations on a Message Channel. Instead, you should be able to focus on your specific domain modelwith 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.Thecomponents responsible for these connections are Message Endpoints.This does not mean that you willnecessarily connect your existing application code directly.The important thing is to achieve separation of concerns between such integration logic and businesslogic. The goal shouldbe 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, theendpoint’s primary role is to connect application code to the messaging framework and to do so in a noninvasivemanner. In other words, the application code should ideally have no awareness of the Messageobjects or the Message Channels. This is similar to the role of a Controller in the MVC paradigm. Just asa Controller handles HTTP requests, the Message Endpoint handles Messages. Just as Controllers aremapped to URL patterns, Message Endpoints are mapped to Message Channels. The goal is the samein 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
* **ServiceActivator**: 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.*

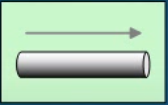
**boolean** send(Message<?> message);

*//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 pollto 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 itssubscribed handlers. This is most often used for sending *Event Messages* whose primary role isnotification. Note that the PublishSubscribeChannel is intended for sending only. Sinceit broadcasts to its subscribers directly when its` send(Message)` method is invoked, consumerscannot poll for Messages. Any subscriber must be a MessageHandler itself, and the subscriber’shandleMessage(Message) method will be invoked in turn.

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

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

To create a PublishSubscribeChannel, use the <publish-subscribe-channel/> element. When using this element, you can also specify the task-executor used for publishing Messages (if none is specified it simply publishes in the sender’s thread):

**<int:publish-subscribe-channel id**=**"pubsubChannel" task-executor**=**"someExecutor"/>**

If you are providing a Resequencer or Aggregator downstream from a PublishSubscribeChannel, then you can set the apply-sequence property on the channel to true. That will indicate that the channel should set the sequence-size and sequence-number Message headers as well as the correlation id prior to passing the Messages along.

**<int:publish-subscribe-channel id**=**"pubsubChannel" apply-sequence**=**"true"/>**

#### RendezvousChannel

The RendezvousChannel enables a "direct-handoff" scenario where a sender will block until anotherparty invokes the channel’s receive() method or vice-versa.The RendezvousChannel is also useful for implementing request-reply operations. The sendercan create a temporary, anonymous instance of RendezvousChannel which it then sets asthe replyChannel header when building a Message. After sending that Message, the sender canimmediately call receive (optionally providing a timeout value) in order to block while waiting for a replyMessage. This is very similar to the implementation used internally by many of Spring Integration’srequest-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 PollableChannelinterface, so it dispatches Messages directly to a subscriber. However,it differs from the PublishSubscribeChannel in that it will only send each Message to a singlesubscribed 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 configurationas DirectChannel. The key differencebetween these two dispatching channel types is that the ExecutorChannel delegates to an instanceof 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 notsupport 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 strategyinterface 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 interceptorthat sends the Message to another channel without otherwise altering the existing flow. It can be veryuseful 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 asa 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.

## MessagingTemplate

Spring Integration provides a MessagingTemplate that supports a variety of operations across the Message Channels, including request/reply scenarios.

MessagingTemplate template = **new** MessagingTemplate();

Message reply = template.sendAndReceive(someChannel, **new** GenericMessage(***"test"***));

A less invasive approach that allows you to invoke simple interfaces with payload and/or header values instead of Message instances is theGatewayProxyFactoryBean.

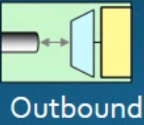
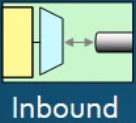
# Adapters

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

## Configuring an Inbound Channel Adapter

An "inbound-channel-adapter" element can invoke any method on a Spring-managed Object and send a non-null return value to a MessageChannel after converting it to a Message. When the adapter’s subscription is activated, a poller will attempt to receive messages from the source. The poller will be scheduled with the TaskScheduler according to the provided configuration. To configure the polling interval or cron expression for an individual channel-adapter, provide a poller element with one of the scheduling attributes, such as fixed-rate or cron.

**<int:inbound-channel-adapter ref**=**"source1" method**=**"method1" channel**=**"channel1">**

**<int:poller fixed-rate**=**"5000"/>**

**</int:inbound-channel-adapter>**

**<int:inbound-channel-adapter ref**=**"source2" method**=**"method2" channel**=**"channel2">**

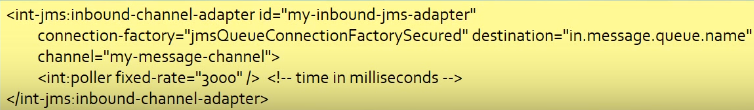
**<int:poller cron**=**"30 \* 9-17 \* \* MON-FRI"/>**

**</int:channel-adapter>**

If no poller is provided, then a single default poller must be registered within the context.

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.



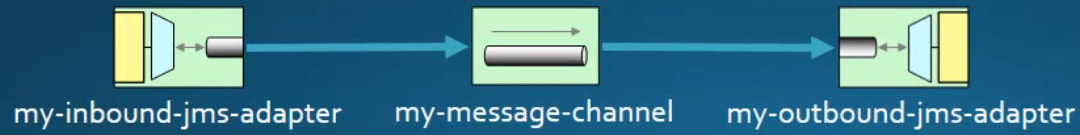
## Configuring An Outbound Channel Adapter

An "outbound-channel-adapter" element can also connect a MessageChannel to any POJO consumer method that should be invoked with the payload of Messages sent to that channel.

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:



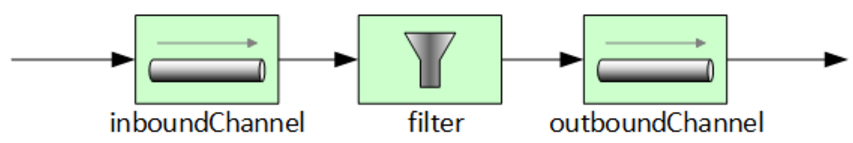
# Filters

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.

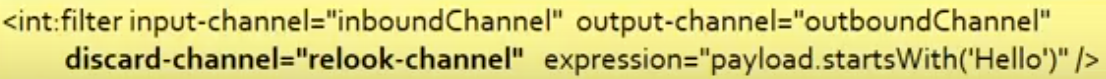
## Built-in filters

* 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*:

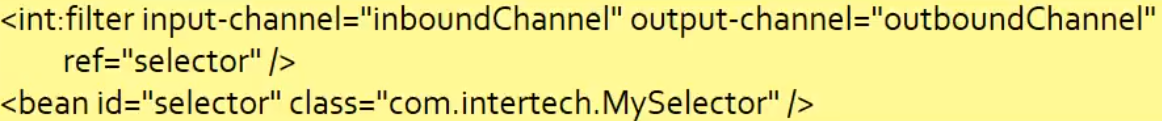


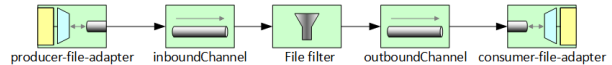
## Custom Filters

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

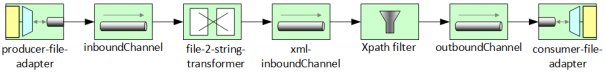
**public interface** MessageSelector **extends** GenericSelector<Message<?>>{  
**boolean** accept(Message<?> message);  
}

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

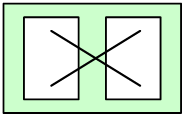




Or



# 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



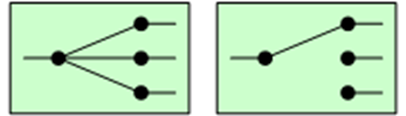
It takes a message with an object payload from the *inboundChannel*, calls toString on the object and puts the result string into the *outboundChannel*.

String to String transformer using Spring Expression Language(SpEL)





# Routers



Routers consume Messages from a Message Channel and distribute each consumed Message to one or more different Message Channels based upon the Message’s content and/or metadata available in the Message Headers.

Router is a sub-class of MessageHandler. It can therefore only take a SubscribableChannel as *Input-Channel*.

A Message Router is often used as a dynamic alternative to a statically configured output channel on a Service Activator or other endpoint capable of sending reply Messages.

## Out-Of-Box Routers Implementation

Spring Integration provides the following routers out-of-the-box.

### Payload Type Router

A PayloadTypeRouter will send Messages to the channel as defined by payload-type mappings.

**<bean id**=**"payloadTypeRouter" class**=**"org.springframework.integration.router.PayloadTypeRouter">**

**<property name**=**"channelMapping">**

**<map>**

**<entry key**=**"java.lang.String" value-ref**=**"stringChannel"/>**

**<entry key**=**"java.lang.Integer" value-ref**=**"integerChannel"/>**

**</map>**

**</property>**

**</bean>**

Or

**<int:payload-type-router input-channel**=**"routingChannel">**

**<int:mapping type**=**"java.lang.String" channel**=**"stringChannel" />**

**<int:mapping type**=**"java.lang.Integer" channel**=**"integerChannel" />**

**</int:payload-type-router>**

### Header Value Router

When a HeaderValueRouter is created it is initialized with the name of the header to be evaluated. The value of the header could be one of two things:

* Arbitrary value
* Channel name

If arbitrary then additional mappings for these header values to channel names is required, otherwise no additional configuration is needed.

Configuration where mapping of header values to channels is required:

**<int:header-value-router input-channel**=**"routingChannel" header-name**=**"testHeader">**

**<int:mapping value**=**"someHeaderValue" channel**=**"channelA" />**

**<int:mapping value**=**"someOtherHeaderValue" channel**=**"channelB" />**

**</int:header-value-router>**

During the resolution process this router may encounter channel resolution failures, causing an exception. If you want to suppress such exceptions and send unresolved messages to the default output channel (identified with the default-output-channel attribute) set resolution-required to false. Normally, messages for which the header value is not explicitly mapped to a channel will be sent to the default-output-channel. However, in cases where the header value is mapped to a channel name but the channel cannot be resolved, setting the resolution-required attribute to false will result in routing such messages to the default-output-channel.

Configuration where mapping of header values to channel names is not required since header values themselves represent channel names:

**<int:header-value-router input-channel**=**"routingChannel" header-name**=**"testHeader"/>**

### RecipientListRouter

A RecipientListRouter will send each received Message to a statically defined list of Message Channels:

**<bean id**=**"recipientListRouter" class**=**"org.springframework.integration.router.RecipientListRouter">**

**<property name**=**"channels">**

**<list>**

**<ref bean**=**"channel1"/>**

**<ref bean**=**"channel2"/>**

**<ref bean**=**"channel3"/>**

**</list>**

**</property>**

**</bean>**

<**int:router** **ref**=**"customRecipientListRouter" input-channel="routingChannel"**

**default-output-channel="errorChannel"** />

Namespace configuration:

**<int:recipient-list-router id**=**"customRouter" input-channel**=**"routingChannel" timeout**=**"1234"**

**ignore-send-failures**=**"true" apply-sequence**=**"true">**

**<int:recipient channel**=**"channel1"/>**

**<int:recipient channel**=**"channel2"/>**

**</int:recipient-list-router>**

The apply-sequence flag here has the same effect as it does for a publish-subscribe-channel, and like a publish-subscribe-channel, it is disabled by default on the recipient-list-router.

Another convenient option when configuring a RecipientListRouter is to use Spring Expression Language (SpEL) support as selectors for individual recipient channels.

**<int:recipient-list-router id**=**"customRouter" input-channel**=**"routingChannel">**

**<int:recipient channel**=**"channel1" selector-expression**=**"payload.equals('foo')"/>**

**<int:recipient channel**=**"channel2" selector-expression**=**"headers.containsKey('bar')"/>**

**</int:recipient-list-router>**

**RecipientListRouterManagement**

Starting with version 4.1, the RecipientListRouter provides several operation to manipulate with recipients dynamically at runtime. These management operations are presented by RecipientListRouterManagement @ManagedResource. They are available using Section “Control Bus” as well as via JMX.

**<control-bus input-channel**=**"controlBus"/>**

**<recipient-list-router id**=**"simpleRouter" input-channel**=**"routingChannelA">**

**<recipient channel**=**"channel1"/>**

**</recipient-list-router>**

**<channel id**=**"channel2"/>**

### Xpath Router

The XPath Router is part of the XML Module.

### Routing and Error handling

Spring Integration also provides a special type-based router called ErrorMessageExceptionTypeRouter for routing Error Messages (Messages whose payload is a Throwable instance). ErrorMessageExceptionTypeRouter is very similar to the PayloadTypeRouter. In fact they are almost identical. The only difference is that while PayloadTypeRouter navigates the instance hierarchy of a payload instance (e.g., payload.getClass().getSuperclass()) to find the most specific type/channel mappings, the ErrorMessageExceptionTypeRouter navigates the hierarchy of exception causes (e.g., payload.getCause()) to find the most specific Throwable type/channel mappings.

**<int:exception-type-router input-channel**=**"inputChannel"**

**default-output-channel**=**"defaultChannel">**

**<int:mapping exception-type**=**"java.lang.IllegalArgumentException" channel**=**"illegalChannel"/>**

**<int:mapping exception-type**=**"java.lang.NullPointerException" channel**=**"npeChannel"/>**

**</int:exception-type-router>**

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

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

## Configuring (Generic) Router

**<int:router ref**=**"payloadTypeRouter" input-channel**=**"input1"**

**default-output-channel**=**"defaultOutput1"/>**

**<int:router ref**=**"recipientListRouter" input-channel**=**"input2"**

**default-output-channel**=**"defaultOutput2"/>**

**<int:router ref**=**"customRouter" input-channel**=**"input3"**

**default-output-channel**=**"defaultOutput3"/>**

**<beans:bean id**=**"customRouterBean" class**=**"org.foo.MyCustomRouter"/>**

**<int:router input-channel**=**"input" ref**=**"somePojo" method**=**"someMethod"/>**

Using a ref attribute is generally recommended if the custom router implementation is referenced in other <router> definitions. However if the custom router implementation should be scoped to a single definition of the <router>, you may provide an inner bean definition:

**<int:router method**=**"someMethod" input-channel**=**"input3"**

**default-output-channel**=**"defaultOutput3">**

**<beans:bean class**=**"org.foo.MyCustomRouter"/>**

**</int:router>**

Generally a SpEL expression is evaluated and the result is mapped to a channel:

**<int:router input-channel**=**"inChannel" expression**=**"payload.paymentType">**

**<int:mapping value**=**"CASH" channel**=**"cashPaymentChannel"/>**

**<int:mapping value**=**"CREDIT" channel**=**"authorizePaymentChannel"/>**

**<int:mapping value**=**"DEBIT" channel**=**"authorizePaymentChannel"/>**

**</int:router>**

To simplify things even more, the SpEL expression may evaluate to a channel name:

**<int:router input-channel**=**"inChannel" expression**=**"payload + 'Channel'"/>**

Another value of SpEL for configuring routers is that an expression can actually return a Collection, effectively making every <router> a *Recipient List Router*. Whenever the expression returns multiple channel values the Message will be forwarded to each channel.

**<int:router input-channel**=**"inChannel" expression**=**"headers.channels"/>**

In the above configuration, if the Message includes a header with the name channels the value of which is a List of channel names then the Message will be sent to each channel in the list. You may also find Collection Projection(**![projectionExpression]**, e.g. *"Members.![placeOfBirth.city]"*) and Collection Selection(**?[selectionExpression]**, e.g. *"Members.?[Nationality == 'Serbian']"*) expressions useful to select multiple channels.

## Dynamic Routers

PayloadTypeRouter, HeaderValueRouter usw. all require *static configuration*. The Dynamic Router pattern describes the mechanisms by which one can change/configure routers dynamically without bringing down the system or individual routers.

Typical flow of router:

1. Compute channel identifier which is a value calculated by the router once it receives the Message.
2. Resolve channel identifier to channel name.
3. Resolve channel name to the actual MessageChannel

**<int:header-value-router input-channel="inputChannel" header-name="testHeader">**

In this example the Step 2 is optional. If mapping is not defined then the channel identifier value will automatically be treated as the channel name, which will be resolved to the actual MessageChannel as in Step 3.

Any router that is a subclass of the AbstractMappingMessageRouter (which includes most framework defined routers) is a Dynamic Router simply because the channelMapping is defined at the AbstractMappingMessageRouter level. That map’s setter method is exposed as a public method along with setChannelMapping and removeChannelMapping methods. These allow you to change/add/remove router mappings at runtime as long as you have a reference to the router itself.

It also means that you could expose these same configuration options via JMX (see Section 9.2, “JMX Support”) or the Spring Integration ControlBus (see Section 9.6, “Control Bus”) functionality.

### Manage Router Mappings using the Control Bus

One way to manage the router mappings is through the Control Bus pattern which exposes a Control Channel where you can send control messages to manage and monitor Spring Integration components, including routers.

### Manage Router Mappings using JMX

You can also expose a router instance with Spring’s JMX support, and then use your favorite JMX client (e.g., JConsole) to manage those operations (methods) for changing the router’s configuration.

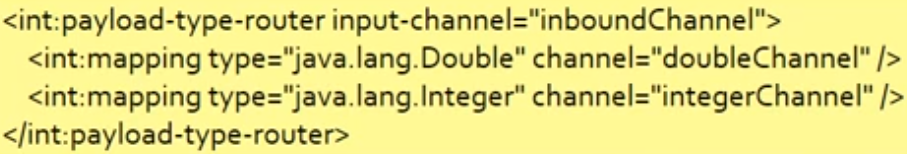
### Routing Slip

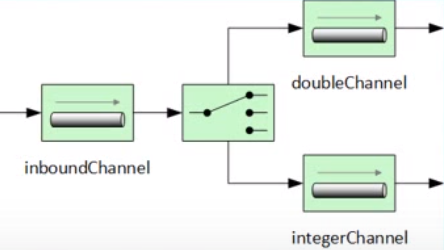
### Process Manager Enterprise Integration Pattern

The EIP also defines the Process Manager pattern. This pattern can now easily be implemented using custom Process Manager logic encapsulated in a RoutingSlipRouteStrategy within the routing slip.

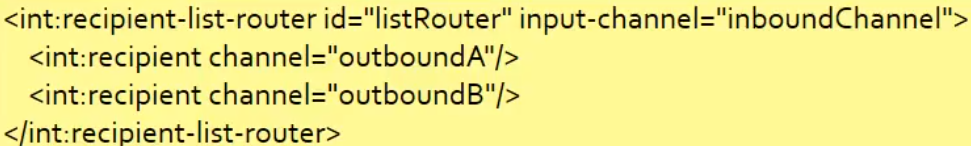
## Example

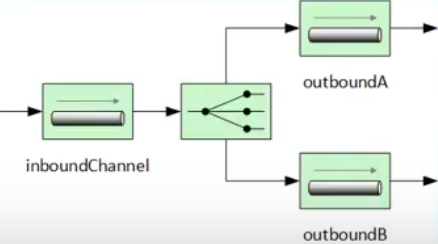
* Below is the configuration for a simple payload type content router:



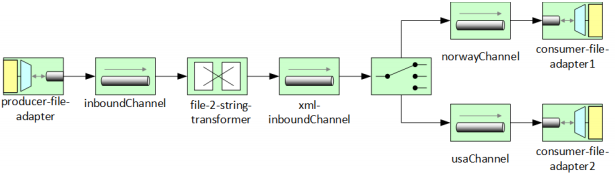


* Recipient list routers:

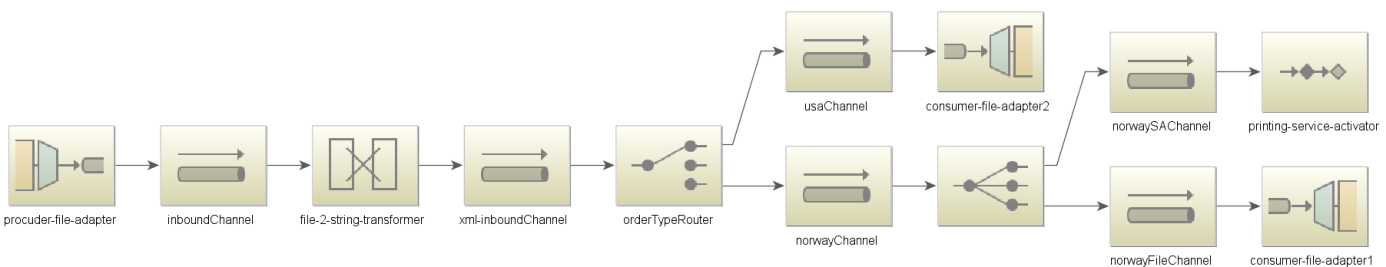




<**int-xml:xpath-router id="orderTypeRouter" input-channel="xml-inboundChannel"**>  
<**int-xml:xpath-expression expression="/shiporder/shipto/country"** />  
<**int-xml:mapping value="Norway" channel="norwayChannel"** />  
<**int-xml:mapping value="USA" channel="usaChannel"** />  
</**int-xml:xpath-router**>



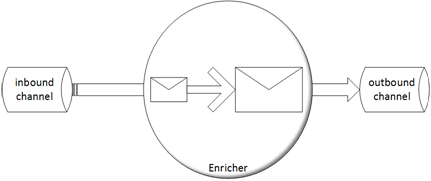
<**int:recipient-list-router input-channel="norwayChannel"**>  
 <**int:recipient channel="norwayFileChannel"**/>  
 <**int:recipient channel="norwaySAChannel"**/>  
</**int:recipient-list-router**>



# Enrichers

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

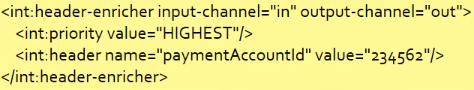
Enrichers is a special type of transformer. Enrichers take a message and enhance it by adding information to its header or payload.



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

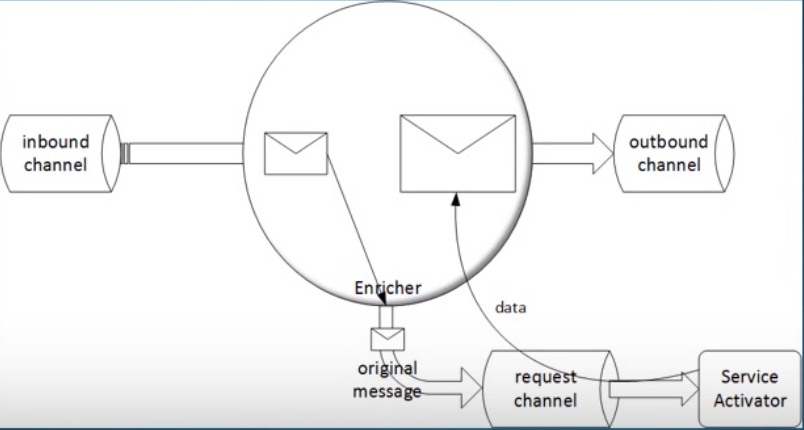
This example takes a message and adds a priority Header as well as a custom header called paymentAccountId.



Payloader enricher, add data to the message payload.

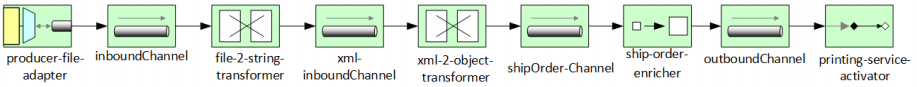


Custom enricher: that uses another Spring integration component. In this case is simple service activator to provide the data backed in the channel.



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

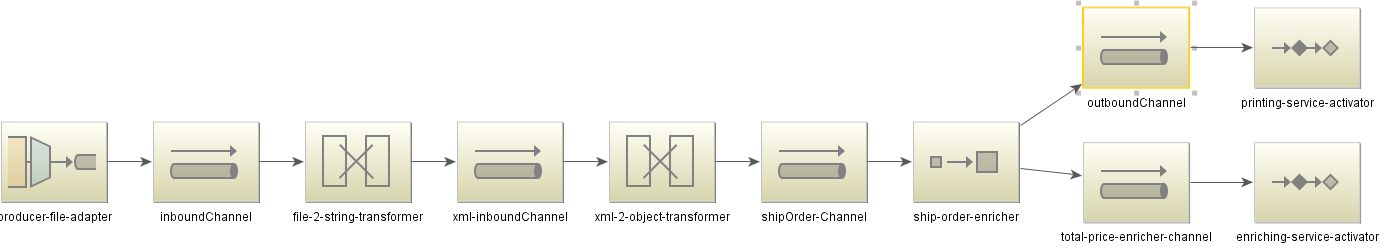
**output-channel="outboundChannel"**>  
<**int:property name="shipped" expression="true"**/>  
</**int:enricher**>



<**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

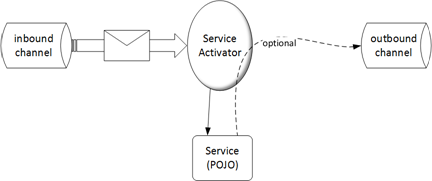
http://cdn.intertech.com/Blog/wp-content/uploads/2014/09/serviceactivator.png

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.

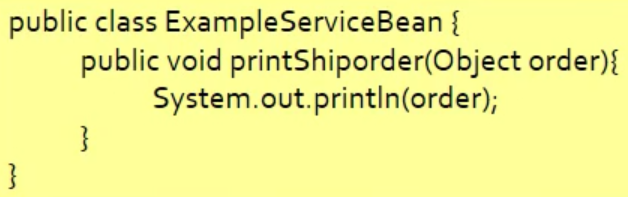
A service activator polls a message channel looking for messages. On the arrival of a message, it calls the processing method of the service bean(which is typically just a POJO).

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.

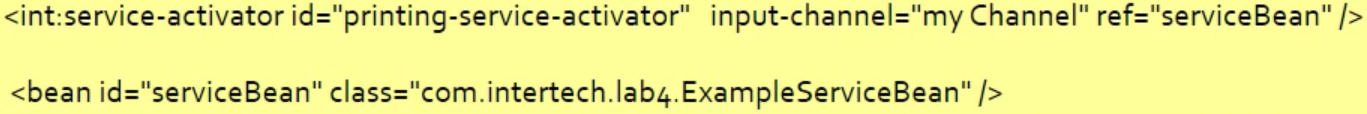
The service’s processing method may also optionally return a value (bundled up into a message) or message. The output, when returned, can be sent to an Spring Integration output channel.



## Example

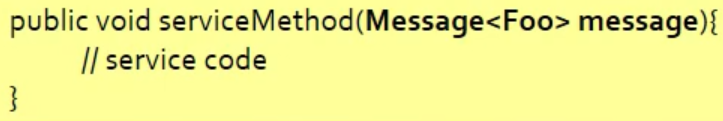


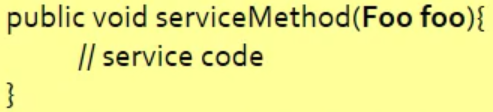
The service activator configuration must specify the message channel that it polls for messages and the class of the service bean.



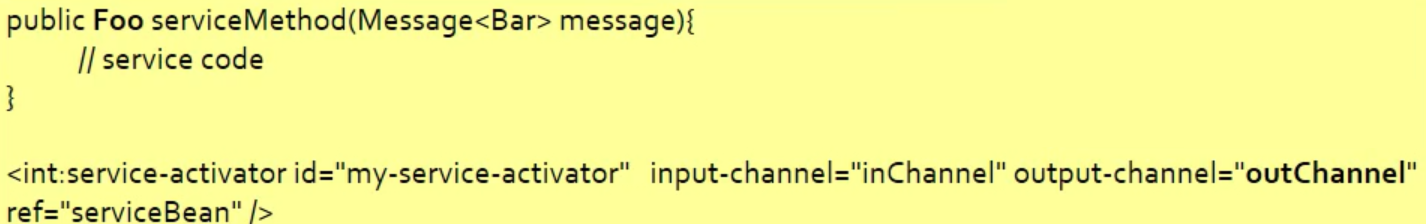
If there were more public methods in the service, the service activator would need a *method* attribute.

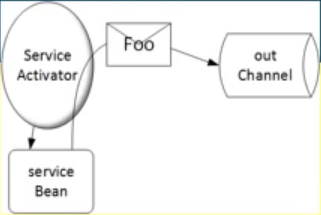
The parameter of the process method can be message (the incoming message self) or an arbitrary type(the payload of the message).

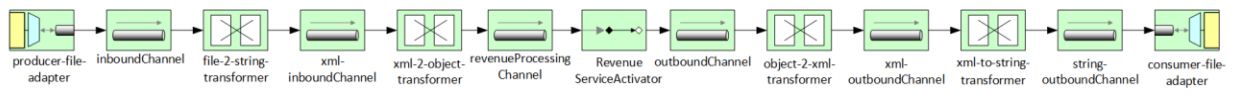




Process method with return value:



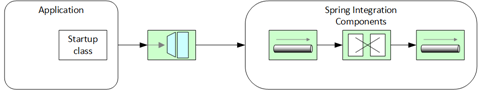




# Gateway

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.



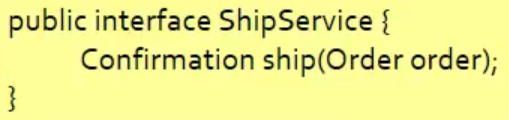
Gateways are defined by an interface.

* 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

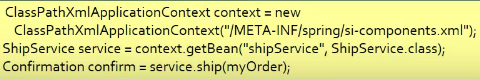
Applications must provide an interface to make requests of the spring integration system.

* 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.



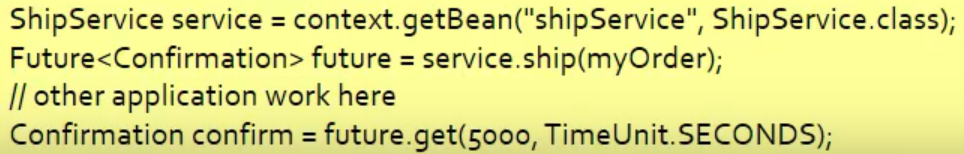
Next Configure the synchronous Gateway in the spring integration configuration.



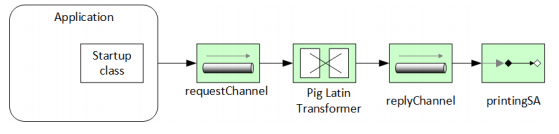


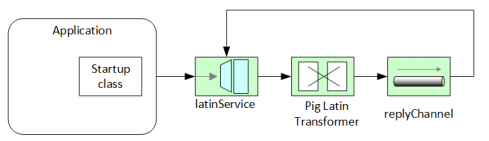
Configuration asynchronous Gateway,





## Example





# Message Handler Chain

# Scatter-Gather

# Thread Barrier

# Message Endpoints

Message Endpoints are responsible for connecting the various messaging components to channels. Sending Messages is quite straightforward. However, receiving is a bit more complicated. The main reason is that there are two types of consumers: *Polling Consumers* and *Event Driven Consumers*.

Event Driven Consumers are much simpler. Without any need to manage and schedule a separate poller thread, they are essentially just listeners with a callback method.

When connecting to one of Spring Integration’s subscribable Message Channels, this simple option works great. However, when connecting to a buffering, pollable Message Channel, some component has to schedule and manage the polling thread(s).

## Message Handler

MessageHandler interface is implemented by many of the components within the framework. In other words, this is not part of the public API. It is used by a Message Consumer for actually handling the consumed Messages. It provides also the foundation ofr most of the components(Routers, Transformers, Splitters, Aggregators, Service Activators, etc).

Spring Integration provides two endpoint implementations that host these callback-based handlers and allow them to be connected to Message Channels.

## Event Driven Consumer

This class accepts a SubscribableChannel und a MessageHandler:

SubscribableChannel inputChannel = context.getBean(**"subscribableChannel"**, SubscribableChannel.**class**);  
EventDrivenConsumer consumer = **new** EventDrivenConsumer(inputChannel, messageHandler);

And the subscribe() method in SubscribableChannel accepts the handler:

inputChannel.subscribe(messageHandler);

## Polling Consumer

It can be instantiated in the same way except that the input channel must implement PollableChannel:

PollableChannel inputChannel = context.getBean(**"pollableChannel"**, PollableChannel.**class**);  
PollingConsumer consumer = **new** PollingConsumer(inputChannel, messageHandler);

### Trigger Configuration

The trigger is a required property. There are three types of Trigger:

PeriodicTrigger, DynamicPeriodicTrigger and CronTrigger.

The PeriodicTrigger is typically defined with a simple interval (in milliseconds), but also supports an initialDelay property and a boolean fixedRate property (the default is false, i.e. fixed delay. If it is true, fixed Rate without delay)

consumer.setTrigger(**new** PeriodicTrigger(30, TimeUnit.***SECONDS***));

trigger.setInitialDelay(5000);  
trigger.setFixedRate(**true**);

Over DynamicPeriodicTrigger you can additionally set the perid at runtime.

The CronTrigger simply requires a valid cron expression:

CronTrigger trigger = **new** CronTrigger(**"\*/10 \* \* \* \* MON-FRI"**);

### MaxMessagesPerPoll

The maxMessagesPerPoll property specifies the maximum number of messages to receive within agiven poll operation. This means that the poller will continue calling receive() without waiting until either null is returned or that max is reached.

PollingConsumer consumer = **new** PollingConsumer(channel, handler);

consumer.setMaxMessagesPerPoll(10);

### ReceiveTimeout

The receiveTimeout property specifies the amount of time the poller should wait if no messages are available when it invokes the receive operation.

PollingConsumer consumer = **new** PollingConsumer(channel, handler);

consumer.setReceiveTimeout(5000);

### TaskExecutor

A Polling Consumer may also delegate to a Spring TaskExecutor:

PollingConsumer consumer = **new** PollingConsumer(channel, handler);

TaskExecutor taskExecutor = context.getBean(***"exampleExecutor"***, TaskExecutor.**class**);

consumer.setTaskExecutor(taskExecutor);

### AdviceChain

Furthermore, a PollingConsumer has a property called adviceChain. This property allows you to specify a List of AOP Advices for handling additional cross cutting concerns including transactions.

These advices are applied around the doPoll() method.

Spring Integration also provides a FactoryBean called ConsumerEndpointFactoryBean that creates the appropriate consumer type based on the type of channel, and there is full XML namespace support to even further hide those details.

### Namespace Support

**<int:poller cron**=**""**

default="false"

error-channel=""

fixed-delay=""

fixed-rate=""

id=""

max-messages-per-poll=""

receive-timeout=""

ref=""

task-executor=""

time-unit="MILLISECONDS" **11**

trigger=""> **12**

**<int:advice-chain /> 13**

**<int:transactional /> 14**

**</int:poller>**

It is also possible to create top-level pollers in which case only a ref is required:

**<int:poller id**=**"weekdayPoller" default**=**"true" cron**=**"\*/10 \* \* \* \* MON-FRI"/>**

**<int:transformer input-channel**=**"pollable" ref**=**"transformer"**

**output-channel**=**"output">**

**<int:poller ref**=**"weekdayPoller"/>**

**</int:transformer>**

*<!-- No <poller/> sub-element is necessary since there is a default -->*

**<int:transformer input-channel**=**"pollable" ref**=**"transformer"**

**output-channel**=**"output"/>**

Transaction Support:

**<int:poller fixed-delay**=**"1000">**

**<int:transactional transaction-manager**=**"txManager" propagation**=**"REQUIRED"**

**isolation**=**"REPEATABLE\_READ" timeout**=**"10000" read-only**=**"false"/>**

**</int:poller>**

AOP Advice Chain:

**<int:service-activator id**=**"advicedSa" input-channel**=**"goodInputWithAdvice"**

**ref**=**"testBean" method**=**"good" output-channel**=**"output">**

**<int:poller max-messages-per-poll**=**"1" fixed-rate**=**"10000">**

**<int:advice-chain>**

**<ref bean**=**"adviceA" />**

**<beans:bean class**=**"org.bar.SampleAdvice" />**

**<ref bean**=**"txAdvice" />**

**</int:advice-chain>**

**</int:poller>**

**</int:service-activator>**

TaskExecutor:

**<int:poller task-executor**=**"pool" fixed-rate**=**"1000"/>**

**<task:executor id**=**"pool" pool-size**=**"5-25" queue-capacity**=**"20" keep-alive**=**"120"/>**

## Endpoint Roles

Endpoints can be assigned to roles. Roles allow endpoints to be started and stopped as a group; this is particularly useful when using leadership election where a set of endpoints can be started or stopped when leadership is granted or revoked respectively.

**<int:inbound-channel-adapter id**=**"ica" channel**=**"someChannel" expression**=**"'foo'"**

**role**=**"cluster">**

**<int:poller fixed-rate**=**"60000" />**

**</int:inbound-channel-adapter>**

@Bean

@ServiceActivator(inputChannel = "sendAsyncChannel")

@Role("cluster")

**public** MessageHandler sendAsyncHandler() {

**return** *// some MessageHandler*

}

@Payload("#args[0].toLowerCase()")

@Role("cluster")

**public** String handle(String payload) {

**return** payload.toUpperCase();

}

@Autowired

**private** SmartLifecycleRoleController roleController;

...

**this**.roleController.addSmartLifeCycleToRole(***"cluster"***, someEndpoint);

...

Invoking roleController.startLifecyclesInRole("cluster") (and the corresponding stop... method) will start/stop the endpoints.

Any object implementing SmartLifecycle can be programmatically added, not just endpoints.