Spring Integration Reference Manual

2.0.0.M7

Copyright © 2005-2010 Mark Fisher, Marius Bogoevici, Iwein Fuld, Jonas Partner, Oleg Zhurakousky, Gary Russell, Josh Long

© SpringSource Inc., 2010

1. Spring Integration Overview 1
1.1. Background 1
1.2. Goals and Principles 1
1.3. Main Components 2
1.3.1. Message 2
1.3.2. Message Channel 2
1.3.3. Message Endpoint 3
1.4. Message Endpoints 3
1.4.1. Transformer 3
1.4.2. Filter 3
1.4.3. Router 4
1.4.4. Splitter 4
1.4.5. Aggregator 4
1.4.6. Service Activator 4
1.4.7. Channel Adapter 5
2. Message Construction 6
2.1. The Message Interface 6
2.2. Message Headers 6
2.3. Message Implementations 7
2.4. The MessageBuilder Helper Class 7
3. Message Channels 10
3.1. The MessageChannel Interface 10
3.1.1. PollableChannel 10
3.1.2. SubscribableChannel 10
3.2. Message Channel Implementations 11
3.2.1. PublishSubscribeChannel 11
3.2.2. QueueChannel 11
3.2.3. PriorityChannel 11
3.2.4. RendezvousChannel 11
3.2.5. DirectChannel 12
3.2.6. ExecutorChannel 13
3.2.7. ThreadLocalChannel 13
3.3. Channel Interceptors 14
3.4. MessageChannelTemplate 15
3.5. Configuring Message Channels 15
3.5.1. DirectChannel Configuration 16
3.5.2. QueueChannel Configuration 16
3.5.3. PublishSubscribeChannel Configuration 16
3.5.4. ExecutorChannel 17
3.5.5. PriorityChannel Configuration 17
3.5.6. RendezvousChannel Configuration 18
3.5.7. ThreadLocalChannel Configuration 18
3.5.8. Channel Interceptor Configuration 18
3.5.9. Global Channel Interceptor Configuration 18
3.5.10. Wire Tap 19
4. Message Endpoints 20
4.1. Message Handler 20
4.2. Event Driven Consumer 20
4.3. Polling Consumer 21
4.4. Namespace Support 22
4.5. Payload Type Conversion 24
5. Service Activator 25

5.1. Introduction ... 25
5.2. The <service-activator/> Element ... 25
6. Channel Adapter ... 27
6.1. The <inbound-channel-adapter> element ... 27
6.2. The <outbound-channel-adapter/> element ... 27
7. Router ... 29
7.1. Router Implementations ... 29
7.1.1. PayloadTypeRouter ... 29
7.1.2. HeaderValueRouter ... 29
7.1.3. RecipientListRouter ... 30
7.2. The <router> element ... 31
7.3. The @Router Annotation ... 31
8. Filter ... 33
8.1. Introduction ... 33
8.2. The <filter> Element ... 33
9. Transformer ... 35
9.1. Introduction ... 35
9.2. The <transformer> Element ... 35
9.3. The @Transformer Annotation ... 37
10. Splitter ... 39
10.1. Introduction ... 39
10.2. Programming model ... 39
10.3. Configuring a Splitter using XML ... 40
10.4. Configuring a Splitter with Annotations ... 40
11. Aggregator ... 41
11.1. Introduction ... 41
11.2. Functionality ... 41
11.3. Programming model ... 41
11.3.1. CorrelatingMessageHandler ... 41
11.3.2. ReleaseStrategy ... 43
11.3.3. CorrelationStrategy ... 43
11.4. Configuring an Aggregator with XML ... 44
11.5. Managing State in an Aggregator: MessageGroupStore ... 46
11.6. Configuring an Aggregator with Annotations ... 47
12. Resequencer ... 49
12.1. Introduction ... 49
12.2. Functionality ... 49
12.3. Configuring a Resequencer with XML ... 49
13. Delayer ... 51
13.1. Introduction ... 51
13.2. The <delayer> Element ... 51
14. Message Handler Chain ... 52
14.1. Introduction ... 52
14.2. The <chain> Element ... 52
15. Messaging Bridge ... 54
15.1. Introduction ... 54
15.2. The <bridge> Element ... 54
16. Inbound Messaging Gateways ... 55
16.1. GatewayProxyFactoryBean ... 55
16.2. Asynchronous Gateway ... 56
17. Message Publishing ... 58
17.1. Message Publishing Configuration ... 58

17.1.1. Annotation-driven approach via @Publisher annotation ... 58
17.1.2. XML-based approach via <publishing-interceptor> element ... 60
17.1.3. Producing and publishing messages based on schedule ... 61
18. Message History ... 63
18.1. Message History Configuration ... 63
19. File Support ... 65
19.1. Introduction ... 65
19.2. Reading Files ... 65
19.3. Writing files ... 66
19.4. File Transformers ... 67
20. JDBC Support ... 69
20.1. Inbound Channel Adapter ... 69
20.1.1. Polling and Transactions ... 69
20.2. Outbound Channel Adapter ... 70
20.3. Outbound Gateway ... 70
20.4. Message Store ... 71
20.4.1. Initializing the Database ... 71
20.4.2. Partitioning a Message Store ... 72
21. JMS Support ... 73
21.1. Inbound Channel Adapter ... 73
21.2. Message-Driven Channel Adapter ... 74
21.3. Outbound Channel Adapter ... 74
21.4. Inbound Gateway ... 75
21.5. Outbound Gateway ... 76
21.6. Message Conversion, Marshalling and Unmarshalling ... 76
21.7. JMS Backed Message Channels ... 76
21.8. JMS Samples ... 78
22. Web Services Support ... 79
22.1. Outbound Web Service Gateways ... 79
22.2. Inbound Web Service Gateways ... 79
22.3. Web Service Namespace Support ... 79
23. RMI Support ... 82
23.1. Introduction ... 82
23.2. Outbound RMI ... 82
23.3. Inbound RMI ... 82
23.4. RMI namespace support ... 82
24. HttpInvoker Support ... 84
24.1. Introduction ... 84
24.2. HttpInvoker Inbound Gateway ... 84
24.3. HttpInvoker Outbound Gateway ... 84
24.4. HttpInvoker Namespace Support ... 85
25. HTTP Support ... 86
25.1. Introduction ... 86
25.2. Http Inbound Gateway ... 86
25.3. Http Outbound Gateway ... 87
25.4. HTTP Namespace Support ... 87
25.5. HTTP Samples ... 88
25.5.1. Multipart HTTP request - RestTemplate (client) and Http Inbound Gateway (server) 88
26. TCP and UDP Support ... 90
26.1. Introduction ... 90
26.2. UDP Adapters ... 90
26.3. TCP Connection Factories ... 92

26.4. TCP Adapters ... 94
26.5. TCP Gateways ... 95
26.6. IP Configuration Attributes ... 95
27. Mail Support ... 101
27.1. Mail-Sending Channel Adapter ... 101
27.2. Mail-Receiving Channel Adapter ... 101
27.3. Mail Namespace Support ... 102
28. JMX Support ... 104
28.1. Notification Listening Channel Adapter ... 104
28.2. Notification Publishing Channel Adapter ... 104
28.3. Attribute Polling Channel Adapter ... 105
28.4. Operation Invoking Channel Adapter ... 105
28.5. Operation Invoking outbound Gateway ... 105
28.6. MBean Exporter ... 106
28.7. Control Bus ... 106
29. XMPP Support ... 107
29.1. Introduction ... 107
29.2. Using The Spring Integration XMPP Namespace ... 107
29.3. XMPP Connection ... 108
29.4. XMPP Messages ... 108
29.4.1. Inbound Message Adapter ... 108
29.4.2. Outbound Message Adapter ... 109
29.5. XMPP Presence ... 110
29.5.1. Inbound Presence Adapter ... 111
29.5.2. Outbound Presence Adapter ... 111
30. Stream Support ... 112
30.1. Introduction ... 112
30.2. Reading from streams ... 112
30.3. Writing to streams ... 112
30.4. Stream namespace support ... 112
31. Spring ApplicationEvent Support ... 114
31.1. Receiving Spring ApplicationEvents ... 114
31.2. Sending Spring ApplicationEvents ... 114
32. Dealing with XML Payloads ... 115
32.1. Introduction ... 115
32.2. Transforming xml payloads ... 115
32.3. Namespace support for xml transformers ... 116
32.4. Splitting xml messages ... 118
32.5. Routing xml messages using XPath ... 118
32.6. Selecting xml messages using XPath ... 119
32.7. XPath components namespace support ... 119
33. Security in Spring Integration ... 121
33.1. Introduction ... 121
33.2. Securing channels ... 121
34. Groovy support ... 122
34.1. Groovy configuration ... 122
A. Spring Integration Samples ... 123
A.1. The Cafe Sample ... 123
A.2. The XML Messaging Sample ... 126
A.3. The OSGi Samples ... 127
B. Configuration ... 131
B.1. Introduction ... 131

B.2. Namespace Support 131
B.3. Configuring the Task Scheduler 132
B.4. Error Handling 133
B.5. Annotation Support 134
B.6. Message Mapping rules and conventions 136
B.6.1. Simple Scenarios 136
B.6.2. Complex Scenarios 138
C. Additional Resources 141
C.1. Spring Integration Home 141

Chapter 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.

1.3.1. 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.

1.3.2. 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”.

1.3.3. 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.

1.4.1. 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.

1.4.2. 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.

1.4.3. 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.

1.4.4. 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.

1.4.5. 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.

1.4.6. 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.

1.4.7. 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.

Chapter 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)
ERROR_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.withPayload("test3")
        .copyHeaders(message1.getHeaders())
        .build();

Message<String> message4 = MessageBuilder.withPayload("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.withPayload(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
}
```

Chapter 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*.

3.1.1. 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.

3.1.2. SubscribableChannel

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

```
public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);

}
```

3.2. Message Channel Implementations

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

3.2.1. PublishSubscribeChannel

The `PublishSubscribeChannel` implementation broadcasts any Message sent to it to all of its subscribed handlers. This is most often used for sending *Event Messages* whose primary role is notification as opposed to *Document Messages* which are generally intended to be processed by a single handler. Note that the `PublishSubscribeChannel` is intended for sending only. Since it broadcasts to its subscribers directly when its `send(Message)` method is invoked, consumers cannot poll for Messages (it does not implement `PollableChannel` and therefore has no `receive()` method). Instead, any subscriber must be a `MessageHandler` itself, and the subscriber's `handleMessage(Message)` method will be invoked in turn.

3.2.2. QueueChannel

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

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

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

3.2.3. 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.

3.2.4. RendezvousChannel

The `RendezvousChannel` enables a "direct-handoff" scenario where a sender will block until another party invokes the channel's `receive()` method or vice-versa. Internally, this implementation is quite similar to the `QueueChannel` except that it uses a `SynchronousQueue` (a zero-capacity implementation of `BlockingQueue`). This works well in situations where the sender and receiver are operating in different threads but simply dropping the message in a queue asynchronously is not appropriate. In other words, with a `RendezvousChannel` at least the sender knows that some receiver has accepted the message, whereas with a `QueueChannel`, the message would have been stored to the internal queue and potentially never received.

Tip

Keep in mind that all of these queue-based channels are storing messages in-memory only. When persistence is required, you can either invoke a database operation within a handler or use Spring Integration's support for JMS-based Channel Adapters. The latter option allows you to take advantage of any JMS provider's implementation for message persistence, and it will be discussed in Chapter 21, *JMS Support*. However, when buffering in a queue is not necessary, the simplest approach is to rely upon the `DirectChannel` discussed next.

The `RendezvousChannel` is also useful for implementing request-reply operations. The sender can create a temporary, anonymous instance of `RendezvousChannel` which it then sets as the 'replyChannel' header when building a Message. After sending that Message, the sender can immediately call receive (optionally providing a timeout value) in order to block while waiting for a reply Message. This is very similar to the implementation used internally by many of Spring Integration's request-reply components.

3.2.5. DirectChannel

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

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

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

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 need to provide buffering or to throttle input, and then modify those to be queue-based `PollableChannels`. Likewise, if a channel needs to broadcast messages, it should not be a `DirectChannel` but rather a `PublishSubscribeChannel`. Below you will see how each of these can be configured.

The `DirectChannel` internally delegates to a Message Dispatcher to invoke its subscribed Message Handlers, and that dispatcher can have a load-balancing strategy. The load-balancer determines how invocations will be ordered in the case that there are multiple handlers subscribed to the same channel. When using the namespace support described below, the default strategy is "round-robin" which essentially load-balances across the handlers in rotation.

> **Note**
> The "round-robin" strategy is currently the only implementation available out-of-the-box in Spring Integration. Other strategy implementations may be added in future versions.

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

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

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

3.2.6. ExecutorChannel

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

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

3.2.7. 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 a RuntimeException). Also, the `preReceive` method can return '`false`' to prevent the receive operation from proceeding.

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

Spring Integration also provides an implementation of the Wire Tap pattern. It is a simple interceptor that sends the Message to another channel without otherwise altering the existing flow. It can be very useful for debugging and monitoring. An example is shown in Section 3.5.10, “Wire Tap”.

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

```
public class CountingChannelInterceptor extends ChannelInterceptorAdapter {

    private final AtomicInteger sendCount = new AtomicInteger();

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

> **Tip**
> The order of invocation for the interceptor methods depends on the type of channel. As described above, the queue-based channels are the only ones where the receive method is intercepted in the

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

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) { ... }
```

Note

A less invasive approach that allows you to invoke simple interfaces with payload and/or header values instead of Message instances is described in Section 16.1, “GatewayProxyFactoryBean”.

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

3.5.1. DirectChannel Configuration

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

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

A default channel will have a *round-robin* load-balancer and will also have failover enabled (See the discussion in Section 3.2.5, “DirectChannel” for more detail). To disable one or both of these, add a <dispatcher/> sub-element and configure the attributes:

```
<channel id="failFastChannel">
    <dispatcher failover="false"/>
</channel>

<channel id="channelWithFixedOrderSequenceFailover">
    <dispatcher load-balancer="none"/>
</channel>
```

3.5.2. QueueChannel Configuration

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

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

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

3.5.3. 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="pubsubChannel" task-executor="someExecutor"/>
```

If you are providing a *Resequencer* or *Aggregator* downstream from a `PublishSubscribeChannel`, then you can set the 'apply-sequence' property on the channel to `true`. That will indicate that the channel should set the sequence-size and sequence-number Message headers as well as the correlation id prior to passing the Messages along. For example, if there are 5 subscribers, the sequence-size would be set to 5, and the Messages would have sequence-number header values ranging from 1 to 5.

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

Note

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

3.5.4. ExecutorChannel

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

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

Note

The "load-balancer" and "failover" options are also both available on the dispatcher sub-element as described above in Section 3.5.1, “DirectChannel Configuration”. The same defaults apply as well. So, the channel will have a round-robin load-balancing strategy with failover enabled unless explicit configuration is provided for one or both of those attributes.

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

3.5.5. PriorityChannel Configuration

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

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

3.5.6. RendezvousChannel Configuration

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

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

3.5.7. ThreadLocalChannel Configuration

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

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

3.5.8. Channel Interceptor Configuration

Message channels may also have interceptors as described in Section 3.3, “Channel Interceptors”. The <interceptors> sub-element can be added within <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">
    <interceptors>
        <ref bean="trafficMonitoringInterceptor"/>
    </interceptors>
</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.

3.5.9. Global Channel Interceptor Configuration

Channel Interceptors allow you for a clean and concise way of applying cross-cutting behavior per individual channel. But what if the same behavior should be applied on multiple channels, configuring the same set of interceptors for each channel *would not be* the most efficient way. The better way would be to configure interceptors globally and apply them on multiple channels in one shot. Spring Integration provides capabilities to configure *Global Interceptors* and apply them on multiple channels. Look at the example below:

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

or

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

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

<channel-interceptor> element allows you to define a global interceptor which will be applied on all channels

that match patterns defined via *pattern* attribute. In the above case the global interceptor will be applied on 'foo' channel and all other channels that begin with 'bar' and 'input'. The *order* attribute allows you to manage the place where this interceptor will be injected. For example, channel 'inputChannel' could have individual interceptors configured locally (see below):

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

The reasonable question would be how global interceptor will be injected in relation to other interceptors configured locally or through other global interceptor definitions? Current implementation provides a very simple and clever mechanism of handling this. Positive number in the *order* attribute will ensure interceptor injection after existing interceptors and negative number will ensure that such interceptors injected before. This means that in the above example global interceptor will be injected *AFTER* (since its order is greater then 0) 'wire-tap' interceptor configured locally. If there was another global interceptor with matching *pattern* their order would be determined based on who's got the higher or lower value in *order* attribute. To inject global interceptor *BEFORE* the existing interceptors use negative value for the *order* attribute.

Note

Note that *order* and *pattern* attributes are optional. The default value for *order* will be 0 and for *pattern* is '*'

3.5.10. Wire Tap

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

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

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

Tip

The 'logging-channel-adapter' also accepts a boolean attribute: *'log-full-message'*. That is *false* by default so that only the payload is logged. Setting that to *true* enables logging of all headers in addition to the payload.

Note

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

Chapter 4. Message Endpoints

The first part of this chapter covers some background theory and reveals quite a bit about the underlying API that drives Spring Integration's various messaging components. This information can be helpful if you want to really understand what's going on behind the scenes. However, if you want to get up and running with the simplified namespace-based configuration of the various elements, feel free to skip ahead to Section 4.4, “Namespace Support” for now.

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 Section 3.1.2, “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("subscribableChannel");

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("pollableChannel");

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

There are many other configuration options for the Polling Consumer. For example, the trigger is a required property:

```
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 a valid 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);
```

The 'maxMessagesPerPoll' property specifies the maximum number of messages to receive within a given poll operation. This means that the poller will continue calling receive() *without waiting* until either `null` is returned or that max is reached. For example, if a poller has a 10 second interval trigger and a 'maxMessagesPerPoll' setting of 25, and it is polling a channel that has 100 messages in its queue, all 100 messages can be retrieved within 40 seconds. It grabs 25, waits 10 seconds, grabs the next 25, and so on.

The 'receiveTimeout' property specifies the amount of time the poller should wait if no messages are available when it invokes the receive operation. For example, consider two options that seem similar on the surface but

are actually quite different: the first has an interval trigger of 5 seconds and a receive timeout of 50 milliseconds while the second has an interval trigger of 50 milliseconds and a receive timeout of 5 seconds. The first one may receive a message up to 4950 milliseconds later than it arrived on the channel (if that message arrived immediately after one of its poll calls returned). On the other hand, the second configuration will never miss a message by more than 50 milliseconds. The difference is that the second option requires a thread to wait, but as a result it is able to respond much more quickly to arriving messages. This technique, known as "long polling", can be used to emulate event-driven behavior on a polled source.

A Polling Consumer may also delegate to a Spring `TaskExecutor`, and it can be configured to 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:

```
 <transformer input-channel="pollable"
              ref="transformer"
              output-channel="output">
     <poller>
         <interval-trigger interval="1000"/>
     </poller>
</transformer>
```

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

```
<transformer input-channel="pollable"
             ref="transformer"
             output-channel="output">
    <poller>
        <cron-trigger expression="*/10 * * * * MON-FRI"/>
    </poller>
</transformer>
```

If the input channel is a `PollableChannel`, then the poller configuration is required. Specifically, as mentioned above, the 'trigger' is a required property of the PollingConsumer class. Therefore, if you omit the "poller" sub-element for a Polling Consumer endpoint's configuration, an Exception may be thrown. However, it is also possible to create top-level pollers in which case only a "ref" is required:

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

<transformer input-channel="pollable"
             ref="transformer"
             output-channel="output">
   <poller ref="weekdayPoller"/>
</transformer>
```

In fact, to simplify the configuration, you can define a global default poller. A single top-level poller within an ApplicationContext may have the `default` attribute with a value of "true". In that case, any endpoint with a PollableChannel for its input-channel that is defined within the same ApplicationContext and has no explicitly configured 'poller' sub-element will use that default.

```
<poller id="defaultPoller" default="true" max-messages-per-poll="5">
    <interval-trigger interval="3" time-unit="SECONDS"/>
</poller>

<!-- No <poller/> sub-element is necessary since there is a default -->
<transformer input-channel="pollable"
             ref="transformer"
             output-channel="output"/>
```

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="REQUIRED"
                   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 of Spring 3.0, there is a "task" namespace in the core Spring Framework, and its <executor/> element supports the creation of a simple thread pool executor. That element accepts attributes for common concurrency settings such as pool-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"/>
</poller>

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

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 default `TaskScheduler` (see Section B.3, “Configuring the 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 "executor" element above is simply provided for convenience.

As mentioned in the background section for Polling Consumers above, you can also configure a Polling Consumer in such a way as to emulate event-driven behavior. With a long receive-timeout and a short interval-trigger, you can ensure a very timely reaction to arriving messages even on a polled message source. Note that this will only apply to sources that have a blocking wait call with a timeout. For example, the File poller does not block, each receive() call returns immediately and either contains new files or not. Therefore, even if a poller contains a long receive-timeout, that value would never be usable in such a scenario. On the other hand when using Spring Integration's own queue-based channels, the timeout value does have a chance to participate. The following example demonstrates how a Polling Consumer will receive Messages nearly instantaneously.

```
<service-activator input-channel="someQueueChannel"
       output-channel="output">
    <poller receive-timeout="30000">
        <interval-trigger interval="10"/>
    </poller>
</service-activator>
```

Using this approach does not carry much overhead since internally it is nothing more then a timed-wait thread which does not require nearly as much CPU resource usage as a thrashing, infinite while loop for example.

4.5. Payload Type Conversion

Throughout the reference manual, you will also see specific configuration and implementation examples of various endpoints which can accept a Message or any arbitrary Object as an input parameter. In the case of an Object, such parameter will be mapped to a Message payload or part of the payload or header (when using Spring Expression Language). However there are times when the type of input parameter of the endpoint method does not match the type of the payload or its part. In this scenario we need to perform type conversion. Spring Integration provides a convenient way for registering type converters (using Spring 3.x ConversionService) within its own instance of the conversion service bean named *integrationConversionService* which is automatically created as soon as the first converter is defined. To register such converter all you need is to implement `org.springframework.core.convert.converter.Converter` and register via cionvinient namespace support:

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

       <bean id="sampleConverter" class="foo.bar.TestConverter"/>
```

or

```
<int:converter>
               <bean class="org.springframework.integration.config.xml.ConverterParserTests$TestConverter3"/>
       </int:converter>
```

Chapter 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 `REPLY_CHANNEL` 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.

The argument in the service method could be either a Message or an arbitrary type. If the latter, then it will be assumed that it is a Message payload, which will be extracted from the message and injected into such service method. This is generally the recommended approach as it follows and promotes a POJO model when working with Spring Integration. Arguments may also have @Header, @Headers or @MessageMapping annotations as described in Section B.5, “Annotation Support”

> **Note**
> Since v1.0.3 of Spring Integration, the service method is not required to have an argument at all, which means you can now implement event-style Service Activators, where all you care about is an invocation of the service method, not worrying about the contents of the message. Think of it as a NULL JMS message. An example use-case for such an implementation could be a simple counter/monitor of messages deposited on the input channel.

Using a "ref" attribute is generally recommended if the custom Service Activator handler implementation can

be reused in other `<service-activator>` definitions. However if the custom Service Activator handler implementation should be scoped to a single definition of the `<service-activator>`, you can use an inner bean definition:

```
<service-activator id="exampleServiceActivator" input-channel="inChannel"
                                                output-channel = "outChannel" method="foo">
                <beans:bean class="org.foo.ExampleServiceActivator"/>
</service-activator>
```

Note

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

Chapter 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, HTTP, Web Services, and Mail. Those will be discussed in upcoming chapters of this reference guide. However, this chapter focuses on the simple but flexible Method-invoking Channel Adapter support. There are both inbound and outbound adapters, and each may be configured with XML elements provided in the core namespace.

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 cron expression for an individual channel-adapter, 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 * 9-17 * * MON-FRI"/>
    </poller>
</channel-adapter>
```

> **Note**
>
> If no poller is provided, then a single default poller must be registered within the context. See Section 4.4, “Namespace Support” for more detail.

6.2. The <outbound-channel-adapter/> element

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

```
<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>
<beans:bean id="target1" class="org.bar.Foo"/>
```

Using a "ref" attribute is generally recommended if the POJO consumer implementation can be reused in other `<outbound-channel-adapter>` definitions. However if the consumer implementation should be scoped to a

single definition of the `<outbound-channel-adapter>`, you can define it as inner bean:

```
<outbound-channel-adapter channel="channel2" method="method2">
        <beans:bean class="org.bar.Foo"/>

</outbound-channel-adapter>
```

Note

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

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

Chapter 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.

7.1.1. 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>
```

Configuration of `PayloadTypeRouter` is also supported via the namespace provided by Spring Integration (see Section B.2, “Namespace Support”), which essentially simplifies configuration by combining `<router/>` configuration and its corresponding implementation defined using `<bean/>` element into a single and more concise configuration element. The example below demonstrates `PayloadTypeRouter` configuration which is equivalent to the one above using Spring Integration's namespace support:

```
<payload-type-router input-channel="routingChannel">
        <mapping type="java.lang.String" channel="stringChannel" />
        <mapping type="java.lang.Integer" channel="integerChannel" />
</payload-type-router>
```

7.1.2. HeaderValueRouter

A `HeaderValueRouter` will send Messages to the channel based on the individual header value mappings. When `HeaderValueRouter` is created it is initialized with the *name* of the header to be evaluated, using `constructor-arg`. The *value* of the header could be one of two things:

1. Arbitrary value

2. Channel name

If arbitrary value, then a `channelResolver` should be provided to map *header values* to *channel names*. The example below uses `MapBasedChannelResolver` to set up a map of header values to channel names.

```
 <bean id="myHeaderValueRouter"
               class="org.springframework.integration.router.HeaderValueRouter">
  <constructor-arg value="someHeaderName" />
  <property name="channelResolver">
    <bean class="org.springframework.integration.channel.MapBasedChannelResolver">
      <property name="channelMap">
        <map>
          <entry key="someHeaderValue" value-ref="channelA" />
          <entry key="someOtherHeaderValue" value-ref="channelB" />
```

```
        </map>
      </property>
    </bean>
  </property>
</bean>
```

If `channelResolver` is not specified, then the *header value* will be treated as a *channel name* making configuration much simpler, where no `channelResolver` needs to be specified.

```
<bean id="myHeaderValueRouter"
        class="org.springframework.integration.router.HeaderValueRouter">
        <constructor-arg value="someHeaderName" />
</bean>
```

Similar to the `PayloadTypeRouter`, configuration of `HeaderValueRouter` is also supported via namespace support provided by Spring Integration (see Section B.2, “Namespace Support”). The example below demonstrates two types of namespace-based configuration of `HeaderValueRouter` which are equivalent to the ones above using Spring Integration namespace support:

1. Configuration where mapping of header values to channels is required

```
<header-value-router input-channel="routingChannel" header-name="testHeader">
        <mapping value="someHeaderValue" channel="channelA" />
        <mapping value="someOtherHeaderValue" channel="channelB" />
</header-value-router>
```

2. Configuration where mapping of header values is not required if header values themselves represent the channel names

```
<header-value-router input-channel="routingChannel" header-name="testHeader"/>
```

Note

The two router implementations shown above 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.

7.1.3. 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>
```

Configuration for `RecipientListRouter` is also supported via namespace support provided by Spring Integration (see Section B.2, “Namespace Support”). The example below demonstrates namespace-based

configuration of `RecipientListRouter` and all the supported attributes using Spring Integration namespace support:

```
<recipient-list-router id="customRouter" input-channel="routingChannel"
                                timeout="1234"
                                ignore-send-failures="true"
                                apply-sequence="true">
        <recipient channel="channel1"/>
        <recipient channel="channel2"/>
</recipient-list-router>
```

Note

The 'apply-sequence' flag here has the same affect as it does for a publish-subscribe-channel, and like publish-subscribe-channel it is disabled by default on the recipient-list-router. Refer to Section 3.5.3, “PublishSubscribeChannel Configuration” for more information.

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 of a custom Router implementation (extending AbstractMessageRouter):

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

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

<router ref="customRouter" input-channel="input3" default-output-channel="defaultOutput3"/>

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

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"/>
```

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

```
<router method="someMethod" input-channel="input3" default-output-channel="defaultOutput3">
        <beans:bean class="org.foo.MyCustomRouter"/>
</router>
```

Note

Using both the "ref" attribute and an inner handler definition in the same `<router>` configuration is not allowed, as it creates an ambiguous condition and will result in an Exception being thrown.

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 @Header parameter annotation that is documented in Section B.5, “Annotation Support”.

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

Note

For routing of XML-based Messages, including XPath support, see Chapter 32, *Dealing with XML Payloads*.

Chapter 8. Filter

8.1. Introduction

Message Filters are used to decide whether a Message should be passed along or dropped based on some criteria such as a Message Header value or even content within the Message itself. Therefore, a Message Filter is similar to a router, except that for each Message received from the filter's input channel, that same Message may or may not be sent to the filter's output channel. Unlike the router, it makes no decision regarding *which* Message Channel to send to but only decides *whether* to send.

> **Note**
> As you will see momentarily, the Filter does also support a discard channel, so in certain cases it *can* play the role of a very simple router (or "switch") based on a boolean condition.

In Spring Integration, a Message Filter may be configured as a Message Endpoint that delegates to some implementation of the `MessageSelector` interface. That interface is itself quite simple:

```
public interface MessageSelector {

    boolean accept(Message<?> message);

}
```

The `MessageFilter` constructor accepts a selector instance:

```
MessageFilter filter = new MessageFilter(someSelector);
```

In combination with the namespace and SpEL very powerful filters can be configured with very little java code.

8.2. The <filter> Element

The <filter> element is used to create a Message-selecting endpoint. In addition to "input-channel" and "output-channel" attributes, it requires a "ref". The "ref" may point to a MessageSelector implementation:

```
<filter input-channel="input" ref="selector" output-channel="output"/>

<bean id="selector" class="example.MessageSelectorImpl"/>
```

Alternatively, the "method" attribute can be added at which point the "ref" may refer to any object. The referenced method may expect either the `Message` type or the payload type of inbound Messages. The return value of the method must be a boolean value. Any time the method returns 'true', the Message *will* be passed along to the output-channel.

```
<filter input-channel="input" output-channel="output"
        ref="exampleObject" method="someBooleanReturningMethod"/>

<bean id="exampleObject" class="example.SomeObject"/>
```

If the selector or adapted POJO method returns `false`, there are a few settings that control the fate of the rejected Message. By default (if configured like the example above), the rejected Messages will be silently dropped. If rejection should instead indicate an error condition, then set the 'throw-exception-on-rejection' flag to `true`:

```
<filter input-channel="input" ref="selector"
     output-channel="output" throw-exception-on-rejection="true"/>
```

If you want the rejected messages to go to a specific channel, provide that reference as the 'discard-channel':

```
<filter input-channel="input" ref="selector"
     output-channel="output" discard-channel="rejectedMessages"/>
```

Note

A common usage for Message Filters is in conjunction with a Publish Subscribe Channel. Many filter endpoints may be subscribed to the same channel, and they decide whether or not to pass the Message for the next endpoint which could be any of the supported types (e.g. Service Activator). This provides a *reactive* alternative to the more *proactive* approach of using a Message Router with a single Point-to-Point input channel and multiple output channels.

Using a "ref" attribute is generally recommended if the custom filter implementation can be reused in other `<filter>` definitions. However if the custom filter implementation should be scoped to a single `<filter>` element, provide an inner bean definition:

```
<filter method="someMethod" input-channel="inChannel" output-channel="outChannel">
        <beans:bean class="org.foo.MyCustomFilter"/>
</filter>
```

Note

Using both the "ref" attribute and an inner handler definition in the same `<filter>` configuration is not allowed, as it creates an ambiguous condition, and it will therefore result in an Exception being thrown.

With the introduction of SpEL Spring Integration has added the `expression` attribute to the filter element. It can be used to avoid Java entirely for simple filters.

```
          <filter input-channel="input" expression="payload.equals(nonsense)"/>
```

The string passed as the expression attribute will be evaluated as a SpEL expression in the context of the message. If it is needed to include the result of an expression in the scope of the application context you can use the #{} notation as defined in the SpEL reference documentation SpEL reference documentation .

```
          <filter input-channel="input" expression="payload.matches(#{filterPatterns.nonsensePattern})"/>
```

Chapter 9. Transformer

9.1. 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 32, *Dealing with XML Payloads*.

9.2. 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.

```
<transformer id="testTransformer" ref="testTransformerBean" input-channel="inChannel"
                                          method="transform" output-channel="outChannel"/>
<beans:bean id="testTransformerBean" class="org.foo.TestTransformer" />
```

Using a "ref" attribute is generally recommended if the custom transformer handler implementation can be reused in other `<transformer>` definitions. However if the custom transformer handler implementation should be scoped to a single definition of the `<transformer>`, you can define an inner bean definition:

```
<transformer id="testTransformer" input-channel="inChannel" method="transform"
                                                                output-channel="outChannel">
        <beans:bean class="org.foo.TestTransformer"/>
</transformer>
```

> **Note**
>
> Using both the "ref" attribute and an inner handler definition in the same `<transformer>` configuration is not allowed, as it creates an ambiguous condition and will result in an Exception being thrown.

The method that is used for transformation may expect either the `Message` type or the payload type of inbound Messages. It may also accept Message header values either individually or as a full map by using the @Header

and @Headers parameter annotations respectively. 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 type is a Map, and the original Message payload was *not* a Map, the entries in that Map will be added to the Message headers of the original Message (the keys must be Strings). If the return value is *null*, then no reply Message will be sent (effectively the same behavior as a Message Filter returning false). Otherwise, the return value will be sent as the payload of an outbound reply Message.

There are a also a few Transformer implementations available out of the box. Because, it is fairly common to use the `toString()` representation of an Object, Spring Integration provides an `ObjectToStringTransformer` whose output is a Message with a String payload. That String is the result of invoking the toString operation on the inbound Message's payload.

```
<object-to-string-transformer input-channel="in" output-channel="out"/>
```

A potential example for this would be sending some arbitrary object to the 'outbound-channel-adapter' in the *file* namespace. Whereas that Channel Adapter only supports String, byte-array, or `java.io.File` payloads by default, adding this transformer immediately before the adapter will handle the necessary conversion. Of course, that works fine as long as the result of the `toString()` call is what you want to be written to the File. Otherwise, you can just provide a custom POJO-based Transformer via the generic 'transformer' element shown previously.

> **Tip**
> When debugging, this transformer is not typically necessary since the 'logging-channel-adapter' is capable of logging the Message payload. Refer to Section 3.5.10, “Wire Tap” for more detail.

If you need to serialize an Object to a byte array or deserialize a byte array back into an Object, Spring Integration provides symmetrical serialization transformers.

```
<payload-serializing-transformer input-channel="objectsIn" output-channel="bytesOut"/>

<payload-deserializing-transformer input-channel="bytesIn" output-channel="objectsOut"/>
```

If you only need to add headers to a Message, and they are not dynamically determined by Message content, then referencing a custom implementation may be overkill. For that reason, Spring Integration provides the 'header-enricher' element.

```
<header-enricher input-channel="in" output-channel="out">
    <header name="foo" value="123"/>
    <header name="bar" ref="someBean"/>
</header-enricher>
```

As added convenience, Spring Integration also provides *Object-to-Map* and *Map-to-Object* transformers which utilize Spring Expression Language (SpEL) to serialize and de-serialize the object graphs. Object hierarchy is introspected to the most primitive types (e.g., String, int etc.). The path to this type is described via SpEL, which becomes the *key*key in the transformed Map with primitive type being the value.

For example:

```
public class Parent{
    private Child child;
    private String name;
    // setters and getters are omitted
}

public class Child{
   private String name;
   private List<String> nickNames;
   // setters and getters are omitted
```

```
}
```

... will be transformed to a Map which looks like this: `{person.name=George, person.child.name=Jenna, person.child.nickNames[0]=Bimbo . . . etc}`

SpEL-based Map allows you to describe the object structure without sharing the actual types allowing you to restore/rebuild the object graph into a differently typed Object graph as long as you maintain the structure.

For example: The above structure could be easily restored back to the following Object graph via Map-to-Object transformer:

```
public class Father{
    private Kid child;
    private String name;
    // setters and getters are omitted
}

public class Kid{
   private String name;
   private List<String> nickNames;
   // setters and getters are omitted
}
```

To configure these transformers, Spring Integration provides namespace support Object-to-Map:

```
<object-to-map-transformer input-channel="directInput" output-channel="output"/>
```

Map-to-Object

```
<int:map-to-object-transformer input-channel="input"
                               output-channel="output"
                                type="org.foo.Person"/>
```

or

```
<int:map-to-object-transformer input-channel="inputA"
                                      output-channel="outputA"
                                      ref="person"/>
<bean id="person" class="org.foo.Person" scope="prototype"/>
```

Note

NOTE: 'ref' and 'type' attributes are mutually exclusive. You can only use either one. Also, if using 'ref' attribute you must point to a 'prototype' scoped bean, otherwise BeanCreationException will be thrown.

9.3. 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 that is documented in

Section B.5, “Annotation Support”

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

Chapter 10. Splitter

10.1. 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.

10.2. Programming model

The API for performing splitting consists from one base class, AbstractMessageSplitter, which is a MessageHandler 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 are 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 developers to focus on the domain objects without having to consider the Messaging system and produces code that is easier to test.
- 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. Since this decouples the code from the Spring Integration API and will typically be easier to test, it is the recommended approach.

10.3. 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. *Optional*. If reference to a bean is not provided, then it is assumed that the *payload* of the Message that arrived on the `input-channel` is an implementation of *java.util.Collection* and the default splitting logic will be applied on such Collection, incorporating each individual element into a Message and depositing it on the `output-channel`.

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)*.

Using a "ref" attribute is generally recommended if the custom splitter handler implementation can be reused in other `<splitter>` definitions. However if the custom splitter handler implementation should be scoped to a single definition of the `<splitter>`, you can configure an inner bean definition:

```
<splitter id="testSplitter" input-channel="inChannel" method="split"
                                                        output-channel="outChannel">
        <beans:bean class="org.foo.TestSplitter"/>
</spliter>
```

Note

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

10.4. 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()
}
```

Chapter 11. Aggregator

11.1. Introduction

Basically a mirror-image of the Splitter, the Aggregator is a type of Message Handler 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. In order to do this it requires a MessageStore

11.2. Functionality

The Aggregator combines a group of related messages, by correlating and storing 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 that aggregated message as output.

An main 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. The other two aspects are correlation and release

In Spring Integration, the grouping of the messages for aggregation (correlation) is done by default based on their CORRELATION_ID message header (i.e. the messages with the same CORRELATION_ID will be grouped together). However, this can be customized, and the users can opt for other ways of specifying how the messages should be grouped together, by using a CorrelationStrategy (see below).

To determine whether or not a group of messages may be processed, a ReleaseStrategy is consulted. The default release strategy for aggregator will release groups that have all messages from the sequence, but this can be entirely customized

11.3. Programming model

The Aggregation API consists of a number of classes:

- The interface `MessageGroupProcessor` and related base class `AbstractAggregatingMessageGroupProcessor` and its subclass `MethodInvokingAggregatingMessageGroupProcessor`
- The `ReleaseStrategy` interface and its default implementation `SequenceSizeReleaseStrategy`
- The `CorrelationStrategy` interface and its default implementation `HeaderAttributeCorrelationStrategy`

11.3.1. CorrelatingMessageHandler

The `CorrelatingMessageHandler` is a `MessageHandler` implementation, encapsulating the common functionalities of an Aggregator (and other correlating use cases), which are:

- correlating messages into a group to be aggregated
- maintaining those messages in a MessageStore until the group may be released
- deciding when the group is in fact may be released
- processing the released group into a single aggregated message
- recognizing and responding to an expired group

The responsibility of deciding how the messages should be grouped together is delegated to a `CorrelationStrategy` instance. The responsibility of deciding whether the message group can be released is delegated to a `ReleaseStrategy` instance.

Here is a brief highlight of the base `AbstractAggregatingMessageGroupProcessor` (the responsibility of implementing the aggregateMessages method is left to the developer):

```
public abstract class AbstractAggregatingMessageGroupProcessor
              implements MessageGroupProcessor {

    protected Map<String, Object> aggregateHeaders(MessageGroup group) {
  ....
    }

          protected abstract Object aggregatePayloads(MessageGroup group);

}
```

The CorrelationStrategy is owned by the `CorrelatingMessageHandler` and it has a default value based on the correlation ID message header:

```
private volatile CorrelationStrategy correlationStrategy =
          new HeaderAttributeCorrelationStrategy(MessageHeaders.CORRELATION_ID);
```

When appropriate, the simplest option is the `DefaultAggregatingMessageGroupProcessor`. It creates a single Message whose payload is a List of the payloads received for a given group. It uses the default `CorrelationStrategy` and `CompletionStrategy` as shown above. This works well for simple Scatter Gather implementations with either a Splitter, Publish Subscribe Channel, or Recipient List Router upstream.

Note

When using a Publish Subscribe Channel or Recipient List Router in this type of scenario, be sure to enable the flag to *apply-sequence*. That will add the necessary headers (correlation id, sequence number and sequence size). That behavior is enabled by default for Splitters in Spring Integration, but it is not enabled for the Publish Subscribe Channel or Recipient List Router because those components may be used in a variety of contexts where those headers are not necessary.

When implementing a specific aggregator object for an application, a developer can extend `AbstractAggregatingMessageGroupProcessor` and implement the `aggregatePayloads` 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.

11.3.2. ReleaseStrategy

The `ReleaseStrategy` interface is defined as follows:

```
public interface ReleaseStrategy {

  boolean canRelease(MessageGroup 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 ready for aggregation, and false otherwise.

When the group is released for aggregation, all its unmarked messages are processed and then marked so they will not be processed again. If the group is also complete (i.e. if all messages from a sequence have arrived or if there is no sequence defined) then the group is removed from the message store. Partial sequences can be released, in which case the next time the `ReleaseStrategy` is called it will be presented with a group containing marked messages (already processed) and unmarked messages (a potential new partial sequence)

Spring Integration provides an out-of-the box implementation for `ReleaseStrategy`, the `SequenceSizerReleaseStrategy`. 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. As shown above, it is also the default strategy.

11.3.3. CorrelationStrategy

The `CorrelationStrategy` interface is defined as follows:

```
public interface CorrelationStrategy {
```

```
  Object getCorrelationKey(Message<?> message);

}
```

The method shall return an Object which represents the correlation key used for grouping messages together. The key must satisfy the criteria used for a key in a Map with respect to the implementation of equals() and hashCode().

In general, any ordinary Java class (i.e. POJO) can implement the correlation decision mechanism, and the rules for mapping a message to a method's argument (or arguments) are the same as for a `ServiceActivator` (including support for @Header annotations). The method must return a value, and the value must not be `null`.

Spring Integration provides an out-of-the box implementation for `CorrelationStrategy`, the `HeaderAttributeCorrelationStrategy`. This implementation returns the value of one of the message headers (whose name is specified by a constructor argument) as the correlation key. By default, the correlation strategy is a HeaderAttributeCorrelationStrategy returning the value of the CORRELATION_ID header attribute.

11.4. Configuring an Aggregator with XML

Spring Integration supports the configuration of an aggregator via XML through the <aggregator/> element. Below you can see an example of an aggregator with all optional parameters defined.

```
<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
    release-strategy="releaseStrategyBean"  ❼
    release-strategy-method="canRelease" ❽
    correlation-strategy="correlationStrategyBean" ❾
    correlation-strategy-method="groupNumbersByLastDigit" ❿
    message-store="messageStore" 11
    send-partial-result-on-expiry="true" 12
    send-timeout="86420000" 13 />

<channel id="outputChannel"/>

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

<bean id="releaseStrategyBean" class="sample.PojoReleaseStrategy"/>

<bean id="correlationStrategyBean" class="sample.PojoCorrelationStrategy"/>
```

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 will use sequence size)* .

❽ A method defined on the bean referenced by `release-strategy`, *that implements the completion decision algorithm. Optional, with restrictions (requires `completion-strategy` to be present).*

❾ A reference to a bean that implements the correlation strategy. The bean can be an implementation of the CorrelationStrategy interface or a POJO. In the latter case the correlation-strategy-method attribute must be defined as well. *Optional (by default, the aggregator will use the correlation id header attribute)* .

❿ A method defined on the bean referenced by `correlation-strategy`, *that implements the correlation key algorithm. Optional, with restrictions (requires `correlation-strategy` to be present).*

A reference to a `MessageGroupStore` that can be used to store groups of messages under their correlation key until they are complete. *Optional* with default a volatile in-memory store.

Whether upon the expiration of the message group, the aggregator will try to aggregate the messages that have already arrived. *Optional (false by default).*

The timeout for sending the aggregated messages to the output or reply channel. *Optional*.

Using a "ref" attribute is generally recommended if a custom aggregator handler implementation can be reused in other `<aggregator>` definitions. However if a custom aggregator handler implementation should be scoped to a concrete definition of the `<aggregator>`, you can use an inner bean definition (starting with version 1.0.3) for custom aggregator handlers within the `<aggregator>` element:

```
<aggregator input-channel="input" method="sum" output-channel="output">
    <beans:bean class="org.foo.ExampleAggregator"/>
</aggregator>
```

Note

Using both a "ref" attribute and an inner bean definition in the same `<aggregator>` configuration is not allowed, as it creates an ambiguous condition. In such cases, an Exception will be thrown.

An example implementation of the aggregator bean 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 PojoReleaseStrategy {
...
  public boolean canRelease(List<Long> numbers) {
    int sum = 0;
    for (long number: numbers) {
      sum += number;
    }
    return sum >= maxValue;
  }
}
```

Note

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

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

```
public class PojoCorrelationStrategy {
...
  public Long groupNumbersByLastDigit(Long number) {
    return number % 10;
  }
}
```

For example, this aggregator would group numbers by some criterion (in our case the remainder after dividing by 10) and will hold the group until the sum of the numbers which represents the payload exceeds a certain value.

Note

Wherever it makes sense, the release strategy method, correlation strategy method and the aggregator method can be combined in a single bean (all of them or any two).

11.5. Managing State in an Aggregator: MessageGroupStore

Aggregator (and some other patterns in Spring Integration) is a stateful pattern that requires decisions to be made based on a group of messages that have arrived over a period of time, all with the same correlation key. The design of the interfaces in the stateful patterns (e.g. `ReleaseStrategy`) is driven by the principle that the components (framework and user) should be to remain stateless. All state is carried by the `MessageGroup` and its management is delegated to the `MessageGroupStore`.

The `MessageGroupStore` accumulates state information in `MessageGroups`, potentially forever. So to prevent stale state from hanging around, and for volatile stores to provide a hook for cleaning up when the application shots down, the `MessageGroupStore` allows the user to register callbacks to apply to `MessageGroups` when they expire. The interface is very straighforward:

```
public interface MessageGroupCallback {

  void execute(MessageGroupStore messageGroupStore, MessageGroup group);

}
```

The callback has access directly to the store and the message group so it can manage the persistent state (e.g. by removing the group from the store entirely).

The MessageGroupStore maintains a list of these callbacks which it applies when asked to all messages whose timestamp is earlier than a time supplied as a parameter:

```
public interface MessageGroupStore {
   void registerMessageGroupExpiryCallback(MessageGroupCallback callback);
   int expireMessageGroups(long timeout);
}
```

The expireMessageGroups method can be called with a timeout value: any message older than the current time

minus this value wiull be expired, and have the callbacks applied. Thus it is the user of the store that defines what is meant by message group "expiry".

As a convenience for users, Spring Integration provides a wrapper for the message expiry in the form of a `MessageGroupStoreReaper`:

```
<bean id="reaper" class="org...MessageGroupStoreReaper">
  <property name="messageGroupStore" ref="messageStore"/>
  <property name="timeout" value="10"/>
</bean>

<task:scheduled-tasks scheduler="scheduler">
  <task:scheduled ref="reaper" method="run" fixed-rate="10000"/>
</task:scheduled-tasks>
```

The reaper is a Runnable, and all that is happening is that the message group store's expire method is being called in the sample above once every 10 seconds. In addition to the reaper, the expiry callbacks are invoked when the application shuts down via a lifecycle callback in the `CorrelatingMessageHandler`.

The `CorrelatingMessageHandler` registers its own expiry callback, and this is the link with the boolean flag `send-partial-result-on-expiry` in the XML configuration of the aggregator. If the flag is set to true, then when the expiry callback is invoked then any unmarked messages in groups that are not yet released can be sent on to the downstream channel.

11.6. Configuring an Aggregator with Annotations

An aggregator configured using annotations can look like this.

```
public class Waiter {
  ...

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

  @ReleaseStrategy ❷
  public boolean releaseChecker(List<Message<?>> messages) {
    ...
  }

  @CorrelationStrategy ❸
  public String correlateBy(OrderItem item) {
    ...
  }

}
```

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 release strategy of an aggregator. If not present on any method, the aggregator will use the SequenceSizeCompletionStrategy.

❸ An annotation indicating that this method shall be used as the correlation strategy of an aggregator. If no correlation strategy is indicated, the aggregator will use the HeaderAttributeCorrelationStrategy based on CORRELATION_ID.

All of the configuration options provided by the xml element are also 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.

Chapter 12. Resequencer

12.1. Introduction

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

12.2. 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 the Resequencer does not process the messages in any way. It simply releases them in the order of their SEQUENCE_NUMBER header values.

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.

12.3. 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" ❶
  input-channel="inputChannel" ❷
  output-channel="outputChannel" ❸
  discard-channel="discardChannel" ❹
  release-partial-sequences="true" ❺
  message-store="messageStore" ❻
  send-partial-result-on-expiry="true" ❼
  send-timeout="86420000" ❽ />
```

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 (false by default)*.
If this flag is not specified (so a complete sequence is defined by the sequence headers) then it can make sense to provide a custom `Comparator` to be used to order the messages when sending (use the XML attribute `comparator` to point to a bean definition). If `release-partial-sequences` is true then there is no way with a custom comparator to define a partial sequence. To do that you would have to provide a `release-strategy` (also a reference to another bean definition, either a POJO or a `ReleaseStrategy`).

❻ A reference to a `MessageGroupStore` that can be used to store groups of messages under their correlation key until they are complete. *Optional* with default a volatile in-memory store.

❼ Whether, upon the expiration of the group, the ordered group should be sent out (even if some of the messages are missing). *Optional (false by default)*. See Section 11.5, “Managing State in an Aggregator: MessageGroupStore”.

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

Note

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

Chapter 13. Delayer

13.1. Introduction

A Delayer is a simple endpoint that allows a Message flow to be delayed by a certain interval. When a Message is delayed, the original sender will not block. Instead, the delayed Messages will be scheduled with an instance of `java.util.concurrent.ScheduledExecutorService` to be sent to the output channel after the delay has passed. This approach is scalable even for rather long delays, since it does not result in a large number of blocked sender Threads. On the contrary, in the typical case a thread pool will be used for the actual execution of releasing the Messages. Below you will find several examples of configuring a Delayer.

13.2. The <delayer> Element

The <delayer> element is used to delay the Message flow between two Message Channels. As with the other endpoints, you can provide the "input-channel" and "output-channel" attributes, but the delayer also requires at least the 'default-delay' attribute with the number of milliseconds that each Message should be delayed.

```
<delayer input-channel="input" default-delay="3000" output-channel="output"/>
```

If you need per-Message determination of the delay, then you can also provide the name of a header within the 'delay-header-name' attribute:

```
<delayer input-channel="input" output-channel="output"
         default-delay="3000" delay-header-name="delay"/>
```

In the example above the 3 second delay would only apply in the case that the header value is not present for a given inbound Message. If you only want to apply a delay to Messages that have an explicit header value, then you can set the 'default-delay' to 0. For any Message that has a delay of 0 (or less), the Message will be sent directly. In fact, if there is not a positive delay value for a Message, it will be sent to the output channel on the calling Thread.

> **Tip**
> The delay handler actually supports header values that represent an interval in milliseconds (any Object whose `toString()` method produces a value that can be parsed into a Long) as well as `java.util.Date` instances representing an absolute time. In the former case, the milliseconds will be counted from the current time (e.g. a value of 5000 would delay the Message for at least 5 seconds from the time it is received by the Delayer). In the latter case, with an actual Date instance, the Message will not be released until that Date occurs. In either case, a value that equates to a non-positive delay, or a Date in the past, will not result in any delay. Instead, it will be sent directly to the output channel in the original sender's Thread.

The delayer delegates to an instance of Spring's `TaskScheduler` abstraction. The default scheduler is a `ThreadPoolTaskScheduler` instance with a pool size of 1. If you want to delegate to a different scheduler, you can provide a reference through the delayer element's 'scheduler' attribute:

```
<delayer input-channel="input" output-channel="output"
         default-delay="0" delay-header-name="delay"
         scheduler="exampleTaskScheduler"/>

<task:scheduler id="exampleTaskScheduler" pool-size="3"/>
```

Chapter 14. Message Handler Chain

14.1. Introduction

The `MessageHandlerChain` is an implementation of `MessageHandler` that can be configured as a single Message Endpoint while actually delegating to a chain of other handlers, such as Filters, Transformers, Splitters, and so on. This can lead to a much simpler configuration when several handlers need to be connected in a fixed, linear progression. For example, it is fairly common to provide a Transformer before other components. Similarly, when providing a *Filter* before some other component in a chain, you are essentially creating a Selective Consumer. In either case, the chain only requires a single input-channel and a single output-channel as opposed to the configuration of channels for each individual component.

> **Tip**
>
> Spring Integration's *Filter* provides a boolean property 'throwExceptionOnRejection'. When providing multiple Selective Consumers on the same point-to-point channel with different acceptance criteria, this value should be set to 'true' (the default is false) so that the dispatcher will know that the Message was rejected and as a result will attempt to pass the Message on to other subscribers. If the Exception were not thrown, then it would appear to the dispatcher as if the Message had been passed on successfully even though the Filter had *dropped* the Message to prevent further processing.

The handler chain simplifies configuration while internally maintaining the same degree of loose coupling between components, and it is trivial to modify the configuration if at some point a non-linear arrangement is required.

Internally, the chain will be expanded into a linear setup of the listed endpoints, separated by direct channels. The reply channel header will not be taken into account within the chain: only after the last handler is invoked will the resulting message be forwarded on to the reply channel or the chain's output channel. Because of this setup all handlers except the last require a `setOutputChannel` implementation. The last handler only needs an output channel if the outputChannel on the MessageHandlerChain is set.

> **Note**
>
> As with other endpoints, the output-channel is optional. If there is a reply Message at the end of the chain, the output-channel takes precedence, but if not available, the chain handler will check for a reply channel header on the inbound Message.

In most cases there is no need to implement MessageHandlers yourself. The next section will focus on namespace support for the chain element. Most Spring Integration endpoints, like Service Activators and Transformers, are suitable for use within a `MessageHandlerChain`.

14.2. The <chain> Element

The <chain> element provides an 'input-channel' attribute, and if the last element in the chain is capable of producing reply messages (optional), it also supports an 'output-channel' attribute. The sub-elements are then filters, transformers, splitters, and service-activators. The last element may also be a router.

```
<chain input-channel="input" output-channel="output">
    <filter ref="someSelector" throw-exception-on-rejection="true"/>
```

```
    <header-enricher error-channel="customErrorChannel">
        <header name="foo" value="bar"/>
    </header-enricher>
    <service-activator ref="someService" method="someMethod"/>
</chain>
```

The <header-enricher> element used in the above example will set a message header with name "foo" and value "bar" on the message. A header enricher is a specialization of Transformer that touches only header values. You could obtain the same result by implementing a MessageHandler that did the header modifications and wiring that as a bean.

Some time you need to make a nested call to another chain from within the chain and then come back and continue execution within the original chain. To accomplish this you can utilize Messaging Gateway by including light-configuration via <gateway> element. For example:

```
         <si:chain id="main-chain" input-channel="inputA" output-channel="inputB">
                <si:header-enricher>
                        <si:header name="name" value="Many" />
                </si:header-enricher>
                <si:service-activator>
                        <bean class="org.foo.SampleService" />
                </si:service-activator>
                <si:gateway request-channel="inputC"/>
        </si:chain>
        <si:chain id="nested-chain-a" input-channel="inputC">
                <si:header-enricher>
                        <si:header name="name" value="Moe" />
                </si:header-enricher>
                <si:gateway request-channel="inputD"/>
                <si:service-activator>
                        <bean class="org.foo.SampleService" />
                </si:service-activator>
        </si:chain>
        <si:chain id="nested-chain-b" input-channel="inputD">
                <si:header-enricher>
                        <si:header name="name" value="Jack" />
                </si:header-enricher>
                <si:service-activator>
                        <bean class="org.foo.SampleService" />
                </si:service-activator>
        </si:chain>
```

In the above example the *nested-chain-a* will be called at the end of *main-chain* processing by the 'gateway' element configured there. While in *nested-chain-a* a call to a *nested-chain-b* will be made after header enrichment and then it will come back to finish execution in *nested-chain-b* finally getting back to the *main-chain*. When light version of <gateway> element is defined in the chain SI will construct an instance `SimpleMessagingGateway` (no need to provide 'service-interface' configuration) which will take the message in its current state and will place it on the channel defined via 'request-channel' attribute. Upon processing `Message` will be returned to the gateway and continue its journey within the current chain.

Chapter 15. Messaging Bridge

15.1. Introduction

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

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

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

15.2. The <bridge> Element

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

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

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

```
<bridge input-channel="pollable" output-channel="subscribable">
    <poller max-messages-per-poll="10">
        <interval-trigger interval="5" time-unit="SECONDS"/>
    </poller>
</bridge>
```

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

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

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

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

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

Note

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

Chapter 16. Inbound Messaging Gateways

16.1. GatewayProxyFactoryBean

Working with Objects instead of Messages is an improvement. However, it would be even better to have no dependency on the Spring Integration API at all - including the gateway class. For that reason, Spring Integration also provides a `GatewayProxyFactoryBean` that generates a proxy for any interface and internally invokes the gateway methods shown above. Namespace support is also provided as demonstrated by the following example.

```
<gateway id="fooService"
         service-interface="org.example.FooService"
         default-request-channel="requestChannel"
         default-reply-channel="replyChannel"/>
```

Then, the "fooService" can be injected into other beans, and the code that invokes the methods on that proxied instance of the FooService interface has no awareness of the Spring Integration API. The general approach is similar to that of Spring Remoting (RMI, HttpInvoker, etc.). See the "Samples" Appendix for an example that uses this "gateway" element (in the Cafe demo).

The reason that the attributes on the 'gateway' element are named 'default-request-channel' and 'default-reply-channel' is that you may also provide per-method channel references by using the @Gateway annotation.

```
public interface Cafe {

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

}
```

It is also possible to pass values to be interpreted as Message headers on the Message that is created and sent to the request channel by using the @Header annotation:

```
public interface FileWriter {

    @Gateway(requestChannel="filesOut")
    void write(byte[] content, @Header(FileHeaders.FILENAME) String filename);

}
```

If you prefer XML way of configuring Gateway methods, you can provide *method* sub-elements to the gateway configuration (see below)

```
<si:gateway id="myGateway" service-interface="org.foo.bar.TestGateway"
                        default-request-channel="inputC">
        <si:method name="echo" request-channel="inputA" reply-timeout="2" request-timeout="200"/>
            <si:method name="echoUpperCase" request-channel="inputB"/>
            <si:method name="echoViaDefault"/>
</si:gateway>
```

You can also provide individual headers per method invocation via XML. This could be very useful if the headers you want to set are static in nature and you don't want to embed them in the gateway's method signature via `@Header` annotations. For example, in the Loan Broker example we want to influence how aggregation of the Loan quotes will be done based on what type of request was initiated (single quote or all quotes). Determining the type of the request by evaluating what gateway method was invoked, although

possible would violate the separation of concerns paradigm (method is a java artifact), but expressing your intention (meta information) via Message headers is natural in a Messaging architecture.

```
<int:gateway id="loanBrokerGateway"
                service-interface="org.springframework.integration.loanbroker.LoanBrokerGateway">
        <int:method name="getLoanQuote" request-channel="loanBrokerPreProcessingChannel">
                <int:header name="RESPONSE_TYPE" value="BEST"/>
        </int:method>
        <int:method name="getAllLoanQuotes" request-channel="loanBrokerPreProcessingChannel">
                <int:header name="RESPONSE_TYPE" value="ALL"/>
        </int:method>
</int:gateway>
```

In the above case you can clearly see how a different header value will be set for the 'RESPONSE_TYPE' header based on the gateway's method.

As with anything else, Gateway invocation might result in errors. By default any error that has occurred downstream will be re-thrown as a MessagingExeption (RuntimeException) upon the Gateway's method invocation. However there are times when you may want to treat an Exception as a valid reply, by mapping it to a Message. To accomplish this our Gateway provides support for Exception mappers via the *exception-mapper* attribute.

```
<si:gateway id="sampleGateway"
                default-request-channel="gatewayChannel"
                service-interface="foo.bar.SimpleGateway"
                exception-mapper="exceptionMapper"/>

<bean id="exceptionMapper" class="foo.bar.SampleExceptionMapper"/>
```

foo.bar.SampleExceptionMapper is the implementation of *org.springframework.integration.message.InboundMessageMapper* which only defines one method: `toMessage(Object object)`.

```
public static class SampleExceptionMapper implements InboundMessageMapper<Throwable>{
        public Message<?> toMessage(Throwable object) throws Exception {
                MessageHandlingException ex = (MessageHandlingException) object;
                return MessageBuilder.withPayload("Error happened in message: " +
                            ex.getFailedMessage().getPayload()).build();
        }

}
```

16.2. Asynchronous Gateway

As a pattern the Messaging Gateway is a very nice way to hide messaging-specific code while still exposing the full capabilities of the messaging system. And `GatewayProxyFactoryBean` provides a convenient way to expose a Proxy over a service-interface thus giving you a POJO-based access to a messaging system (based on objects in your own domain, or primitives/Strings, etc). But when a gateway is exposed via simple POJO methods which return values it does imply that for each Request message (generated when the method is invoked) there must be a Reply message (generated when the method has returned). Since Messaging systems naturally are asynchronous you may not always be able to guarantee the contract where *"for each request there will always be be a reply"*. With Spring Integration 2.0 we are introducing support for an *Asynchronous Gateway* which is a convenient way to initiate flows where you may not know if a reply is expected or how long will it take for it to arrive.

A natural way to handle these types of scenarios in Java would be relying upon *java.util.concurrent.Future* instances, and that is exactly what Spring Integration uses to support an *Asynchronous Gateway*.

From the XML configuration, there is nothing different and you still define *Asynchronous Gateway* the same way as a regular Gateway.

```
<int:gateway id="mathService"
     service-interface="org.springframework.integration.sample.gateway.futures.MathServiceGateway"
     default-request-channel="requestChannel"/>
```

However the Gateway Interface (service-interface) is a bit different.

```
public interface MathServiceGateway {
        Future<Integer> multiplyByTwo(int i);
}
```

As you can see from the example above the return type for the gateway method is `Future`. When `GatewayProxyFactoryBean` sees that the return type of the gateway method is `Future`, it immediately switches to the async mode by utilizing an `AsyncTaskExecutor`. That is all. The call to a method always returns immediately with `Future` encapsulating the interaction with the framework. Now you can interact with the `Future` at your own pace to get the result, timeout, get the exception etc...

```
MathServiceGateway mathService = ac.getBean("mathService", MathServiceGateway.class);
Future<Integer> result = mathService.multiplyByTwo(number);
// do something else here since the reply might take a moment
int finalResult =  result.get(1000, TimeUnit.SECONDS);
```

For a more detailed example, please refer to the *async-gateway* sample distributed within the Spring Integration samples.

Chapter 17. Message Publishing

The AOP Message Publishing feature allows you to construct and send a message as a by-product of method invocation. For example, imagine you have a component and every time the state of this component changes you would like to be notified via a Message. The easiest way to send such notifications would be to send a message to a dedicated channel, but how would you connect the method invocation that changes the state of the object to a message sending process, and how should the notification Message be structured? The AOP Message Publishing feature handles these responsibilities with a configuration-driven approach.

17.1. Message Publishing Configuration

Spring Integration provides two approaches: XML and Annotation-driven.

17.1.1. Annotation-driven approach via @Publisher annotation

The annotation-driven approach allows you to annotate any method with the `@Publisher` annotation, specifying 'channel' attribute. The Message will be constructed from the return value of method invocation and sent to a channel specified by 'channel' attribute. To further manage message structure you can also use a combination of both `@Payload` and `@Header` annotations.

Internally message publishing feature of Spring Integration uses both Spring AOP by defining `PublisherAnnotationAdvisor` and Spring 3.0 Expression Language (SpEL) support, giving you considerable flexibility and control over the structure of the *Message* it will build.

`PublisherAnnotationAdvisor` defines and binds the following variables:

- *#return* - will bind to a return value allowing you to reference it or its attributes (e.g., *#return.foo* where 'foo' is an attribute of the object bound to *#return*)
- *#exception* - will bind to an exception if one is thrown by the method invocation.
- *#args* - will bind to method arguments, so individual arguments could be extracted by name (e.g., *#args.fname* as in the above method)

Let's look at couple of examples:

```
@Publisher
public String defaultPayload(String fname, String lname) {
        return fname + " " + lname;
}
```

In the above example the Message will be constructed with the following structure:

- Message payload - will be the return type and value of the method. This is the default.
- A newly constructed message will be sent to a default publisher channel configured with annotation post processor (see the end of this section).

```
@Publisher(channel="testChannel")
public String defaultPayload(String fname, @Header("last") String lname) {
        return fname + " " + lname;
```

```
}
```

In this example everything is the same as above, however we are not using default publishing channel. Instead we are specifying the publishing channel via 'channel' attribute of @Publisher annotation. We are also adding @Header annotation which results in the Message header with the name 'last' and the value of 'lname' input parameter to be added to the newly constructed Message.

```
@Publisher(channel="testChannel")
@Payload
public String defaultPayloadButExplicitAnnotation(String fname, @Header String lname) {
        return fname + " " + lname;
}
```

The above example is almost identical to the previous one. The only difference here is that we are using @Payload annotation on the method, thus explicitly specifying that the return value of the method should be used as a payload of the Message.

```
@Publisher(channel="testChannel")
@Payload("#return + #args.lname")
public String setName(String fname, String lname, @Header("x") int num) {
        return fname + " " + lname;
}
```

Here we are expending on the previous configuration by using Spring Expression language in the @Payload annotation further instructing the framework on how the message should be constructed. In this particular case the message will be a concatenation of the return value of the method invocation and 'lname' input argument. Message header 'x' with value of 'num' input argument will be added to the newly constructed Message.

```
@Publisher(channel="testChannel")
public String argumentAsPayload(@Payload String fname, @Header String lname) {
        return fname + " " + lname;
}
```

In the above example you see another usage of @Payload annotation. Here we are annotating method argument which will become a payload of newly constructed message.

As with most other annotation-driven features in Spring, you will need to register a post-processor (`PublisherAnnotationBeanPostProcessor`).

```
<bean class="org.springframework.integration.aop.PublisherAnnotationBeanPostProcessor"/>
```

You can also use namespace support for added convenience:

```
<si:annotation-config default-publisher-channel="defaultChannel"/>
```

Similar to other Spring annotations (e.g., @Controller), `@Publisher` is a meta-annotation, which means you can define your own annotations that will be treated as `@Publisher`

```
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Publisher(channel="auditChannel")
public @interface Audit {
}
```

Here we defined `@Audit` annotation which itself is a `@Publisher`. Also note that you can define `channel`

attribute on the meta-annotation thus encapsulating the behavior of where messages will be sent inside of this annotation. Now you can annotate any method:

```
@Audit
public String test() {
    return "foo";
}
```

In the above example every invocation of `test()` method will result in Message with payload which is the return value of the method invocation to be sent to *auditChannel* You can also annotate the class which would mean that the properties of this annotation will be applied on every public method of this class

```
@Audit
static class BankingOperationsImpl implements BankingOperations {

  public String debit(String amount) {
     . . .
  }

  public String credit(String amount) {
     . . .
  }
}
```

17.1.2. XML-based approach via <publishing-interceptor> element

The XML-based approach allows you to configure the same AOP-based Message Publishing functionality with simple namespace-based configuration of a `MessagePublishingInterceptor`. It certainly has some benefits over the annotation-driven approach since it allows you to use AOP pointcut expressions, thus possibly intercepting multiple methods at once or intercepting and publishing methods to which you don't have the source code.

To configure Message Publishing via XML, you only need to do the following two things:

- Provide configuration for `MessagePublishingInterceptor` via the `<publishing-interceptor>` XML element.
- Provide AOP configuration to apply the `MessagePublishingInterceptor` to managed objects.

```
<aop:config>
        <aop:advisor advice-ref="interceptor" pointcut="bean(testBean)" />
</aop:config>
<publishing-interceptor id="interceptor" default-channel="defaultChannel">
  <method pattern="echo" payload="'Echoing: ' + #return" channel="echoChannel">
    <header name="foo" value="bar"/>
  </method>
  <method pattern="repl*" payload="'Echoing: ' + #return" channel="echoChannel">
    <header name="foo" expression="'bar'.toUpperCase()"/>
  </method>
  <method pattern="echoDef*" payload="#return"/>
</publishing-interceptor>
```

As you can see the `<publishing-interceptor>` configuration look rather similar to Annotation-based approach and it also utilizes the power of the Spring 3.0 Expression Language.

In the above example the execution of the `echo` method of a `testBean` will render a *Message* with the following structure:

- The Message payload will be of type String and value of "Echoing: [value]" where `value` is the value

returned by an executed method.

- The Message will have header with the key "foo" value "bar".
- The Message will be sent to `echoChannel`.

The second method is very similar to the first. Here every method that begins with 'repl' will render a Message with the following structure:

- The Message payload will be the same as in the above sample
- The Message will have header with the key "foo" and value that is the result of the SpEL expression `'bar'.toUpperCase()`.
- The Message will be sent to `echoChannel`.

The second method, mapping the execution of any method that begins with `echoDef` of `testBean`, will produce a Message with the following structure.

- The Message payload will be the value returned by an executed method.
- Since the `channel` attribute is not provided explicitly, the Message will be sent to the `defaultChannel` defined by the *publisher*.

For simple mapping rules you can rely on the *publisher* defaults. For example:

```
<publishing-interceptor id="anotherInterceptor"/>
```

This will map the return value of every method that matches the pointcut expression to a payload and will be sent to a *default-channel*. If the *defaultChannel*is not specified (as above) the messages will be sent to the global *nullChannel*.

17.1.3. Producing and publishing messages based on schedule

In the above sections we looked at the Message publishing feature of Spring Integration which constructs and publishes messages as by-products of Method invocations. However you are still responsible to invoke the method. With scheduling support added to Spring Framework 3.0 we've added another useful feature to Spring Integration - support for scheduled Message producers/publishers. Scheduling could be based on several triggers. Currently we support `cron`, `fixed-rate`, `fixed-delay` as well as the custom triggers implemented by you.

Support for scheduled producers/publishers is provided via <*scheduled-producer*> xml element. Lets look at couple of examples:

```
<scheduled-producer id="fixedDelayProducer"
                        fixed-delay="1000"
                        payload-expression="'fixedDelayTest'"
                        channel="fixedDelayChannel"/>
```

In the above example scheduled producer will be created which will construct the Message with payload being the result of the expression defined in `payload-expression` attribute. Such message will be created and sent every time after a delay specified in the `fixed-delay` attribute.

```
<scheduled-producer id="fixedDelayProducer"
                        fixed-rate="1000"
                        payload-expression="'fixedDelayTest'"
                        channel="fixedDelayChannel"/>
```

This example is very similar to the previous one, except that we are using `fixed-rate` attribute which will allow us to send messages at the fixed rate.

```
<scheduled-producer id="fixedDelayProducer"
                        cron="7 6 5 4 3 ?"
                        payload-expression="'fixedDelayTest'"
                        channel="fixedDelayChannel"/>
```

This example demonstrates how you can apply Cron trigger specified by `cron` attribute.

```
<scheduled-producer id="headerExpressionsProducer" fixed-delay="99"
                        payload-expression="'headerExpressionsTest'"
                        channel="headerExpressionsChannel"
                        auto-startup="false">
        <header name="foo" expression="6 * 7"/>
        <header name="bar" value="x"/>
</scheduled-producer>
```

Here you can see that in a way very similar to Message publishing feature we are enriching a newly constructed Message with extra Message headers which could take scalar values as well as Spring expressions.

If you need to implement your own custom trigger you can use `trigger` attribute pointing to any spring configured bean which implements `org.springframework.scheduling.Trigger` interface.

```
<scheduled-producer id="triggerRefProducer" trigger="customTrigger"
                        payload-expression="'triggerRefTest'" channel="triggerRefChannel"/>

<beans:bean id="customTrigger" class="org.springframework.scheduling.support.PeriodicTrigger">
        <beans:constructor-arg value="9999"/>
</beans:bean>
```

Chapter 18. Message History

The key benefit of messaging architecture is loose coupling where participating components do not maintain any awareness about one another. This fact alone makes you architecture extremely flexible allowing you to change components without affecting the rest of the flow, change messaging routs, message consuming styles (polling vs event driven) etc... However, this unassuming style of architecture could prove to be problematic when things go wrong. For example, if something happened you would probably like to get as much information about the message as you can (its origin, where it was etc.)

Message History is one of those patterns that could help by giving you an option to maintain some level of awareness of a message path either for debugging purposes or to maintain an audit trail. Spring integration provides a simple way to configure your message flows to maintain Message History by adding Message History header to a Message every time a message goes through a tracked component.

18.1. Message History Configuration

To enable Message History all you need is define `message-history` element in your configuration.

```
<int:message-history/>
```

Now every named component (component that has an 'id' defined) will be tracked. The framework will set the '$history' header in your Message who's value is very simple - `List<Properties>`. The need for this simple structure is mandated by the loosely coupled architecture of messaging systems where the framework must not require you to share any dependencies outside of Java itself.

```
<int:gateway id="sampleGateway"
    service-interface="org.springframework.integration.history.sample.SampleGateway"
    default-request-channel="bridgeInChannel"/>

<int:chain id="sampleChain" input-channel="chainChannel" output-channel="filterChannel">
        <int:header-enricher>
                <int:header name="baz" value="baz"/>
        </int:header-enricher>
</int:chain>
```

The above configuration will produce a very simple Message History structure:

```
[{name=sampleGateway, type=gateway, timestamp=1283281668091},
 {name=sampleChain, type=chain, timestamp=1283281668094}]
```

To get access to Message History all you need is access the MessageHistory header. For example:

```
Iterator<Properties> historyIterator =
    message.getHeaders().get(MessageHistory.HEADER_NAME, MessageHistory.class).iterator();
assertTrue(historyIterator.hasNext());
Properties gatewayHistory = historyIterator.next();
assertEquals("sampleGateway", gatewayHistory.get("name"));
assertTrue(historyIterator.hasNext());
Properties chainHistory = historyIterator.next();
assertEquals("sampleChain", chainHistory.get("name"));
```

Some times you might not want to track all of the components. To accomplish this all you need is provide `tracked-components` attribute where you can specify comma delimited list of component names and/or patterns you want to track.

```
<int:message-history tracked-components="*Gateway, sample*, foo"/>
```

In the above example, Message History will only be maintained for all of the components that end with 'Gateway', all components that start with 'sample' and 'foo' component.

Note

Remember, that by definition History is immutable (you can't re-write history,although some try), therefore Message History can not be changed once written. Every attempt will end in exception.

Chapter 19. File Support

19.1. Introduction

Spring Integration's File support extends the Spring Integration Core with a dedicated vocabulary to deal with reading, writing, and transforming files. It provides a namespace that enables elements defining Channel Adapters dedicated to files and support for Transformers that can read file contents into strings or byte arrays.

This section will explain the workings of `FileReadingMessageSource` and `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.

19.2. 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}"/>
```

To prevent creating messages for certain files, you may supply a `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}"
        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. The `CompositeFileListFilter` enables the composition.

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

```
<file:inbound-channel-adapter id="filesIn"
      directory="file:${input.directory}" prevent-duplicates="true"/>

<file:inbound-channel-adapter id="filesIn"
       directory="file:${input.directory}"
       filter="customFilterBean" />

<file:inbound-channel-adapter id="filesIn"
       directory="file:${input.directory}"
       filename-pattern="^test.*$" />
```

The first channel adapter is relying on the default filter that just prevents duplication, the second is using a custom filter, and the third is using the *filename-pattern* attribute to add a `Pattern` based filter to the `FileReadingMessageSource`. The *file-name-pattern* and *filter* attributes are mutually exclusive, but you can use a `CompositeFileListFilter` to use any combination of filters, including a pattern based filter to fit your particular needs.

When multiple processes are reading from the same directory it can be desirable to lock files to prevent them from being picked up concurrently. To do this you can use a `FileLocker`. There is a java.nio based implementation available out of the box, but it is also possible to implement your own locking scheme. The nio locker can be injected as follows

```
<file:inbound-channel-adapter id="filesIn"
      directory="file:${input.directory}" prevent-duplicates="true">
   <file:nio-locker/>
 </file:inbound-channel-adapter>
```

A custom locker you can configure like this:

```
<file:inbound-channel-adapter id="filesIn"
      directory="file:${input.directory}" prevent-duplicates="true">
   <file:locker ref="customLocker"/>
 </file:inbound-channel-adapter>
```

When filtering and locking files is not enough it might be needed to control the way files are listed entirely. To implement this type of requirement you can use an implementation of `DirectoryScanner`. This scanner allows you to determine entirely what files are listed each poll. This is also the interface that Spring Integration uses internally to wire FileListFilters FileLocker to the FileReadingMessageSource. A custom DirectoryScanner can be injected into the <file:inbound-channel-adapter/> on the `scanner` attribute.

```
<file:inbound-channel-adapter id="filesIn"
      directory="file:${input.directory}" prevent-duplicates="true" scanner="customDirectoryScanner"/>
```

This gives you full freedom to choose the ordering, listing and locking strategies.

19.3. Writing files

To write messages to the file system you can use a `FileWritingMessageHandler`. This class can deal with File, String, or byte array payloads. In its simplest form the `FileWritingMessageHandler` only requires a destination directory for writing the files. The name of the file to be written is determined by the handler's `FileNameGenerator`. The default implementation looks for a Message header whose key matches the constant defined as `FileHeaders.FILENAME`.

Additionally, you can configure the encoding and the charset that will be used in case of a String payload.

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}"/>
```

The namespace based configuration also supports a `delete-source-files` attribute. If set to `true`, it will trigger deletion of the original source files after writing to a destination. The default value for that flag is `false`.

```
<file:outbound-channel-adapter id="filesOut"
            directory="file:${output.directory}"
            delete-source-files="true"/>
```

Note

The `delete-source-files` attribute will only have an effect if the inbound Message has a File payload or if the `FileHeaders.ORIGINAL_FILE` header value contains either the source File instance or a String representing the original file path.

In cases where you want to continue processing messages based on the written File you can use the `outbound-gateway` instead. It plays a very similar role as the `outbound-channel-adapter`. However after writing the File, it will also send it to the reply channel as the payload of a Message.

```
<file:outbound-gateway id="mover" request-channel="moveInput"
           reply-channel="output"
           directory="${output.directory}"
           delete-source-files="true"/>
```

Note

The 'outbound-gateway' works well in cases where you want to first move a File and then send it through a processing pipeline. In such cases, you may connect the file namespace's 'inbound-channel-adapter' element to the 'outbound-gateway' and then connect that gateway's reply-channel to the beginning of the pipeline.

If you have more elaborate requirements or need to support additional payload types as input to be converted to file content you could extend the FileWritingMessageHandler, but a much better option is to rely on a `Transformer`.

19.4. 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 Spring's `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. If nothing else, this can be useful for debugging (consider using with a Wire Tap).

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

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

<file:file-to-string-transformer input-channel="input" output-channel="output
            delete-files="true" charset="UTF-8"/>
```

The *delete-files* option signals to the transformer that it should delete the inbound File after the transformation is complete. This is in no way a replacement for using the `AcceptOnceFileListFilter` when the FileReadingMessageSource is being used in a multi-threaded environment (e.g. Spring Integration in general).

Chapter 20. JDBC Support

Spring Integration provides Channel Adapters for receiving and sending messages via database queries.

20.1. Inbound Channel Adapter

The main function of an inbound Channel Adapter is to execute a SQL `SELECT` query and turn the result set into a message. The message payload is the whole result set, expressed as a `List`, and the types of the items in the list depends on the row-mapping strategy that is used. The default strategy is a generic mapper that just returns a `Map` for each row i nthe query. Optionally this can be changed by adding a reference to requires a reference to a `RowMapper` instance (see the Spring JDBC documentation for more detailed information about row mapping).

> **Note**
>
> If you want to convert rows in the SELECT query result to individual messages you can use a downstream splitter.

The inbound adapter also requires a reference to either `JdbcTemplate` instance or `DataSource`.

As well as the `SELECT` statement to generate the messages, the adapter above also has an `UPDATE` statement that is being used to mark the records as processed, so they don't show up in the next poll. The update can be parameterised by the list of ids from the original select. This is done through a naming convention by default (a column in the input result set called "id" is translated into a list in the parameter map for the update called "id"). The following example defines an inbound Channel Adapter with an update query and a `DataSource` reference.

```
<jdbc:inbound-channel-adapter query="select * from item where status=2"
  channel="target" data-source="dataSource"
  update="update item set status=10 where id in (:id)" />
```

> **Note**
>
> The parameters in the update query are specified with a colon (:) prefix to the name of a parameter (which in this case is an expression to be applied to each of the rows in the polled result set). This is a standard feature of the named parameter JDBC support in Spring JDBC combined with a convention (projection onto the polled result list) adopted in Spring Integration. The underlying Spring JDBC features limit the available expressions (e.g. most special characters other than period are disallowed), but since the target is usually a list of or an individual object addressable by simple bean paths this isn't unduly restrictive.

To change the parameter generation strategy you can inject a `SqlParameterSourceFactory` into the adapter to override the default behaviour (the adapter has a `sql-parameter-source-factory` attribute).

20.1.1. Polling and Transactions

The inbound adapter accepts a regular Spring Integration poller as a sub element, so for instance the frequency of the polling can be controlled. A very important feature of the poller for JDBC usage is the option to wrap the poll operation in a transaction, for example:

```
<jdbc:inbound-channel-adapter query="..."
  channel="target" data-source="dataSource"
  update="...">
  <poller>
    <interval-trigger interval="1000"/>
```

```
    <transactional/>
  </poller>
</jdbc:inbound-channel-adapter>
```

Note

If a poller is not explicitly specified a default value will be used (and as per normal with Spring Integration can be defined as a top level bean)

In this example the database is polled every 1000 milliseconds, and the update and select queries are both executed in the same transaction. The transaction manager configuration is not shown, but as long as it is aware of the data source then the poll is transactional. A common use case is for the downstream channels to be direct channels (the default), so that the endpoints are invoked in the same thread, and hence the same transaction. then if any of them fails, the transaction rolls back and the input data are reverted to their original state.

20.2. Outbound Channel Adapter

The outbound Channel Adapter is the inverse of the inbound: its role is to handle a message and use it to execute a SQL query. The message payload and headers are available by default as input parameters to the query, for instance:

```
<jdbc:outbound-channel-adapter
  query="insert into foos (id, status, name) values (:headers[$id], 0, :payload[foo])"
  channel="input" data-source="dataSource"/>
```

In the example above, messages arriving on the channel "input" have a payload of a map with key "foo", so the `[]` operator dereferences that value from the map. The headers are also accessed as a map.

Note

The parameters in the query above are bean property expressions on the incoming message (not Spring EL expressions). This behaviour is part of the `SqlParameterSource` which is the default source created by the outbound adapter. Other behaviour is possible in the adapter, and requires the user to inject a different `SqlParameterSourceFactory` .

The outbound adapter requires a reference to either a DataSource or a JdbcTemplate. It can also have a `SqlParameterSourceFactory` injected to control the binding of incoming message to the query.

If the input channel is a direct channel then the outbound adapter runs its query in the same thread, and therefor ethe same transaction (if there is one) as the sender of the message.

20.3. Outbound Gateway

The outbound Gateway is like a combination of the outbound and inbound adapters: its role is to handle a message and use it to execute a SQL query and then respond with the result sending it to a reply channel. The message payload and headers are available by default as input parameters to the query, for instance:

```
<jdbc:outbound-gateway
  update="insert into foos (id, status, name) values (:headers[$id], 0, :payload[foo])"
  request-channel="input" reply-channel="output" data-source="dataSource" />
```

The result of the above would be to insert a record into the "foos" table and return a message to the output

channel indicating the number of rows affected (the payload is a map `{UPDATED=1}`.

If the update query is an insert with auto-generated keys, the reply message can be populated with the generated keys by adding `keys-generated="true"` to the above example (this is not the default because it is not supported by some database platforms). For example:

```
<jdbc:outbound-gateway
  update="insert into foos (status, name) values (0, :payload[foo])"
  request-channel="input" reply-channel="output" data-source="dataSource"
  keys-generated="true"/>
```

Instead of the update count or the generated keys, you can also provide a select query to execute and generate a reply message that way (like the inbound adapter), e.g:

```
<jdbc:outbound-gateway
  update="insert into foos (id, status, name) values (:headers[$id], 0, :payload[foo])"
  query="select * from foos where id=:headers[$id]"
  request-channel="input" reply-channel="output" data-source="dataSource" />
```

Like with the adapters there is also the option to provide `SqlParameterSourceFactory` instances for request and reply. The default is the same as for the outbound adapter, so the request message is available as the root of an expression. If keys-generated="true" then the root of the expression is the generated keys (a map if there is only one or a list of maps if multi-valued).

The outbound gateway requires a reference to either a DataSource or a JdbcTemplate. It can also have a `SqlParameterSourceFactory` injected to control the binding of incoming message to the query.

20.4. Message Store

The JDBC module provides an implementation of the Spring Integration `MessageStore` (important in the Claim Check pattern) and `MessageGroupStore` (important in stateful patterns like Aggregator) backed by a database. Both interfaces are implemented by the JdbcMessageStore and there is also support for configuring store instances in XML. For example:

```
<jdbc:message-store id="messageStore" data-source="dataSource"/>
```

A `JdbcTemplate` can be specified instead of a `DataSource`.

Other optional attributes are show in the next example:

```
<jdbc:message-store id="messageStore" data-source="dataSource"
  lob-handler="lobHandler" table-prefix="MY_INT_"/>
```

Here we have specified a `LobHandler` for dealing with messages as large objects (e.g. often necessary if using Oracle) and a prefix for the table names in the queries generated by the store. The table name prefix defaults to "INT_".

20.4.1. Initializing the Database

Spring Integration ships with some sample scripts that can be used to initialize a database. In the spring-integration-jdbc JAR file you will find scripts in the `org.springframework.integration.jdbc` package: there is a create and a drop script example for a range of common database platforms. A common way to use these scripts is to reference them in a Spring JDBC data source initializer. Note that the scripts are

provided as samples or specifications of the the required table and column names. You may find that you need to enhance them for production use (e.g. with index declarations).

20.4.2. Partitioning a Message Store

It is common to use a `JdbcMessageStore` as a global store for a group of applications, or nodes in the same application. To provide some portection against name clashes, and to give control over the database meta-data configuration, the message store allows the tables to be partitioned in two ways. One is to use separate table names, by changing the prefix as described above, and the other is to specify a "region" name for partitioning data within a single table. An important use case for this is using the store to manage persistent queues backing a Spring Integration channel. The message data for a persistent channel is keyed in the store on the channel name, so if the channel names are not globally unique then there is the danger of channels picking up data that was not intended for them. To avoid this the message store region can be used to keep data separate for different physical channels that happen to have the same logical name.

Chapter 21. JMS Support

Spring Integration provides Channel Adapters for receiving and sending JMS messages. There are actually two JMS-based inbound Channel Adapters. The first uses Spring's `JmsTemplate` to receive based on a polling period. The second is "message-driven" and relies upon a Spring MessageListener container. There is also an outbound Channel Adapter which uses the `JmsTemplate` to convert and send a JMS Message on demand.

Whereas the JMS Channel Adapters are intended for unidirectional Messaging (send-only or receive-only), Spring Integration also provides inbound and outbound JMS Gateways for request/reply operations. The inbound gateway relies on one of Spring's MessageListener container implementations 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 sends a JMS Message to a "request-destination" and then receives a reply Message. The "reply-destination" reference (or "reply-destination-name") can be configured explicitly or else the outbound gateway will use a JMS TemporaryQueue.

21.1. 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">
    <integration:poller>
        <integration:interval-trigger interval="30" time-unit="SECONDS"/>
    </integration:poller>
</jms:inbound-channel-adapter>
```

Tip

Notice from the configuration that the inbound-channel-adapter is a Polling Consumer. That means that it invokes receive() when triggered. This should only be used in situations where polling is done relatively infrequently and timeliness is not important. For all other situations (a vast majority of JMS-based use-cases), the *message-driven-channel-adapter* described below is a better option.

Note

All of the JMS adapters that require a reference to the ConnectionFactory will automatically look for a bean named "connectionFactory" by default. That is why you don't see a "connection-factory" attribute in many of the examples. However, if your JMS ConnectionFactory has a different bean name, then you will need to provide that attribute.

If 'extract-payload' is set to true (which is the default), the received JMS Message will be passed through the MessageConverter. When relying on the default SimpleMessageConverter, this means that the resulting Spring Integration Message will have the JMS Message's body as its payload. A JMS TextMessage will produce a String-based payload, a JMS BytesMessage will produce a byte array payload, and a JMS ObjectMessage's Serializable instance will become the Spring Integration Message's payload. If instead you prefer to have the raw JMS Message as the Spring Integration Message's payload, then set 'extract-payload' to false.

```
<jms:inbound-channel-adapter id="jmsIn"
                             destination="inQueue"
                             channel="exampleChannel"
                             extract-payload="false"/>
    <integration:poller>
        <integration:interval-trigger interval="30" time-unit="SECONDS"/>
```

```
    </integration:poller>
</jms:inbound-channel-adapter>
```

21.2. Message-Driven Channel Adapter

The "message-driven-channel-adapter" requires a reference to either an instance of a Spring MessageListener container (any subclass of `AbstractMessageListenerContainer`) or both `ConnectionFactory` and `Destination` (a 'destinationName' can be provided in place of the 'destination' reference). The following example defines a message-driven Channel Adapter with a `Destination` reference.

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

> **Note**
> The Message-Driven adapter also accepts several properties that pertain to the MessageListener container. These values are only considered if you do not provide an actual 'container' reference. In that case, an instance of DefaultMessageListenerContainer will be created and configured based on these properties. For example, you can specify the "transaction-manager" reference, the "concurrent-consumers" value, and several other property references and values. Refer to the JavaDoc and Spring Integration's JMS Schema (spring-integration-jms-1.0.xsd) for more detail.

The 'extract-payload' property has the same effect as described above, and once again its default value is 'true'. The poller sub-element is not applicable for a message-driven Channel Adapter, as it will be actively invoked. For most usage scenarios, the message-driven approach is better since the Messages will be passed along to the `MessageChannel` as soon as they are received from the underlying JMS consumer.

21.3. 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"/>
```

As with the inbound Channel Adapters, there is an 'extract-payload' property. However, the meaning is reversed for the outbound adapter. Rather than applying to the JMS Message, the boolean property applies to the Spring Integration Message payload. In other words, the decision is whether to pass the Spring Integration Message *itself* as the JMS Message body or whether to pass the Spring Integration Message's payload as the JMS Message body. The default value is once again 'true'. Therefore, if you pass a Spring Integration Message whose payload is a String, a JMS TextMessage will be created. If on the other hand you want to send the actual Spring Integration Message to another system via JMS, then simply set this to 'false'.

> **Note**
> Regardless of the boolean value for payload extraction, the Spring Integration MessageHeaders will map to JMS properties as long as you are relying on the default converter or provide a

reference to another instance of HeaderMappingMessageConverter (the same holds true for 'inbound' adapters except that in those cases, it's the JMS properties mapping *to* Spring Integration MessageHeaders).

21.4. Inbound Gateway

Spring Integration's message-driven JMS inbound-gateway delegates to a `MessageListener` container, supports dynamically adjusting concurrent consumers, and can also handle replies. The inbound gateway requires references to a `ConnectionFactory`, and a request `Destination` (or 'requestDestinationName'). The following example defines a JMS "inbound-gateway" that receives from the JMS queue referenced by the bean id "inQueue" and sends to the Spring Integration channel named "exampleChannel".

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

Since the gateways provide request/reply behavior instead of unidirectional send *or* receive, they also have two distinct properties for the "payload extraction" (as discussed above for the Channel Adapters' 'extract-payload' setting). For an inbound-gateway, the 'extract-request-payload' property determines whether the received JMS Message body will be extracted. If 'false', the JMS Message itself will become the Spring Integration Message payload. The default is 'true'.

Similarly, for an inbound-gateway the 'extract-reply-payload' property applies to the Spring Integration Message that is going to be converted into a reply JMS Message. If you want to pass the whole Spring Integration Message (as the body of a JMS ObjectMessage) then set this to 'false'. By default, it is also 'true' such that the Spring Integration Message *payload* will be converted into a JMS Message (e.g. String payload becomes a JMS TextMessage).

As with anything else, Gateway invocation might result in error. By default Producer will not be notified of the errors thta might have occurredon ythe consumer side and will time out waiting for the reply. However there might be times when you to communicate error condition back to the consumer, in other words treat the Exception as a valid reply valid reply by mapping it to a Message. To accomplish this JMS Inbound Gateway provides support for Exception mappers via *exception-mapper* attribute.

```
<int-jms:inbound-gateway request-destination="requestQueue"
          request-channel="jmsinputchannel"
          exception-mapper="errorMessageMapper"/>

<bean id="exceptionMapper" class="foo.bar.SampleExceptionMapper"/>
```

foo.bar.SampleExceptionMapper is the implementation of *org.springframework.integration.message.InboundMessageMapper* which only defines one method `toMessage(Object object)`.

```
public static class SampleExceptionMapper implements InboundMessageMapper<Throwable>{
        public Message<?> toMessage(Throwable object) throws Exception {
                MessageHandlingException ex = (MessageHandlingException) object;
                return MessageBuilder.withPayload("Error happened in message: " +
                            ex.getFailedMessage().getPayload()).build();
        }

}
```

21.5. Outbound Gateway

The outbound Gateway creates JMS Messages from Spring Integration Messages and then sends to a 'request-destination'. It will then handle the JMS reply Message either by using a selector to receive from the 'reply-destination' that you configure, or if no 'reply-destination' is provided, it will create JMS TemporaryQueues. Notice that the "reply-channel" is also provided.

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

The 'outbound-gateway' payload extraction properties are inversely related to those of the 'inbound-gateway' (see the discussion above). That means that the 'extract-request-payload' property value applies to the Spring Integration Message that is being converted into a JMS Message to be *sent as a request*, and the 'extract-reply-payload' property value applies to the JMS Message that is *received as a reply* and then converted into a Spring Integration Message to be subsequently sent to the 'reply-channel' as shown in the example configuration above.

21.6. Message Conversion, Marshalling and Unmarshalling

If you need to convert the message, all JMS adapters and gateways, allow you to provide a `MessageConverter` via *message-converter* attribute. Simply provide the bean name of an instance of `MessageConverter` that is available within the same ApplicationContext. Also, to provide some consistency with Marshaller and Unmarshaller interfaces Spring provides `MarshallingMessageConverter` which you can configure with your own custom Marshallers and Unmarshallers

```
<int-jms:inbound-gateway request-destination="requestQueue"
                         request-channel="inbound-gateway-channel"
                         message-converter="marshallingMessageConverter"/>

<bean id="marshallingMessageConverter"
      class="org.springframework.jms.support.converter.MarshallingMessageConverter">
  <constructor-arg>
     <bean class="org.bar.SampleMarshaller"/>
  </constructor-arg>
  <constructor-arg>
     <bean class="org.bar.SampleUnmarshaller"/>
  </constructor-arg>
</bean>
```

Note

Note, however, that when you provide your own MessageConverter instance, it will still be wrapped within the HeaderMappingMessageConverter. This means that the 'extract-request-payload' and 'extract-reply-payload' properties may effect what actual objects are passed to your converter. The HeaderMappingMessageConverter itself simply delegates to a target MessageConverter while also mapping the Spring Integration MessageHeaders to JMS Message properties and vice-versa.

21.7. JMS Backed Message Channels

The Channel Adapters and Gateways featured above are all intended for applications that are integrating with

other external systems. The inbound options assume that some other system is sending JMS Messages to the JMS Destination and the outbound options assume that some other system is receiving from the Destination. The other system may or may not be a Spring Integration application. Of course, when sending the Spring Integration Message instance as the body of the JMS Message itself (with the 'extract-payload' value set to false), it is assumed that the other system is based on Spring Integration. However, that is by no means a requirement. That flexibility is one of the benefits of using a Message-based integration option with the abstraction of "channels" or Destinations in the case of JMS.

There are cases where both the producer and consumer for a given JMS Destination are intended to be part of the same application, running within the same process. This could be accomplished by using a pair of inbound and outbound Channel Adapters. The problem with that approach is that two adapters are required even though conceptually the goal is to have a single Message Channel. A better option is supported as of Spring Integration version 2.0. Now it is possible to define a single "channel" when using the JMS namespace.

```
<jms:channel id="jmsChannel" queue="exampleQueue"/>
```

The channel in the above example will behave much like a normal <channel/> element from the main Spring Integration namespace. It can be referenced by both "input-channel" and "output-channel" attributes of any endpoint. The difference is that this channel is backed by a JMS Queue instance named "exampleQueue". This means that asynchronous messaging is possible between the producing and consuming endpoints, but unlike the simpler asynchronous Message Channels created by adding a <queue/> sub-element within a non-JMS <channel/> element, the Messages are not just stored in an in-memory queue. Instead those Messages are passed within a JMS Message body, and the full power of the underlying JMS provider is then available for that channel. Probably the most common rationale for using this alternative would be to take advantage of the persistence made available by the *store and forward* approach of JMS messaging. If configured properly, the JMS-backed Message Channel also supports transactions. In other words, a producer would not actually write to a transactional JMS-backed channel if its send operation is part of a transaction that rolls back. Likewise, a consumer would not physically remove a JMS Message from the channel if the reception of that Message is part of a transaction that rolls back. Note that the producer and consumer transactions are separate in such a scenario. This is significantly different than the propagation of a transactional context across the simple, synchronous <channel/> element that has no <queue/> sub-element.

Since the example above is referencing a JMS Queue instance, it will act as a point-to-point channel. If on the other hand, publish/subscribe behavior is needed, then a separate element can be used, and a JMS Topic can be referenced instead.

```
<jms:publish-subscribe-channel id="jmsChannel" topic="exampleTopic"/>
```

For either type of JMS-backed channel, the name of the destination may be provided instead of a reference.

```
<jms:channel id="jmsQueueChannel" queue-name="exampleQueueName"/>

<jms:publish-subscribe-channel id="jmsTopicChannel" topic-name="exampleTopicName"/>
```

In the examples above, the Destination names would be resolved by Spring's default `DynamicDestinationResolver` implementation, but any implementation of the `DestinationResolver` interface could be provided. Also, the JMS `ConnectionFactory` is a required property of the channel, but by default the expected bean name would be "connectionFactory". The example below provides both a custom instance for resolution of the JMS Destination names and a different name for the ConnectionFactory.

```
<jms:channel id="jmsChannel" queue-name="exampleQueueName"
             destination-resolver="customDestinationResolver"
             connection-factory="customConnectionFactory"/>
```

21.8. JMS Samples

To experiment with these JMS adapters, check out the samples available within the "samples/jms" directory 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" file 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"). Both of the samples accept input via stdin and then echo back to stdout. Look at the configuration to see how these messages are routed over JMS.

Chapter 22. Web Services Support

22.1. 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 a Spring Web Services `DestinationProvider` for determining the URI of the Web Service to be called.

```
simpleGateway = new SimpleWebServiceOutboundGateway(destinationProvider);

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

> **Note**
> When using the namespace support described below, you will only need to set a URI. Internally, the parser will configure a fixed URI DestinationProvider implementation. If you do need dynamic resolution of the URI at runtime, however, then the DestinationProvider can provide such behavior as looking up the URI from a registry. See the Spring Web Services javadoc for more information about the DestinationProvider strategy.

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.

22.2. Inbound Web Service Gateways

To send a message to a channel upon receiving a Web Service invocation, there are two options again: `SimpleWebServiceInboundGateway` and `MarshallingWebServiceInboundGateway`. The former will extract a `javax.xml.transform.Source` from the `WebServiceMessage` and set it as the message payload. The latter provides support for implementation of the `Marshaller` and `Unmarshaller` interfaces. If the incoming web service message is a SOAP message the SOAP Action header will be added to the headers of the `Message` that is forwarded onto the request channel.

```
simpleGateway = new SimpleWebServiceInboundGateway();
simpleGateway.setRequestChannel(forwardOntoThisChannel);
simpleGateway.setReplyChannel(listenForResponseHere); //Optional

marshallingGateway = new MarshallingWebServiceInboundGateway(marshaller);
//set request and optionally reply channel
```

Both gateways implement the Spring Web Services `MessageEndpoint` interface, so they can be configured with a `MessageDispatcherServlet` as per standard Spring Web Services configuration.

For more detail on how to use these components, see the Spring Web Services reference guide's chapter covering creating a Web Service. The chapter covering Object/XML mapping is also applicable again.

22.3. 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"/>
```

Note

Notice that this example does not provide a 'reply-channel'. If the Web Service were to return a non-empty response, the Message containing that response would be sent to the reply channel provided in the request Message's REPLY_CHANNEL header, and if that were not available a channel resolution Exception would be thrown. If you want to send the reply to another channel instead, then provide a 'reply-channel' attribute on the 'outbound-gateway' element.

Tip

When invoking a Web Service that returns an empty response after using a String payload for the request Message, *no reply Message will be sent by default*. Therefore you don't need to set a 'reply-channel' or have a REPLY_CHANNEL header in the request Message. If for any reason you actually *do* want to receive the empty response as a Message, then provide the 'ignore-empty-responses' attribute with a value of *false* (this only applies for Strings, because using a Source or Document object simply leads to a NULL response and will therefore *never* generate a reply Message).

To set up an inbound Web Service Gateway, use the "inbound-gateway":

```
<ws:inbound-gateway id="simpleGateway"
                    request-channel="inputChannel"/>
```

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

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

And for inbound:

```
<ws:inbound-gateway id="marshallingGateway"
                    request-channel="requestChannel"
                    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" on the outbound gateways.

For either outbound gateway type, a "destination-provider" attribute can be specified instead of the "uri" (exactly one of them is required). You can then reference any Spring Web Services DestinationProvider implementation (e.g. to lookup the URI at runtime from a registry).

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

For the simple inbound gateway type, the "extract-payload" attribute can be set to false to forward the entire

`WebServiceMessage` instead of just its payload as a `Message` to the request channel. This might be useful, for example, when a custom Transformer works against the `WebServiceMessage` directly.

Chapter 23. RMI Support

23.1. 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.

23.2. 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>
```

23.3. 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>
```

23.4. 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"/>
```

Chapter 24. HttpInvoker Support

24.1. 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.

24.2. HttpInvoker Inbound Gateway

To receive messages over http you can 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.

24.3. 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.

24.4. 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"/>
```

Chapter 25. HTTP Support

25.1. Introduction

The HTTP support allows for the execution of HTTP requests and the processing of inbound HTTP requests. Because interaction over HTTP is always synchronous, even if all that is returned is a 200 status code, the HTTP support consists of two gateway implementations: `HttpInboundEndpoint` and `HttpRequestExecutingMessageHandler`.

25.2. Http Inbound Gateway

To receive messages over HTTP you need to use an HTTP inbound Channel Adapter or Gateway. In common with the HttpInvoker support the HTTP inbound adapters need to be deployed within a servlet container. The easiest way to do this is to provide a servlet definition in *web.xml*, see Section 24.2, “HttpInvoker Inbound Gateway” for further details. Below is an example bean definition for a simple HTTP inbound endpoint.

```
<bean id="httpInbound" class="org.springframework.integration.http.HttpRequestHandlingMessagingGateway">
        <property name="requestChannel" ref="httpRequestChannel" />
        <property name="replyChannel" ref="httpReplyChannel" />
</bean>
```

The `HttpRequestHandlingMessagingGateway` accepts a list of `HttpMessageConverter` instances or else relies on a default list. The converters allow customization of the mapping from `HttpServletRequest` to `Message`. The default converters encapsulate simple strategies, which for example will create a String message for a *POST* request where the content type starts with "text", see the Javadoc for full details.

Starting with this release MultiPart File support was implemented. If the request has been wrapped as a *MultipartHttpServletRequest*, when using the default converters, that request will be converted to a Message payload that is a MultiValueMap containing values that may be byte arrays, Strings, or instances of Spring's `MultipartFile` depending on the content type of the individual parts.

> **Note**
> The HTTP inbound Endpoint will locate a MultipartResolver in the context if one exists with the bean name "multipartResolver" (the same name expected by Spring's DispatcherServlet). If it does in fact locate that bean, then the support for MultipartFiles will be enabled on the inbound request mapper. Otherwise, it will fail when trying to map a multipart-file request to a Spring Integration Message. For more on Spring's support for MultipartResolvers, refer to the Spring Reference Manual.

In sending a response to the client there are a number of ways to customize the behavior of the gateway. By default the gateway will simply acknowledge that the request was received by sending a 200 status code back. It is possible to customize this response by providing a 'viewName' to be resolved by the Spring MVC `ViewResolver`. In the case that the gateway should expect a reply to the `Message` then setting the expectReply flag (constructor argument) will cause the gateway to wait for a reply `Message` before creating an HTTP response. Below is an example of a gateway configured to serve as a Spring MVC Controller with a view name. Because of the constructor arg value of TRUE, it wait for a reply. This also shows how to customize the HTTP methods accepted by the gateway, which are *POST* and *GET* by default.

```
<bean id="httpInbound" class="org.springframework.integration.http.HttpRequestHandlingController">
        <constructor-arg value="true" /> <!-- indicates that a reply is expected -->
        <property name="requestChannel" ref="httpRequestChannel" />
        <property name="replyChannel" ref="httpReplyChannel" />
```

```
        <property name="viewName" value="jsonView" />
        <property name="supportedMethodNames" >
                <list>
                        <value>GET</value>
                        <value>DELETE</value>
                </list>
        </property>
        <property name="expectReply" value="true" />
</bean>
```

The reply message will be available in the Model map. The key that is used for that map entry by default is 'reply', but this can be overridden by setting the 'replyKey' property on the endpoint's configuration.

25.3. Http Outbound Gateway

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

```
<bean id="httpOutbound" class="org.springframework.integration.http.HttpRequestExecutingMessageHandler" >
        <constructor-arg value="http://localhost:8080/example" />
        <property name="outputChannel" ref="responseChannel" />
</bean>
```

This bean definition will execute HTTP requests by delegating to a `RestTemplate`. That template in turn delegates to a list of HttpMessageConverters to generate the HTTP request body from the Message payload. You can configure those converters as well as the ClientHttpRequestFactory instance to use:

```
<bean id="httpOutbound" class="org.springframework.integration.http.HttpRequestExecutingMessageHandler" >
        <constructor-arg value="http://localhost:8080/example" />
        <property name="outputChannel" ref="responseChannel" />
        <property name="messageConverters" ref="messageConverterList" />
        <property name="requestFactory" ref="customRequestFactory" />
</bean>
```

By default the HTTP request will be generated using an instance of `SimpleClientHttpRequestFactory` which uses the JDK `HttpURLConnection`. Use of the Apache Commons HTTP Client is also supported through the provided `CommonsClientHttpRequestFactory` which can be injected as shown above.

25.4. HTTP Namespace Support

Spring Integration provides an "http" 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/http'. The schema location should then map to 'http://www.springframework.org/schema/integration/http/spring-integration-http.xsd'.

To configure an inbound http channel adapter which is an instance of `HttpInboundEndpoint` configured not to expect a response.

```
<http:inbound-channel-adapter id="httpChannelAdapter" channel="requests" supported-methods="PUT, DELETE"/>
```

To configure an inbound http gateway which expects a response.

```
<http:inbound-gateway id="inboundGateway" request-channel="requests" reply-channel="responses"/>
```

To configure the outbound gateway you can use the namespace support as well. The following code snippet shows the different configuration options for an outbound Http gateway. Most importantly, notice that the 'http-method' and 'expected-response-type' are provided. Those are two of the most commonly configured

values. The default http-method is POST, and the default response type is *null*. With a null response type, the payload of the reply Message would only contain the status code (e.g. 200) as long as it's a successful status (non-successful status codes will throw Exceptions). If you are expecting a different type, such as a `String`, then provide that fully-qualified class name as shown below.

```
<http:outbound-gateway id="example"
                request-channel="requests"
                url="http://localhost/test"
                http-method="POST"
                extract-request-payload="false"
                expected-response-type="java.lang.String"
                charset="UTF-8"
                request-factory="requestFactory"
                request-timeout="1234"
                reply-channel="replies"/>
```

If your outbound adapter is to be used in a unidirectional way, then you can use an outbound-channel-adapter instead. This means that a successful response will simply execute without sending any Messages to a reply channel. In the case of any non-successful response status code, it will throw an exception. The configuration looks very similar to the gateway:

```
<http:outbound-channel-adapter id="example"
                        url="http://localhost/example"
                        http-method="GET"
                        channel="requests"
                        charset="UTF-8"
                        extract-payload="false"
                        expected-response-type="java.lang.String"
                        request-factory="someRequestFactory"
                        order="3"
                        auto-startup="false"/>
```

25.5. HTTP Samples

25.5.1. Multipart HTTP request - RestTemplate (client) and Http Inbound Gateway (server)

This example demonstrates how simple it is to send a Multipart HTTP request via Spring's RestTemplate and receive it by Spring Integration HTTP Inbound Adapter. All we are doing is creating `MultiValueMap` and populating it with multi-part data. `RestTemplate` will take care of the rest by converting it to `MultipartHttpServletRequest` THis particular client will send a multipart Http Request which contains the name of the company as well as the image file with company logo.

```
RestTemplate template = new RestTemplate();
String uri = "http://localhost:8080/multipart-http/inboundAdapter.htm";
Resource s2logo =
   new ClassPathResource("org/springframework/integration/samples/multipart/spring09_logo.png");
MultiValueMap map = new LinkedMultiValueMap();
map.add("company", "SpringSource");
map.add("company-logo", s2logo);
HttpHeaders headers = new HttpHeaders();
headers.setContentType(new MediaType("multipart", "form-data"));
HttpEntity request = new HttpEntity(map, headers);
ResponseEntity<?> httpResponse = template.exchange(uri, HttpMethod.POST, request, null);
```

That is all for the client.

On the server side we have the following configuration:

```
<int-http:inbound-channel-adapter id="httpInboundAdapter"
        channel="receiveChannel"
        name="/inboundAdapter.htm"
        supported-methods="GET, POST" />

<int:channel id="receiveChannel"/>

<int:service-activator input-channel="receiveChannel">
        <bean class="org.springframework.integration.samples.multipart.MultipartReceiever"/>
</int:service-activator>

<bean id="multipartResolver"
    class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>
```

The 'httpInboundAdapter' will receive the request, convert it to a `Message` with a payload as `LinkedMultiValueMap` which we are parsing in the 'multipartReceiver' service-activator;

```
public void recieve(LinkedMultiValueMap<String, Object> multipartRequest){
  System.out.println("### Successfully recieved multipart request ###");
  for (String elementName : multipartRequest.keySet()) {
    if (elementName.equals("company")){
      System.out.println("\t" + elementName + " - " +
        ((String[]) multipartRequest.getFirst("company"))[0]);
    } else if (elementName.equals("company-logo")){
      System.out.println("\t" + elementName + " - as UploadedMultipartFile: " +
        ((UploadedMultipartFile) multipartRequest.getFirst("company-logo")).getOriginalFilename());
    }
  }
}
```

You should see the following output:

```
### Successfully recieved multipart request ###
   company - SpringSource
   company-logo - as UploadedMultipartFile: spring09_logo.png
```

Chapter 26. TCP and UDP Support

Spring Integration provides Channel Adapters for receiving and sending messages over internet protocols. Both UDP (User Datagram Protocol) and TCP (Transmission Control Protocol) adapters are provided. Each adapter provides for one-way communication over the underlying protocol. In addition, simple inbound and outbound tcp gateways are provided. These are used when two-way communication is needed.

26.1. Introduction

Two flavors each of UDP inbound and outbound adapters are provided `UnicastSendingMessageHandler` sends a datagram packet to a single destination. `UnicastReceivingChannelAdapter` receives incoming datagram packets. `MulticastSendingMessageHandler` sends (broadcasts) datagram packets to a multicast address. `MulticastReceivingChannelAdapter` receives incoming datagram packets by joining to a multicast address.

TCP inbound and outbound adapters are provided `TcpSendingMessageHandler` sends messages over TCP. `TcpReceivingChannelAdapter` receives messages over TCP.

> **Tip**
> If you have been using an earlier 2.0 milestone, note that the adapters are no longer configured with connection options directly; instead, they are given a reference to a connection factory. See below.

An inbound TCP gateway is provided; this allows for simple request/response processing. While the gateway can support any number of connections, each connection can only process serially. The thread that reads from the socket waits for, and sends, the response before reading again. If the connection factory is configured for single use connections, the connection is closed after the socket times out.

An outbound TCP gateway is provided; this allows for simple request/response processing. If the associated connection factory is configured for single use connections, a new connection is immediately created for each new request. Otherwise, if the connection is in use, the calling thread blocks on the connection until either a response is received or an I/O error occurs.

26.2. UDP Adapters

```
<ip:udp-outbound-channel-adapter id="udpOut"
                host="somehost"
                port="11111"
                multicast="false"
                channel="exampleChannel" />
```

A simple UDP outbound channel adapter.

> **Tip**
> When setting multicast to true, provide the multicast address in the host attribute.

UDP is an efficient, but unreliable protocol. Two attributes are added to improve reliability. When check-length is set to true, the adapter precedes the message data with a length field (4 bytes in network byte order). This enables the receiving side to verify the length of the packet received. If a receiving system uses a buffer that is too short the contain the packet, the packet can be truncated. The length header provides a mechanism to detect this.

```
<ip:udp-outbound-channel-adapter id="udpOut"
               host="somehost"
               port="11111"
               multicast="false"
               check-length="true"
               channel="exampleChannel" />
```

An outbound channel adapter that adds length checking to the datagram packets.

Tip

The recipient of the packet must also be configured to expect a length to precede the actual data. For a Spring Integration UDP inbound channel adapter, set its `check-length` attribute.

The second reliability improvement allows an application-level acknowledgment protocol to be used. The receiver must send an acknowledgment to the sender within a specified time.

```
<ip:udp-outbound-channel-adapter id="udpOut"
               host="somehost"
               port="11111"
               multicast="false"
               check-length="true"
               acknowledge="true"
               ack-host="thishost"
               ack-port="22222"
               ack-timeout="10000"
               channel="exampleChannel" />
```

An outbound channel adapter that adds length checking to the datagram packets and waits for an acknowledgment.

Tip

Setting acknowledge to true implies the recipient of the packet can interpret the header added to the packet containing acknowledgment data (host and port). Most likely, the recipient will be a Spring Integration inbound channel adapter.

Tip

When multicast is true, an additional attribute min-acks-for-success specifies how many acknowledgments must be received within the ack-timeout.

For even more reliable networking, TCP can be used.

```
<ip:udp-inbound-channel-adapter id="udpReceiver"
               channel="udpOutChannel"
               port="11111"
               receive-buffer-size="500"
               multicast="false"
               check-length="true" />
```

A basic unicast inbound udp channel adapter.

```
<ip:udp-inbound-channel-adapter id="udpReceiver"
               channel="udpOutChannel"
               port="11111"
               receive-buffer-size="500"
               multicast="true"
               multicast-address="225.6.7.8"
               check-length="true" />
```

A basic multicast inbound udp channel adapter.

26.3. TCP Connection Factories

For TCP, the configuration of the underlying connection is provided using a Connection Factory. Two types of connection factory are provided; a client connection factory and a server connection factory. Client connection factories are used to establish outgoing connections; Server connection factories listen for incoming connections.

A client connection factory is used by an outbound channel adapter but a reference to a client connection factory can also be provided to an inbound channel adapter and that adapter will receive any incoming messages received on connections created by the outbound adapter.

A server connection factory is used by an inbound channel adapter (in fact the connection factory will not function without one). A reference to a server connection factory can also be provided to an outbound adapter; that adapter can then be used to send replies to incoming messages to the same connection.

> **Tip**
> Reply messages will only be routed to the connection if the reply contains the header $ip_connection_id that was inserted into the original message by the connection factory.

> **Tip**
> This is the extent of message correlation performed when sharing connection factories between inbound and outbound adapters. Such sharing allows for asynchronous two-way communication over TCP. Only payload information is transferred using TCP; therefore any message correlation must be performed by downstream components such as aggregators or other endpoints.

A maximum of one adapter of each type may be given a reference to a connection factory.

Connection factories using `java.net.Socket` and `java.nio.channel.SocketChannel` are provided.

```
<ip:tcp-connection-factory id="server"
        type="server"
        port="1234"
/>
```

A simple server connection factory that uses `java.net.Socket` connections.

```
<ip:tcp-connection-factory id="server"
        type="server"
        port="1234"
        using-nio="true"
/>
```

A simple server connection factory that uses `java.nio.channel.SocketChannel` connections.

```
<ip:tcp-connection-factory id="client"
        type="client"
        host="localhost"
        port="1234"
        single-use="true"
        so-timeout="10000"
/>
```

A client connection factory that uses `java.net.Socket` connections and creates a new connection for each message.

```
<ip:tcp-connection-factory id="client"
        type="client"
        host="localhost"
        port="1234"
        single-use="true"
        so-timeout="10000"
        using-nio=true
/>
```

A client connection factory that uses `java.nio.channel.Socket` connections and creates a new connection for each message.

TCP is a streaming protocol; this means that some structure has to be provided to data transported over TCP, so the receiver can demarcate the data into discrete messages. Connection factories are configured to use converters to convert between the message payload and the bits that are sent over TCP. This is accomplished by providing an input converter and output converter for inbound and outbound messages respectively. Four standard converters are provided; the first is `ByteArrayCrlfConverter`, which can convert a String or byte array to a stream of bytes followed by carriage return and linefeed characters (\r\n). This is the default converter and can be used with telnet as a client, for example. The second is is `ByteArrayStxEtxConverter`, which can convert a String or byte array to a stream of bytes preceded by an STX (0x02) and followed by an ETX (0x03). The third is `ByteArrayLengthHeaderConverter`, which can convert a String or byte array to a stream of bytes preceded by a 4 byte binary length in network byte order. Each of these converts an input stream containing the corresponding format to a byte array payload. The fourth converter is `JavaSerializationConverter` which can be used to convert any Serializable objects. We expect to provide other serialization technologies but you may also supply your own by implementing the `InputStreamingConverter` and `OutputStreamingConverter` interfaces. If you do not wish to use the default converters, you must supply `input-converter` and `output-converter` attributes on the connection factory (example below).

Tip

This converter mechanism replaces the previous mechanism of subclassing the NxxSocketReader and NxxSocketWriter

```
<bean id="serial"
      class="org.springframework.commons.serializer.JavaSerializationConverter" />

<ip:tcp-connection-factory id="server"
        type="server"
        port="1234"
        input-converter="serial"
        output-converter="serial"
/>
```

A server connection factory that uses `java.net.Socket` connections and uses Java serialization on the wire.

Tip

Normally, with shared connections, one would expect the the same wire protocol to be used for both inbound and outbound messages; however, the configuration allows them to be different. Note, however that if you only specify one converter the other direction will use the default converter.

Connection factories can be configured with a reference to a `TcpConnectionInterceptorFactoryChain`.

Interceptors can be used to add behavior to connections, such as negotiation, security, and other setup. Further documentation to follow.

For a full reference of the attributes available on connection factories, see the reference at the end of this section.

26.4. TCP Adapters

TCP inbound and outbound channel adapters that utilize the above connection factories are provided. These adapters have just 2 attributes `connection-factory` and `channel`. The channel attribute specifies the channel on which messages arrive at an outbound adapter and on which messages are placed by an inbound adapter. The connection factory indicates which connection factory is to be used to manage connections for the adapter. While both inbound and outbound adapters can share a connection factory, server connection factories are always 'owned' by an inbound adapter; client connection factories are always 'owned' by an outbound adapter. One, and only one, adapter of each type may get a reference to a connection factory.

```
<bean id="serializer" class="org.springframework.commons.serializer.JavaSerializationConverter" />

<int-ip:tcp-connection-factory id="server"
        type="server"
        port="1234"
        input-converter="serializer"
        output-converter="serializer"
        using-nio="true"
        single-use="true"
/>

<int-ip:tcp-connection-factory id="client"
        type="client"
        host="localhost"
        port="#{server.port}"
        single-use="true"
        so-timeout="10000"
        input-converter="serializer"
        output-converter="serializer"
/>

<int:channel id="input" />

<int:channel id="replies">
        <int:queue/>
</int:channel>

<int-ip:tcp-outbound-channel-adapter id="outboundClient"
        channel="input"
        connection-factory="client"/>

<int-ip:tcp-inbound-channel-adapter id="inboundClient"
        channel="replies"
        connection-factory="client"/>

<int-ip:tcp-inbound-channel-adapter id="inboundServer"
        channel="loop"
        connection-factory="server"/>

<int-ip:tcp-outbound-channel-adapter id="outboundServer"
        channel="loop"
        connection-factory="server"/>

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

In this configuration, messages arriving in channel 'input' are serialized over connections created by 'client' received at the server and placed on channel 'loop'. Since 'loop' is the input channel for 'outboundServer' the message is simply looped back over the same connection and received by 'inboundClient' and deposited in channel 'replies'.

26.5. TCP Gateways

The inbound TCP gateway `TcpInboundGateway` and oubound TCP gateway `TcpOutboundGateway` use a server and client connection factory respectively. Each connection can process a single request/response at a time.

After constructing a message with the incoming payload and sending it to the requestChannel, it waits for a response and sends the payload from the response message by writing it to the connection.

After sending a message over the connection, the thread waits for a response and constructs a response message Communications over the connections are single-threaded. Users should be aware that only one message can be handled at a time and if another thread attempts to send a message before the current response has been received, it will block until any previous requests are complete (or time out). If, however, the client connection factory is configured for single-use connections each new request gets its own connection and is processed immediately.

```
<ip:tcp-inbound-gateway id="inGateway"
        request-channel="tcpChannel"
        reply-channel="replyChannel"
        connection-factory="cfServer"
        reply-timeout="10000"
/>
```

A simple inbound TCP gateway; if a default connection factory is used, messages will be \r\n delimited data and the gateway can be used by a simple client such as telnet.

```
<ip:tcp-outbound-gateway id="outGateway"
        request-channel="tcpChannel"
        reply-channel="replyChannel"
        connection-factory="cfClient"
        request-timeout="10000"
        reply-timeout="10000"
        />
```

A simple oubound TCP gateway.

26.6. IP Configuration Attributes

Table 26.1. Connection Factory Attributes

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
type	Y	Y	client, server	Determines whether the connection factory is a client or server.
host	Y	N		The host name or ip address of the destination.
port	Y	Y		The port.
using-nio	Y	Y	true, false	Whether or not the tcp adapter is using NIO. Refer to the java.nio package for more information. Default false.
using-direct-buffers	Y	N	true, false	When using NIO, whether or not the tcp

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
				adapter uses direct buffers. Refer to `java.nio.ByteBuffer` documentation for more information. Must be false if using-nio is false.
so-timeout	Y	Y		See `java.net.Socket` setSoTimeout() methods for more information.
so-send-buffer-size	Y	Y		See `java.net.Socket` setSendBufferSize() methods for more information.
so-receive-buffer-size	Y	Y		See `java.net.Socket` setReceiveBufferSize() methods for more information.
so-keep-alive	Y	Y	true, false	See `java.net.Socket. setKeepAlive()`.
so-linger	Y	Y		Sets linger to true with supplied value. See `java.net.Socket. setSoLinger()`.
so-tcp-no-delay	Y	Y	true, false	See `java.net.Socket. setTcpNoDelay()`.
so-traffic-class	Y	Y		See `java.net.Socket. setTrafficClass()`.
local-address	N	Y		On a multi-homed system, specifies an IP address for the interface to which the socket will be bound.
task-executor	Y	Y		Specifies a specific Executor to be used for socket handling. If not supplied, an internal pooled executor will be used. Needed on some platforms that require the use of specific task executors such as a WorkManagerTaskExecutor. See pool-size for thread requirements, depending on other options.
single-use	Y	Y	true, false	Specifies whether a connection can be used for multiple messages. If true, a new connection will be used for each message.
pool-size	Y	Y		Specifies the concurrency. For tcp, not using nio, specifies the number of concurrent connections supported by the adapter. For tcp, using nio, specifies the number of tcp fragments that are concurrently reassembled into complete messages. It only applies in this sense if task-executor is not configured. However, pool-size is also used for the server socket backlog, regardless of whether an external task executor is used. Defaults to 5.
interceptor-factory-chain	Y	Y		Documentation to be supplied.

Table 26.2. UDP Outbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description		
host		The host name or ip address of the destination. For multicast udp adapters, the multicast address.		
port		The port on the destination.		
multicast	true, false	Whether or not the udp adapter uses multicast.		
acknowledge	true, false	Whether or not a udp adapter requires an acknowledgment from the destination. when enabled, requires setting the following 4 attributes.		
ack-host		When acknowledge is true, indicates the host or ip address to which the acknowledgment should be sent. Usually the current host, but may be different, for example when Network Address Transation (NAT) is being used.		
ack-port		When acknowledge is true, indicates the port to which the acknowledgment should be sent. The adapter listens on this port for acknowledgments.		
ack-timeout		When acknowledge is true, indicates the time in milliseconds that the adapter will wait for an acknowlegment. If an acknowlegment is not received in time, the adapter will throw an exception.		
min-acks-for-success		Defaults to 1. For multicast adapters, you can set this to a larger value, requiring acknowlegments from multiple destinations.		
check-length	true, false	Whether or not a udp adapter includes a data length field in the packet sent to the destination.		
time-to-live		For multicast adapters, specifies the time to live attribute for the `MulticastSocket`; controls the scope of the multicasts. Refer to		

Attribute Name	Allowed Values	Attribute Description		
		the Java API documentation for more information.		
so-timeout		See `java.net.DatagramSocket` setSoTimeout() methods for more information.		
so-send-buffer-size		See `java.net.DatagramSocket` setSendBufferSize() methods for more information.		
so-receive-buffer-size		Used for udp acknowlegment packets. See `java.net.DatagramSocket` setReceiveBufferSize() methods for more information.		
local-address	N	Y		On a multi-homed system, for the UDP adapter, specifies an IP address for the interface to which the socket will be bound for reply messages. For a multicast adapter it is also used to determine which interface the multicast packets will be sent over.
task-executor		Specifies a specific Executor to be used for acknowledgment handling. If not supplied, an internal single threaded executor will be used. Needed on some platforms that require the use of specific task executors such as a WorkManagerTaskExecutor. One thread will be dedicated to handling acknowledgments (if the acknowledge option is true).		

Table 26.3. UDP Inbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description		
port		The port on which the adapter listens.		
multicast	true, false	Whether or not the udp adapter uses multicast.		
multicast-address		When multicast is true, the multicast address to which the adapter joins.		
pool-size		Specifies the concurrency. Specifies how many packets can be handled concurrently. It only applies if task-executor is not configured. Defaults to 5.		
task-executor		Specifies a specific Executor to be used for socket handling. If not supplied, an internal pooled executor will be used. Needed on some platforms that require the use of specific task executors such as a WorkManagerTaskExecutor. See pool-size for thread requirements.		
receive-buffer-size		The size of the buffer used to receive DatagramPackets. Usually set to the MTU size. If a smaller buffer is used than the size of the sent packet, truncation can occur. This can be detected by means of the check-length attribute..		
check-length	true, false	Whether or not a udp adapter expects a data length field in the packet received. Used to detect packet truncation.		
so-timeout		See `java.net.DatagramSocket` setSoTimeout() methods for more information.		
so-send-buffer-size		Used for udp acknowlegment packets. See `java.net.DatagramSocket` setSendBufferSize() methods for more information.		
so-receive-buffer-size	Y	Y		See `java.net.DatagramSocket` setReceiveBufferSize() for more information.

Attribute Name	Allowed Values	Attribute Description		
local-address		On a multi-homed system, specifies an IP address for the interface to which the socket will be bound.		

Table 26.4. TCP Inbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
port		The port on which the gateway listens.

Table 26.5. TCP Outbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
host		The host name or ip address of the destination.

Chapter 27. Mail Support

27.1. 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` has various mapping strategies that 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. For simple text-based emails, you can provide a String-based Message payload. In that case, a MailMessage will be created with that String as the text content. If you are working with a Message payload type whose toString() method returns appropriate mail text content, then consider adding Spring Integration's *ObjectToStringTransformer* prior to the outbound Mail adapter (see the example within Section 9.2, “The <transformer> Element” for more detail).

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
```

> **Note**
> `MailHeaders` also allows you to override corresponding `MailMessage` values. For example: If `MailMessage.to` is set to 'foo@bar.com' and `MailHeaders.TO` Message header is provided it will take precedence and override the corresponding value in `MailMessage`

27.2. Mail-Receiving Channel Adapter

Spring Integration also provides support for inbound email with the `MailReceivingMessageSource`. It delegates to a configured instance of Spring Integration's own `MailReceiver` interface, and there are two implementations: `Pop3MailReceiver` and `ImapMailReceiver`. The easiest way to instantiate either of these is by passing the 'uri' for a Mail store to the receiver's constructor. For example:

```
MailReceiver receiver = new Pop3MailReceiver("pop3://usr:pwd@localhost/INBOX");
```

Another option for receiving mail is the IMAP "idle" command (if supported by the mail server you are using). Spring Integration provides the `ImapIdleChannelAdapter` which is itself a Message-producing endpoint. It delegates to an instance of the `ImapMailReceiver` but enables asynchronous reception of Mail Messages. There are examples in the next section of configuring both types of inbound Channel Adapter with Spring

Integration's namespace support in the 'mail' schema.

27.3. 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.

To configure an inbound Channel Adapter, you have the choice between polling or event-driven (assuming your mail server supports IMAP IDLE - if not, then polling is the only option). A polling Channel Adapter simply requires the store URI and the channel to send inbound Messages to. The URI may begin with "pop3" or "imap":

```
<int-mail:inbound-channel-adapter id="imapAdapter"
                        store-uri="imaps://[username]:[password]@imap.gmail.com/INBOX"
                        java-mail-properties="javaMailProperties"
                        channel="recieveChannel"
                        should-delete-messages="true"
                        should-mark-messages-as-read="true"
                        auto-startup="true">
                        <int:poller max-messages-per-poll="1">
                                <int:interval-trigger interval="5000"/>
                        </int:poller>
</int-mail:inbound-channel-adapter>
```

If you do have IMAP idle support, then you may want to configure the "imap-idle-channel-adapter" element instead. Since the "idle" command enables event-driven notifications, no poller is necessary for this adapter. It will send a Message to the specified channel as soon as it receives the notification that new mail is available:

```
<int-mail:imap-idle-channel-adapter id="customAdapter"
                        store-uri="imaps://[username]:[password]@imap.gmail.com/INBOX"
                        channel="recieveChannel"
                        auto-startup="true"
                        should-delete-messages="false"
                        should-mark-messages-as-read="true"
                        java-mail-properties="javaMailProperties"/>
```

Note

When using *should-mark-messages-as-read* be aware of the protocol you are using ti retrieve messages. For example POP3 does not support this flag which means setting it to 'true' will have no effect.

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.

```
<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"/>
```

Chapter 28. JMX Support

Spring Integration provides Channel Adapters for receiving and publishing JMX Notifications. There is also an inbound Channel Adapter for polling JMX MBean attribute values, and an outbound Channel Adapter for invoking JMX MBean operations.

28.1. Notification Listening Channel Adapter

The Notification-listening Channel Adapter requires a JMX ObjectName for the MBean that publishes Notifications to which this listener should be registered. A very simple configuration might look like this:

```
<jmx:notification-listening-channel-adapter id="adapter"
                  channel="channel"
                  object-name="example.domain:name=publisher"/>
```

> **Tip**
> The *notification-listening-channel-adapter* registers with an MBeanServer at startup, and the default bean name is "mbeanServer" which happens to be the same bean name generated when using Spring's <context:mbean-server/> element. If you need to use a different name be sure to include the "mbean-server" attribute.

The adapter can also accept a reference to a NotificationFilter and a "handback" Object to provide some context that is passed back with each Notification. Both of those attributes are optional. Extending the above example to include those attributes as well as an explicit MBeanServer bean name would produce the following:

```
<jmx:notification-listening-channel-adapter id="adapter"
                  channel="channel"
                  mbean-server="someServer"
                  object-name="example.domain:name=somePublisher"
                  notification-fliter="notificationFilter"
                  handback="myHandback"/>
```

Since the notification-listening adapter is registered with the MBeanServer directly, it is event-driven and does not require any poller configuration.

28.2. Notification Publishing Channel Adapter

The Notification-publishing Channel Adapter is relatively simple. It only requires a JMX ObjectName in its configuration as shown below.

```
<context:mbean:export/>

<jmx:notification-publishing-channel-adapter id="adapter"
                             channel="channel"
                             object-name="example.domain:name=publisher"/>
```

It does also require that an MBeanExporter be present in the context. That is why the <context:mbean-export/> element is shown above as well.

When Messages are sent to the channel for this adapter, the Notification is created from the Message content. If the payload is a String it will be passed as the "message" text for the Notification. Any other payload type will be passed as the "userData" of the Notification.

JMX Notifications also have a "type", and it should be a dot-delimited String. There are two ways to provide

the type. Precedence will always be given to a Message header value associated with the JmxHeaders.NOTIFICATION_TYPE key. On the other hand, you can rely on a fallback "default-notification-type" attribute provided in the configuration.

```
<context:mbean:export/>

<jmx:notification-publishing-channel-adapter id="adapter"
                              channel="channel"
                              object-name="example.domain:name=publisher"
                              default-notification-type="some.default.type"/>
```

28.3. Attribute Polling Channel Adapter

The attribute polling adapter is useful when you have a requirement to periodically check on some value that is available through an MBean as a managed attribute. The poller can be configured in the same way as any other polling adapter in Spring Integration (or it's possible to rely on the default poller). The "object-name" and "attribute-name" are required. An MBeanServer reference is also required, but it will automatically check for a bean named "mbeanServer" by default just like the notification-listening-channel-adapter described above.

```
<jmx:attribute-polling-channel-adapter id="adapter"
               channel="channel"
               object-name="example.domain:name=someService"
               attribute-name="InvocationCount">
    <si:poller max-messages-per-poll="1">
        <si:interval-trigger interval="5000"/>
    </si:poller>
</jmx:attribute-polling-channel-adapter>
```

28.4. Operation Invoking Channel Adapter

The *operation-invoking-channel-adapter* enables Message-driven invocation of any managed operation exposed by an MBean. Each invocation requires the operation name to be invoked and the ObjectName of the target MBean. Both of these must be explicitly provided via adapter configuration:

```
<jmx:operation-invoking-channel-adapter id="adapter"
               object-name="example.domain:name=TestBean"
               operation-name="ping"/>
```

Then the adapter only needs to be able to discover the "mbeanServer" bean. If a different bean name is required, then provide the "mbean-server" attribute with a reference.

The payload of the Message will be mapped to the parameters of the operation, if any. A Map-typed payload with String keys is treated as name/value pairs whereas a List or array would be passed as a simple argument list (with no explicit parameter names). If the operation requires a single parameter value, then the payload can represent that single value, and if the operation requires no parameters, then the payload would be ignored.

If you want to expose a channel for a single common operation to be invoked by Messages that need not contain headers, then that option works well.

28.5. Operation Invoking outbound Gateway

Similar to *operation-invoking-channel-adapter* Spring Integration also provides *operation-invoking-outbound-gateway* which could be used when dealing with non-void operations and return

value is required. Such return value will be sent as message payload to the 'reply-channel' specified by this Gateway.

```
<jmx:operation-invoking-outbound-gateway request-channel="requestChannel"
  reply-channel="replyChannel"
  object-name="org.springframework.integration.jmx.config:type=TestBean,name=testBeanGateway"
  operation-name="testWithReturn"/>
```

Another way of provideing the 'reply-channel' is by setting `MessageHeaders.REPLY_CHANNEL` Message Header

28.6. MBean Exporter

Spring Integration components themselves may be exposed as MBeans when the `IntegrationMBeanExporter` is configured. To create an instance of the `IntegrationMBeanExporter`, define a bean and provide a reference to an MBeanServer and a domain name (if desired). The domain can be left out in which case the default domain is "spring.application".

```
<jmx:mbean-exporter domain="my.company.domain" mbean-server="mbeanServer"/>

<bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
    <property name="locateExistingServerIfPossible" value="true"/>
</bean>
```

The MBean exporter is orthogonal to the one provided in Spring core - it registers message channels and message handlers, but not itself (you can expose the exporter itself using the standard `<context:mbean-export/>` tag).

28.7. Control Bus

As described in (EIP), the idea behind the Control Bus is that the same messaging system can be used for monitoring and managing the components within the framework as is used for "application-level" messaging. In Spring Integration we build upon the adapters described above so that it's possible to send Messages as a means of invoking exposed operations. Internally, the Control Bus uses a Spring MBeanExporter instance to expose the various endpoints and channels. To create an instance of the Control Bus, define a bean and provide a reference to an MBeanServer and a domain name.

```
<jmx:control-bus mbean-exporter="mbeanExporter" operation-channel="operationChannel"/>

<jmx:mbean-exporter id="mbeanExporter" mbean-server="mbeanServer"/>
```

The Control Bus has an "operationChannel" that can be accessed for invoking operations on the MBeans that it has exported. This will also be covered by namespace support soon to make it easier to configure references to that channel for other producers. We will likely add some other channels for notifications and attribute polling as well.

The Control Bus functionality is a work in progress. At this time, one can perform some basic monitoring of Message Channels and/or invoke Lifecycle operations (start/stop) on Message Endpoints. Now that the foundation is available, however, we will be able to extend the attributes and operations that are being exposed.

Chapter 29. XMPP Support

Spring Integration provides Channel Adapters for XMPP.

29.1. Introduction

Spring Integration provides adapters for sending and receiving both XMPP messages and status changes from other entries in your roster as well as XMPP.

XMPP describes a way for multiple agents to communicate with each other in a distributed system. The canonical use case is to send and receive instant messages, though XMPP can be, and is, used for far more applications. XMPP is used to describe a network of actors. Within that network, actors may address each other directly, as well as broadcast status changes.

XMPP provides the messaging fabric that underlies some of the biggest Instant Messaging networks in the world, including Google Talk (GTalk) - which is also available from within GMail - and Facebook Chat. There are many good open-source XMPP servers available. Two popular implementations are *Openfire* and *ejabberd* .

In XMPP, *rosters* (the roster corresponds to the notion of a "buddy list" in your typical IM client) are used to manage a list of other agents ("contacts", or "buddies", in an IM client) in the system, called*roster items*. The roster item contains - at a minimum - the roster item's JID which is its unique ID on the network. An actor may subscribe to the state changes of another actor in the system. The subscription can be bidirectional, as well. The subscription settings determine whose status updates are broadcast, and to whom. These subscriptions are stored on the XMPP server, and are thus durable.

29.2. Using The Spring Integration XMPP Namespace

Using the Spring Integration XMPP namespace support is simple. Its use is like any other module in the Spring framework: import the XML schema, and use it to define elements. A prototypical XMPP-based integration might feature the following header. We won't repeat this in subsequent examples, because it is uninteresting.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans
    xmlns="http://www.springframework.org/schema/integration"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:xmpp="http://www.springframework.org/schema/integration/xmpp"
    xmlns:tool="http://www.springframework.org/schema/tool"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="
http://www.springframework.org/schema/integration/xmpp
http://www.springframework.org/schema/integration/xmpp/spring-integration-xmpp.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/integration
http://www.springframework.org/schema/integration/spring-integration.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
">
    ...

</beans:beans>
```

29.3. XMPP Connection

To participate in the network, an actor must connect to an XMPP server. Typically this requires - at a minimum - a `user`, `apassword`, `ahost`, and `aport`. To create an XMPP connection, you may use the XML namespace.

```
<xmpp:xmpp-connection
  id="myConnection"
  user="user"
  password="password"
  host="host"
  port="port"
  resource="theNameOfTheResource"
  subscription-mode="accept_all"
/>
```

29.4. XMPP Messages

29.4.1. Inbound Message Adapter

The Spring Integration adapters support receiving messages from other users in the system. To do this, the adapter "logs in" as a user on your behalf and receives the messages sent to that user. Those messages are then forwarded to your Spring Integration client. The payload of the inbound Spring Integration message may be of the raw type `org.jivesoftware.smack.packet.Message`, or of the type `java.lang.String` - which is the type of the raw `Message`'s `body` property - depending on whether you specify `extract-payload` on the adapter's configuration or not. Inbound Messages are typically small and are text-oriented. Messages received using the adapter have a pretty standard layout, with known headers (all headers have keys defined on `org.springframework.integration.xmpp.XmppHeaders`):

Table 29.1. Header Values

Header Name	What It Describes
XmppHeaders.TYPE	The value of the the `org.jivesoftware.smack.packet.Message.Type` enum that describes the inbound message. Possible values are: `normal`, `chat`, `groupchat`, `headline`, `error`.
XmppHeaders.CHAT	A reference to the `org.jivesoftware.smack.Chat` class which represents the threaded conversation containing the message.

This adapter requires a reference to an XMPP Connection. You may use the xmpp-connection element to define one. An example might look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans ... >

  <context:component-scan
            base-package="com.myxmppclient.inbound"/>

  <context:property-placeholder
            location="#{ systemProperties['user.home'] }/xmpp/xmppclient.properties"/>
```

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

  <xmpp:xmpp-connection
    id="testConnection"
       ...
  />

    <xmpp:message-inbound-channel-adapter
        channel="xmppInbound"
        xmpp-connection="testConnection"/>

  <service-activator input-channel="xmppInbound"
        ref="xmppMessageConsumer"/>

</beans:beans>
```

In this example, the message is received from the XMPP adapter and passed to a `service-activator` component. Here's the declaration of the`service-activator`.

```
package com.myxmppclient.inbound ;

import org.jivesoftware.smack.packet.Message;

import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.stereotype.Component;

@Component
public class XmppMessageConsumer {

   @ServiceActivator
   public void consume(Message input) throws Throwable {
      String text = input.getBody();
      System.out.println( "Received message: " + text  ) ;
   }

}
```

29.4.2. Outbound Message Adapter

You may also send messages to other users on XMPP using the `outbound-message-channel-adapter` adapter. The is configured like the xmpp-message-inbound-channel-adapter. The adapter takes an `xmpp-connection` reference. Here is a (necessarily) contrived example solution using the outbound adapter.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans:beans ... >

    <context:component-scan
        base-package="com.myxmppproducer.outbound"/>

    <context:property-placeholder
        location="#{ systemProperties['user.home'] }/xmpp/xmppclient.properties"/>

    <beans:bean id="xmppProducer"
      class="com.myxmppproducer.outbound.XmppMessageProducer"
      p:recipient="${user.2.login}"/>

    <poller default="true">
        <interval-trigger fixed-rate="true"
           interval="10" time-unit="SECONDS"/>
    </poller>

    <xmpp:xmpp-connection
        id="testConnection"
        ...
    />

    <inbound-channel-adapter ref="xmppProducer"
      channel="outboundChannel"/>
```

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

    <xmpp:message-outbound-channel-adapter
          channel="outboundChannel"
          xmpp-connection="testConnection"/>

</beans:beans>
```

The adapter expects as its input - at a minimum - a payload of type `java.lang.String`, and a header value for `XmppHeaders.CHAT_TO_USER` that specifies to which the user the payload body should be sent to. To create a message destined for the`outbound-message-channel-adapter`, you might use the following Java code:

```
  Message<String> xmppOutboundMsg = MessageBuilder.withPayload("Hello, world!" )
                                    .setHeader(XmppHeaders.CHAT_TO_USER, "userhandle")
                                    .build();
```

It's easy enough to use Java to update the `XmppHeaders.CHAT_TO_USER` header, and this has the advantage of dynamically updating the header at runtime in Java code. If, however, the target is more static in nature, you can configure it using the XMPP enricher support. Here is an example using the enricher. The enricher enriches the Spring Integration message to support the header values that the outbound XMPP adapters expect.

```
<channel id="input"/>
<channel id="output"/>

<xmpp:header-enricher input-channel="input" output-channel="output">
  <xmpp:message-to value="test1@example.org"/>
</xmpp:header-enricher>
```

29.5. XMPP Presence

XMPP also supports broadcasting state. You can use this capability to let people who have you on their roster see your state changes. This happens all the time with your IM clients - you change your away status, and then set an away message, and everybody who has you on their roster sees your icon or username change to reflect this new state, and additionally might see your new "away" message. If you would like to receive notification, or notify others, of state changes, you can use Spring Integration's "presence" adapters.

The most important data for these adapters resides in the headers. The header keys are enumerated on the `org.springframework.integration.xmpp.XmppHeaders` class. The header keys specific to these "presence" adapters start with the token "PRESENCE_". Not all headers are available for both inbound and outbound.

Table 29.2. Header Values

Header Name	What It Describes
XmppHeaders.PRESENCE_LANGUAGE	The `java.lang.String` language in which the message was written.
XmppHeaders.PRESENCE_PRIORITY	The priority (int) of the message. Arbitrary, but it can be used to help assign relevance to a message which in turn might be used in its handling.
XmppHeaders.PRESENCE_MODE	An instance of the enum

Header Name	What It Describes
	`org.jivesoftware.smack.packet.Presence.Mode` that has one of the following values: `chat`, `available`, `away`, `xa`, `dnd`
XmppHeaders.PRESENCE_TYPE	An instance of the enum `org.jivesoftware.smack.packet.Presence.Type` that has one of the following values: `available`, `unavailable`, `subscribe`, `subscribed`, `unsubscribe`, `unsubscribed`, and `error`.
XmppHeaders.PRESENCE_STATUS	A `java.lang.String` string representing the status of the agent. This corresponds to an agents "away" message.
XmppHeaders.PRESENCE_FROM	A `java.lang.String` string representing the handle of the user whose state is being received.

29.5.1. Inbound Presence Adapter

The first adapter supports receiving messages whenever an agent on your roster has updated its state. Most of the important data comes in through the headers.

29.5.2. Outbound Presence Adapter

TBD

Chapter 30. Stream Support

30.1. 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.

30.2. 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>
```

30.3. 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 described in more detail in Chapter 6, *Channel Adapter*.

```
<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>
```

30.4. 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.x
```

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"/>
```

Chapter 31. Spring ApplicationEvent Support

Spring Integration provides support for inbound and outbound `ApplicationEvents` as defined by the underlying Spring Framework. For more information about the events and listeners, refer to the Spring Reference Manual.

31.1. Receiving Spring ApplicationEvents

To receive events and send them to a channel, simply define an instance of Spring Integration's `ApplicationEventListeningChannelAdapter`. This class is 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.

For convenience namespace support was provided to configure `ApplicationEventListeningChannelAdapter` via *inbound-channel-adapter*

```
<int-event:inbound-channel-adapter channel="input" event-types="foo.bar.FooApplicationEvent, foo.bar.BarApplicat

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

In the above sample, all Application Context events that are of type specified by the 'event-types' (optional) attribute will be delivered as Spring Integration Messages to 'sampleEventChannel'.

31.2. Sending Spring ApplicationEvents

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`.

For convenience namespace support was provided to configure `ApplicationEventPublishingMessageHandler` via *outbound-channel-adapter* element

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

<int-event:outbound-channel-adapter channel="input"/>
```

If you are using PollableChannel (e.g., Queue), you can also provide *poller* as sub-element of *outbound-channel-adapter*, optionally providing *task-executor*

```
<int:channel id="input">
        <int:queue/>
</int:channel>

<int-event:outbound-channel-adapter channel="input">
        <int:poller max-messages-per-poll="1" task-executor="executor">
                <int:interval-trigger interval="100" time-unit="MILLISECONDS"/>
        </int:poller>
</int-event:outbound-channel-adapter>

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

In the above sample, all messages sent to an 'input' channel will be published as ApplicationEvents to Spring Application sContext

Chapter 32. Dealing with XML Payloads

32.1. 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.

32.2. Transforming xml payloads

This section will explain the workings of `UnmarshallingTransformer`, `MarshallingTransformer`, `XsltPayloadTransformer` and how to configure them as *beans*. All of the provided xml transformers extend `AbstractTransformer` or `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 `MessageTransformingHandler`. 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 elements in XML.

`UnmarshallingTransformer` 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.UnmarshallingTransformer">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb1Marshaller">
                    <property name="contextPath" value="org.example" />
        </bean>
    </constructor-arg>
</bean>
```

The `MarshallingTransformer` allows an object graph to be converted into xml using a Spring OXM `Marshaller`. By default the `MarshallingTransformer` 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.MarshallingTransformer">
    <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>
```

By default, the `MarshallingTransformer` will pass the payload Object to the `Marshaller`, but if its boolean "extractPayload" property is set to "false", the entire `Message` instance will be passed to the `Marshaller` instead. That may be useful for certain custom implementations of the `Marshaller` interface, but typically the payload is the appropriate source Object for marshalling when delegating to any of the various out-of-the-box `Marshaller` implementations.

`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>
```

32.3. 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 `EventDrivenConsumer` or `PollingConsumer` 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 `UnmarshallingTransformer` is shown below. Since the namespace is now creating an endpoint instance 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 attribute. 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"/>
```

Very often to assist with transformation you may need to have access to Message data (e.g., Message Headers). For example; you may need to get access to certain Message Headers and pass them on as parameters to a transformer (e.g., transformer.setParameter(..)). Spring Integration provides two convenient ways to accomplish this. Just look at the following XML snippet.

```
<si-xml:xslt-transformer id="paramHeadersCombo"
            input-channel="paramHeadersComboChannel"
                output-channel="output"
                xsl-resource="classpath:transformer.xslt"
                xslt-param-headers="testP*, *foo, bar, baz">

    <int-xml:xslt-param name="helloParameter" value="hello"/>
    <int-xml:xslt-param name="firstName" expression="headers.fname"/>
</int-xml:xslt-transformer>
```

If message header names match 1:1 to parameter names, you can simply use *xslt-param-headers attribute*. There you can also use wildcards for simple pattern matching which supports the following simple pattern styles: "xxx*", "*xxx", "*xxx*" and "xxx*yyy".

You can also configure individual xslt parameters via *xslt-param* sub element. There you can

or `value` attribute. The `expression` attribute should be any valid SpEL expression with Message being the root object of the expression evaluation context. The `value` attribute just like any `value` in Spring beans allows you to specify simple scalar vallue. YOu can also use property placeholders (e.g., ${some.value}) So as you can see, with the `expression` and `value` attribute xslt parameters could now be mapped to any accessible part of the Message as well as any literal value.

32.4. 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 `MessageHandler` and should therefore be configured in conjunction with an appropriate endpoint (see the namespace support below for a simpler configuration alternative).

```
<bean id="splittingEndpoint"
      class="org.springframework.integration.endpoint.EventDrivenConsumer">
    <constructor-arg ref="orderChannel" />
    <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>
</bean>
```

32.5. 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 `ChannelResolver` to convert the result(s) of the XPath expression to a channel name. By default a `BeanFactoryChannelResolver` 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 `ChannelResolver` can be used. Where there is a simple mapping from Xpath result to channel name the provided `MapBasedChannelResolver` can be used.

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

```
    <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>
</bean>
```

32.6. 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>
```

32.7. XPath components namespace support

All XPath based components have namespace support allowing them to be configured as Message Endpoints 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 Message Endpoint 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-splitter>

<!-- 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-splitter>
```

XPath router namespace support allows for the creation of a Message Endpoint 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-router>

<!-- 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-router>
```

Chapter 33. Security in Spring Integration

33.1. 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.

33.2. 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-1

<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>
```

Chapter 34. Groovy support

With Spring Integration 2.0 we've added Groovy support allowing you to use Groovy scripting language to provide integration and business logic for various integration components similar to the way Spring Expression Language (SpEL) is use to implement routing, transformation and other integration concerns. For more information about Groovy please refer to Groovy documentation which you can find here: http://groovy.codehaus.org/

34.1. Groovy configuration

Depending on the complexity of your integration requirements Groovy scripts could be provided inline as CDATA in XML configuration or as a reference to a file containing Groovy script. To enable Groovy support Spring Integration defines `GroovyScriptExecutingMessageProcessor` which will create a groovy Binding object identifying Message Payload as `payload` variable and Message Headers as `headers` variable. All that is left for you to do is write script that uses these variables. Below are couple of sample configurations:

Filter

```
<filter input-channel="referencedScriptInput">
   <groovy:script location="some/path/to/groovy/file/GroovyFilterTests.groovy"/>
</filter>

<filter input-channel="inlineScriptInput">
           <groovy:script><![CDATA[
     return payload == 'good'
   ]]></groovy:script>
</filter>
```

You see that script could be included inline or via `location` attribute using the groovy namespace sport.

Other supported elements are *router*, *service-activator*, *transformer*, *splitter*

Another interesting aspect of using Groovy support is framework's ability to update (reload) scripts without restarting the Application Context. To accomplish this all you need is specify `refresh-check-delay` attribute on *script* element. The reason for this attribute is to make reloading of the script more efficient.

```
<groovy:script location="..." refresh-check-delay="5000"/>
```

In the above example for the next 5 seconds after you update the script you'll still be using the old script and after 5 seconds the context will be updated with the new script. This is a good example where 'near real time' is acceptable.

```
<groovy:script location="..." refresh-check-delay="0"/>
```

In the above example the context will be updated with the new script every time the script is modified. Basically this is the example of the 'real-time' and might not be the most efficient way.

```
<groovy:script location="..." refresh-check-delay="-1"/>
```

Any negative number value means the script will never be refreshed after initial initialization of application context. DEFAULT BEHAVIOR

> **Important**
> Inline defined script can not be reloaded.

Appendix A. Spring Integration Samples

Note

Starting with the current release of Spring Integration the *samples* are distributed as independent Maven-based projects (http://maven.apache.org/) to minimize the setup time. Since each project is also an Eclipse-based project, they can be imported as is using the Eclipse Import wizard. If you prefer another IDE, configuration should be very trivial, since a special Maven profile was setup to download all of the required dependencies for all samples. Detailed instructions on how to build and run the samples are provided in the `README.txt` file located in the *samples* directory of the main distribution.

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). 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">

    <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"/>

    <poller id="poller" default="true">
        <interval-trigger interval="1000"/>
    </poller>

</beans:beans>
```

As you can see, each Message Endpoint is connected to input and/or output channels. Each endpoint will manage its own Lifecycle (by default endpoints start automatically upon initialization - to prevent that add the "auto-startup" attribute with a value of "false"). Most importantly, 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);
    }
}
```

Tip

To run this sample as well as 8 others, refer to the `README.txt` within the "samples" directory of the main distribution as described at the beginning of this chapter.

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>

<task:executor id="pool" pool-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 task-scheduler (the caller) to invoke the operation.

> **Note**
> In addition to experimenting with the poller's concurrency settings, you can also add the 'transactional' sub-element and then refer to any PlatformTransactionManager instance within the context.

A.2. The XML Messaging Sample

The xml messaging sample in the org.springframework.integration.samples.xml illustrates how to use some of the provided components which deal with xml payloads. The sample uses the idea of processing an order for books represented as xml.

First the order is split into a number of messages, each one representing a single order item using the XPath splitter component.

```
<si-xml:xpath-splitter id="orderItemSplitter" input-channel="ordersChannel"
            output-channel="stockCheckerChannel" create-documents="true">
    <si-xml:xpath-expression expression="/orderNs:order/orderNs:orderItem" namespace-map="orderNamespaceMap" />
</si-xml:xpath-splitter>
```

A service activator is then used to pass the message into a stock checker POJO. The order item document is enriched with information from the stock checker about order item stock level. This enriched order item message is then used to route the message. In the case where the order item is in stock the message is routed to the warehouse. The XPath router makes use of a `MapBasedChannelResolver` which maps the XPath evaluation result to a channel reference.

```
<si-xml:xpath-router id="instockRouter" channel-resolver="mapChannelResolver"
        input-channel="orderRoutingChannel" resolution-required="true">
    <si-xml:xpath-expression expression="/orderNs:orderItem/@in-stock" namespace-map="orderNamespaceMap"  />
</si-xml:xpath-router>

<bean id="mapChannelResolver"
    class="org.springframework.integration.channel.MapBasedChannelResolver">
    <property name="channelMap">
        <map>
```

```
            <entry key="true" value-ref="warehouseDispatchChannel" />
            <entry key="false" value-ref="outOfStockChannel" />
        </map>
    </property>
</bean>
```

Where the order item is not in stock the message is transformed using xslt into a format suitable for sending to the supplier.

```
<si-xml:xslt-transformer input-channel="outOfStockChannel" output-channel="resupplyOrderChannel"
     xsl-resource="classpath:org/springframework/integration/samples/xml/bigBooksSupplierTransformer.xsl"/>
```

A.3. The OSGi Samples

This release of Spring Integration includes several samples that are OSGi enabled as well as samples that were specifically designed to show some of the other benefits of OSGi and Spring Integration when used together. First lets look at the two familiar examples that are also configured to be valid OSGi bundles. These are *Hello World* and *Cafe*. All you need to do to see these samples work in an OSGi environment is deploy the generated JAR into such an environment.

Use Maven to generate the JAR by executing the 'mvn install' command on either of these projects. This will generate the JAR file in the target directory. Now you can simply drop that JAR file into the deployment directory of your OSGi platform. For example, if you are using SpringSource dm Server, drop the files into the 'pickup' directory within the dm Server home directory.

> **Note**
> Prior to deploying and testing Spring Integration samples in the dm Server or any other OSGi server platform, you must have the Spring Integration and Spring bundles installed on that platform. For example, to install Spring Integration into SpringSource dm Server, copy all JAR files that are located in the 'dist' directory of your Spring Integration distribution into the 'repository/bundles/usr' directory of your dm Server instance (see the dm Server User Guide for more detail on how to install bundles).

The Spring Integration samples require a few other bundles to be installed. For the 1.0.3 release, the full list including transitive dependencies is:

- org.apache.commons.codec-1.3.0.jar
- org.apache.commons.collections-3.2.0.jar
- org.apache.commons.httpclient-3.1.0.jar
- org.apache.ws.commons.schema-1.3.2.jar
- org.springframework.oxm-1.5.5.A.jar
- org.springframework.security-2.0.4.A.jar
- org.springframework.ws-1.5.5.A.jar
- org.springframework.xml-1.5.5.A.jar

These are all located within the 'lib' directory of the Spring Integration distribution. So, you can simply copy

those JARs into the dm Server 'repository/bundles/usr' directory as well.

Note

The Spring Framework bundles (aop, beans, context, etc.) are also included in the 'lib' directory of the Spring Integration distribution, but they do not need to be installed since they are already part of the dm Server infrastructure. Also, note that the versions listed above are those included with the Spring Integration 1.0.3 release. Newer versions of individual JARs may be used as long as they are within the range specified in the MANIFEST.MF files of those bundles that depend upon them.

Tip

The bundles listed above are appropriate for a SpringSource dm Server 1.0.x deployment environment with a Spring Framework 2.5.x foundation. That is the version against which Spring Integration 1.0.3 has been developed and tested. However, as of the time of the Spring Integration 1.0.3 release, the Spring Framework 3.0 release candidates are about to be available, and the dm Server 2.0.x milestones are available. If you want to try running these samples in that environment, then you will need to replace the Spring Security and Spring Web Services bundles with versions that support Spring 3.0. The OXM functionality is moving into the Spring Framework itself for the 3.0 release. Otherwise, Spring Integration 1.0.3 has been superficially tested against the Spring 3.0 snapshots available at the time of its release. In fact, some internal changes were made in the 1.0.3 release specifically to support Spring 3.0 (whereas 1.0.2 does not). Spring Integration 2.0 will be built upon a Spring 3.0 foundation.

To demonstrate some of the benefits of running Spring Integration projects in an OSGi environment (e.g. modularity, OSGi service dynamics, etc.), we have included a couple new samples that are dedicated to highlighting those benefits. In the 'samples' directory, you will find the following two projects:

- osgi-inbound (producer bundle)
- osgi-outbound (consumer bundle)

Unlike the other samples in the distribution, these are not Maven enabled. Instead, we have simply configured them as valid dm Server Bundle projects. That means you can import these projects directly into an STS workspace using the "Existing Projects into Workspace" option from the Eclipse Import wizard. Then, you can take advantage of the STS dm Server tools to deploy them into a SpringSource dm Server instance.

Note

A simple Ant 'build.xml' file has been included within each of these projects as well. The build files contain a single 'jar' target. Therefore, after these projects have been built within Eclipse/STS, you can generate the bundle (JAR) directly and deploy it manually.

The structure of these projects is very simple, yet the concepts they showcase are quite powerful. The 'osgi-inbound' module enables sending a Message to a Publish-Subscribe Channel using a Spring Integration Gateway proxy. The interesting part, however, is that the Publish-Subscribe Channel is exported as an OSGi service via the <osgi:service/> element. As a result, any other bundles can be developed, deployed, and maintained independently yet still subscribe to that channel.

The 'osgi-outbound' module is an example of such a subscribing consumer bundle. It uses the corresponding <osgi:reference/> element to locate the channel exported by the 'osgi-inbound' bundle. It also contains configuration for a <file:outbound-gateway/> which is a subscriber to that channel and will write the Message content to a file once it arrives. It then sends a response Message with the name of the file and its location.

To make it easy to run, we've exposed a command-line interface where you can type in the command, the

message, and the file name to execute the demo. This is exposed through the OSGi console. Likewise, the response that provides the name and location of the resulting file will also be visible within the OSGi console.

To run these samples, make sure your OSGi environment is properly configured to host Spring Integration bundles (as described in the note above). Deploy the producer bundle (osgi-inbound) first, and then deploy the consumer bundle (osgi-outbound). After you have deployed these bundles, open the OSGi console and type the following command:

```
osgi> help
```

You will see the following amidst the output:

```
---Spring Integration CLI-based OSGi Demo---
        siSend <message> <filename> - send text to be written to a file
```

As you can see, that describes the command that you will be able to use to send messages. If you are interested in how it is implemented or want to customize message sending logic or even create a new command look at `InboundDemoBundleActivator.java` in the consumer bundle.

Tip

When using the SpringSource Tool Suite, you can open the OSGi console by first opening the dm Server view and then choosing the 'Server Console' tab at the bottom (to open the dm Server view, navigate to the dm Server instance listed in the 'Servers' view and either double-click or hit F3). Alternatively, you can open the OSGi console by connecting to port 2401 via telnet (as long as that is enabled, and for dm Server, it is enabled by default):

```
telnet localhost 2401
```

Now send a message by typing:

```
osgi> siSend "Hello World" hello.txt
```

You will see something similar to the following in the OSGi console:

```
Sending message: 'Hello World'
Message sent and its contents were written to:
/usr/local/dm-server/work/tmp/spring-integration-samples/output/hello.txt
```

Note

It is not necessary to wrap the message in quotes if it does not contain spaces. Go ahead and open up the file and verify that the message content was written to it.

Let's assume you wanted to change the directory to which the files are written or make any other change to the consumer bundle (osgi-outboud). You only need to update the consumer bundle and not the producer bundle. So, go ahead and change the directory in the 'osgi-outbound.xml' file located within 'src/META-INF/spring' and refresh the consumer bundle.

Tip

If using STS to deploy to dm Server, the refresh will happen automatically. If replacing bundles manually, you can issue the command 'refresh n' in the OSGi console (where n would be the ID of the bundle as displayed at any point after issuing the 'ss' command to see the short status output).

You will see that the change takes affect immediately. Not only that, you could even start developing and

deploying new bundles that subscribe to the messages produced by the producer bundle the same way as the existing consumer bundle (osgi-outbound) does. With a publish-subscribe-channel any newly deployed bundles would start receiving each Message as well.

> **Note**
>
> If you also want to modify and refresh the producer bundle, be sure to refresh the consumer bundle afterwards as well. This is necessary because the consumer's subscription must be explicitly re-enabled after the producer's channel disappears. You could alternatively deploy a relatively static bundle that defines channels so that producers and consumers can be completely dynamic without affecting each other at all. In Spring Integration 2.0, we plan to support automatic re-subscription and more through the use of a *Control Bus*.

That pretty much wraps up this very simple example. Hopefully it has successfully demonstrated the benefits of modularity and OSGi service dynamics while working with Spring Integration. Feel free to experiment by following some of the suggestions mentioned above. For deeper coverage of the applicability of OSGi when used with Spring Integration, read this blog by Spring Integration team member Iwein Fuld.

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 Task Scheduler

In Spring Integration, the ApplicationContext plays the central role of a Message Bus, and there are only a couple configuration options to be aware of. First, you may want to control the central TaskScheduler instance. You can do so by providing a single bean with the name "taskScheduler". This is also defined as a constant:

```
IntegrationContextUtils.TASK_SCHEDULER_BEAN_NAME
```

By default Spring Integration uses the `SimpleTaskScheduler` implementation. That in turn just delegates to any instance of Spring's `TaskExecutor` abstraction. Therefore, it's rather trivial to supply your own configuration. The "taskScheduler" bean is then responsible for managing all pollers. The TaskScheduler will startup automatically by default. If you provide your own instance of SimpleTaskScheduler however, you can set the 'autoStartup' property to *false* instead.

When Polling Consumers provide an explicit task-executor reference in their configuration, the invocation of the handler 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, it will be invoked by one of the main scheduler's threads.

> **Note**
>
> An endpoint is a *Polling Consumer* if its input channel is one of the queue-based (i.e. pollable) channels. On the other hand, *Event Driven Consumers* are those whose input channels have dispatchers instead of queues (i.e. they are subscribable). Such endpoints have no poller configuration since their handlers will be invoked directly.

The next section will describe what happens if Exceptions occur within the asynchronous invocations.

B.4. Error Handling

As described in the overview at the very beginning of this manual, one of the main motivations behind a Message-oriented framework like Spring Integration is to promote loose-coupling between components. The Message Channel plays an important role in that producers and consumers do not have to know about each other. However, the advantages also have some drawbacks. Some things become more complicated in a very loosely coupled environment, and one example is error handling.

When sending a Message to a channel, the component that ultimately handles that Message may or may not be operating within the same thread as the sender. If using a simple default DirectChannel (with the <channel> element that has no <queue> sub-element and no 'task-executor' attribute), the Message-handling will occur in the same thread as the Message-sending. In that case, if an Exception is thrown, it can be caught by the sender (or it may propagate past the sender if it is an uncaught RuntimeException). So far, everything is fine. This is the same behavior as an Exception-throwing operation in a normal call stack. However, when adding the asynchronous aspect, things become much more complicated. For instance, if the 'channel' element *does* provide a 'queue' sub-element, then the component that handles the Message *will* be operating in a different thread than the sender. The sender may have dropped the Message into the channel and moved on to other things. There is no way for the Exception to be thrown directly back to that sender using standard Exception throwing techniques. Instead, to handle errors for asynchronous processes requires an asynchronous error-handling mechanism as well.

Spring Integration supports error handling for its components by publishing errors to a Message Channel. Specifically, the Exception will become the payload of a Spring Integration Message. That Message will then be sent to a Message Channel that is resolved in a way that is similar to the 'replyChannel' resolution. First, if the request Message being handled at the time the Exception occurred contains an 'errorChannel' header (the header name is defined in the constant: MessageHeaders.ERROR_CHANNEL), the ErrorMessage will be sent to that channel. Otherwise, the error handler will send to a "global" channel whose bean name is "errorChannel" (this is also defined as a constant: IntegrationContextUtils.ERROR_CHANNEL_BEAN_NAME).

Whenever relying on Spring Integration's XML namespace support, a default "errorChannel" bean will be created behind the scenes. However, you can just as easily define your own if you want to control the settings.

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

> **Note**
> The default "errorChannel" is a PublishSubscribeChannel.

The most important thing to understand here is that the messaging-based error handling will only apply to Exceptions that are thrown by a Spring Integration task that is executing within a TaskExecutor. This does *not* apply to Exceptions thrown by a handler that is operating within the same thread as the sender (e.g. through a DirectChannel as described above).

> **Note**
> When Exceptions occur in a scheduled poller task's execution, those exceptions will be wrapped in `ErrorMessages` and sent to the 'errorChannel' as well.

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.

B.5. 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 using XML configuration in combination with annotations. 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 that case, the annotation prevents ambiguity even when no "method" attribute exists on the <service-activator/> element.

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 {

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

}
```

When the method parameter should be mapped from a value in the `MessageHeaders`, 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) {
        ...
    }

}
```

A more powerful and flexible way to map Messages to method arguments is to use `@MessageMapping` annotation which allows you to define expression via Spring 3.0 Expression Language support to help parse the message payload and/or header and map the parsed values to method arguments.

For example:

```
public void fromMessageToMethod(@MessageMapping("headers.day") String argA,
                @MessageMapping("#this") Message message,
                @MessageMapping("payload") Employee payloadArg,
                @MessageMapping("payload.fname") String value,
                @MessageMapping("headers") Map headers) { ... }
```

As you can see, the above method takes 5 arguments where:

- First - will be mapped to the value of 'day' header
- Second - will be mapped to the Message itself
- Third - will be mapped to the Payload
- Fourth - will be mapped to the 'fname' property of a Payload object
- Fifth - will be mapped to MessageHeaders

Tip

A Map-typed argument does not strictly require the use of the @Headers annotation. In other words the following is also valid:

```
public void bar(String payload, Map<String, Object> headerMap)
```

However this can lead to unresolvable ambiguities if the payload is itself a Map. For that reason, we highly recommend using the annotation whenever expecting the headers. For a much more detailed description, see the javadoc for `MethodParameterMessageMapper`.

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 such an endpoint's output channel will be used if available, and the REPLY_CHANNEL message header value will be used as a fallback.

Tip

The combination of output channels on endpoints and the reply channel message header enables a pipeline approach where multiple components have an output channel, and the final component simply allows the reply message to be forwarded to the reply channel as specified in the original

request message. In other words, the final component depends on the information provided by the original sender and can dynamically support any number of clients as a result. This is an example of Return Address.

In addition to the examples shown here, these annotations also support inputChannel and outputChannel properties.

```
public class FooService {

    @ServiceActivator(inputChannel="input", outputChannel="output")
    public void bar(String payload, @Headers Map<String, Object> headerMap) {
        ...
    }

}
```

That provides a pure annotation-driven alternative to the XML configuration. However, it is generally recommended to use XML for the endpoints, since it is easier to keep track of the overall configuration in a single, external location (and besides the namespace-based XML configuration is not very verbose). If you do prefer to provide channels with the annotations however, you just need to enable a SI Annotations BeanPostProcessor. The following element should be added:

```
<int:annotation-config/>
```

Note

When configuring the "inputChannel" and "outputChannel" with annotations, the "inputChannel" *must* be a reference to a `SubscribableChannel` instance. Otherwise, it would be necessary to also provide the full poller configuration via annotations, and those settings (e.g. the trigger for scheduling the poller) should be externalized rather than hard-coded within an annotation. If the input channel that you want to receive Messages from is indeed a `PollableChannel` instance, one option to consider is the Messaging Bridge. Spring Integration's "bridge" element can be used to connect a PollableChannel directly to a SubscribableChannel. Then, the polling metadata is externally configured, but the annotation option is still available. For more detail see Chapter 15, *Messaging Bridge*.

B.6. Message Mapping rules and conventions

Spring Integration implements a flexible facility to map Messages to Methods and their arguments without providing extra configuration by relying on some default rules as well as defining certain conventions.

B.6.1. Simple Scenarios

Single un-annotated parameter (object or primitive) which is not a Map/Properties with non-void return type;

```
public String foo(Object o);
```

Details:

Input parameter is Message Payload. If parameter type is not compatible with Message Payload an attempt will be made to convert it using Conversion Service provided by Spring 3.0. The return value will be incorporated as a Payload of the returned Message

Single un-annotated parameter (object or primitive) which is not a Map/Properties with Message return type;

```
public Message  foo(Object o);
```

Details:

Input parameter is Message Payload. If parameter type is not compatible with Message Payload an attempt will be made to convert it using Conversion Service provided by Spring 3.0. The return value is a newly constructed Message that will be sent to the next destination.

Single parameter which is a Message or its subclass with arbitrary object/primitive return type;

```
public int foo(Message  msg);
```

Details:

Input parameter is Message itself. The return value will become a payload of the Message that will be sent to the next destination.

Single parameter which is a Message or its subclass with Message or its subclass as a return type;

```
public Message foo(Message msg);
```

Details:

Input parameter is Message itself. The return value is a newly constructed Message that will be sent to the next destination.

Single parameter which is of type Map or Properties with Message as a return type;

```
public Message foo(Map m);
```

Details:

This one is a bit interesting. Although at first it might seem like an easy mapping straight to Message Headers, the preference is always given to a Message Payload. This means that if Message Payload is of type Map, this input argument will represent Message Payload. However if Message Payload is not of type Map, then no conversion via Conversion Service will be attempted and the input argument will be mapped to Message Headers.

Two parameters where one of them is arbitrary non-Map/Properties type object/primitive and another is Map/Properties type object (regardless of the return)

```
public Message foo(Map h, <T> t);
```

Details:

This combination contains two input parameters where one of them is of type Map. Naturally the non-Map parameters (regardless of the order) will be mapped to a Message Payload and the Map/Properties (regardless of the order) will be mapped to Message Headers giving you a nice POJO way of interacting with Message structure.

No parameters (regardless of the return)

```
public String foo();
```

Details:

This Message Handler method will be invoked based on the Message sent to the input channel this handler is hooked up to, however no Message data will be mapped, thus making Message act as event/trigger to invoke such handlerThe output will be mapped according to the rules above

No parameters, void return

```
public void foo();
```

Details:

Same as above, but no output

Annotation based mappings

Annotation based mapping is the safest and least ambiguous approach to map Messages to Methods. There wil be many pointers to annotation based mapping throughout this manual, however here are couple of examples:

```
public String foo(@Payload String s,  @Header("foo") String b)
```

Very simple and explicite way of mapping Messages to method. As you'll see later on without annotation this signature would result in the ambiguous condition, however by explicitly mapping first argument to a Message Payload and second argument to a value of the 'foo' Message Header we have avoided ambiguity.

```
public String foo(@Payload String s,  @RequestParam("foo") String b)
```

Looks almost identical to the previous example, however @RequestMapping or any other non-SI mapping annotation is irrelevant and therefore will be ignored leaving the second parameter unmapped. And although the second parameters could easily be mapped to a Payload, there can only be one Payload, therefore this method becomes ambiguous.

```
public String foo(String s,  @Header("foo") String b)
```

The same as above. The only difference is that the first argument will be mapped to Message Payload implicitly.

```
public String foo(@Headers Map m,  @Header("foo")Map f, @Header("bar") String bar)
```

Yet another signature that would definitely be treated as ambiguous because it has more then 2 arguments, plus two of them are Maps, however with annotation-based mapping ambiguity is easily avoided. In this example the first argument is mapped to all the Message Headers, while second and third argument map to the values of Message Headers 'foo' and 'bar'.

B.6.2. Complex Scenarios

Multiple parameters:

Multiple parameters could create a lot of ambiguity with regards to determining the appropriate mappings. The general advice is to annotate your method parameters with @Payload and/or @Header/@Headers Below are

some of the examples of ambiguous conditions which result in exception being raised.

```
public String foo(String s, int i)
```

- the two parameters are equal in weight, therefore no way to determine which one is a payload and what to do with another.

```
public String foo(String s, Map m, String b)
```

- almost the same as above. Although Map could be easily mapped to Message Headers, there is no way to determine what to do with two Strings.

```
public String foo(Map m, Map f)
```

- although one might argue that one Map could be mapped to Message Payload and another one to Message Headers, it would be unreasonable to rely on the order (e.g., first is Payload, second Headers)

> **Tip**
> Basically any method signature with more then one method argument which is not (Map, <T>) and those parameters are not annotated will result in the ambiguous condition thus triggering an exception.

Multiple methods:

Message Handlers with multiple methods are mapped based on the same rules that are described above, however some scenarios might still look confusing.

Multiple methods (same or different name) with legal (mappable) signatures:

```
public class Foo{
        public String foo(String str, Map m);

        public String foo(Map m)
}
```

As you can see, the Message could be mapped to either method. The first method would be invoked where Message Payload could be mapped to 'str' and Message Headers could be mapped to 'm'. The second method could easily also be a candidate where only Message Headers are mapped to 'm'. To make meters worse both methods have the same name which at first might look very ambiguous considering the following configuration:

```
<si:service-activator input-channel="input" output-channel="output" method="foo">
        <bean class="org.bar.Foo"/>
</si:service-activator>
```

At this point it would be important to understand Spring Integration mapping Conventions where at the very core, mappings are based on Payload first and everything else next. In other words the method whose argument could be mapped to a Payload will take precedence over all other methods.

On the other hand let's look at slightly different example:

```
public class Foo{
        public String foo(String str, Map m);
```

```
        public String foo(String str)
}
```

If you look at it you can probably see a truly an ambiguous condition. In this example since both methods have signatures that could be mapped to a Message Payload. They also have the same name. Such handler will trigger an exception. However if method names were different you could influence the mapping with 'method' attribute (see below):

```
public class Foo{
        public String foo(String str, Map m);

        public String bar(String str)
}
```

```
<si:service-activator input-channel="input" output-channel="output" method="bar">
        <bean class="org.bar.Foo"/>
</si:service-activator>
```

Now there is no ambiguity since the configuration explicitly maps to 'bar' method which has no name conflicts.

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.springsource.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.