Spring Integration Reference Manual

Mark Fisher
Marius Bogoevici
Iwein Fuld
Jonas Partner

1.0.0.RC1 (Release Candidate 1)

Table of Contents

1. Spring Integration Overview

1.1 Background

One of the key themes of the Spring Framework is *inversion of control*. In its broadest sense, this means that the framework handles responsibilities on behalf of the components that are managed within its context. The components themselves are simplified since they are relieved of those responsibilities. For example, *dependency injection* relieves the components of the responsibility of locating or creating their dependencies. Likewise, *aspect-oriented programming* relieves business components of generic cross-cutting concerns by modularizing them into reusable aspects. In each case, the end result is a system that is easier to test, understand, maintain, and extend.

Furthermore, the Spring framework and portfolio provide a comprehensive programming model for building enterprise applications. Developers benefit from the consistency of this model and especially the fact that it is based upon well-established best practices such as programming to interfaces and favoring composition over inheritance. Spring's simplified abstractions and powerful support libraries boost developer productivity while simultaneously increasing the level of testability and portability.

Spring Integration is a new member of the Spring portfolio motivated by these same goals and principles. It extends the Spring programming model into the messaging domain and builds upon Spring's existing enterprise integration support to provide an even higher level of abstraction. It supports message-driven architectures where inversion of control applies to runtime concerns, such as *when* certain business logic should execute and *where* the response should be sent. It supports routing and transformation of messages so that different transports and different data formats can be integrated without impacting testability. In other words, the messaging and integration concerns are handled by the framework, so business components are further isolated from the infrastructure and developers are relieved of complex integration responsibilities.

As an extension of the Spring programming model, Spring Integration provides a wide variety of configuration options including annotations, XML with namespace support, XML with generic "bean" elements, and of course direct usage of the underlying API. That API is based upon well-defined strategy interfaces and non-invasive, delegating adapters. Spring Integration's design is inspired by the recognition of a strong affinity between common patterns within Spring and the well-known Enterprise Integration Patterns as described in the book of the same name by Gregor Hohpe and Bobby Woolf (Addison Wesley, 2004). Developers who have read that book should be immediately comfortable with the Spring Integration concepts and terminology.

1.2 Goals and Principles

Spring Integration is motivated by the following goals:

- Provide a simple model for implementing complex enterprise integration solutions.

- Facilitate asynchronous, message-driven behavior within a Spring-based application.
- Promote intuitive, incremental adoption for existing Spring users.

Spring Integration is guided by the following principles:

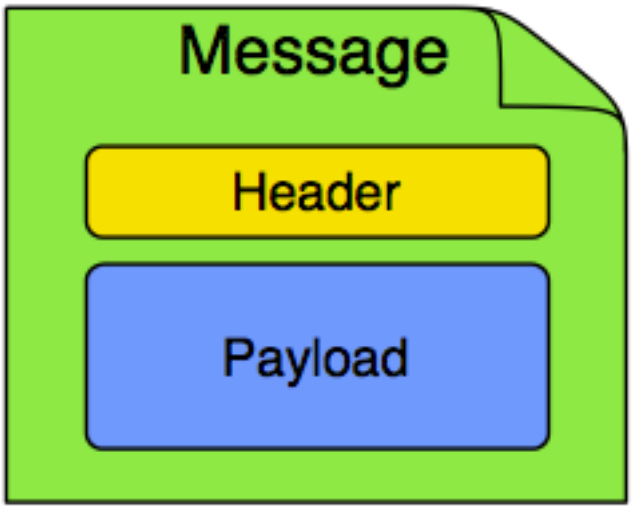
- Components should be *loosely coupled* for modularity and testability.
- The framework should enforce *separation of concerns* between business logic and integration logic.
- Extension points should be abstract in nature but within well-defined boundaries to promote *reuse* and *portability*.

1.3 Main Components

From the *vertical* perspective, a layered architecture facilitates separation of concerns, and interface-based contracts between layers promote loose coupling. Spring-based applications are typically designed this way, and the Spring framework and portfolio provide a strong foundation for following this best practice for the full-stack of an enterprise application. Message-driven architectures add a *horizontal* perspective, yet these same goals are still relevant. Just as "layered architecture" is an extremely generic and abstract paradigm, messaging systems typically follow the similarly abstract "pipes-and-filters" model. The "filters" represent any component that is capable of producing and/or consuming messages, and the "pipes" transport the messages between filters so that the components themselves remain loosely-coupled. It is important to note that these two high-level paradigms are not mutually exclusive. The underlying messaging infrastructure that supports the "pipes" should still be encapsulated in a layer whose contracts are defined as interfaces. Likewise, the "filters" themselves would typically be managed within a layer that is logically above the application's service layer, interacting with those services through interfaces much in the same way that a web-tier would.

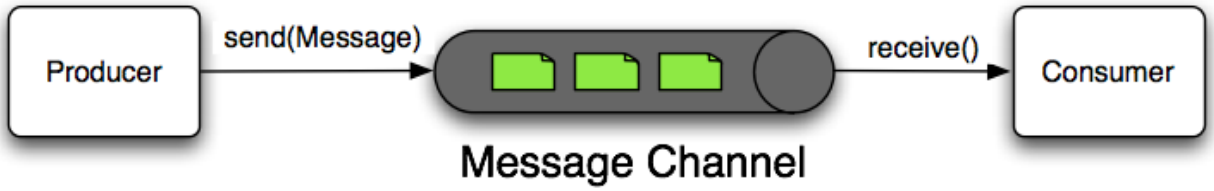
Message

In Spring Integration, a Message is a generic wrapper for any Java object combined with metadata used by the framework while handling that object. It consists of a payload and headers. The payload can be of any type and the headers hold commonly required information such as id, timestamp, expiration, and return address. Headers are also used for passing values to and from connected transports. For example, when creating a Message from a received File, the file name may be stored in a header to be accessed by downstream components. Likewise, if a Message's content is ultimately going to be sent by an outbound Mail adapter, the various properties (to, from, cc, subject, etc.) may be configured as Message header values by an upstream component. Developers can also store any arbitrary key-value pairs in the headers.



Message Channel

A Message Channel represents the "pipe" of a pipes-and-filters architecture. Producers send Messages to a channel, and consumers receive Messages from a channel. The Message Channel therefore decouples the messaging components, and also provides a convenient point for interception and monitoring of Messages.



A Message Channel may follow either Point-to-Point or Publish/Subscribe semantics. With a Point-to-Point channel, at most one consumer can receive each Message sent to the channel. Publish/Subscribe channels, on the other hand, will attempt to broadcast each Message to all of its subscribers. Spring Integration supports both of these.

Whereas "Point-to-Point" and "Publish/Subscribe" define the two options for *how many* consumers will ultimately receive each Message, there is another important consideration: should the channel buffer messages? In Spring Integration, *Pollable Channels* are capable of buffering Messages within a queue. The advantage of buffering is that it allows for throttling the inbound Messages and thereby prevents overloading a consumer. However, as the name suggests, this also adds some complexity, since a consumer can only receive the Messages from such a channel if a *poller* is configured. On the other hand, a consumer connected to a *Subscribable Channel* is simply Message-driven. The variety of channel implementations available in Spring Integration will be discussed in detail in Section 3.2, “Message Channel Implementations”.

Message Endpoint

One of the primary goals of Spring Integration is to simplify the development of enterprise integration

solutions through *inversion of control*. This means that you should not have to implement consumers and producers directly, and you should not even have to build Messages and invoke send or receive operations on a Message Channel. Instead, you should be able to focus on your specific domain model with an implementation based on plain Objects. Then, by providing declarative configuration, you can "connect" your domain-specific code to the messaging infrastructure provided by Spring Integration. The components responsible for these connections are Message Endpoints. This does not mean that you will necessarily connect your existing application code directly. Any real-world enterprise integration solution will require some amount of code focused upon integration concerns such as *routing* and *transformation*. The important thing is to achieve separation of concerns between such integration logic and business logic. In other words, as with the Model-View-Controller paradigm for web applications, the goal should be to provide a thin but dedicated layer that translates inbound requests into service layer invocations, and then translates service layer return values into outbound replies. The next section will provide an overview of the Message Endpoint types that handle these responsibilities, and in upcoming chapters, you will see how Spring Integration's declarative configuration options provide a non-invasive way to use each of these.

1.4 Message Endpoints

A Message Endpoint represents the "filter" of a pipes-and-filters architecture. As mentioned above, the endpoint's primary role is to connect application code to the messaging framework and to do so in a non-invasive manner. In other words, the application code should ideally have no awareness of the Message objects or the Message Channels. This is similar to the role of a Controller in the MVC paradigm. Just as a Controller handles HTTP requests, the Message Endpoint handles Messages. Just as Controllers are mapped to URL patterns, Message Endpoints are mapped to Message Channels. The goal is the same in both cases: isolate application code from the infrastructure. These concepts are discussed at length along with all of the patterns that follow in the Enterprise Integration Patterns book. Here, we provide only a high-level description of the main endpoint types supported by Spring Integration and their roles. The chapters that follow will elaborate and provide sample code as well as configuration examples.

Transformer

A Message Transformer is responsible for converting a Message's content or structure and returning the modified Message. Probably the most common type of transformer is one that converts the payload of the Message from one format to another (e.g. from XML Document to java.lang.String). Similarly, a transformer may be used to add, remove, or modify the Message's header values.

Filter

A Message Filter determines whether a Message should be passed to an output channel at all. This simply requires a boolean test method that may check for a particular payload content type, a property value, the presence of a header, etc. If the Message is accepted, it is sent to the output channel, but if not it will be dropped (or for a more severe implementation, an Exception could be thrown). Message Filters are often

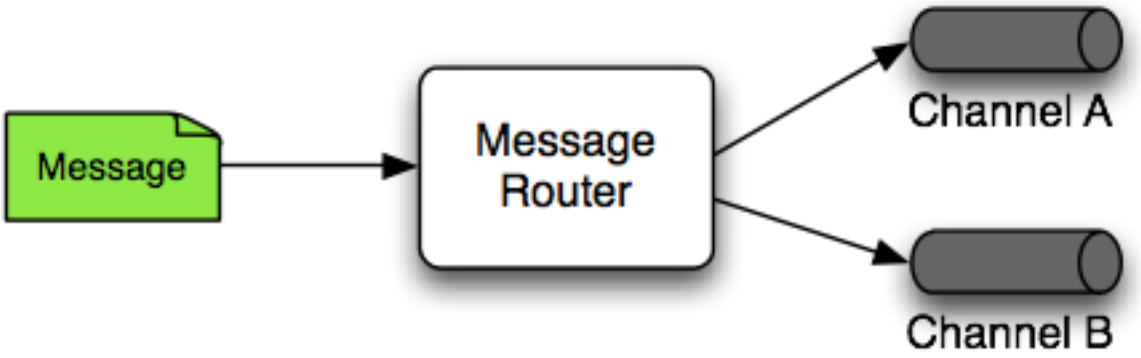
used in conjunction with a Publish Subscribe channel, where multiple consumers may receive the same Message and use the filter to narrow down the set of Messages to be processed based on some criteria.

Note

Be careful not to confuse the generic use of "filter" within the Pipes-and-Filters architectural pattern with this specific endpoint type that selectively narrows down the Messages flowing between two channels. The Pipes-and-Filters concept of "filter" matches more closely with Spring Integration's Message Endpoint: any component that can be connected to Message Channel(s) in order to send and/or receive Messages.

Router

A Message Router is responsible for deciding what channel or channels should receive the Message next (if any). Typically the decision is based upon the Message's content and/or metadata available in the Message Headers. 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. Likewise, a Message Router provides a proactive alternative to the reactive Message Filters used by multiple subscribers as described above.



Splitter

A Splitter is another type of Message Endpoint whose responsibility is to accept a Message from its input channel, split that Message into multiple Messages, and then send each of those to its output channel. This is typically used for dividing a "composite" payload object into a group of Messages containing the sub-divided payloads.

Aggregator

Basically a mirror-image of the Splitter, the Aggregator is a type of Message Endpoint that receives multiple Messages and combines them into a single Message. In fact, Aggregators are often downstream consumers in a pipeline that includes a Splitter. Technically, the Aggregator is more complex than a Splitter, because it is required to maintain state (the Messages to-be-aggregated), to decide when the

complete group of Messages is available, and to timeout if necessary. Furthermore, in case of a timeout, the Aggregator needs to know whether to send the partial results or to discard them to a separate channel. Spring Integration provides a `CompletionStrategy` as well as configurable settings for timeout, whether to send partial results upon timeout, and the discard channel.

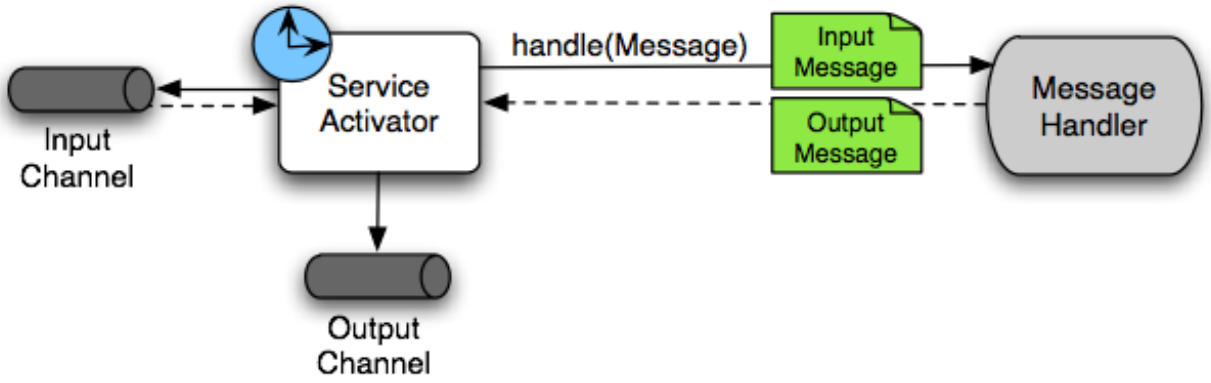
Service Activator

A Service Activator is a generic endpoint for connecting a service instance to the messaging system. The input Message Channel must be configured, and if the service method to be invoked is capable of returning a value, an output Message Channel may also be provided.

Note

The output channel is optional, since each Message may also provide its own 'Return Address' header. This same rule applies for all consumer endpoints.

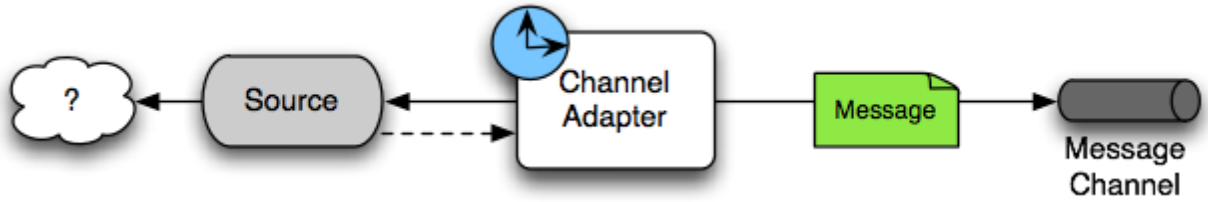
The Service Activator invokes an operation on some service object to process the request Message, extracting the request Message's payload and converting if necessary (if the method does not expect a Message-typed parameter). Whenever the service object's method returns a value, that return value will likewise be converted to a reply Message if necessary (if it's not already a Message). That reply Message is sent to the output channel. If no output channel has been configured, then the reply will be sent to the channel specified in the Message's "return address" if available.



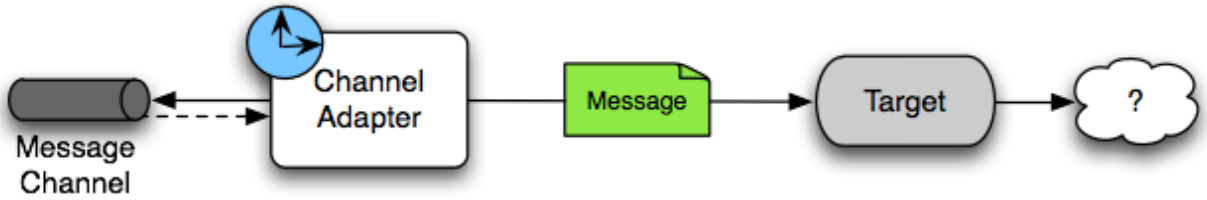
A request-reply "Service Activator" endpoint connects a target object's method to input and output Message Channels.

Channel Adapter

A Channel Adapter is an endpoint that connects a Message Channel to some other system or transport. Channel Adapters may be either inbound or outbound. Typically, the Channel Adapter will do some mapping between the Message and whatever object or resource is received-from or sent-to the other system (File, HTTP Request, JMS Message, etc). Depending on the transport, the Channel Adapter may also populate or extract Message header values. Spring Integration provides a number of Channel Adapters, and they will be described in upcoming chapters.



An inbound "Channel Adapter" endpoint connects a source system to a MessageChannel.



An outbound "Channel Adapter" endpoint connects a MessageChannel to a target system.

2. Message Construction

The Spring Integration `Message` is a generic container for data. Any object can be provided as the payload, and each `Message` also includes headers containing user-extensible properties as key-value pairs.

2.1 The Message Interface

Here is the definition of the `Message` interface:

```
public interface Message<T> {

    T getPayload();

    MessageHeaders getHeaders();

}
```

The `Message` is obviously a very important part of the API. By encapsulating the data in a generic wrapper, the messaging system can pass it around without any knowledge of the data's type. As an application evolves to support new types, or when the types themselves are modified and/or extended, the messaging system will not be affected by such changes. On the other hand, when some component in the messaging system *does* require access to information about the `Message`, such metadata can typically be stored to and retrieved from the metadata in the Message Headers.

2.2 Message Headers

Just as Spring Integration allows any Object to be used as the payload of a Message, it also supports any Object types as header values. In fact, the `MessageHeaders` class implements the *java.util.Map* interface:

```
public final class MessageHeaders implements Map<String, Object>, Serializable {
        ...
}
```

Note

Even though the MessageHeaders implements Map, it is effectively a read-only implementation. Any attempt to *put* a value in the Map will result in an `UnsupportedOperationException`. The same applies for *remove* and *clear*. Since Messages may be passed to multiple consumers, the structure of the Map cannot be modified. Likewise, the Message's payload Object can not be *set* after the initial creation. However, the mutability of the header values themselves (or the payload Object) is intentionally left as a decision for the framework user.

As an implementation of Map, the headers can obviously be retrieved by calling `get(..)` with the name of the header. Alternatively, you can provide the expected *Class* as an additional parameter. Even better, when retrieving one of the pre-defined values, convenient getters are available. Here is an example of each of these three options:

```
Object someValue = message.getHeaders().get("someKey");

CustomerId customerId = message.getHeaders().get("customerId", CustomerId.class);

Long timestamp = message.getHeaders().getTimestamp();
```

The following Message headers are pre-defined:

Table 2.1. Pre-defined Message Headers

Header Name	Header Type
ID	java.util.UUID
TIMESTAMP	java.lang.Long
EXPIRATION_DATE	java.lang.Long
CORRELATION_ID	java.lang.Object
REPLY_CHANNEL	java.lang.Object (can be a String or MessageChannel)
SEQUENCE_NUMBER	java.lang.Integer
SEQUENCE_SIZE	java.lang.Integer
PRIORITY	MessagePriority (an *enum*)

Many inbound and outbound adapter implementations will also provide and/or expect certain headers, and additional user-defined headers can also be configured.

2.3 Message Implementations

The base implementation of the `Message` interface is `GenericMessage<T>`, and it provides two constructors:

```
new GenericMessage<T>(T payload);

new GenericMessage<T>(T payload, Map<String, Object> headers)
```

When a Message is created, a random unique id will be generated. The constructor that accepts a Map of headers will copy the provided headers to the newly created Message.

There are also two convenient subclasses available: `StringMessage` and `ErrorMessage`. The former accepts a String as its payload:

```
StringMessage message = new StringMessage("hello world");

String s = message.getPayload();
```

And, the latter accepts any `Throwable` object as its payload:

```
ErrorMessage message = new ErrorMessage(someThrowable);

Throwable t = message.getPayload();
```

Notice that these implementations take advantage of the fact that the `GenericMessage` base class is parameterized. Therefore, as shown in both examples, no casting is necessary when retrieving the Message payload Object.

2.4 The MessageBuilder Helper Class

You may notice that the Message interface defines retrieval methods for its payload and headers but no setters. The reason for this is that a Message cannot be modified after its initial creation. Therefore, when a Message instance is sent to multiple consumers (e.g. through a Publish Subscribe Channel), if one of those consumers needs to send a reply with a different payload type, it will need to create a new Message. As a result, the other consumers are not affected by those changes. Keep in mind, that multiple consumers may access the same payload instance or header value, and whether such an instance is itself immutable is a decision left to the developer. In other words, the contract for Messages is similar to that of an *unmodifiable Collection*, and the MessageHeaders' map further exemplifies that; even though the MessageHeaders class implements `java.util.Map`, any attempt to invoke a *put* operation (or 'remove' or 'clear') on the MessageHeaders will result in an `UnsupportedOperationException`.

Rather than requiring the creation and population of a Map to pass into the GenericMessage constructor, Spring Integration does provide a far more convenient way to construct Messages: `MessageBuilder`. The MessageBuilder provides two factory methods for creating Messages from either an existing Message or with a payload Object. When building from an existing Message, the headers *and payload* of that Message will be copied to the new Message:

```
Message<String> message1 = MessageBuilder.withPayload("test")
        .setHeader("foo", "bar")
        .build();

Message<String> message2 = MessageBuilder.fromMessage(message1).build();

assertEquals("test", message2.getPayload());
assertEquals("bar", message2.getHeaders().get("foo"));
```

If you need to create a Message with a new payload but still want to copy the headers from an existing Message, you can use one of the 'copy' methods.

```
Message<String> message3 = MessageBuilder.fromPayload("test3")
        .copyHeaders(message1.getHeaders())
        .build();
```

```
Message<String> message4 = MessageBuilder.fromPayload("test4")
        .setHeader("foo", 123)
        .copyHeadersIfAbsent(message1.getHeaders())
        .build();

assertEquals("bar", message3.getHeaders().get("foo"));
assertEquals(123, message4.getHeaders().get("foo"));
```

Notice that the `copyHeadersIfAbsent` does not overwrite existing values. Also, in the second example above, you can see how to set any user-defined header with `setHeader`. Finally, there are set methods available for the predefined headers as well as a non-destructive method for setting any header (MessageHeaders also defines constants for the pre-defined header names).

```
Message<Integer> importantMessage = MessageBuilder.fromPayload(99)
        .setPriority(MessagePriority.HIGHEST)
        .build();

assertEquals(MessagePriority.HIGHEST, importantMessage.getHeaders().getPriority());

Message<Integer> anotherMessage = MessageBuilder.fromMessage(importantMessage)
        .setHeaderIfAbsent(MessageHeaders.PRIORITY, MessagePriority.LOW)
        .build();

assertEquals(MessagePriority.HIGHEST, anotherMessage.getHeaders().getPriority());
```

The `MessagePriority` is only considered when using a `PriorityChannel` (as described in the next chapter). It is defined as an *enum* with five possible values:

```
public enum MessagePriority {
    HIGHEST,
    HIGH,
    NORMAL,
    LOW,
    LOWEST
}
```

3. Message Channels

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

3.1 The MessageChannel Interface

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

```
public interface MessageChannel {

    String getName();

    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);

    List<Message<?>> clear();

    List<Message<?>> purge(MessageSelector selector);

}
```

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 consumers. Therefore, they do not provide receive methods for polling, but instead define methods for handling those subscribers:

```
public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageConsumer consumer);
```

```
    boolean unsubscribe(MessageConsumer consumer);
}
```

3.2 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 consumers. 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 consumer. 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 `MessageConsumer` itself, and the subscriber's `send(Message)` method will be invoked in turn.

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. 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 calling 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 too dangerous. For example, the sender's thread could roll back a transaction if the send operation times out, whereas with a `QueueChannel`, the message would have been stored to the internal queue and potentially never received.

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.

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 `MessageConsumer`. Its primary purpose is to enable a single thread to perform the operations on "both sides" of the channel. For example, if a consumer is subscribed to a `DirectChannel`, then sending a Message to that channel will trigger invocation of that consumer's `onMessage(Message)` method *directly in the sender's thread*. 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 consumer's invocation (e.g. updating a database record) can play a role in determining the ultimate result of that transaction (commit or rollback).

Note

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 needs to provide buffering to throttle input, and to 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 these can be configured.

ThreadLocalChannel

The final channel implementation type is `ThreadLocalChannel`. This channel also delegates to a queue internally, but the queue is bound to the current thread. 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 probably the least common type of channel, this is useful for 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 a `ThreadLocalChannel`, the original sending thread can collect its replies from it.

3.3 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 `Messages` 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);

    boolean preReceive(MessageChannel channel);

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

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 an Exception). Also, the `preReceive` method can return '`false`' to prevent the receive operation from proceeding.

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 parameter 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;
```

```
    }
}
```

3.4 MessageChannelTemplate

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 `MessageChannelTemplate` 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.

```
MessageChannelTemplate template = new MessageChannelTemplate();

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

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 Message<?> message, final MessageChannel channel) { ... }

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

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

3.5 Configuring Message Channels

To create a Message Channel instance, you can use the 'channel' element:

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

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

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

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:

```
<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:

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

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

However, you can also provide a variety of "queue" sub-elements to create the channel types (as described in Section 3.2, “Message Channel Implementations”). Examples of each are shown below.

DirectChannel Configuration

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

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

QueueChannel Configuration

To create a `QueueChannel`, use the "queue" sub-element. You must specify the channel's capacity:

```
<channel id="exampleChannel">
    <queue capacity="25"/>
</channel>
```

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):

```
<publish-subscribe-channel id="exampleChannel" task-executor="someTaskExecutor"/>
```

PriorityChannel Configuration

To create a `PriorityChannel`, use the "priority-queue" sub-element:

```
<channel id="exampleChannel">
    <priority-queue capacity="20"/>
</channel>
```

By default, the channel will consult the `MessagePriority` 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:

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

RendezvousChannel Configuration

The `RendezvousChannel` does not provide any additional configuration options.

```
<channel id="exampleChannel"/>
    <rendezvous-queue/>
</channel>
```

ThreadLocalChannel Configuration

The `ThreadLocalChannel` does not provide any additional configuration options.

```
<thread-local-channel id="exampleChannel"/>
```

Message channels may also have interceptors as described in Section 3.3, “Channel Interceptors”. One or more <interceptor> elements can be added as sub-elements of <channel> (or the more specific element types). Provide the "ref" attribute to reference any Spring-managed object that implements the `ChannelInterceptor` interface:

```
<channel id="exampleChannel">
    <interceptor ref="trafficMonitoringInterceptor"/>
</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.

4. Message Endpoints

As mentioned in the overview, Message Endpoints are responsible for connecting the various messaging components to channels. Over the next several chapters, you will see a number of different components that consume Messages. Some of these are also capable of sending reply Messages. Sending Messages is quite straightforward. As shown above in Chapter 3, *Message Channels*, it's easy to *send* a Message to a Message Channel. However, receiving is a bit more complicated. The main reason is that there are two types of consumers: Polling Consumers and Event-Driven Consumers.

Of the two, 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). Spring Integration provides two different endpoint implementations to accommodate these two types of consumers. Therefore, the consumers themselves can simply implement the callback interface. When polling is required, the endpoint acts as a "container" for the consumer instance. The benefit is similar to that of using a container for hosting Message-Driven Beans, but since these consumers are simply Spring-managed Objects running within an ApplicationContext, it more closely resembles Spring's own MessageListener containers.

4.1 Message Handler

Spring Integration's `MessageHandler` interface is implemented by many of the components within the framework. In other words, this is not part of the public API, and a developer would not typically implement `MessageHandler` directly. Nevertheless, it is used by a Message Consumer for actually handling the consumed Messages, and so being aware of this strategy interface does help in terms of understanding the overall role of a consumer. The interface is defined as follows:

```
public interface MessageHandler {

    void handleMessage(Message<?> message);

}
```

Despite its simplicity, this provides the foundation for most of the components that will be covered in the following chapters (Routers, Transformers, Splitters, Aggregators, Service Activators, etc). Those components each perform very different functionality with the Messages they handle, but the requirements for actually receiving a Message are the same, and the choice between polling and event-driven behavior is also the same. Spring Integration provides two endpoint implementations that "host" these callback-based handlers and allow them to be connected to Message Channels.

4.2 Event-Driven Consumer

Because it is the simpler of the two, we will cover the Event-Driven Consumer endpoint first. You may

recall that the `SubscribableChannel` interface provides a `subscribe()` method and that the method accepts a `MessageHandler` parameter (as shown in the section called “SubscribableChannel”):

```
subscribableChannel.subscribe(messageHandler);
```

Since a handler that is subscribed to a channel does not have to actively poll that channel, this is an Event-Driven Consumer, and the implementation provided by Spring Integration accepts a a `SubscribableChannel` and a `MessageHandler`:

```
SubscribableChannel channel = (SubscribableChannel) context.getBean("exampleSubscribableChannel");

EventDrivenConsumer consumer = new EventDrivenConsumer(channel, exampleHandler);
```

4.3 Polling Consumer

Spring Integration also provides a `PollingConsumer`, and it can be instantiated in the same way except that the channel must implement `PollableChannel`:

```
PollableChannel channel = (PollableChannel) context.getBean("examplePollableChannel");

PollingConsumer consumer = new PollingConsumer(channel, exampleHandler);
```

There are many other configuration options for the Polling Consumer. For example, the trigger can be provided:

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

consumer.setTrigger(new IntervalTrigger(30, TimeUnit.SECONDS));
```

Spring Integration currently provides two implementations of the `Trigger` interface: `IntervalTrigger` and `CronTrigger`. The `IntervalTrigger` 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):

```
IntervalTrigger trigger = new IntervalTrigger(1000);
trigger.setInitialDelay(5000);
trigger.setFixedRate(true);
```

The `CronTrigger` simply requires the cron expression (see the Javadoc for details):

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

In addition to the trigger, several other polling-related configuration properties may be specified:

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

consumer.setMaxMessagesPerPoll(10);

consumer.setReceiveTimeout(5000);
```

A Polling Consumer may even delegate to a Spring `TaskExecutor` and participate in Spring-managed

transactions. The following example shows the configuration of both:

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

TaskExecutor taskExecutor = (TaskExecutor) context.getBean("exampleExecutor");
consumer.setTaskExecutor(taskExecutor);

PlatformTransactionManager txManager = (PlatformTransationManager) context.getBean("exampleTxManager");
consumer.setTransactionManager(txManager);
```

The examples above show dependency lookups, but keep in mind that these consumers will most often be configured as Spring *bean definitions*. In fact, Spring Integration also provides a `FactoryBean` 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. The namespace-based configuration will be featured as each component type is introduced.

Note

Many of the `MessageHandler` implementations are also capable of generating reply Messages. As mentioned above, sending Messages is trivial when compared to the Message reception. Nevertheless, *when* and *how many* reply Messages are sent depends on the handler type. For example, an *Aggregator* waits for a number of Messages to arrive and is often configured as a downstream consumer for a *Splitter* which may generate multiple replies for each Message it handles. When using the namespace configuration, you do not strictly need to know all of the details, but it still might be worth knowing that several of these components share a common base class, the `AbstractReplyProducingMessageHandler`, and it provides a `setOutputChannel(..)` method.

4.4 Namespace Support

Throughout the reference manual, you will see specific configuration examples for endpoint elements, such as router, transformer, service-activator, and so on. Most of these will support an "input-channel" attribute and many will support an "output-channel" attribute. After being parsed, these endpoint elements produce an instance of either the `PollingConsumer` or the `EventDrivenConsumer` depending on the type of the "input-channel" that is referenced: `PollableChannel` or `SubscribableChannel` respectively. When the channel is pollable, then the polling behavior is determined based on the endpoint element's "poller" sub-element. For example, a simple interval-based poller with a 1-second interval would be configured like this:

```
<poller>
    <interval-trigger interval="1000"/>
</poller>
```

For a poller based on a Cron expression, use the "cron-trigger" child element instead:

```
<poller>
    <cron-trigger expression="*/10 * * * * MON-FRI"/>
</poller>
```

Spring Integration also provides transaction support for the pollers so that each receive-and-forward operation can be performed as an atomic unit-of-work. To configure transactions for a poller, simply add the <transactional/> sub-element. The attributes for this element should be familiar to anyone who has experience with Spring's Transaction management:

```
<poller>
    <interval-trigger interval="1000"/>
    <transactional transaction-manager="txManager"
                   propagation="REQUIRES_NEW"
                   isolation="REPEATABLE_READ"
                   timeout="10000"
                   read-only="false"/>
</poller>
```

The polling threads may be executed by any instance of Spring's `TaskExecutor` abstraction. This enables concurrency for an endpoint or group of endpoints. As a convenience, there is also namespace support for creating a simple thread pool executor. The <thread-pool-task-executor/> element defines attributes for common concurrency settings such as core-size, max-size, and queue-capacity. Configuring a thread-pooling executor can make a substantial difference in how the endpoint performs under load. These settings are available per-endpoint since the performance of an endpoint is one of the major factors to consider (the other major factor being the expected volume on the channel to which the endpoint subscribes). To enable concurrency for a polling endpoint that is configured with the XML namespace support, provide the 'task-executor' reference on its <poller/> element and then provide one or more of the properties shown below:

```
<poller task-executor="pool"/>
    <interval-trigger interval="5" time-unit="SECONDS"/>
    <thread-pool-task-executor id="pool"
                               core-size="5"
                               max-size="25"
                               queue-capacity="20"
                               keep-alive-seconds="120"/>
</poller>
```

If no 'task-executor' is provided, the consumer's handler will be invoked in the caller's thread. Note that the "caller" is usually the MessageBus' task scheduler. Also, keep in mind that the 'task-executor' attribute can provide a reference to any implementation of Spring's `TaskExecutor` interface by specifying the bean name. The thread pool elements is simply provided for convenience.

5. Service Activator

5.1 Introduction

The Service Activator is the endpoint type for connecting any Spring-managed Object to an input channel so that it may play the role of a service. If the service produces output, it may also be connected to an output channel. Alternatively, an output producing service may be located at the end of a processing pipeline or message flow in which case, the inbound Message's "replyChannel" header can be used. This is the default behavior if no output channel is defined, and as with most of the configuration options you'll see here, the same behavior actually applies for most of the other components we have seen.

5.2 The <service-activator/> Element

To create a Service Activator, use the 'service-activator' element with the 'input-channel' and 'ref' attributes:

```
<service-activator input-channel="exampleChannel" ref="exampleHandler"/>
```

The configuration above assumes that "exampleHandler" either contains a single method annotated with the @ServiceActivator annotation or that it contains only one public method at all. To delegate to an explicitly defined method of any object, simply add the "method" attribute.

```
<service-activator input-channel="exampleChannel" ref="somePojo" method="someMethod"/>
```

In either case, when the service method returns a non-null value, the endpoint will attempt to send the reply message to an appropriate reply channel. To determine the reply channel, it will first check if an "output-channel" was provided in the endpoint configuration:

```
<service-activator input-channel="exampleChannel" output-channel="replyChannel"
                   ref="somePojo" method="someMethod"/>
```

If no "output-channel" is available, it will then check the Message's `RETURN_ADDRESS` header value. If that value is available, it will then check its type. If it is a `MessageChannel`, the reply message will be sent to that channel. If it is a `String`, then the endpoint will attempt to resolve the channel name to a channel instance. If the channel cannot be resolved, then a `ChannelResolutionException` will be thrown.

6. 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, etc. 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 is an inbound and outbound adapter, and each may be configured with XML elements provided in the core namespace.

6.1 The <inbound-channel-adapter> element

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 'cronExpression' for an individual channel-adapter's schedule, provide a 'poller' element with either an 'interval-trigger' (in milliseconds) or 'cron-trigger' sub-element:

```
<inbound-channel-adapter ref="source1" method="method1" channel="channel1">
    <poller>
        <interval-trigger interval="5000"/>
    </poller>
</inbound-channel-adapter>

<inbound-channel-adapter ref="source2" method="method2" channel="channel2">
    <poller>
        <cron-trigger expression="30 * * * * MON-FRI"/>
    </poller>
</channel-adapter>
```

6.2 The <outbound-channel-adapter/> element

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

```
<outbound-channel-adapter channel="channel1" ref="target1" method="method1"/>
```

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

```
<outbound-channel-adapter channel="channel2" ref="target2" method="method2">
    <poller>
        <interval-trigger interval="3000"/>
    </poller>
</outbound-channel-adapter>
```

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.

7. Router

7.1 Router Implementations

Since content-based routing often requires some domain-specific logic, most use-cases will require Spring Integration's options for delegating to POJOs using the XML namespace support and/or Annotations. Both of these are discussed below, but first we present a couple implementations that are available out-of-the-box since they fulfill generic, but common, requirements.

PayloadTypeRouter

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="payloadTypeChannelMap">
        <map>
            <entry key="java.lang.String" value-ref="stringChannel"/>
            <entry key="java.lang.Integer" value-ref="integerChannel"/>
        </map>
    </property>
</bean>
```

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>
```

Note

The router implementations share some common properties, such as "defaultOutputChannel" and "resolutionRequired". If "resolutionRequired" is set to "true", and the router is unable to determine a target channel (e.g. there is no matching payload for a PayloadTypeRouter and no "defaultOutputChannel" has been specified), then an Exception will be thrown.

7.2 The <router> element

The "router" element provides a simple way to connect a router to an input channel, and also accepts the optional default output channel. The "ref" may provide the bean name to one of the implementations described above:

```
<router ref="payloadTypeRouter" input-channel="input1" default-output-channel="defaultOutput1"/>

<router ref="recipientListRouter" input-channel="input2" default-output-channel="defaultOutput2"/>
```

Alternatively, the "ref" may point to a simple Object that contains the @Router annotation (see below), or the "ref" may be combined with an explicit "method" name. When specifying a "method", the same behavior applies as described in the @Router annotation section below.

```
<router input-channel="input" ref="somePojo" method="someMethod"/>
```

7.3 The @Router Annotation

When using the `@Router` annotation, the annotated method can return either the `MessageChannel` or `String` type. In the case of the latter, the endpoint will resolve the channel name as it does for the default output. Additionally, the method can return either a single value or a collection. When a collection is returned, the reply message will be sent to multiple channels. To summarize, the following method signatures are all valid.

```
@Router
public MessageChannel route(Message message) {...}

@Router
public List<MessageChannel> route(Message message) {...}

@Router
public String route(Foo payload) {...}

@Router
public List<String> route(Foo payload) {...}
```

In addition to payload-based routing, a common requirement is to route based on metadata available within the message header as either a property or attribute. Rather than requiring use of the `Message` type as the method parameter, the `@Router` annotation may also use the same @Header parameter annotation that was introduced above.

```
@Router
public List<String> route(@Header("orderStatus") OrderStatus status)
```

Note

For routing of XML-based Messages, including XPath support, see Chapter 20, *Dealing with XML Payloads*.

8. Transformer

8.1 Introduction

Message Transformers play a very important role in enabling the loose-coupling of Message Producers and Message Consumers. Rather than requiring every Message-producing component to know what type is expected by the next consumer, Transformers can be added between those components. Generic transformers, such as one that converts a String to an XML Document, are also highly reusable.

For some systems, it may be best to provide a Canonical Data Model, but Spring Integration's general philosophy is not to require any particular format. Rather, for maximum flexibility, Spring Integration aims to provide the simplest possible model for extension. As with the other endpoint types, the use of declarative configuration in XML and/or Annotations enables simple POJOs to be adapted for the role of Message Transformers. These configuration options will be described below.

Note

For the same reason of maximizing flexibility, Spring does not require XML-based Message payloads. Nevertheless, the framework does provide some convenient Transformers for dealing with XML-based payloads if that is indeed the right choice for your application. For more information on those transformers, see Chapter 20, *Dealing with XML Payloads*.

8.2 The <transformer> Element

The <transformer> element is used to create a Message-transforming endpoint. In addition to "input-channel" and "output-channel" attributes, it requires a "ref". The "ref" may either point to an Object that contains the @Transformer annotation on a single method (see below) or it may be combined with an explicit method name value provided via the "method" attribute.

The method that is used for transformation may expect either the `Message` type or the payload type of inbound Messages. The return value of the method can be any type. If the return value is itself a `Message`, that will be passed along to the transformer's output channel. If the return value is *null*, then no reply Message will be sent (effectively the same behavior as a Message Filter). Otherwise, the return value will be sent as the payload of a Message.

8.3 The @Transformer Annotation

The `@Transformer` annotation can also be added to methods that expect either the `Message` type or the message payload type. The return value will be handled in the exact same way as described above in the section describing the <transformer> element.

```
@Transformer
Order generateOrder(String productId) {
    return new Order(productId);
}
```

Transformer methods may also accept the @Header and @Headers annotations. For example, one of those annotations may complement the payload Object as an additional parameter:

```
@Transformer
Order generateOrder(String productId, @Header("customerName") String customer) {
    return new Order(productId, customer);
}
```

9. Splitter

9.1 Introduction

The Splitter is a component whose role is to partition a message in several parts, and send the resulting messages to be processed independently. Very often, they are upstream producers in a pipeline that includes an Aggregator.

9.2 Functionality

9.3 Programming model

The API for performing splitting consists from one base class, AbstractMessageSplitter, which is a MessageConsumer implementation, encapsulating features which are common to splitters, such as filling in the appropriate message headers CORRELATION_ID, SEQUENCE_SIZE, and SEQUENCE_NUMBER on the messages that are produced. This allows to track down the messages and the results of their processing (in a typical scenario, these headers would be copied over to the messages that are produced by the various transforming endpoints), and use them, for example, in a Composed Message Processor scenario.

An excerpt from AbstractMessageSplitter can be seen below:

```
public abstract class AbstractMessageSplitter
                extends AbstractReplyProducingMessageConsumer {
  ...
  protected abstract Object splitMessage(Message<?> message);

}
```

For implementing a specific Splitter in an application, a developer can extend AbstractMessageSplitter and implement the splitMessage method, thus defining the actual logic for splitting the messages. The return value can be one of the following:

- a Collection (or subclass thereof) or an array of Message objects - in this case the messages will be sent as such (after the CORRELATION_ID, SEQUENCE_SIZE and SEQUENCE_NUMBER will be populated). Using this approach gives more control to the developer, for example for populating custom message headers as part of the splitting process.

- a Collection (or subclass thereof) or an array of non-Message objects - works like the prior case, except that each collection element will be used as a Message payload. Using this approach allows to focus on the

- a Message or non-Message object (but not a Collection or an Array) - it works like the previous cases, except that there is a single message to be sent out.

In Spring Integration, any POJO can implement the splitting algorithm, provided that it defines a method that accepts a single argument and has a return value. In this case, the return value of the method will be interpreted as described above. The input argument might either be a Message or a simple POJO. In the latter case, the splitter will receive the payload of the incoming message.

9.4 Configuring a Splitter using XML

A splitter can be configured through XML as follows:

```
<channel id="inputChannel"/>

<splitter id="splitter" ❶
  ref="splitterBean" ❷
  method="split" ❸
  input-channel="inputChannel" ❹
  output-channel="outputChannel" ❺/>

<channel id="outputChannel"/>

<beans:bean id="splitterBean" class="sample.PojoSplitter"/>
```

❶ The id of the splitter is *optional*.
❷ A reference to a bean defined in the application context. The bean must implement the splitting logic as described in the section above. *Required*.
❸ The method (defined on the bean specified above) that implements the splitting logic. *Optional*.
❹ The input channel of the splitter. *Required*.
❺ The channel where the splitter will send the results of splitting the incoming message. *Optional (because incoming messages can specify a reply channel themselves)*.

9.5 Configuring a Splitter with Annotations

The `@Splitter` annotation is applicable to methods that expect either the `Message` type or the message payload type, and the return values of the method should be a collection of any type. If the returned values are not actual `Message` objects, then each of them will be sent as the payload of a message. Those messages will be sent to the output channel as designated for the endpoint on which the `@Splitter` is defined.

```
@Splitter
List<LineItem> extractItems(Order order) {
    return order.getItems()
}
```

10. Aggregator

10.1 Introduction

Basically a mirror-image of the Splitter, the Aggregator is a type of Message Consumer that receives multiple Messages and combines them into a single Message. In fact, Aggregators are often downstream consumers in a pipeline that includes a Splitter.

Technically, the Aggregator is more complex than a Splitter, because it is required to maintain state (the Messages to-be-aggregated), to decide when the complete group of Messages is available, and to timeout if necessary. Furthermore, in case of a timeout, the Aggregator needs to know whether to send the partial results or to discard them to a separate channel.

10.2 Functionality

The Aggregator combines a group of related messages, by storing and grouping them, until the group is deemed complete. At that point, the Aggregator will create a single message by processing the whole group, and will send the result message further.

As messages might arrive with a certain delay (or certain messages from the group might not arrive at all), the Aggregator can specify a timeout (counted from the moment when the first message in the group has arrived), and whether, in the case of a timeout, the group should be discarded, or the Aggregator should merely attempt to create a single message out of what has arrived so far. An important aspect of implementing an Aggregator is providing the logic that has to be executed when the aggregation (creation of a single message out of many) takes place.

In Spring Integration, the grouping of the messages for Aggregation is done based on their CORRELATION_ID message header (i.e. the messages with the same CORRELATION_ID will be grouped together).

An important concern with respect to the timeout is, what happens if late messages arrive after the aggregation has taken place? In this case, a configuration option allows the user to decide whether they should be discarded or not.

10.3 Programming model

The Aggregation API consists of a number of classes:

- The base class `AbstractMessageAggregator` and its subclass `MethodInvokingMessageAggregator`

- The `CompletionStrategy` interface and its default implementation `SequenceSizeCompletionStrategy`

The `AbstractMessageAggregator` is a `MessageConsumer` implementation, encapsulating the common functionalities of an Aggregator, which are: storing messages until the message sequence to aggregate is complete (and grouping them according to their CORRELATION_ID), and implementing the timeout functionality. The responsibility of deciding whether the message sequence is complete is delegated to a `CompletionStrategy` instance.

A brief highlight of the base `AbstractMessageAggregator` (the responsibility of implementing the aggregateMessages method is left to the developer):

```
public abstract class AbstractMessageAggregator
              extends AbstractMessageBarrierConsumer {

  private volatile CompletionStrategy completionStrategy
                           = new SequenceSizeCompletionStrategy();
  ....

  protected abstract Message<?> aggregateMessages(List<Message<?>> messages);

}
```

For implementing a specific aggregator object for an application, a developer can extend `AbstractMessageAggregator` and implement the `aggregateMessages` method. However, there are better suited (which reads, less coupled to the API) solutions for implementing the aggregation logic, which can be configured easily either through XML or through annotations.

In general, any ordinary Java class (i.e. POJO) can implement the aggregation algorithm. For doing so, it must provide a method that accepts as an argument a single java.util.List (parametrized lists are supported as well). This method will be invoked for aggregating messages, as follows:

- if the argument is a parametrized java.util.List, and the parameter type is assignable to Message, then the whole list of messages accumulated for aggregation will be sent to the aggregator
- if the argument is a non-parametrized java.util.List or the parameter type is not assignable to Message, then the method will receive the payloads of the accumulated messages
- if the return type is not assignable to Message, then it will be treated as the payload for a Message that will be created automatically by the framework.

Note

In the interest of code simplicity, and promoting best practices such as low coupling, testability, etc., the preferred way of implementing the aggregation logic is through a POJO, and using the XML or annotation support for setting it up in the application.

The `CompletionStrategy` interface is defined as follows:

```
public interface CompletionStrategy {

  boolean isComplete(List<Message<?>> messages);

}
```

In general, any ordinary Java class (i.e. POJO) can implement the completion decision mechanism. For doing so, it must provide a method that accepts as an argument a single java.util.List (parametrized lists are supported as well), and returns a boolean value. This method will be invoked after the arrival of a new message, to decide whether the group is complete or not, as follows:

- if the argument is a parametrized java.util.List, and the parameter type is assignable to Message, then the whole list of messages accumulated in the group will be sent to the method
- if the argument is a non-parametrized java.util.List or the parameter type is not assignable to Message, then the method will receive the payloads of the accumulated messages
- the method must return true if the message group is complete and ready for aggregation, and false otherwise.

Spring Integration provides an out-of-the box implementation for `CompletionStrategy`, the `SequenceSizeCompletionStrategy` This implementation uses the SEQUENCE_NUMBER and SEQUENCE_SIZE of the arriving messages for deciding when a message group is complete and ready to be aggregated.

10.4 Configuring an Aggregator with XML

Spring Integration supports the configuration of an aggregator via XML through the <aggregator/> element. A completely defined sample on how to define such an element is presented below, as well as it:

```
<channel id="inputChannel"/>

<aggregator id="completelyDefinedAggregator" ❶
    input-channel="inputChannel" ❷
    output-channel="outputChannel"  ❸
    discard-channel="discardChannel"  ❹
    ref="aggregatorBean" ❺
    method="add" ❻
    completion-strategy="completionStrategyBean"  ❼
    completion-strategy-method="checkCompleteness" ❽
    timeout="42" ❾
    send-partial-result-on-timeout="true" ❿
    reaper-interval="135" 11
    tracked-correlation-id-capacity="99" 12
    send-timeout="86420000" 13 />

<channel id="outputChannel"/>

<bean id="aggregatorBean" class="sample.PojoAggregator"/>

<bean id="completionStrategyBean" class="sample.PojoCompletionStrategy"/>
```

❶ The id of the aggregator is *optional*.
❷ The input channel of the aggregator. *Required*.
❸ The channel where the aggregator will send the aggregation results. *Optional (because incoming messages can specify a reply channel themselves)*.
❹ The channel where the aggregator will send the messages that timed out (if `send-partial-results-on-timeout` is *false)*. *Optional*.
❺ A reference to a bean defined in the application context. The bean must implement the aggregation logic as described above. *Required*.
❻ A method defined on the bean referenced by `ref`, that implements the message aggregation algorithm. *Optional, with restrictions (see above).*
❼ A reference to a bean that implements the decision algorithm as to whether a given message group is complete. The bean can be an implementation of the CompletionStrategy interface or a POJO. In the latter case the completion-strategy-method attribute must be defined as well. *Optional (by default, the aggregator* .
❽ A method defined on the bean referenced by `completion-strategy`, that implements the completion decision algorithm. *Optional, with restrictions (requires `completion-strategy` to be present).*
❾ The timeout for aggregating messages (counted from the arrival of the first message). *Optional*.
❿ Whether upon the expiration of the timeout, the aggregator shall try to aggregate the already arrived messages. *Optional (false by default)*.
The interval (in milliseconds) at which a reaper task is executed, checking if there are any timed out groups. *Optional*.
The capacity of the correlation id tracker. Remembers the already processed correlation ids, preventing the formation of new groups for messages that arrive after their group has been already processed (aggregated or discarded). *Optional*.
The timeout for sending out messages. *Optional*.

An implementation of the aggregator bean, for example, looks as follows:

```
public class PojoAggregator {

  public Long add(List<Long> results) {
    long total = 0l;
    for (long partialResult: results) {
      total += partialResult;
    }
    return total;
  }

}
```

An implementation of the completion strategy bean for the example above may be as follows:

```
public class PojoCompletionStrategy {
...
  public boolean checkCompleteness(List<Long> numbers) {
    int sum = 0;
    for (long number: numbers) {
      sum += number;
    }
    return sum >= maxValue;
  }
```

```
}
```

Wherever it makes sense, the completion strategy method and the aggregator method can be combined in a single bean.

10.5 Configuring an Aggregator with Annotations

An aggregator configured using annotations can look like this.

```
public class Waiter {
  ...

  @Aggregator #
  public Delivery aggregatingMethod(List<OrderItem> items) {
    ...
  }

  @CompletionStrategy #
  public boolean completionChecker(List<Message<?>> messages) {
    ...
  }

}
```

❶ An annotation indicating that this method shall be used as an aggregator. Must be specified if this class will be used as an aggregator.

❷ An annotation indicating that this method shall be used as the completion strategy of an aggregator. If not present of the method, the aggregator will use the SequenceSizeCompletionStrategy.

All the configuration options provided by xml element are available for the @Aggregator annotation.

The aggregator can be either referenced explicitly from XML or, if the @MessageEndpoint is defined on the class, detected automatically through classpath scanning.

11. Resequencer

11.1 Introduction

Related to the Aggregator, albeit different from a functional standpoint, is the Resequencer.

11.2 Functionality

The Resequencer works in a similar way to the Aggregator, in the sense that it uses the CORRELATION_ID to store messages in groups, the difference being that all what the Resequencer does, is to release them in the order of their SEQUENCE_NUMBER.

With respect to that, the user might opt to release all messages at once (after the whole sequence, according to the SEQUENCE_SIZE, has been released), or as soon as a valid sequence is available. Another option is to set a timeout, deciding whether to drop the whole sequence if the timeout has expired, and not all messages have arrived, or to release the messages accumulated so far, in the appropriate order.

11.3 Configuring a Resequencer with XML

Configuring a resequencer requires only including the appropriate element in XML.

A sample resequencer configuration is shown below.

```
<channel id="inputChannel"/>

<channel id="outputChannel"/>

<resequencer id="completelyDefinedResequencer" #
  input-channel="inputChannel" #
  output-channel="outputChannel" #
  discard-channel="discardChannel" #
  release-partial-sequences="true" #
  timeout="42" #
  send-partial-result-on-timeout="true" #
  reaper-interval="135" #
  tracked-correlation-id-capacity="99" #
  send-timeout="86420000" # />
```

❶ The id of the resequencer is *optional*.
❷ The input channel of the resequencer. *Required*.
❸ The channel where the resequencer will send the reordered messages. *Optional*.
❹ The channel where the resequencer will send the messages that timed out (if `send-partial-result-on-timeout` is *false)*. *Optional*.

❺ Whether to send out ordered sequences as soon as they are available, or only after the whole message group arrives. *Optional (true by default)*.

❻ The timeout for reordering message sequences (counted from the arrival of the first message). *Optional*.

❼ Whether, upon the expiration of the timeout, the ordered group shall be sent out (even if some of the messages are missing). *Optional (false by default)*.

❽ The interval (in milliseconds) at which a reaper task is executed, checking if there are any timed out groups. *Optional*.

❾ The capacity of the correlation id tracker. Remembers the already processed correlation ids, preventing the formation of new groups for messages that arrive after their group has been already processed (reordered or discarded). *Optional*.

❿ The timeout for sending out messages. *Optional*.

Note

Since there is no custom behaviour to be implemented in Java classes for resequencers, there is no annotation support for it.

12. File Support

12.1 Introduction

Spring Integration File extends the Spring Integration Core with dedicated vocabulary to deal with reading, writing and transforming files. There is a namespace that enables elements that define channel adapters dedicated to files and support for transformers that transform files into strings or byte arrays.

This section will explain the workings of `FileReadingMessageSource`, `FileWritingMessageHandler` and how to configure them as *beans*. Also the support for dealing with files through file specific implementations of `Transformer` will be discussed. Finally the file specific namespace will be explained.

12.2 Reading Files

A `FileReadingMessageSource` can be used to consume files from the filesystem. This is an implementation of `MessageSource` that creates messages from a file system directory.

```
<bean id="pollableFileSource"
      class="org.springframework.integration.file.FileReadingMessageSource"
        p:inputDirectory="file:${input.directory.property}"/>
```

To prevent creating messages for certain files, you may supply a {@link FileListFilter}. By default, an `AcceptOnceFileListFilter` is used. This filter ensures files are picked up only once from the directory.

```
<bean id="pollableFileSource"
      class="org.springframework.integration.file.FileReadingMessageSource"
        p:inputDirectory="file:${input.directory.property}"
        p:filter-ref="customFilterBean"/>
```

A common problem with reading files is that a file may be detected before it is ready. The default `AcceptOnceFileListFilter` does not prevent this. In most cases, this can be prevented if the file-writing process renames each file as soon as it is ready for reading. A pattern-matching filter that accepts only files that are ready (e.g. based on a known suffix), composed with the default `AcceptOnceFileListFilter` allows for this.

```
<bean id="pollableFileSource"
      class="org.springframework.integration.file.FileReadingMessageSource"
        p:inputDirectory="file:${input.directory.property}"
        p:filter-ref="compositeFilter"/>
<bean id="compositeFilter" class="org.springframework.integration.file.CompositeFileListFilter">
    <constructor-arg>
        <list>
            <bean class="org.springframework.integration.file.AcceptOnceFileListFilter" />
            <bean class="org.springframework.integration.file.PatternMatchingFileListFilter">
                <constructor-arg value="^test.*$"/>
            </bean>
```

```
        </list>
    </constructor-arg>
</bean>
```

The configuration can be simplified using the file specific namespace. To do this use the following template.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:integration="http://www.springframework.org/schema/integration"
        xmlns:file="http://www.springframework.org/schema/integration/file"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                        http://www.springframework.org/schema/integration
                        http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
                        http://www.springframework.org/schema/integration/file
                        http://www.springframework.org/schema/integration/file/spring-integration-file-1.0.xsd"
</beans>
```

Within this namespace you can reduce the FileReadingMessageSource and wrap it in an InboundChannelAdapter like this:

```
<file:inbound-channel-adapter id="filesIn"
       directory="file:${input.directory.property}"/>                          (1)
<file:inbound-channel-adapter id="filesIn"
       directory="file:${input.directory.property}"
       filter="customFilterBean" />                                        (2)
<file:inbound-channel-adapter id="filesIn"
       directory="file:${input.directory.property}"
       filename-pattern="^test.*$" />                                          (3)
```

Where (1) is relying on the default filter that just prevents duplication, (2) is using a custom filter and (3) is using the *filename-pattern* attribute to add a `Pattern` based filter to the `FileReadingMessageSource`

12.3 Writing files

To write messages to the file system you can use a `FileWritingMessageHandler` . This class can deal with File or byte[] payloads and otherwise invokes the toString() method on the payload to establish the contents of the File. In its simplest form the `FileWritingMessageHandler` just needs a parent directory for the files.

Additionally, you can configure the encoding and the charset that will be used in case of a fallback on the toString() method.

To make things easier you can configure the FileWritingMessageHandler as part of an outbound channel adapter using the namespace.

```
        <file:outbound-channel-adapter id="filesOut" directory="file:${input.directory.property}"/>
```

If you have more elaborate requirements to the payload to file conversion you could extend the FileWritingMessageHandler, but a much better option is to rely on a `Transformer`

12.4 File Transformers

To transform data read from the file system to objects and the other way around you need to do some work. Contrary to `FileReadingMessageSource` and to a lesser extent `FileWritingMessageHandler` it is very likely that you will need your own mechanism to get the job done. For this you can implement the `Transformer` interface. Or extend the `AbstractFilePayloadTransformer` for inbound messages. Some obvious implementations have been provided.

`FileToByteArrayTransformer` transforms Files into byte[]s using `FileCopyUtils` . It is often better to use a sequence of transformers than to put all transformations in a single class, in that case the File to byte[] conversion might be a logical first step.

`FileToStringTransformer` will convert Files to Strings as the name suggests. This is mainly useful for debugging.

To configure File specific transformers you can use the appropriate elements from the file namespace.

```
<file-to-bytes-transformer input-channel="intput" output-channel="output" delete-files="true"/>
```

The *delete-files* option signals the transformer to delete the File after the transformation is done. This is in no way a replacement for using the `AcceptOnceFileListFilter` with the PollableFileSource in a multi-threaded environment (e.g. Spring Integration in general).

13. JMS Support

Spring Integration provides Channel Adapters for receiving and sending JMS messages. The inbound Channel Adapter uses Spring's `JmsTemplate` to receive based on a polling period, and the outbound Channel Adapter uses the `JmsTemplate` to convert and send a JMS Message on demand.

Spring Integration also provides inbound and outbound JMS Gateways. The inbound gateway relies on Spring's `DefaultMessageListenerContainer` for Message-driven reception that is also capable of sending a return value to the "reply-to" Destination as provided by the received Message. The outbound Gateway uses a JMS `QueueRequestor` for request/reply operations. In other words, while the inbound and outbound Channel Adapters are for unidirectional Messaging, the Gateways are intended for handling request/reply operations.

13.1 Inbound Channel Adapter

The inbound Channel Adapter requires a reference to either a single `JmsTemplate` instance or both `ConnectionFactory` and `Destination` (a 'destinationName' can be provided in place of the 'destination' reference). The following example defines an inbound Channel Adapter with a `Destination` reference.

```
<jms:inbound-channel-adapter id="jmsIn" destination="inQueue" channel="exampleChannel"/>
```

13.2 Outbound Channel Adapter

The `JmsSendingMessageHandler` implements the `MessageHandler` interface and is capable of converting Spring Integration `Messages` to JMS messages and then sending to a JMS destination. It requires either a 'jmsTemplate' reference or both 'connectionFactory' and 'destination' references (again, the 'destinationName' may be provided in place of the 'destination). As with the inbound Channel Adapter, the easiest way to configure this adapter is with the namespace support. The following configuration will produce an adapter that receives Spring Integration Messages from the "exampleChannel" and then converts those into JMS Messages and sends them to the JMS Destination reference whose bean name is "outQueue".

```
<jms:outbound-channel-adapter id="jmsOut" destination="outQueue" channel="exampleChannel"/>
```

13.3 Inbound Gateway

Spring Integration's message-driven `JmsInboundGateway` delegates to a `MessageListener` container, supports dynamically adjusting concurrent consumers, and can also handle replies. The `JmsInboundGateway` requires references to a `ConnectionFactory`, and a `Destination` (or 'destinationName'). The following example defines a `JmsInboundGateway` that receives from the

JMS queue called "exampleQueue" and sends to the Spring Integration channel named "exampleChannel".

```
<jms:inbound-gateway id="jmsInGateway" destination="inQueue" request-channel="exampleChannel"/>
```

Note

The default behavior for an outbound gateway is to reply with the Spring Integration Message as the JMS Message body. This works well when you are connecting two systems that are both running Spring Integration-based applications. However, if you prefer to just send the Spring Integration Message's payload as the JMS reply Message's body, then set the "extract-payload-for-reply" attribute to "true".

13.4 Outbound Gateway

The outbound Gateway uses a `QueueRequestor` so that reply Messages will be automatically correlated. Notice that the "reply-channel" is also provided.

```
<jms:outbound-gateway id="jmsOutGateway"
                      jms-queue="outQueue"
                      request-channel="outboundJmsRequests"
                      reply-channel="jmsReplies"/>
```

Note

For all of these JMS adapters, you can also specify your own "message-converter" reference. Simply provide the bean name of an instance of `MessageConverter` that is available within the same ApplicationContext.

13.5 JMS Samples

To experiment with these JMS adapters, check out the samples available within the "jms" package of the "org.springramework.integration.samples" module (in the distribution). There are two samples included. One provides inbound and outbound Channel Adapters, and the other provides inbound and outbound Gateways. They are configured to run with an embedded ActiveMQ process, but the "common.xml" can easily be modified to support either a different JMS provider or a standalone ActiveMQ process. In other words, you can split the configuration so that the inbound and outbound adapters are running in separate JVMs. If you have ActiveMQ installed, simply modify the "brokerURL" property within the configuration to use "tcp://localhost:61616" for example (instead of "vm://localhost").

14. Web Services Support

14.1 Outbound Web Service Gateways

To invoke a Web Service upon sending a message to a channel, there are two options - both of which build upon the Spring Web Services project: `SimpleWebServiceOutboundGateway` and `MarshallingWebServiceOutboundGateway`. The former will accept either a `String` or `javax.xml.transform.Source` as the message payload. The latter provides support for any implementation of the `Marshaller` and `Unmarshaller` interfaces. Both require the URI of the Web Service to be called.

```
simpleGateway = new SimpleWebServiceOutboundGateway(uri);

marshallingGateway = new MarshallingWebServiceOutboundGateway(uri, marshaller);
```

For more detail on the inner workings, see the Spring Web Services reference guide's chapter covering client access as well as the chapter covering Object/XML mapping.

14.2 Web Service Namespace Support

To configure an outbound Web Service Gateway, use the "outbound-gateway" element from the "ws" namespace:

```
<ws:outbound-gateway id="simpleGateway"
                     request-channel="inputChannel"
                     uri="http://example.org"/>
```

To use Spring OXM Marshallers and/or Unmarshallers, provide bean references:

```
<ws:outbound-gateway id="marshallingGateway"
                     request-channel="requestChannel"
                     uri="http://example.org"
                     marshaller="someMarshaller"
                     unmarshaller="someUnmarshaller"/>
```

Note

Most `Marshaller` implementations also implement the `Unmarshaller` interface. When using such a Marshaller, only the "marshaller" attribute is necessary. Even when using a Marshaller, you may also provide a reference for the "request-callback".

For either gateway type, the "message-factory" attribute can also be configured with a reference to any Spring Web Services `WebServiceMessageFactory` implementation.

15. RMI Support

15.1 Introduction

This Chapter explains how to use RMI specific channel adapters to distribute a system over multiple JVMs. The first section will deal with sending messages over RMI. The second section shows how to receive messages over RMI. The last section shows how to define rmi channel adapters through the namespace support

15.2 Outbound RMI

To send messages from a channel over RMI, simply define an `RmiOutboundGateway`. This gateway will use Spring's RmiProxyFactoryBean internally to create a proxy for a remote gateway. Note that to invoke a remote interface that doesn't use Spring Integration you should use a service activator in combination with Spring's RmiProxyFactoryBean.

To configure the outbound gateway write a bean definition like this:

```
        <bean id="rmiOutGateway" class=org.spf.integration.rmi.RmiOutboundGateway>
                <constructor-arg value="rmi://host"/>
                <property name="replyChannel" value="replies"/>
        </bean>
```

15.3 Inbound RMI

To receive messages over RMI you need to use a `RmiInboundGateway`. This gateway can be configured like this

```
<bean id="rmiOutGateway" class=org.spf.integration.rmi.RmiInboundGateway>
    <property name="requestChannel" value="requests"/>
</bean>
```

15.4 RMI namespace support

To configure the inbound gateway you can choose to use the namespace support for it. The following code snippet shows the different configuration options that are supported.

```
<rmi:inbound-gateway id="gatewayWithDefaults" request-channel="testChannel"/>

<rmi:inbound-gateway id="gatewayWithCustomProperties" request-channel="testChannel"
                     expect-reply="false" request-timeout="123" reply-timeout="456"/>
```

```
<rmi:inbound-gateway id="gatewayWithHost" request-channel="testChannel"
                     registry-host="localhost"/>

<rmi:inbound-gateway id="gatewayWithPort" request-channel="testChannel"
                     registry-port="1234"/>

<rmi:inbound-gateway id="gatewayWithExecutorRef" request-channel="testChannel"
                     remote-invocation-executor="invocationExecutor"/>
```

To configure the outbound gateway you can use the namespace support as well. The following code snippet shows the different configuration for an outbound rmi gateway.

```
<rmi:outbound-gateway id="gateway"
                      request-channel="localChannel"
                      remote-channel="testChannel"
                      host="localhost"/>
```

16. HttpInvoker Support

16.1 Introduction

HttpInvoker is a Spring-specific remoting option that essentially enables Remote Procedure Calls (RPC) over HTTP. In order to accomplish this, an outbound representation of a method invocation is serialized using standard Java serialization and then passed within an HTTP POST request. After being invoked on the target system, the method's return value is then serialized and written to the HTTP response. There are two main requirements. First, you must be using Spring on both sides since the marshalling to and from HTTP requests and responses is handled by the client-side invoker and server-side exporter. Second, the Objects that you are passing must implement `Serializable` and be available on both the client and server.

While traditional RPC provides *physical* decoupling, it does not offer nearly the same degree of *logical* decoupling as a messaging-based system. In other words, both participants in an RPC-based invocation must be aware of a specific interface and specific argument types. Interestingly, in Spring Integration, the "parameter" being sent is a Spring Integration Message, and the interface is an internal detail of Spring Integration's implementation. Therefore, the RPC mechanism is being used as a *transport* so that from the end user's perspective, it is not necessary to consider the interface and argument types. It's just another adapter to enable messaging between two systems.

16.2 HttpInvoker Inbound Gateway

To receive messages over http you need to use an `HttpInvokerInboundGateway`. Here is an example bean definition:

```
<bean id="inboundGateway"
      class="org.springframework.integration.httpinvoker.HttpInvokerInboundGateway">
    <property name="requestChannel" ref="requestChannel"/>
    <property name="replyChannel" ref="replyChannel"/>
    <property name="requestTimeout" value="30000"/>
    <property name="replyTimeout" value="10000"/>
</bean>
```

Because the inbound gateway must be able to receive HTTP requests, it must be configured within a Servlet container. The easiest way to do this is to provide a servlet definition in *web.xml*:

```
<servlet>
    <servlet-name>inboundGateway</servlet-name>
    <servlet-class>org.springframework.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>
```

Notice that the servlet name matches the bean name.

Note

If you are running within a Spring MVC application and using the BeanNameHandlerMapping, then the servlet definition is not necessary. In that case, the bean

name for your gateway can be matched against the URL path just like a Spring MVC Controller bean.

16.3 HttpInvoker Outbound Gateway

To configure the `HttpInvokerOutboundGateway` write a bean definition like this:

```
<bean id="outboundGateway"
      class="org.springframework.integration.httpinvoker.HttpInvokerOutboundGateway">
    <property name="replyChannel" ref="replyChannel"/>
</bean>
```

The outbound gateway is a `MessageHandler` and can therefore be registered with either a `PollingConsumer` or `EventDrivenConsumer`. The URL must match that defined by an inbound HttpInvoker Gateway as described in the previous section.

16.4 HttpInvoker Namespace Support

Spring Integration provides an "httpinvoker" namespace and schema definition. To include it in your configuration, simply provide the following URI within a namespace declaration: 'http://www.springframework.org/schema/integration/httpinvoker'. The schema location should then map to 'http://www.springframework.org/schema/integration/httpinvoker/spring-integration-httpinvoker-1.0.xsd'.

To configure the inbound gateway you can choose to use the namespace support for it. The following code snippet shows the different configuration options that are supported.

```
<httpinvoker:inbound-gateway id="inboundGateway"
                             request-channel="requestChannel"
                             request-timeout="10000"
                             expect-reply="false"
                             reply-timeout="30000"/>
```

Note

A 'reply-channel' may also be provided, but it is recommended to rely on the temporary anonymous channel that will be created automatically for handling replies.

To configure the outbound gateway you can use the namespace support as well. The following code snippet shows the different configuration for an outbound HttpInvoker gateway. Only the 'url' and 'request-channel' are required.

```
<httpinvoker:outbound-gateway id="outboundGateway"
                              url="http://localhost:8080/example"
                              request-channel="requestChannel"
                              request-timeout="5000"
                              reply-channel="replyChannel"
```

```
                                reply-timeout="10000"/>
```

17. Mail Support

17.1 Mail-Sending Channel Adapter

Spring Integration provides support for outbound email with the `MailSendingMessageHandler`. It delegates to a configured instance of Spring's `JavaMailSender`:

```
JavaMailSender mailSender = (JavaMailSender) context.getBean("mailSender");

MailSendingMessageHandler mailSendingHandler = new MailSendingMessageHandler(mailSender);
```

`MailSendingMessageHandler` various mapping strategies use Spring's `MailMessage` abstraction. If the received Message's payload is already a MailMessage instance, it will be sent directly. Therefore, it is generally recommended to precede this consumer with a Transformer for non-trivial MailMessage construction requirements. However, a few simple Message mapping strategies are supported out-of-the-box. For example, if the message payload is a byte array, then that will be mapped to an attachment. If the payload is neither a MailMessage or byte array, then a MailMessage will be created with text content corresponding to the value returned from the Spring Integration Message payload's `toString()` method. For simple text-based emails, simply provide a String-based Message payload.

The outbound MailMessage may also be configured with certain values from the `MessageHeaders`. If available, values will be mapped to the outbound mail's properties, such as the recipients (TO, CC, and BCC), the from/reply-to, and the subject. The header names are defined by the following constants:

```
MailHeaders.SUBJECT
MailHeaders.TO
MailHeaders.CC
MailHeaders.BCC
MailHeaders.FROM
MailHeaders.REPLY_TO
```

17.2 Mail Namespace Support

Spring Integration provides a namespace for mail-related configuration. To use it, configure the following schema locations.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mail="http://www.springframework.org/schema/integration/mail"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.springframework.org/schema/integration/mail
            http://www.springframework.org/schema/integration/mail/spring-integration-mail-1.0.xsd">
```

To configure an outbound Channel Adapter, provide the channel to receive from, and the MailSender:

```
<mail:outbound-channel-adapter channel="outboundMail"
                               mail-sender="mailSender"/>
```

Alternatively, provide the host, username, and password:

```
<mail:outbound-channel-adapter channel="outboundMail"
                               host="somehost" username="someuser" password="somepassword"/>
```

Note

Keep in mind, as with any outbound Channel Adapter, if the referenced channel is a PollableChannel, a <poller> sub-element should be provided with either an interval-trigger or cron-trigger.

When using the namespace support, a *header-enricher* Message Transformer is also available. This simplifies the application of the headers mentioned above to any Message prior to sending to the Mail-sending Channel Adapter. Also, note that a boolean value c

```
<mail:header-enricher subject="Example Mail"
                      to="to@example.org"
                      cc="cc@example.org"
                      bcc="bcc@example.org"
                      from="from@example.org"
                      reply-to="replyTo@example.org"
                      overwrite="false"/>
```

18. Stream Support

18.1 Introduction

In many cases application data is obtained from a stream. It is *not* recommended to send a reference to a Stream as a message payload to a consumer. Instead messages are created from data that is read from an input stream and message payloads are written to an output stream one by one.

18.2 Reading from streams

Spring Integration provides two adapters for streams. Both `ByteStreamReadingMessageSource` and `CharacterStreamReadingMessageSource` implement `MessageSource`. By configuring one of these within a channel-adapter element, the polling period can be configured, and the Message Bus can automatically detect and schedule them. The byte stream version requires an `InputStream`, and the character stream version requires a `Reader` as the single constructor argument. The `ByteStreamReadingMessageSource` also accepts the 'bytesPerMessage' property to determine how many bytes it will attempt to read into each `Message`. The default value is 1024

```
<bean class="org.springframework.integration.stream.ByteStreamReadingMessageSource">
        <constructor-arg ref="someInputStream"/>
        <property name="bytesPerMessage" value="2048"/>
</bean>

<bean class="org.springframework.integration.stream.CharacterStreamReadingMessageSource">
        <constructor-arg ref="someReader"/>
</bean>
```

18.3 Writing to streams

For target streams, there are also two implementations: `ByteStreamWritingMessageHandler` and `CharacterStreamWritingMessageHandler`. Each requires a single constructor argument - `OutputStream` for byte streams or `Writer` for character streams, and each provides a second constructor that adds the optional 'bufferSize'. Since both of these ultimately implement the `MessageHandler` interface, they can be referenced from a *channel-adapter* configuration as will be described in more detail in ???.

```
<bean class="org.springframework.integration.stream.ByteStreamWritingMessageHandler">
        <constructor-arg ref="someOutputStream"/>
        <constructor-arg value="1024"/>
</bean>

<bean class="org.springframework.integration.stream.CharacterStreamWritingMessageHandler">
        <constructor-arg ref="someWriter"/>
</bean>
```

18.4 Stream namespace support

To reduce the configuration needed for stream related channel adapters there is a namespace defined. The following schema locations are needed to use it.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration/stream"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:beans="http://www.springframework.org/schema/beans"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                        http://www.springframework.org/schema/integration/stream
                        http://www.springframework.org/schema/integration/stream/spring-integration-stream-1.0.
```

To configure the inbound channel adapter the following code snippet shows the different configuration options that are supported.

```
<stdin-channel-adapter id="adapterWithDefaultCharset"/>

<stdin-channel-adapter id="adapterWithProvidedCharset" charset="UTF-8"/>
```

To configure the outbound channel adapter you can use the namespace support as well. The following code snippet shows the different configuration for an outbound channel adapters.

```
<stdout-channel-adapter id="stdoutAdapterWithDefaultCharset" channel="testChannel"/>

<stdout-channel-adapter id="stdoutAdapterWithProvidedCharset" charset="UTF-8" channel="testChannel"/>

<stderr-channel-adapter id="stderrAdapter" channel="testChannel"/>

<stdout-channel-adapter id="newlineAdapter" append-newline="true" channel="testChannel"/>
```

19. Spring ApplicationEvent Support

Spring Integration also provides support for inbound and outbound `ApplicationEvents`. To receive events and send to a channel, simply define an instance of Spring Integration's `ApplicationEventListeningChannelAdapter`. This class in an implementation of Spring's `ApplicationListener` interface. By default it will pass all received events as Spring Integration Messages. To limit based on the type of event, configure the list of event types that you want to receive with the 'eventTypes' property.

To send Spring `ApplicationEvents`, create an instance of the `ApplicationEventPublishingMessageHandler` and register it within an endpoint. This implementation of the `MessageHandler` interface also implements Spring's `ApplicationEventPublisherAware` interface and thus acts as a bridge between Spring Integration Messages and `ApplicationEvents`.

20. Dealing with XML Payloads

20.1 Introduction

Spring Integration's XML support extends the Spring Integration Core with implementations of splitter, transformer, selector and router designed to make working with xml messages in Spring Integration simple. The provided messaging components are designed to work with xml represented in a range of formats including instances of `java.lang.String`, `org.w3c.dom.Document` and `javax.xml.transform.Source`. It should be noted however that where a DOM representation is required, for example in order to evaluate an XPath expression, the `String` payload will be converted into the required type and then converted back again to `String`. Components that require an instance of `DocumentBuilder` will create a namespace aware instance if one is not provided. Where greater control of the document being created is required an appropriately configured instance of `DocumentBuilder` should be provided.

20.2 Transforming xml payloads

This section will explain the workings of `XmlPayloadUnmarshallingTransformer`, `XmlPayloadMarshallingTransformer`, `XsltPayloadTransformer` and how to configure them as *beans*. All of the provided xml transformers extend `AbstractPayloadTransformer` and therefore implement `Transformer`. When configuring xml transformers as beans in Spring Integration you would normally configure the transformer in conjunction with either a `MessageTransformingChannelInterceptor` or a `MessageTransformingConsumer`. This allows the transformer to be used as either an interceptor, which transforms the message as it is sent or received to the channel, or as an endpoint. Finally the namespace support will be discussed which allows for the simple configuration of the transformers as `MessageEndpoint` instances.

`XmlPayloadUnmarshallingTransformer` allows an xml `Source` to be unmarshalled using implementations of Spring OXM `Unmarshaller`. Spring OXM provides several implementations supporting marshalling and unmarshalling using JAXB, Castor and JiBX amongst others. Since the unmarshaller requires an instance of `Source` where the message payload is not currently an instance of `Source`, conversion will be attempted. Currently `String` and `org.w3c.dom.Document` payloads are supported. Custom conversion to a `Source` is also supported by injecting an implementation of `SourceFactory`.

```
<bean id="unmarshallingTransformer"
      class="org.springframework.integration.xml.transformer.XmlPayloadUnmarshallingTransformer">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb1Marshaller">
                    <property name="contextPath" value="org.example" />
        </bean>
    </constructor-arg>
</bean>
```

The `XmlPayloadMarshallingTransformer` allows an object graph to be converted into xml using a Spring OXM `Marshaller`. By default the `XmlPayloadMarshallingTransformer` will return a `DomResult`. However the type of result can be controlled by configuring an alternative `ResultFactory` such as `StringResultFactory`. In many cases it will be more convenient to transform the payload into an alternative xml format. To achieve this configure a `ResultTransformer`. Two implementations are provided, one which converts to `String` and another which converts to `Document`.

```
<bean id="marshallingTransformer"
      class="org.springframework.integration.xml.transformer.XmlPayloadMarshallingTransformer">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb1Marshaller">
            <property name="contextPath" value="org.example" />
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="org.springframework.integration.xml.transformer.ResultToDocumentTransformer" />
    </constructor-arg>
</bean>
```

`XsltPayloadTransformer` transforms xml payloads using xsl. The transformer requires an instance of either `Resource` or `Templates`. Passing in a `Templates` instance allows for greater configuration of the `TransformerFactory` used to create the template instance. As in the case of `XmlPayloadMarshallingTransformer` by default `XsltPayloadTransformer` will create a message with a `Result` payload. This can be customised by providing a `ResultFactory` and/or a `ResultTransformer`.

```
<bean id="xsltPayloadTransformer"
      class="org.springframework.integration.xml.transformer.XsltPayloadTransformer">
    <constructor-arg value="classpath:org/example/xsl/transform.xsl" />
    <constructor-arg>
        <bean class="org.springframework.integration.xml.transformer.ResultToDocumentTransformer" />
    </constructor-arg>
</bean>
```

20.3 Namespace support for xml transformers

Namespace support for all xml transformers is provided in the Spring Integration xml namespace, a template for which can be seen below. The namespace support for transformers creates an instance of either `SubscribingConsumerEndpoint` or `PollingConsumerEndpoint` according to the type of the provided input channel. The namespace support is designed to reduce the amount of xml configuration by allowing the creation of an endpoint and transformer using one element.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:integration="http://www.springframework.org/schema/integration"
       xmlns:si-xml="http://www.springframework.org/schema/integration/xml"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
           http://www.springframework.org/schema/integration/xml
```

```
            http://www.springframework.org/schema/integration/xml/spring-integration-xml-1.0.xsd">
</beans>
```

The namespace support for `XmlPayloadUnmarshallingTransformer` is shown below. Since the namespace is now creating an instance of `MessageEndpoint` rather than a transformer a poller can also be nested within the element to control the polling of the input channel.

```
<si-xml:unmarshalling-transformer id="defaultUnmarshaller"
    input-channel="input"
    output-channel="output"
    unmarshaller="unmarshaller"/>

<si-xml:unmarshalling-transformer id="unmarshallerWithPoller"
    input-channel="input"
    output-channel="output"
    unmarshaller="unmarshaller">
    <si:poller>
        <si:interval-trigger interval="2000"/>
    </si:poller>
<si-xml:unmarshalling-transformer/>
```

The namespace support for the marshalling transformer requires an input channel, output channel and a reference to a marshaller. The optional result-type attribute can be used to control the type of result created, valid values are StringResult or DomResult (the default). Where the provided result types are not sufficient a reference to a custom implementation of `ResultFactory` can be provided as an alternative to setting the result-type attribute using the result-factory attrbitue. An optional result-transformer can also be specified in order to convert the created `Result` after marshalling.

```
<si-xml:marshalling-transformer
     input-channel="marshallingTransformerStringResultFactory"
     output-channel="output"
     marshaller="marshaller"
     result-type="StringResult" />

<si-xml:marshalling-transformer
    input-channel="marshallingTransformerWithResultTransformer"
    output-channel="output"
    marshaller="marshaller"
    result-transformer="resultTransformer" />

<bean id="resultTransformer"
      class="org.springframework.integration.xml.transformer.ResultToStringTransformer"/>
```

Namespace support for the `XsltPayloadTransformer` allows either a resource to be passed in in order to create the `Templates` instance or alternatively a precreated `Templates` instance can be passed in as a reference. In common with the marshalling transformer the type of the result output can be controlled by specifying either the result-factory or result-type attribute. A result-transfomer attribute can also be used to reference an implementation of `ResultTransfomer` where conversion of the result is required before sending.

```
<si-xml:xslt-transformer id="xsltTransformerWithResource"
    input-channel="withResourceIn"
    output-channel="output"
    xsl-resource="org/springframework/integration/xml/config/test.xsl"/>
<si-xml:xslt-transformer id="xsltTransformerWithTemplatesAndResultTransformer"
    input-channel="withTemplatesAndResultTransformerIn"
    output-channel="output"
    xsl-templates="templates"
```

```
    result-transformer="resultTransformer"/>
```

20.4 Splitting xml messages

`XPathMessageSplitter` supports messages with either `String` or `Document` payloads. The splitter uses the provided XPath expression to split the payload into a number of nodes. By default this will result in each `Node` instance becoming the payload of a new message. Where it is preferred that each message be a Document the `createDocuments` flag can be set. Where a `String` payload is passed in the payload will be converted then split before being converted back to a number of String messages. The XPath splitter implements `MessageConsumer` and should therefore be configured in conjunction with an appropriate endpoint.

```
<bean id="splittingEndpoint"
      class="org.springframework.integration.endpoint.SubscribingConsumerEndpoint">
    <constructor-arg>
        <bean class="org.springframework.integration.xml.splitter.XPathMessageSplitter">
            <constructor-arg value="/order/items" />
            <property name="documentBuilder" ref="customisedDocumentBuilder" />
            <property name="outputChannel" ref="orderItemsChannel" />
        </bean>
    </constructor-arg>
        <constructor-arg ref="orderChannel" />
</bean>
```

20.5 Routing xml messages using XPath

Two Router implementations based on XPath are provided `XPathSingleChannelRouter` and `XPathMultiChannelRouter`. The implementations differ in respect to how many channels any given message may be routed to, exactly one in the case of the single channel version or zero or more in the case of the multichannel router. Both evaluate an XPath expression against the xml payload of the message, supported payload types by default are `Node`, `Document` and `String`. For other payload types a custom implementation of `XmlPayloadConverter` can be provided. The router implementations use `ChannelNameResolver` to convert the result(s) of the XPath expression to a channel name. By default a `BeanFactoryChannelName` strategy will be used, this means that the string returned by the XPath evaluation should correspond directly to the name of a channel. Where this is not the case an alternative implementation of `ChannelNameResolver` can be used. Where there is a simple mapping from Xpath result to channel name the provided `MapBasedChannelName` can be used.

```
<!-- Expects a channel for each value of order type to exist  -->
<bean id="singleChannelRoutingEndpoint"
      class="org.springframework.integration.endpoint.SubscribingConsumerEndpoint">
    <constructor-arg>
        <bean class="org.springframework.integration.xml.router.XPathSingleChannelRouter">
            <constructor-arg value="/order/@type" />
        </bean>
    </constructor-arg>
        <constructor-arg ref="orderChannel" />
</bean>
```

```
<!-- Multi channel router which uses a map channel resolver to resolve the channel name
     based on the XPath evaluation result Since the router is multi channel it may deliver
     message to one or both of the configured channels -->
<bean id="multiChannelRoutingEndpoint"
      class="org.springframework.integration.endpoint.SubscribingConsumerEndpoint">
    <constructor-arg>
        <bean class="org.springframework.integration.xml.router.XPathMultiChannelRouter">
            <constructor-arg value="/order/recipient" />
            <property name="channelResolver">
                <bean class="org.springframework.integration.channel.MapBasedChannelResolver">
                    <constructor-arg>
                        <map>
                            <entry key="accounts"
                                   value-ref="accountConfirmationChannel" />
                            <entry key="humanResources"
                                   value-ref="humanResourcesConfirmationChannel" />
                        </map>
                     </constructor-arg>
                     </bean>
            </property>
        </bean>
    </constructor-arg>
    <constructor-arg ref="orderChannel" />
</bean>
```

20.6 Selecting xml messages using XPath

Two `MessageSelector` implementations are provided, `BooleanTestXPathMessageSelector` and `StringValueTestXPathMessageSelector`. `BooleanTestXPathMessageSelector` requires an XPathExpression which evaluates to a boolean, for example *boolean(/one/two)* which will only select messages which have an element named two which is a child of a root element named one. `StringValueTestXPathMessageSelector` evaluates any XPath expression as a `String` and compares the result with the provided value.

```
<!-- Interceptor which rejects messages that do not have a root element order  -->
<bean id="orderSelectingInterceptor"
      class="org.springframework.integration.channel.interceptor.MessageSelectingInterceptor">
    <constructor-arg>
        <bean class="org.springframework.integration.xml.selector.BooleanTestXPathMessageSelector">
            <constructor-arg value="boolean(/order)" />
        </bean>
    </constructor-arg>
</bean>

<!--  Interceptor which rejects messages that are not version one orders -->
<bean id="versionOneOrderSelectingInterceptor"
      class="org.springframework.integration.channel.interceptor.MessageSelectingInterceptor">
    <constructor-arg>
        <bean class="org.springframework.integration.xml.selector.StringValueTestXPathMessageSelector">
            <constructor-arg value="/order/@version" index="0"/>
            <constructor-arg value="1" index="1"/>
        </bean>
    </constructor-arg>
</bean>
```

20.7 XPath components namespace support

All XPath based components have namespace support allowing them to be configured as instances of `MessageEndpoint` with the exception of the XPath selectors which are not designed to act as endpoints. Each component allows the XPath to either be referenced at the top level or configured via a nested xpath-expression element. So the following configurations of an xpath-selector are all valid and represent the general form of XPath namespace support. All forms of XPath expression result in the creation of an `XPathExpression` using the Spring `XPathExpressionFactory`

```
<si-xml:xpath-selector id="xpathRefSelector"
                       xpath-expression="refToXpathExpression"
                       evaluation-result-type="boolean" />

<si-xml:xpath-selector id="selectorWithNoNS" evaluation-result-type="boolean" >
    <si-xml:xpath-expression expression="/name"/>
</si-xml:xpath-selector>

<si-xml:xpath-selector id="selectorWithOneNS" evaluation-result-type="boolean" >
    <si-xml:xpath-expression expression="/ns1:name"
                             ns-prefix="ns1" ns-uri="www.example.org" />
</si-xml:xpath-selector>

<si-xml:xpath-selector id="selectorWithTwoNS" evaluation-result-type="boolean" >
    <si-xml:xpath-expression expression="/ns1:name/ns2:type">
        <map>
            <entry key="ns1" value="www.example.org/one" />
            <entry key="ns2" value="www.example.org/two" />
        </map>
    </si-xml:xpath-expression>
</si-xml:xpath-selector>

<si-xml:xpath-selector id="selectorWithNamespaceMapRef" evaluation-result-type="boolean" >
    <si-xml:xpath-expression expression="/ns1:name/ns2:type"
                             namespace-map="defaultNamespaces"/>
</si-xml:xpath-selector>

<util:map id="defaultNamespaces">
    <util:entry key="ns1" value="www.example.org/one" />
    <util:entry key="ns2" value="www.example.org/two" />
</util:map>
```

XPath splitter namespace support allows the creation of a MessageEndpoint with an input channel and output channel.

```
<!-- Split the order into items creating a new message for each item node -->
<si-xml:xpath-splitter id="orderItemSplitter"
                       input-channel="orderChannel"
                       output-channel="orderItemsChannel">
    <si-xml:xpath-expression expression="/order/items"/>
</si-xml:xpath-selector>

<!-- Split the order into items creating a new document for each item-->
<si-xml:xpath-splitter id="orderItemDocumentSplitter"
                       input-channel="orderChannel"
                       output-channel="orderItemsChannel"
                       create-documents="true">
    <si-xml:xpath-expression expression="/order/items"/>
    <si:poller>
        <si:interval-trigger interval="2000"/>
    </si:poller>
</si-xml:xpath-selector>
```

XPath router namespace support allows for the creation of a MessageEndpoint with an input channel but no output channel since the output channel is determined dynamically. The multi-channel attribute causes the creation of a multi channel router capable of routing a single message to many channels when true and a single channel router when false.

```
<!-- route the message according to exactly one order type channel -->
<si-xml:xpath-router id="orderTypeRouter" input-channel="orderChannel" multi-channel="false">
    <si-xml:xpath-expression expression="/order/type"/>
</si-xml:xpath-selector>

<!-- route the order to all responders-->
<si-xml:xpath-router id="responderRouter" input-channel="orderChannel" multi-channel="true">
    <si-xml:xpath-expression expression="/request/responders"/>
    <si:poller>
        <si:interval-trigger interval="2000"/>
    </si:poller>
</si-xml:xpath-selector>
```

21. Security in Spring Integration

21.1 Introduction

Spring Integration provides integration with the Spring Security project to allow role based security checks to be applied to channel send and receive invocations.

21.2 Securing channels

Spring Integration provides the interceptor `ChannelSecurityInterceptor`, which extends `AbstractSecurityInterceptor` and intercepts send and receive calls on the channel. Access decisions are then made with reference to `ChannelInvocationDefinitionSource` which provides the definition of the send and receive security constraints. The interceptor requires that a valid `SecurityContext` has been established by authenticating with Spring Security, see the Spring Security reference documentation for details.

Namespace support is provided to allow easy configuration of security constraints. This consists of the secured channels tag which allows definition of one or more channel name patterns in conjunction with a definition of the security configuration for send and receive. The pattern is a `java.util.regexp.Pattern`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
         xmlns:si-security="http://www.springframework.org/schema/integration/security"
        xmlns:beans="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:security="http://www.springframework.org/schema/security"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                        http://www.springframework.org/schema/security
                        http://www.springframework.org/schema/security/spring-security-2.0.xsd
                        http://www.springframework.org/schema/integration
                        http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
                        http://www.springframework.org/schema/integration/security
                        http://www.springframework.org/schema/integration/security/spring-integration-security-

<si-security:secured-channels>
    <si-security:access-policy pattern="admin.*" send-access="ROLE_ADMIN"/>
    <si-security:access-policy pattern="user.*" receive-access="ROLE_USER"/>
</si-security:secured-channels>
```

By default the secured-channels namespace element expects a bean named *authenticationManager* which implements `AuthenticationManager` and a bean named *accessDecisionManager* which implements `AccessDecisionManager`. Where this is not the case references to the appropriate beans can be configured as attributes of the *secured-channels* element as below.

```
<si-security:secured-channels access-decision-manager="customAccessDecisionManager"
                              authentication-manager="customAuthenticationManager">
    <si-security:access-policy pattern="admin.*" send-access="ROLE_ADMIN"/>
    <si-security:access-policy pattern="user.*" receive-access="ROLE_USER"/>
```

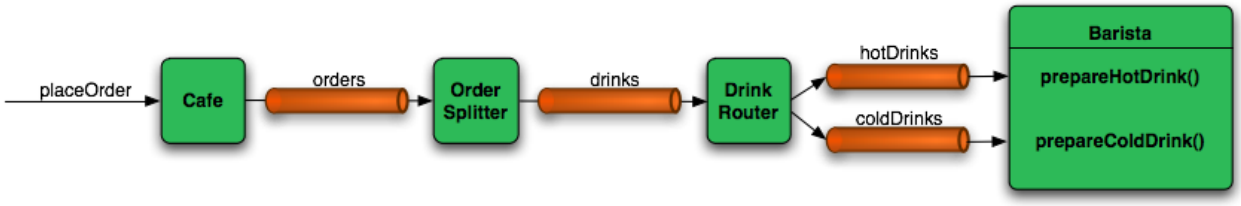
```
</si-security:secured-channels>
```

Appendix A. Spring Integration Samples

A.1 The Cafe Sample

In this section, we will review a sample application that is included in the Spring Integration distribution. This sample is inspired by one of the samples featured in Gregor Hohpe's Ramblings.

The domain is that of a Cafe, and the basic flow is depicted in the following diagram:



The `Order` object may contain multiple `OrderItems`. Once the order is placed, a *Splitter* will break the composite order message into a single message per drink. Each of these is then processed by a *Router* that determines whether the drink is hot or cold (checking the `OrderItem` object's 'isIced' property). Finally the `Barista` prepares each drink, but hot and cold drink preparation are handled by two distinct methods: 'prepareHotDrink' and 'prepareColdDrink'. The prepared drinks are then sent to the Waiter where they are aggregated into a `Delivery` object.

Here is the XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:stream="http://www.springframework.org/schema/integration/stream"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
            http://www.springframework.org/schema/integration/stream
            http://www.springframework.org/schema/integration/stream/spring-integration-stream-1.0.xsd">

    <message-bus/>

    <gateway id="cafe" service-interface="org.springframework.integration.samples.cafe.Cafe"/>

    <channel id="orders"/>
    <splitter input-channel="orders" ref="orderSplitter" method="split" output-channel="drinks"/>

    <channel id="drinks"/>
    <router input-channel="drinks" ref="drinkRouter" method="resolveOrderItemChannel"/>

    <channel id="coldDrinks">
        <queue capacity="10"/>
```

```
    </channel>
    <service-activator input-channel="coldDrinks" ref="barista"
                       method="prepareColdDrink" output-channel="preparedDrinks"/>

    <channel id="hotDrinks">
        <queue capacity="10"/>
    </channel>
    <service-activator input-channel="hotDrinks" ref="barista"
                       method="prepareHotDrink" output-channel="preparedDrinks"/>

    <channel id="preparedDrinks"/>
    <aggregator input-channel="preparedDrinks" ref="waiter"
                method="prepareDelivery" output-channel="deliveries"/>

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

    <beans:bean id="orderSplitter"
                class="org.springframework.integration.samples.cafe.xml.OrderSplitter"/>

    <beans:bean id="drinkRouter"
                class="org.springframework.integration.samples.cafe.xml.DrinkRouter"/>

    <beans:bean id="barista" class="org.springframework.integration.samples.cafe.xml.Barista"/>

    <beans:bean id="waiter" class="org.springframework.integration.samples.cafe.xml.Waiter"/>

</beans:beans>
```

Notice that the Message Bus is defined. It will automatically detect and register all channels and endpoints, and it will manage the Lifecycle for each endpoint. Notice that the objects are simple POJOs with strongly typed method arguments. For example, here is the Splitter.

```
public class OrderSplitter {

    public List<OrderItem> split(Order order) {
        return order.getItems();
    }
}
```

In the case of the Router, the return value does not have to be a `MessageChannel` instance (although it can be). As you see in this example, a String-value representing the channel name is returned instead.

```
public class DrinkRouter {

    public String resolveOrderItemChannel(OrderItem orderItem) {
        return (orderItem.isIced()) ? "coldDrinks" : "hotDrinks";
    }
}
```

Now turning back to the XML, you see that there are two <service-activator> elements. Each of these is delegating to the same `Barista` instance but different methods: 'prepareHotDrink' or 'prepareColdDrink' corresponding to the two channels where order items have been routed.

```
public class Barista {

    private long hotDrinkDelay = 5000;
    private long coldDrinkDelay = 1000;

    private AtomicInteger hotDrinkCounter = new AtomicInteger();
    private AtomicInteger coldDrinkCounter = new AtomicInteger();

    public void setHotDrinkDelay(long hotDrinkDelay) {
        this.hotDrinkDelay = hotDrinkDelay;
```

```
    }

    public void setColdDrinkDelay(long coldDrinkDelay) {
        this.coldDrinkDelay = coldDrinkDelay;
    }

    public Drink prepareHotDrink(OrderItem orderItem) {
        try {
            Thread.sleep(this.hotDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                    + " prepared hot drink #" + hotDrinkCounter.incrementAndGet()
                    + " for order #" + orderItem.getOrder().getNumber() + ": " + orderItem);
            return new Drink(orderItem.getOrder().getNumber(), orderItem.getDrinkType(),
                    orderItem.isIced(), orderItem.getShots());
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
    }

    public Drink prepareColdDrink(OrderItem orderItem) {
        try {
            Thread.sleep(this.coldDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                    + " prepared cold drink #" + coldDrinkCounter.incrementAndGet()
                    + " for order #" + orderItem.getOrder().getNumber() + ": " + orderItem);
            return new Drink(orderItem.getOrder().getNumber(), orderItem.getDrinkType(),
                    orderItem.isIced(), orderItem.getShots());
        }
        catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            return null;
        }
    }

}
```

As you can see from the code excerpt above, the barista methods have different delays (the hot drinks take 5 times as long to prepare). This simulates work being completed at different rates. When the `CafeDemo` 'main' method runs, it will loop 100 times sending a single hot drink and a single cold drink each time. It actually sends the messages by invoking the 'placeOrder' method on the Cafe interface. Above, you will see that the <gateway> element is specified in the configuration file. This triggers the creation of a proxy that implements the given 'service-interface' and connects it to a channel. The channel name is provided on the @Gateway annotation of the `Cafe` interface.

```
public interface Cafe {

    @Gateway(requestChannel="orders")
    void placeOrder(Order order);

}
```

Finally, have a look at the `main()` method of the `CafeDemo` itself.

```
public static void main(String[] args) {
    AbstractApplicationContext context = null;
    if (args.length > 0) {
        context = new FileSystemXmlApplicationContext(args);
    }
    else {
        context = new ClassPathXmlApplicationContext("cafeDemo.xml", CafeDemo.class);
    }
```

```
    Cafe cafe = (Cafe) context.getBean("cafe");
    for (int i = 1; i <= 100; i++) {
        Order order = new Order(i);
        order.addItem(DrinkType.LATTE, 2, false);
        order.addItem(DrinkType.MOCHA, 3, true);
        cafe.placeOrder(order);
    }
}
```

To run this demo, go to the "samples" directory within the root of the Spring Integration distribution. On Unix/Mac you can run 'cafeDemo.sh', and on Windows you can run 'cafeDemo.bat'. Each of these will by default create a Spring `ApplicationContext` from the 'cafeDemo.xml' file that is in the "spring-integration-samples" JAR and hence on the classpath (it is the same as the XML above). However, a copy of that file is also available within the "samples" directory, so that you can provide the file name as a command line argument to either 'cafeDemo.sh' or 'cafeDemo.bat'. This will allow you to experiment with the configuration and immediately run the demo with your changes. It is probably a good idea to first copy the original file so that you can make as many changes as you want and still refer back to the original to compare.

When you run cafeDemo, you will see that the cold drinks are initially prepared more quickly than the hot drinks. Because there is an aggregator, the cold drinks are effectively limited by the rate of the hot drink preparation. This is to be expected based on their respective delays of 1000 and 5000 milliseconds. However, by configuring a poller with a concurrent task executor, you can dramatically change the results. For example, you could use a thread pool executor with 5 workers for the hot drink barista while keeping the cold drink barista as it is:

```
<service-activator input-channel="hotDrinks"
                   ref="barista"
                   method="prepareHotDrink"
                   output-channel="preparedDrinks"/>

<service-activator input-channel="hotDrinks"
                   ref="barista"
                   method="prepareHotDrink"
                   output-channel="preparedDrinks">
    <poller task-executor="pool">
        <interval-trigger interval="1000"/>
    </poller>
</service-activator>

<thread-pool-task-executor id="pool" core-size="5"/>
```

Also, notice that the worker thread name is displayed with each invocation. You will see that the hot drinks are prepared by the task-executor threads. If you provide a much shorter poller interval (such as 100 milliseconds), then you will notice that occasionally it throttles the input by forcing the message-bus (the caller) to invoke the operation.

In addition to experimenting with the poller's concurrency settings, you can also add the 'transactional' sub-element. If you want to explore the sample in more detail, the source JAR is available in the "src" directory: 'org.springframework.integration.samples-sources-1.0.0.RC1.jar'.

Appendix B. Configuration

B.1 Introduction

Spring Integration offers a number of configuration options. Which option you choose depends upon your particular needs and at what level you prefer to work. As with the Spring framework in general, it is also possible to mix and match the various techniques according to the particular problem at hand. For example, you may choose the XSD-based namespace for the majority of configuration combined with a handful of objects that are configured with annotations. As much as possible, the two provide consistent naming. XML elements defined by the XSD schema will match the names of annotations, and the attributes of those XML elements will match the names of annotation properties. Direct usage of the API is of course always an option, but we expect that most users will choose one of the higher-level options, or a combination of the namespace-based and annotation-driven configuration.

B.2 Namespace Support

Spring Integration components can be configured with XML elements that map directly to the terminology and concepts of enterprise integration. In many cases, the element names match those of the Enterprise Integration Patterns.

To enable Spring Integration's core namespace support within your Spring configuration files, add the following namespace reference and schema mapping in your top-level 'beans' element:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:integration="http://www.springframework.org/schema/integration"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-1.0.xsd">
```

You can choose any name after "xmlns:"; *integration* is used here for clarity, but you might prefer a shorter abbreviation. Of course if you are using an XML-editor or IDE support, then the availability of auto-completion may convince you to keep the longer name for clarity. Alternatively, you can create configuration files that use the Spring Integration schema as the primary namespace:

```
<beans:beans xmlns="http://www.springframework.org/schema/integration"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:beans="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-1.0.xsd">
```

When using this alternative, no prefix is necessary for the Spring Integration elements. On the other hand, if you want to define a generic Spring "bean" within the same configuration file, then a prefix would be

required for the bean element (<beans:bean ... />). Since it is generally a good idea to modularize the configuration files themselves based on responsibility and/or architectural layer, you may find it appropriate to use the latter approach in the integration-focused configuration files, since generic beans are seldom necessary within those same files. For purposes of this documentation, we will assume the "integration" namespace is primary.

Many other namespaces are provided within the Spring Integration distribution. In fact, each adapter type (JMS, File, etc.) that provides namespace support defines its elements within a separate schema. In order to use these elements, simply add the necessary namespaces with an "xmlns" entry and the corresponding "schemaLocation" mapping. For example, the following root element shows several of these namespace declarations:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:integration="http://www.springframework.org/schema/integration"
    xmlns:file="http://www.springframework.org/schema/integration/file"
    xmlns:jms="http://www.springframework.org/schema/integration/jms"
    xmlns:mail="http://www.springframework.org/schema/integration/mail"
    xmlns:rmi="http://www.springframework.org/schema/integration/rmi"
    xmlns:ws="http://www.springframework.org/schema/integration/ws"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
            http://www.springframework.org/schema/integration/file
            http://www.springframework.org/schema/integration/file/spring-integration-file-1.0.xsd
            http://www.springframework.org/schema/integration/jms
            http://www.springframework.org/schema/integration/jms/spring-integration-jms-1.0.xsd
            http://www.springframework.org/schema/integration/mail
            http://www.springframework.org/schema/integration/mail/spring-integration-mail-1.0.xsd
            http://www.springframework.org/schema/integration/rmi
            http://www.springframework.org/schema/integration/rmi/spring-integration-rmi-1.0.xsd
            http://www.springframework.org/schema/integration/ws
            http://www.springframework.org/schema/integration/ws/spring-integration-ws-1.0.xsd">
 ...
</beans>
```

The reference manual provides specific examples of the various elements in their corresponding chapters. Here, the main thing to recognize is the consistency of the naming for each namespace URI and schema location.

B.3 Configuring the Message Bus

The Message Bus plays a central role, but its configuration is quite simple since it is primarily concerned with managing internal details based on the configuration of channels and endpoints. The bus is aware of its host application context, and therefore is also capable of auto-detecting the channels and endpoints. The Message Bus can be configured with a single empty element:

```
<message-bus/>
```

The Message Bus provides default error handling for its components in the form of a configurable error channel, and it will first check for a channel bean named 'errorChannel' within the context:

```
<message-bus/>

<channel id="errorChannel" capacity="500"/>
```

When exceptions occur in a scheduled poller task's execution, those exceptions will be wrapped in `ErrorMessages` and sent to the 'errorChannel' by default. To enable global error handling, simply register a handler on that channel. For example, you can configure Spring Integration's `ErrorMessageExceptionTypeRouter` as the handler of an endpoint that is subscribed to the 'errorChannel'. That router can then spread the error messages across multiple channels based on `Exception` type. However, since most of the errors will already have been wrapped in `MessageDeliveryException` or `MessageHandlingException`, the `ErrorMessageExceptionTypeRouter` is typically a better option.

The 'message-bus' element accepts several more optional attributes. First, you can control whether the `MessageBus` will be started automatically (the default) or will require explicit startup by invoking its `start()` method (`MessageBus` implements Spring's `Lifecycle` interface):

```
<message-bus auto-startup="false"/>
```

Another configurable property is the reference to a `TaskScheduler` implementation. If not provided, a default will be created. The scheduler is responsible for managing the pollers.

```
<message-bus task-scheduler="someScheduler"/>
```

When the endpoints are concurrency-enabled with their own 'taskExecutor' reference, the invocation of the handling methods will happen within that executor's thread pool and not the main scheduler pool. However, when no task-executor is provided for an endpoint's poller, then it will be invoked in the dispatcher's thread (with the exception of subscribable channels where the subscribers may be invoked directly).

B.4 Annotation Support

In addition to the XML namespace support for configuring Message Endpoints, it is also possible to use annotations. First, Spring Integration provides the class-level `@MessageEndpoint` as a *stereotype* annotation meaning that is itself annotated with Spring's @Component annotation and therefore is recognized automatically as a bean definition when using Spring component-scanning.

Even more importantly are the various Method-level annotations that indicate the annotated method is capable of handling a message. The following example demonstrates both:

```
@MessageEndpoint
public class FooService {

    @ServiceActivator
    public void processMessage(Message message) {
        ...
    }
}
```

Exactly what it means for the method to "handle" the Message depends on the particular annotation. The following are available with Spring Integration, and the behavior of each is described in its own chapter or section within this reference: @Transformer, @Router, @Splitter, @Aggregator, @ServiceActivator, and @ChannelAdapter.

Note

The @MessageEndpoint is not required. If you want to configure a POJO reference from the "ref" attribute of a <service-activator/> element, it is sufficient to provide the method-level annotations.

In most cases, the annotated handler method should not require the `Message` type as its parameter. Instead, the method parameter type can match the message's payload type.

```
public class FooService {

    @Handler
    public void bar(Foo foo) {
        ...
    }

}
```

When the method parameter should be mapped from a value in the `MessageHeader`, another option is to use the parameter-level `@Header` annotation. In general, methods annotated with the Spring Integration annotations can either accept the `Message` itself, the message payload, or a header value (with @Header) as the parameter. In fact, the method can accept a combination, such as:

```
public class FooService {

    @ServiceActivator
    public void bar(String payload, @Header("x") int valueX, @Header("y") int valueY) {
        ...
    }

}
```

There is also a @Headers annotation that provides all of the Message headers as a Map:

```
public class FooService {

    @ServiceActivator
    public void bar(String payload, @Headers Map<String, Object> headerMap) {
        ...
    }

}
```

For several of these annotations, when a Message-handling method returns a non-null value, the endpoint will attempt to send a reply. This is consistent across both configuration options (namespace and annotations) in that the such an endpoint's output channel will be used if available, and the message header's REPLY_CHANNEL value will be the fallback.

Appendix C. Additional Resources

C.1 Spring Integration Home

The definitive source of information about Spring Integration is the Spring Integration Home at http://www.springframework.org. That site serves as a hub of information and is the best place to find up-to-date announcements about the project as well as links to articles, blogs, and new sample applications.