4. Messaging Channels

4.1 Message Channels

While the `Message` plays the crucial role of encapsulating data, it is the `MessageChannel` that decouples message producers from message consumers.

4.1.1 The MessageChannel Interface

Spring Integration's top-level `MessageChannel` interface is defined as follows.

```
public interface MessageChannel {

    boolean send(Message message);

    boolean send(Message message, long timeout);
}
```

When sending a message, the return value will be *true* if the message is sent successfully. If the send call times out or is interrupted, then it will return *false*.

PollableChannel

Since Message Channels may or may not buffer Messages (as discussed in the overview), there are two sub-interfaces defining the buffering (pollable) and non-buffering (subscribable) channel behavior. Here is the definition of `PollableChannel`.

```
public interface PollableChannel extends MessageChannel {

    Message<?> receive();

    Message<?> receive(long timeout);

}
```

Similar to the send methods, when receiving a message, the return value will be *null* in the case of a timeout or interrupt.

SubscribableChannel

The `SubscribableChannel` base interface is implemented by channels that send Messages directly to their subscribed `MessageHandler` s. Therefore, they do not provide receive methods for polling, but instead define methods for managing those subscribers:

```
public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);

}
```

4.1.2 Message Channel Implementations

Spring Integration provides several different Message Channel implementations. Each is briefly described in the sections below.

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 as opposed to *Document Messages* which are generally intended to be processed by a single handler. 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 (it does not implement `PollableChannel` and therefore has no `receive()` method). Instead, any subscriber must be a `MessageHandler` itself, and the subscriber's `handleMessage(Message)` method will be invoked in turn.

Prior to version 3.0, invoking the send method on a `PublishSubscribeChannel` that had no subscribers returned `false`. When used in conjunction with a `MessagingTemplate`, a `MessageDeliveryException` was thrown. 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.

QueueChannel

The `QueueChannel` implementation wraps a queue. Unlike the `PublishSubscribeChannel`, the `QueueChannel` has point-to-point semantics. In other words, even if the channel has multiple consumers, only one of them should receive any Message sent to that channel. It provides a default no-argument constructor (providing an essentially unbounded capacity of `Integer.MAX_VALUE`) as well as a constructor that accepts the queue capacity:

```
public QueueChannel(int capacity)
```

A channel that has not reached its capacity limit will store messages in its internal queue, and the `send()` method will return immediately even if no receiver is ready to handle the message. If the queue has reached capacity, then the sender will block until room is available. Or, if using the send call that accepts a timeout, it will block until either room is available or the timeout period elapses, whichever occurs first. Likewise, a receive call will return immediately if a message is available on the queue, but if the queue is empty, then a receive call may block until either a message is available or the timeout elapses. In either case, it is possible to force an immediate return regardless of the queue's state by passing a timeout value of 0. Note however, that calls to the no-arg versions of `send()` and `receive()` will block indefinitely.

PriorityChannel

Whereas the `QueueChannel` enforces first-in/first-out (FIFO) ordering, the `PriorityChannel` is an alternative implementation that allows for messages to be ordered within the channel based upon a priority. By default the priority is determined by the *`priority`* header within each message. However, for custom priority determination logic, a comparator of type `Comparator<Message<?>>` can be provided to the `PriorityChannel`'s constructor.

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. Internally, this implementation is quite similar to the `QueueChannel` except that it uses a `SynchronousQueue` (a zero-capacity implementation of `BlockingQueue`). This works well in situations where the sender and receiver are operating in different threads but simply dropping the message in a queue asynchronously is not appropriate. In other words, with a `RendezvousChannel` at least the sender knows that some receiver has accepted the

message, whereas with a `QueueChannel`, the message would have been stored to the internal queue and potentially never received.

Keep in mind that all of these queue-based channels are storing messages in-memory only by default. When persistence is required, you can either provide a *message-store* attribute within the *queue* element to reference a persistent MessageStore implementation, or you can replace the local channel with one that is backed by a persistent broker, such as a JMS-backed channel or Channel Adapter. The latter option allows you to take advantage of any JMS provider's implementation for message persistence, and it will be discussed in Chapter 21, *JMS Support*. However, when buffering in a queue is not necessary, the simplest approach is to rely upon the `DirectChannel` discussed next.

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.

DirectChannel

The `DirectChannel` has point-to-point semantics but otherwise is more similar to the `PublishSubscribeChannel` than any of the queue-based channel implementations described above. It implements the `SubscribableChannel` interface instead of the `PollableChannel` interface, so it dispatches Messages directly to a subscriber. As a point-to-point channel, however, it differs from the `PublishSubscribeChannel` in that it will only send each Message to a *single* subscribed `MessageHandler`.

In addition to being the simplest point-to-point channel option, one of its most important features is that it enables a single thread to perform the operations on "both sides" of the channel. For example, if a handler is subscribed to a `DirectChannel`, then sending a Message to that channel will trigger invocation of that handler's `handleMessage(Message)` method *directly in the sender's thread*, before the send() method invocation can return.

The key motivation for providing a channel implementation with this behavior is to support transactions that must span across the channel while still benefiting from the abstraction and loose coupling that the channel provides. If the send call is invoked within the scope of a transaction, then the outcome of the handler's invocation (e.g. updating a database record) will play a role in determining the ultimate result of that transaction (commit or rollback).

Since the `DirectChannel` is the simplest option and does not add any additional overhead that would be required for scheduling and managing the threads of a poller, it is the default channel type within Spring Integration. The general idea is to define the channels for an application and then to consider which of those need to provide buffering or to throttle input, and then modify those to be queue-based `PollableChannels`. Likewise, if a channel needs to broadcast messages, it should not be a `DirectChannel` but rather a `PublishSubscribeChannel`. Below you will see how each of these can be configured.

The `DirectChannel` internally delegates to a Message Dispatcher to invoke its subscribed Message Handlers, and that dispatcher can have a load-balancing strategy exposed via *load-balancer* or *load-balancer-ref* attributes (mutually exclusive). The load balancing strategy is used by the Message Dispatcher to help determine how Messages are distributed amongst Message Handlers in the case that there are multiple Message Handlers subscribed to the same channel. As a convenience the *load-balancer* attribute exposes enumeration of values pointing to pre-existing implementations of `LoadBalancingStrategy`. The "round-robin" (load-balances across the handlers in rotation) and "none" (for the cases where one wants to explicitely disable load balancing) are the only available values. Other strategy implementations may be added in future versions. However, since version 3.0 you can provide your own implementation of the `LoadBalancingStrategy` and inject it using *load-balancer-ref* attribute which should point to a bean that implements `LoadBalancingStrategy`.

```
<int:channel id="lbRefChannel">
  <int:dispatcher load-balancer-ref="lb"/>
</int:channel>

<bean id="lb" class="foo.bar.SampleLoadBalancingStrategy"/>
```

Note that *load-balancer* or *load-balancer-ref* attributes are mutually exclusive.

The load-balancing also works in combination with a boolean *failover* property. If the "failover" value is true (the default), then the dispatcher will fall back to any subsequent handlers as necessary when preceding handlers throw Exceptions. The order is determined by an optional order value defined on the handlers themselves or, if no such value exists, the order in which the handlers are subscribed.

If a certain situation requires that the dispatcher always try to invoke the first handler, then fallback in the same fixed order sequence every time an error occurs, no load-balancing strategy should be provided. In other words, the dispatcher still supports the failover boolean property even when no load-balancing is enabled. Without load-balancing, however, the invocation of handlers will always begin with the first according to their order. For example, this approach works well when there is a clear definition of primary, secondary, tertiary, and so on. When using the namespace support, the "order" attribute on any endpoint will determine that order.

Keep in mind that load-balancing and failover only apply when a channel has more than one subscribed Message Handler. When using the namespace support, this means that more than one endpoint shares the same channel reference in the "input-channel" attribute.

ExecutorChannel

The `ExecutorChannel` is a point-to-point channel that supports the same dispatcher configuration as `DirectChannel` (load-balancing strategy and the failover boolean property). 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*.

Note that there are occasions where the sender may block. For example, when using a TaskExecutor with a rejection-policy that throttles back on the client (such as the `ThreadPoolExecutor.CallerRunsPolicy`), the sender's thread will execute the method directly anytime the thread pool is at its maximum capacity and the executor's work queue is full. Since that situation would only occur in a non-predictable way, that obviously cannot be relied upon for transactions.

Scoped Channel

Spring Integration 1.0 provided a `ThreadLocalChannel` implementation, but that has been removed as of 2.0. Now, there is a more general way for handling the same requirement by simply adding a "scope" attribute to a channel. The value of the attribute can be any name of a Scope that is available within the context. For example, in a web environment, certain Scopes are available, and any custom Scope implementations can be registered with the context. Here's an example of a ThreadLocal-based scope being applied to a channel, including the registration of the Scope itself.

```
<int:channel id="threadScopedChannel" scope="thread">
     <int:queue />
</int:channel>

<bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
    <property name="scopes">
        <map>
            <entry key="thread" value="org.springframework.context.support.SimpleThreadScope" />
        </map>
    </property>
</bean>
```

The channel above also delegates to a queue internally, but the channel is bound to the current thread, so the contents of the queue are as well. That way the thread that sends to the channel will later be able to receive those same Messages, but no other thread would be able to access them. While thread-scoped channels are rarely needed, they can be useful in situations where `DirectChannels` are being used to enforce a single thread of operation but any reply Messages should be sent to a "terminal" channel. If that terminal channel is thread-scoped, the original sending thread can collect its replies from it.

Now, since any channel can be scoped, you can define your own scopes in addition to Thread Local.

4.1.3 Channel Interceptors

One of the advantages of a messaging architecture is the ability to provide common behavior and capture meaningful information about the messages passing through the system in a non-invasive way. Since the `Message` s are being sent to and received from `MessageChannels`, those channels provide an opportunity for intercepting the send and receive operations. 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);
}
```

After implementing the interface, registering the interceptor with a channel is just a matter of calling:

```
channel.addInterceptor(someChannelInterceptor);
```

The methods that return a Message instance can be used for transforming the Message or can return *null* to prevent further processing (of course, any of the methods can throw a RuntimeException). Also, the `preReceive` method can return `false` to prevent the receive operation from proceeding.

Keep in mind that `receive()` calls are only relevant for `PollableChannels`. In fact the `SubscribableChannel` interface does not even define a `receive()` method. The reason for this is that when a Message is sent to a `SubscribableChannel` it will be sent directly to one or more subscribers depending on the type of channel (e.g. a PublishSubscribeChannel sends to all of its subscribers). Therefore, the `preReceive(..)`, `postReceive(..)` and `afterReceiveCompletion(..)` interceptor methods are only invoked when the interceptor is applied to a `PollableChannel`.

Spring Integration also provides an implementation of the Wire Tap pattern. 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. An example is shown in the section called “Wire Tap”.

Because it is rarely necessary to implement all of the interceptor methods, a `ChannelInterceptorAdapter` class is also available for sub-classing. It provides no-op methods (the `void` method is empty, the `Message` returning methods return the Message as-is, and the `boolean` method returns `true`). Therefore, it is often easiest to extend that class and just implement the method(s) that you need as in the following example.

```
public class CountingChannelInterceptor extends ChannelInterceptorAdapter {

    private final AtomicInteger sendCount = new AtomicInteger();

    @Override
    public Message<?> preSend(Message<?> message, MessageChannel channel) {
        sendCount.incrementAndGet();
        return message;
    }
}
```

The order of invocation for the interceptor methods depends on the type of channel. As described above, the queue-based channels are the only ones where the receive method is intercepted in the first place. Additionally, the relationship between send and receive interception depends on the timing of separate sender and receiver threads. For example, if a receiver is already blocked while waiting for a message the order could be: preSend, preReceive, postReceive, postSend. However, if a receiver polls after the sender has placed a message on the channel and already returned, the order would be: preSend, postSend, (some-time-elapses) preReceive, postReceive. The time that elapses in such a case depends on a number of factors and is therefore generally unpredictable (in fact, the receive may never happen!). Obviously, the type of queue also plays a role (e.g. rendezvous vs. priority). The bottom line is that you cannot rely on the order beyond the fact that preSend will precede postSend and preReceive will precede postReceive.

Starting with *Spring Framework 4.1* and Spring Integration 4.1, the `ChannelInterceptor` provides new methods - `afterSendCompletion()` and `afterReceiveCompletion()`. They are invoked after `send()/receive()` calls, regardless of any exception that is raised, thus allowing for resource cleanup. Note, the Channel invokes these methods on the ChannelInterceptor List in the reverse order of the initial `preSend()/preReceive()` calls.

4.1.4 MessagingTemplate

As you will see when the endpoints and their various configuration options are introduced, Spring Integration provides a foundation for messaging components that enables non-invasive invocation of your application code *from the messaging system*. However, sometimes it is necessary to invoke the messaging system *from your application code*. For convenience when implementing such use-cases, Spring Integration provides a `MessagingTemplate` that supports a variety of operations across the Message Channels, including request/reply scenarios. For example, it is possible to send a request and wait for a reply.

```
MessagingTemplate template = new MessagingTemplate();

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

In that example, a temporary anonymous channel would be created internally by the template. The *sendTimeout* and *receiveTimeout* properties may also be set on the template, and other exchange types are also supported.

```
public boolean send(final MessageChannel channel, final Message<?> message) { ...
}

public Message<?> sendAndReceive(final MessageChannel channel, final Message<?> request) { ..
}

public Message<?> receive(final PollableChannel<?> channel) { ...
}
```

A less invasive approach that allows you to invoke simple interfaces with payload and/or header values instead of Message instances is described in Section 8.4.1, "Enter the GatewayProxyFactoryBean".

4.1.5 Configuring Message Channels

To create a Message Channel instance, you can use the <channel/> element:

```
<int:channel id="exampleChannel"/>
```

The default channel type is *Point to Point*. To create a *Publish Subscribe* channel, use the `<publish-subscribe-channel/>` element:

```
<int:publish-subscribe-channel id="exampleChannel"/>
```

When using the `<channel/>` element without any sub-elements, it will create a `DirectChannel` instance (a `SubscribableChannel`).

However, you can alternatively provide a variety of `<queue/>` sub-elements to create any of the pollable channel types (as described in Section 4.1.2, "Message Channel Implementations"). Examples of each are shown below.

DirectChannel Configuration

As mentioned above, `DirectChannel` is the default type.

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

A default channel will have a *round-robin* load-balancer and will also have failover enabled (See the discussion in the section called "DirectChannel" for more detail). 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>
```

Datatype Channel Configuration

There are times when a consumer can only process a particular type of payload and you need to therefore ensure the payload type of input Messages. Of course the first thing that comes to mind is Message Filter. However all that Message Filter will do is filter out Messages that are not compliant with the requirements of the consumer. Another way would be to use a Content Based Router and route Messages with non-compliant data-types to specific Transformers to enforce transformation/conversion to the required data-type. This of course would work, but a simpler way of accomplishing the same thing is to apply the Datatype Channel pattern. You can use separate Datatype Channels for each specific payload data-type.

To create a Datatype Channel that only accepts messages containing a certain payload type, provide the fully-qualified class

name in the channel element's `datatype` attribute:

```
<int:channel id="numberChannel" datatype="java.lang.Number"/>
```

Note that the type check passes for any type that is *assignable* to the channel's datatype. In other words, the "numberChannel" above would accept messages whose payload is `java.lang.Integer` or `java.lang.Double`. Multiple types can be provided as a comma-delimited list:

```
<int:channel id="stringOrNumberChannel" datatype="java.lang.String,java.lang.Number"/>
```

So the *numberChannel* above will only accept Messages with a data-type of `java.lang.Number`. But what happens if the payload of the Message is not of the required type? It depends on whether you have defined a bean named `integrationConversionService` that is an instance of Spring's Conversion Service. If not, then an Exception would be thrown immediately, but if you do have an "integrationConversionService" bean defined, it will be used in an attempt to convert the Message's payload to the acceptable type.

You can even register custom converters. For example, let's say you are sending a Message with a String payload to the *numberChannel* we configured above.

```
MessageChannel inChannel = context.getBean("numberChannel", MessageChannel.class);
inChannel.send(new GenericMessage<String>("5"));
```

Typically this would be a perfectly legal operation, however since we are using Datatype Channel the result of such operation would generate an exception:

```
Exception in thread "main" org.springframework.integration.MessageDeliveryException:
Channel 'numberChannel'
expected one of the following datataypes [class java.lang.Number],
but received [class java.lang.String]
…
```

And rightfully so since we are requiring the payload type to be a Number while sending a String. So we need something to convert String to a Number. All we need to do is implement a Converter.

```
public static class StringToIntegerConverter implements Converter<String, Integer> {
    public Integer convert(String source) {
        return Integer.parseInt(source);
    }
}
```

Then, register it as a Converter with the Integration Conversion Service:

```
<int:converter ref="strToInt"/>

<bean id="strToInt" class="org.springframework.integration.util.Demo.StringToIntegerConverter"/>
```

When the *converter* element is parsed, it will create the "integrationConversionService" bean on-demand if one is not already defined. With that Converter in place, the send operation would now be successful since the Datatype Channel will use that Converter to convert the String payload to an Integer.

For more information regarding Payload Type Conversion, please read Section 8.1.6, "Payload Type Conversion".

Beginning with *version 4.0*, the `integrationConversionService` is invoked by the `DefaultDatatypeChannelMessageConverter`, which looks up the conversion service in the application context. To use a different conversion technique, you can specify the `message-converter` attribute on the channel. This must be a reference to a `MessageConverter` implementation. Only the `fromMessage` method is used, which provides the converter with access to the message headers (for example if the conversion might need information from the headers, such as `content-type`). The

method can return just the converted payload, or a full `Message` object. If the latter, the converter must be careful to copy all the headers from the inbound message.

Alternatively, declare a `<bean/>` of type `MessageConverter` with an id `"datatypeChannelMessageConverter"` and that converter will be used by all channels with a `datatype`.

QueueChannel Configuration

To create a `QueueChannel`, use the `<queue/>` sub-element. You may specify the channel's capacity:

```
<int:channel id="queueChannel">
    <queue capacity="25"/>
</int:channel>
```

If you do not provide a value for the *capacity* attribute on this `<queue/>` sub-element, the resulting queue will be unbounded. To avoid issues such as OutOfMemoryErrors, it is highly recommended to set an explicit value for a bounded queue.

Persistent QueueChannel Configuration

Since a `QueueChannel` provides the capability to buffer Messages, but does so in-memory only by default, it also introduces a possibility that Messages could be lost in the event of a system failure. To mitigate this risk, a `QueueChannel` may be backed by a persistent implementation of the `MessageGroupStore` strategy interface. For more details on `MessageGroupStore` and `MessageStore` see Section 10.4, "Message Store".

Important

The `capacity` attribute is not allowed when the `message-store` attribute is used.

When a `QueueChannel` receives a Message, it will add it to the Message Store, and when a Message is polled from a `QueueChannel`, it is removed from the Message Store.

By default, a `QueueChannel` stores its Messages in an in-memory Queue and can therefore lead to the lost message scenario mentioned above. However Spring Integration provides persistent stores, such as the `JdbcChannelMessageStore`.

You can configure a Message Store for any `QueueChannel` by adding the `message-store` attribute as shown in the next example.

```
<int:channel id="dbBackedChannel">
    <int:queue message-store="channelStore"/>
</int:channel>

<bean id="channelStore" class="o.s.i.jdbc.store.JdbcChannelMessageStore">
    <property name="dataSource" ref="dataSource"/>
    <property name="channelMessageStoreQueryProvider" ref="queryProvider"/>
</bean>
```

The Spring Integration JDBC module also provides schema DDL for a number of popular databases. These schemas are located in the *org.springframework.integration.jdbc.store.channel* package of that module (spring-integration-jdbc).

Important

One important feature is that with any transactional persistent store (e.g., `JdbcChannelMessageStore`), as long as the poller has a transaction configured, a Message removed from the store will only be permanently removed if the transaction completes successfully, otherwise the transaction will roll back and the Message will not be lost.

Many other implementations of the Message Store will be available as the growing number of Spring projects related to "NoSQL" data stores provide the underlying support. Of course, you can always provide your own implementation of the MessageGroupStore interface if you cannot find one that meets your particular needs.

Since *version 4.0*, it is recommended that `QueueChannel` s are configured to use a `ChannelMessageStore` if possible. These are generally optimized for this use, when compared with a general message store. If the `ChannelMessageStore` is a `ChannelPriorityMessageStore` the messages will be received in FIFO within priority order. The notion of priority is determined by the message store implementation. For example the Java Configuration for the Section 23.3.1, "MongoDB Channel Message Store":

```
@Bean
public BasicMessageGroupStore mongoDbChannelMessageStore(MongoDbFactory mongoDbFactory) {
    MongoDbChannelMessageStore store = new MongoDbChannelMessageStore(mongoDbFactory);
    store.setPriorityEnabled(true);
    return store;
}

@Bean
public PollableChannel priorityQueue(BasicMessageGroupStore mongoDbChannelMessageStore) {
    return new PriorityChannel(new MessageGroupQueue(mongoDbChannelMessageStore, "priorityQueue"));
}
```

Pay attention to the `MessageGroupQueue` class. That is a `BlockingQueue` implementation to utilize the `MessageGroupStore` operations.

The same with Java DSL may look like:

```
@Bean
public IntegrationFlow priorityFlow(PriorityCapableChannelMessageStore mongoDbChannelMessageStore) {
    return IntegrationFlows.from((Channels c) ->
            c.priority("priorityChannel", mongoDbChannelMessageStore, "priorityGroup"))
            ....
            .get();
}
```

Another option to customize the `QueueChannel` environment is provided by the `ref` attribute of the `<int:queue>` sub-element or particular constructor. This attribute implies the reference to any `java.util.Queue` implementation. For example Hazelcast distributed `IQueue`:

```
@Bean
public HazelcastInstance hazelcastInstance() {
    return Hazelcast.newHazelcastInstance(new Config()
                                          .setProperty("hazelcast.logging.type", "log4j"));
}

@Bean
public PollableChannel distributedQueue() {
    return new QueueChannel(hazelcastInstance()
                            .getQueue("springIntegrationQueue"));
}
```

PublishSubscribeChannel Configuration

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. For example, if there are 5 subscribers, the sequence-size would be set to 5, and the Messages would have sequence-number header values ranging from 1 to 5.

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

The `apply-sequence` value is `false` by default so that a Publish Subscribe Channel can send the exact same Message instances to multiple outbound channels. Since Spring Integration enforces immutability of the payload and header references, the channel creates new Message instances with the same payload reference but different header values when the flag is set to `true`.

ExecutorChannel

To create an `ExecutorChannel`, add the <dispatcher> sub-element along with a `task-executor` attribute. Its value can reference any `TaskExecutor` within the context. For example, this enables configuration of a thread-pool for dispatching messages to subscribed handlers. As mentioned above, this does break the "single-threaded" execution context between sender and receiver so that any active transaction context will not be shared by the invocation of the handler (i.e. the handler may throw an Exception, but the send invocation has already returned successfully).

```
<int:channel id="executorChannel">
    <int:dispatcher task-executor="someExecutor"/>
</int:channel>
```

The `load-balancer` and `failover` options are also both available on the <dispatcher/> sub-element as described above in the section called “DirectChannel Configuration”. The same defaults apply as well. So, the channel will have a round-robin load-balancing strategy with failover enabled unless explicit configuration is provided for one or both of those attributes.

```
<int:channel id="executorChannelWithoutFailover">
    <int:dispatcher task-executor="someExecutor" failover="false"/>
</int:channel>
```

PriorityChannel Configuration

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 with the QueueChannel, it also supports a `capacity` attribute. The following example demonstrates all of these:

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

Since *version 4.0*, the `priority-channel` child element supports the `message-store` option (`comparator` and `capacity` are not allowed in that case). The message store must be a `PriorityCapableChannelMessageStore` and, in

this case. Implementations of the `PriorityCapableChannelMessageStore` are currently provided for `Redis`, `JDBC` and `MongoDB`. See the section called "QueueChannel Configuration" and Section 10.4, "Message Store" for more information. You can find sample configuration in Section 19.4.3, "Backing Message Channels".

RendezvousChannel Configuration

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>
```

Scoped Channel Configuration

Any channel can be configured with a "scope" attribute.

```
<int:channel id="threadLocalChannel" scope="thread"/>
```

Channel Interceptor Configuration

Message channels may also have interceptors as described in Section 4.1.3, "Channel Interceptors". The `<interceptors/>` sub-element can be added within a `<channel/>` (or the more specific element types). Provide the `ref` attribute to reference any Spring-managed object that implements the `ChannelInterceptor` interface:

```
<int:channel id="exampleChannel">
    <int:interceptors>
        <ref bean="trafficMonitoringInterceptor"/>
    </int:interceptors>
</int:channel>
```

In general, it is a good idea to define the interceptor implementations in a separate location since they usually provide common behavior that can be reused across multiple channels.

Global Channel Interceptor Configuration

Channel Interceptors provide a clean and concise way of applying cross-cutting behavior per individual channel. If the same behavior should be applied on multiple channels, configuring the same set of interceptors for each channel *would not be* the most efficient way. To avoid repeated configuration while also enabling interceptors to apply to multiple channels, Spring Integration provides *Global Interceptors*. Look at the example below:

```
<int:channel-interceptor pattern="input*, bar*, foo, !baz*" order="3">
    <bean class="foo.barSampleInterceptor"/>
</int:channel-interceptor>
```

or

```
<int:channel-interceptor ref="myInterceptor" pattern="input*, bar*, foo, !baz*" order="3"/>

<bean id="myInterceptor" class="foo.barSampleInterceptor"/>
```

Each `<channel-interceptor/>` element allows you to define a global interceptor which will be applied on all channels that match any patterns defined via the `pattern` attribute. In the above case the global interceptor will be applied on the *foo* channel and all other channels that begin with *bar* or *input* and not to channel starting with *baz* (starting with *version 5.0*).

> The addition of this syntax to the pattern causes one possible (although perhaps unlikely) problem. If you have a bean `"!foo"` **and** you included a pattern `"!foo"` in your channel-interceptor's `pattern` patterns; it will no long match; the pattern will now match all beans **not** named `foo`. In this case, you can escape the `!` in the pattern with `\`. The pattern `"\!foo"` means match a bean named `"!foo"`.

The *order* attribute allows you to manage where this interceptor will be injected if there are multiple interceptors on a given channel. For example, channel *inputChannel* could have individual interceptors configured locally (see below):

```
<int:channel id="inputChannel">
  <int:interceptors>
    <int:wire-tap channel="logger"/>
  </int:interceptors>
</int:channel>
```

A reasonable question is how will a global interceptor be injected in relation to other interceptors configured locally or through other global interceptor definitions? The current implementation provides a very simple mechanism for defining the order of interceptor execution. A positive number in the `order` attribute will ensure interceptor injection after any existing interceptors and a negative number will ensure that the interceptor is injected before existing interceptors. This means that in the above example, the global interceptor will be injected *AFTER* (since its order is greater than 0) the *wire-tap* interceptor configured locally. If there were another global interceptor with a matching `pattern`, its order would be determined by comparing the values of the `order` attribute. To inject a global interceptor *BEFORE* the existing interceptors, use a negative value for the `order` attribute.

> Note that both the `order` and `pattern` attributes are optional. The default value for `order` will be 0 and for `pattern`, the default is * (to match all channels).

Starting with *version 4.3.15*, you can configure a property `spring.integration.postProcessDynamicBeans = true` to apply any global interceptors to dynamically created `MessageChannel` beans. See Section E.5, "Global Properties" for more information.

Wire Tap

As mentioned above, Spring Integration provides a simple *Wire Tap* interceptor out of the box. You can configure a *Wire Tap* on any channel within an `<interceptors/>` element. This is especially useful for debugging, and can be used in conjunction with Spring Integration's logging Channel Adapter as follows:

```
<int:channel id="in">
    <int:interceptors>
        <int:wire-tap channel="logger"/>
    </int:interceptors>
</int:channel>

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

> The *logging-channel-adapter* also accepts an *expression* attribute so that you can evaluate a SpEL expression against *payload* and/or *headers* variables. Alternatively, to simply log the full Message toString() result, provide a value of "true" for the *log-full-message* attribute. That is `false` by default so that only the payload is logged. Setting that to `true` enables logging of all headers in addition to the payload. The *expression* option does provide the most flexibility, however (e.g. expression="payload.user.name").

A little more on Wire Tap

One of the common misconceptions about the wire tap and other similar components (Section B.1, "Message Publishing Configuration") is that they are automatically asynchronous in nature. 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. That is one of the primary benefits of the Message Channel abstraction. From the inception of the framework, we have always emphasized the need and the value of the *Message Channel* as a first-class citizen of the framework. It is not just an internal, implicit realization of the EIP pattern, it is fully exposed as a configurable component to the end user. So, the 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. Is that flow *synchronous* or *asynchronous*? The answer simply depends on the type of *Message Channel* that *channelB* is. And, now you know that we have: *Direct Channel*, *Pollable Channel*, and *Executor Channel* as options. The last two do break the thread boundary making communication via such channels *asynchronous* simply because the dispatching of the message from that channel to its subscribed handlers happens on a different thread than the one used to send the message to that channel. That is what is going to make your wire-tap flow *sync* or *async*. It is consistent with other components within the framework (e.g., Message Publisher) and actually brings a level of consistency and simplicity by sparing you from worrying in advance (other than writing thread safe code) whether a particular piece of code should be implemented as *sync* or *async*. The actual wiring of two pieces of code (component A and component B) via *Message Channel* is what makes their collaboration *sync* or *async*. You may even want to change from *sync* to *async* in the future and *Message Channel* is what's going to allow you to do it swiftly without ever touching the code.

One final point regarding the Wire Tap is that, despite the rationale provided above for not being async by default, one should keep in mind it is usually desirable to hand off the Message as soon as possible. Therefore, it would be quite common to use an asynchronous channel option as the wire-tap's outbound channel. Nonetheless, another reason that we do not enforce asynchronous behavior by default is that you might not want to break a transactional boundary. Perhaps you are using the Wire Tap for auditing purposes, and you DO want the audit Messages to be sent within the original transaction. As an example, you might connect the wire-tap to a JMS outbound-channel-adapter. That way, you get the best of both worlds: 1) the sending of a JMS Message can occur within the transaction while 2) it is still a "fire-and-forget" action thereby preventing any noticeable delay in the main message flow.

Important

Starting with *version 4.0*, it is important to avoid circular references when an interceptor (such as `WireTap`) references a channel itself. You need to exclude such channels from those being intercepted by the current interceptor. This can be done with appropriate `patterns` or programmatically. If you have a custom `ChannelInterceptor` that references a `channel`, consider implementing `VetoCapableInterceptor`. That way, the framework will ask the interceptor if it's OK to intercept each channel that is a candidate based on the pattern. You can also add runtime protection in the interceptor methods that ensures that the channel is not one that is referenced by the interceptor. The `WireTap` uses both of these techniques.

Starting with *version 4.3*, the `WireTap` has additional constructors that take a `channelName` instead of a `MessageChannel` instance. This can be convenient for Java Configuration and when channel auto-creation logic is being used. The target `MessageChannel` bean is resolved from the provided `channelName` later, on the first interaction with the interceptor.

Important

Channel resolution requires a `BeanFactory` so the wire tap instance must be a Spring-managed bean.

This *late-binding* approach also allows simplification of typical wire-tapping patterns with Java DSL configuration:

```
@Bean
public PollableChannel myChannel() {
    return MessageChannels.queue()
            .wireTap("loggingFlow.input")
            .get();
}

@Bean
public IntegrationFlow loggingFlow() {
    return f -> f.log();
}
```

Conditional Wire Taps

Wire taps can be made conditional, using the `selector` or `selector-expression` attributes. The `selector` references a `MessageSelector` bean, which can determine at runtime whether the message should go to the tap channel. Similarly, the` selector-expression` is a boolean SpEL expression that performs the same purpose - if the expression evaluates to true, the message will be sent to the tap channel.

Global Wire Tap Configuration

It is possible to configure a global wire tap as a special case of the the section called “Global Channel Interceptor Configuration”. Simply configure a top level `wire-tap` element. Now, in addition to the normal `wire-tap` namespace support, the `pattern` and `order` attributes are supported and work in exactly the same way as with the `channel-interceptor`

```
<int:wire-tap pattern="input*, bar*, foo" order="3" channel="wiretapChannel"/>
```

A global wire tap provides a convenient way to configure a single channel wire tap externally without modifying the existing channel configuration. Simply set the `pattern` attribute to the target channel name. For example, This technique may be used to configure a test case to verify messages on a channel.

4.1.6 Special Channels

If namespace support is enabled, there are two special channels defined within the application context by default: `errorChannel` and `nullChannel`. The *nullChannel* acts like `/dev/null`, simply logging any Message sent to it at DEBUG level and returning immediately. Any time you face channel resolution errors for a reply that you don’t care about, you can set the affected component’s `output-channel` attribute to *nullChannel* (the name *nullChannel* is reserved within the application context). The *errorChannel* is used internally for sending error messages and may be overridden with a custom configuration. This is discussed in greater detail in Section E.4, “Error Handling”.

See also Section 9.4, “Message Channels” in Java DSL chapter for more information about message channel and interceptors.

4.2 Poller

4.2.1 Polling Consumer

When Message Endpoints (Channel Adapters) are connected to channels and instantiated, they produce one of the following 2 instances:

- PollingConsumer
- EventDrivenConsumer

The actual implementation depends on which type of channel these Endpoints are connected to. A channel adapter connected to a channel that implements the org.springframework.messaging.SubscribableChannel interface will produce an instance of `EventDrivenConsumer`. On the other hand, a channel adapter connected to a channel that implements the org.springframework.messaging.PollableChannel interface (e.g. a QueueChannel) will produce an instance of `PollingConsumer`.

Polling Consumers allow Spring Integration components to actively poll for Messages, rather than to process Messages in an event-driven manner.

They represent a critical cross cutting concern in many messaging scenarios. In Spring Integration, Polling Consumers are based on the pattern with the same name, which is described in the book "Enterprise Integration Patterns" by Gregor Hohpe and Bobby Woolf. You can find a description of the pattern on the book's website at:

http://www.enterpriseintegrationpatterns.com/PollingConsumer.html

4.2.2 Pollable Message Source

Furthermore, in Spring Integration a second variation of the Polling Consumer pattern exists. When Inbound Channel Adapters are being used, these adapters are often wrapped by a `SourcePollingChannelAdapter`. For example, when retrieving messages from a remote FTP Server location, the adapter described in Section 16.4, "FTP Inbound Channel Adapter" is configured with a *poller* to retrieve messages periodically. So, when components are configured with Pollers, the resulting instances are of one of the following types:

- PollingConsumer
- SourcePollingChannelAdapter

This means, Pollers are used in both inbound and outbound messaging scenarios. Here are some use-cases that illustrate the scenarios in which Pollers are used:

- Polling certain external systems such as FTP Servers, Databases, Web Services
- Polling internal (pollable) Message Channels
- Polling internal services (E.g. repeatedly execute methods on a Java class)

AOP Advice classes can be applied to pollers, in an `advice-chain`. An example being a transaction advice to start a transaction. Starting with *version 4.1* a `PollSkipAdvice` is provided. Pollers use triggers to determine the time of the next poll. The `PollSkipAdvice` can be used to suppress (skip) a poll, perhaps because there is some downstream condition that would prevent the message to be processed properly. To use this advice, you have to provide it with an implementation of a `PollSkipStrategy`. Starting with *version 4.2.5*, a `SimplePollSkipStrategy` is provided. Add an instance as a bean to the application context, inject it into a `PollSkipAdvice` and add that to the poller's advice chain. To skip polling, call `skipPolls()`, to resume polling, call `reset()`. *Version 4.2* added more flexibility in this area - see Section 4.2.4, "Conditional Pollers for Message Sources".

This chapter is meant to only give a high-level overview regarding Polling Consumers and how they fit into the concept of message channels - Section 4.1, "Message Channels" and channel adapters - Section 4.3, "Channel Adapter". For more in-depth information regarding Messaging Endpoints in general and Polling Consumers in particular, please see Section 8.1, "Message Endpoints".

4.2.3 Deferred Acknowledgment Pollable Message Source

Starting with *version 5.0.1*, certain modules provide `MessageSource` implementations that support deferring acknowledgment until the downstream flow completes (or hands off the message to another thread). This is currently limited to the `AmqpMessageSource` and the `KafkaMessageSource` provided by the spring-kafka-integration extension project, version 3.0.1 or higher.

With these message sources, the `IntegrationMessageHeaderAccessor.ACKNOWLEDGMENT_CALLBACK` header (see the section called "MessageHeaderAccessor API") is added to the message. The value of the header is an instance of `AcknowledgmentCallback`:

```
@FunctionalInterface
public interface AcknowledgmentCallback {

    void acknlowledge(Status status);

    boolean isAcknowledged();

    void noAutoAck();

    default boolean isAutoAck();

    enum Status {

        /**
         * Mark the message as accepted.
         */
        ACCEPT,

        /**
         * Mark the message as rejected.
         */
        REJECT,

        /**
         * Reject the message and requeue so that it will be redelivered.
         */
        REQUEUE

    }

}
```

Not all message sources (e.g. Kafka) support the `REJECT` status; it is treated the same as `ACCEPT`.

Applications can acknowledge a message at any time:

```
Message<?> received = source.receive();

...

StaticMessageHeaderAccessor.getAcknowledgmentCallback(received)
        .acknowledge(Status.ACCEPT);
```

If the `MessageSource` is wired into a `SourcePollingChannelAdapter`, when the poller thread returns to the adapter after the downstream flow completes, the adapter will check if the acknowledgment has already been acknowledged and, if not, `ACCEPT` it (or `REJECT` it if the flow throws an exception).

To perform ad-hoc polling of a `MessageSource` a `MessageSourcePollingTemplate` is provided; this, too will take care of `ACCEPT` ing or `REJECT` ing the `AcknowledgmentCallback` when the `MessageHandler` callback returns (or throws an exception).

```
MessageSourcePollingTemplate template =
    new MessageSourcePollingTemplate(this.source);
template.poll(h -> {
    ...
});
```

In both cases (`SourcePollingChannelAdapter` and `MessageSourcePollingTemplate`), you can disable auto ack/nack

by calling `noAutoAck()` on the callback. You might do this if you hand off the message to another thread and wish to acknowledge later. Not all implementations support this (for example Apache Kafka because the offset commit has to be performed on the same thread).

4.2.4 Conditional Pollers for Message Sources

Background

`Advice` objects, in an `advice-chain` on a poller, advise the whole polling task (message retrieval and processing). These "around advice" methods do not have access to any context for the poll, just the poll itself. This is fine for requirements such as making a task transactional, or skipping a poll due to some external condition as discussed above. What if we wish to take some action depending on the result of the `receive` part of the poll, or if we want to adjust the poller depending on conditions?

"Smart" Polling

Version 4.2 introduced the `AbstractMessageSourceAdvice`. Any `Advice` objects in the `advice-chain` that subclass this class, are applied to just the receive operation. Such classes implement the following methods:

```
beforeReceive(MessageSource<?> source)
```

This method is called before the `MessageSource.receive()` method. It enables you to examine and or reconfigure the source at this time. Returning `false` cancels this poll (similar to the `PollSkipAdvice` mentioned above).

```
Message<?> afterReceive(Message<?> result, MessageSource<?> source)
```

This method is called after the `receive()` method; again, you can reconfigure the source, or take any action perhaps depending on the result (which can be `null` if there was no message created by the source). You can even return a different message!

Advice Chain Ordering

It is important to understand how the advice chain is processed during initialization. `Advice` objects that do not extend `AbstractMessageSourceAdvice` are applied to the whole poll process and are all invoked first, in order, before any `AbstractMessageSourceAdvice`; then `AbstractMessageSourceAdvice` objects are invoked in order around the `MessageSource` `receive()` method. If you have, say `Advice` objects `a, b, c, d`, where `b` and `d` are `AbstractMessageSourceAdvice`, they will be applied in the order `a, c, b, d`. Also, if a `MessageSource` is already a `Proxy`, the `AbstractMessageSourceAdvice` will be invoked after any existing `Advice` objects. If you wish to change the order, you should wire up the proxy yourself.

SimpleActiveIdleMessageSourceAdvice

This advice is a simple implementation of `AbstractMessageSourceAdvice`, when used in conjunction with a `DynamicPeriodicTrigger`, it adjusts the polling frequency depending on whether or not the previous poll resulted in a message or not. The poller must also have a reference to the same `DynamicPeriodicTrigger`.

Important: Async Handoff

This advice modifies the trigger based on the `receive()` result. This will only work if the advice is called on the poller thread. It will **not** work if the poller has a `task-executor`. To use this advice where you wish to use async operations after the result of a poll, do the async handoff later, perhaps by using an `ExecutorChannel`.

CompoundTriggerAdvice

This advice allows the selection of one of two triggers based on whether a poll returns a message or not. Consider a poller that uses a `CronTrigger`; `CronTrigger` s are immutable so cannot be altered once constructed. Consider a use case where we want to use a cron expression to trigger a poll once each hour but, if no message is received, poll once per minute and, when a message is retrieved, revert to using the cron expression.

The advice (and poller) use a `CompoundTrigger` for this purpose. The trigger's `primary` trigger can be a `CronTrigger`. When the advice detects that no message is received, it adds the secondary trigger to the `CompoundTrigger`. When the `CompoundTrigger` 's `nextExecutionTime` method is invoked, it will delegate to the secondary trigger, if present; otherwise the primary trigger.

The poller must also have a reference to the same `CompoundTrigger`.

The following shows the configuration for the hourly cron expression with fall-back to every minute…

```
<int:inbound-channel-adapter channel="nullChannel" auto-startup="false">
    <bean class="org.springframework.integration.endpoint.PollerAdviceTests.Source" />
    <int:poller trigger="compoundTrigger">
        <int:advice-chain>
            <bean class="org.springframework.integration.aop.CompoundTriggerAdvice">
                <constructor-arg ref="compoundTrigger"/>
                <constructor-arg ref="secondary"/>
            </bean>
        </int:advice-chain>
    </int:poller>
</int:inbound-channel-adapter>

<bean id="compoundTrigger" class="org.springframework.integration.util.CompoundTrigger">
    <constructor-arg ref="primary" />
</bean>

<bean id="primary" class="org.springframework.scheduling.support.CronTrigger">
    <constructor-arg value="0 0 * * * *" /> <!-- top of every hour -->
</bean>

<bean id="secondary" class="org.springframework.scheduling.support.PeriodicTrigger">
    <constructor-arg value="60000" />
</bean>
```

Important: Async Handoff

This advice modifies the trigger based on the `receive()` result. This will only work if the advice is called on the poller thread. It will **not** work if the poller has a `task-executor`. To use this advice where you wish to use async operations after the result of a poll, do the async handoff later, perhaps by using an `ExecutorChannel`.

4.3 Channel Adapter

A Channel Adapter is a Message Endpoint that enables connecting a single sender or receiver to a Message Channel. Spring Integration provides a number of adapters out of the box to support various transports, such as JMS, File, HTTP, Web Services, Mail, and more. Those will be discussed in upcoming chapters of this reference guide. However, this chapter focuses on the simple but flexible Method-invoking Channel Adapter support. There are both inbound and outbound adapters, and each may be configured with XML elements provided in the core namespace. These provide an easy way to extend Spring Integration as long as you have a method that can be invoked as either a source or destination.

4.3.1 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>
```

Also see Section 4.3.3, “Channel Adapter Expressions and Scripts”.

If no poller is provided, then a single default poller must be registered within the context. See Section 8.1.4, “Endpoint Namespace Support” for more detail.

Important: Poller Configuration

Some `inbound-channel-adapter` types are backed by a `SourcePollingChannelAdapter` which means they contain Poller configuration which will poll the `MessageSource` (invoke a custom method which produces the value that becomes a `Message` payload) based on the configuration specified in the Poller.
For example:

```
<int:poller max-messages-per-poll="1" fixed-rate="1000"/>

<int:poller max-messages-per-poll="10" fixed-rate="1000"/>
```

In the the first configuration the polling task will be invoked once per poll and during such task (poll) the method (which results in the production of the Message) will be invoked once based on the `max-messages-per-poll` attribute value. In the second configuration the polling task will be invoked 10 times per poll or until it returns *null* thus possibly producing 10 Messages per poll while each poll happens at 1 second intervals. However what if the configuration looks like this:

```
<int:poller fixed-rate="1000"/>
```

Note there is no `max-messages-per-poll` specified. As you'll learn later the identical poller configuration in the `PollingConsumer` (e.g., service-activator, filter, router etc.) would have a default value of -1 for `max-messages-per-poll` which means "execute poling task non-stop unless polling method returns null (e.g., no more Messages in the QueueChannel)" and then sleep for 1 second.
However in the SourcePollingChannelAdapter it is a bit different. The default value for `max-messages-per-poll` will be set to 1 by default unless you explicitly set it to a negative value (e.g., -1). It is done so to make sure that poller can react to a LifeCycle events (e.g., start/stop) and prevent it from potentially spinning in the infinite loop if the implementation of the custom method of the `MessageSource` has a potential to never return null and happened to be non-interruptible.
However if you are sure that your method can return null and you need the behavior where you want to poll for as many sources as available per each poll, then you should explicitly set `max-messages-per-poll` to a negative value.

```
<int:poller max-messages-per-poll="-1" fixed-rate="1000"/>
```

4.3.2 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.

```
<int:outbound-channel-adapter channel="channel1" ref="target" method="handle"/>

<beans:bean id="target" class="org.Foo"/>
```

If the channel being adapted is a `PollableChannel`, provide a poller sub-element:

```
<int:outbound-channel-adapter channel="channel2" ref="target" method="handle">
    <int:poller fixed-rate="3000" />
</int:outbound-channel-adapter>

<beans:bean id="target" class="org.Foo"/>
```

Using a "ref" attribute is generally recommended if the POJO consumer implementation can be reused in other `<outbound-channel-adapter>` definitions. However if the consumer implementation is only referenced by a single definition of the `<outbound-channel-adapter>`, you can define it as inner bean:

```
<int:outbound-channel-adapter channel="channel" method="handle">
    <beans:bean class="org.Foo"/>
</int:outbound-channel-adapter>
```

Using both the "ref" attribute and an inner handler definition in the same `<outbound-channel-adapter>` configuration is not allowed as it creates an ambiguous condition. Such a configuration will result in an Exception being thrown.

Any Channel Adapter can be created without a "channel" reference in which case it will implicitly create an instance of `DirectChannel`. The created channel's name will match the "id" attribute of the `<inbound-channel-adapter>` or `<outbound-channel-adapter>` element. Therefore, if the "channel" is not provided, the "id" is required.

4.3.3 Channel Adapter Expressions and Scripts

Like many other Spring Integration components, the `<inbound-channel-adapter>` and `<outbound-channel-adapter>` also provide support for SpEL expression evaluation. To use SpEL, provide the expression string via the *expression* attribute instead of providing the *ref* and *method* attributes that are used for method-invocation on a bean. When an Expression is evaluated, it follows the same contract as method-invocation where: the *expression* for an `<inbound-channel-adapter>` will generate a message anytime the evaluation result is a *non-null* value, while the *expression* for an `<outbound-channel-adapter>` must be the equivalent of a *void* returning method invocation.

Starting with Spring Integration 3.0, an `<int:inbound-channel-adapter/>` can also be configured with a SpEL `<expression/>` (or even with `<script/>`) sub-element, for when more sophistication is required than can be achieved with the simple *expression* attribute. If you provide a script as a `Resource` using the `location` attribute, you can also set the *refresh-check-delay* allowing the resource to be refreshed periodically. If you want the script to be checked on each poll, you would need to coordinate this setting with the poller's trigger:

```
<int:inbound-channel-adapter ref="source1" method="method1" channel="channel1">
    <int:poller max-messages-per-poll="1" fixed-delay="5000"/>
    <script:script lang="ruby" location="Foo.rb" refresh-check-delay="5000"/>
</int:inbound-channel-adapter>
```

Also see the `cacheSeconds` property on the `ReloadableResourceBundleExpressionSource` when using the `<expression/>` sub-element. For more information regarding expressions see Appendix A, *Spring Expression Language (SpEL)*, and for scripts - Section 8.8, "Groovy support" and Section 8.7, "Scripting support".

Important

> The `<int:inbound-channel-adapter/>` is an endpoint that starts a message flow via periodic triggering to poll some underlying `MessageSource`. Since, at the time of polling, there is not yet a message object, expressions and scripts don't have access to a root `Message`, so there are no *payload* or *headers* properties that are available in most other messaging SpEL expressions. Of course, the script **can** generate and return a complete `Message` object with headers and payload, or just a payload, which will be added to a message with basic headers.

4.4 Messaging Bridge

4.4.1 Introduction

A Messaging Bridge is a relatively trivial endpoint that simply connects two Message Channels or Channel Adapters. For example, you may want to connect a `PollableChannel` to a `SubscribableChannel` so that the subscribing endpoints do not have to worry about any polling configuration. Instead, the Messaging Bridge provides the polling configuration.

By providing an intermediary poller between two channels, a Messaging Bridge can be used to throttle inbound Messages. The poller's trigger will determine the rate at which messages arrive on the second channel, and the poller's "maxMessagesPerPoll" property will enforce a limit on the throughput.

Another valid use for a Messaging Bridge is to connect two different systems. In such a scenario, Spring Integration's role would be limited to making the connection between these systems and managing a poller if necessary. It is probably more common to have at least a *Transformer* between the two systems to translate between their formats, and in that case, the channels would be provided as the *input-channel* and *output-channel* of a Transformer endpoint. If data format translation is not required, the Messaging Bridge may indeed be sufficient.

4.4.2 Configuring a Bridge with XML

The <bridge> element is used to create a Messaging Bridge between two Message Channels or Channel Adapters. Simply provide the "input-channel" and "output-channel" attributes:

```
<int:bridge input-channel="input" output-channel="output"/>
```

As mentioned above, a common use case for the Messaging Bridge is to connect a `PollableChannel` to a `SubscribableChannel`, and when performing this role, the Messaging Bridge may also serve as a throttler:

```
<int:bridge input-channel="pollable" output-channel="subscribable">
     <int:poller max-messages-per-poll="10" fixed-rate="5000"/>
 </int:bridge>
```

Connecting Channel Adapters is just as easy. Here is a simple echo example between the "stdin" and "stdout" adapters from Spring Integration's "stream" namespace.

```
<int-stream:stdin-channel-adapter id="stdin"/>

 <int-stream:stdout-channel-adapter id="stdout"/>

 <int:bridge id="echo" input-channel="stdin" output-channel="stdout"/>
```

Of course, the configuration would be similar for other (potentially more useful) Channel Adapter bridges, such as File to JMS, or Mail to File. The various Channel Adapters will be discussed in upcoming chapters.

If no *output-channel* is defined on a bridge, the reply channel provided by the inbound Message will be used, if available. If neither output or reply channel is available, an Exception will be thrown.

4.4.3 Configuring a Bridge with Java Configuration

```
@Bean
public PollableChannel polled() {
    return new QueueChannel();
}

@Bean
@BridgeFrom(value = "polled", poller = @Poller(fixedDelay = "5000", maxMessagesPerPoll = "10"))
public SubscribableChannel direct() {
    return new DirectChannel();
}
```

or

```
@Bean
@BridgeTo(value = "direct", poller = @Poller(fixedDelay = "5000", maxMessagesPerPoll = "10"))
public PollableChannel polled() {
    return new QueueChannel();
}

@Bean
public SubscribableChannel direct() {
    return new DirectChannel();
}
```

Or, using a `BridgeHandler`:

```
@Bean
@ServiceActivator(inputChannel = "polled",
        poller = @Poller(fixedRate = "5000", maxMessagesPerPoll = "10"))
public BridgeHandler bridge() {
    BridgeHandler bridge = new BridgeHandler();
    bridge.setOutputChannelName("direct");
    return bridge;
}
```

4.4.4 Configuring a Bridge with the Java DSL

```
@Bean
public IntegrationFlow bridgeFlow() {
    return IntegrationFlows.from("polled")
            .bridge(e -> e.poller(Pollers.fixedDelay(5000).maxMessagesPerPoll(10)))
            .channel("direct")
            .get();
}
```