Spring Integration Reference Manual

Mark Fisher
Marius Bogoevici
Iwein Fuld
Jonas Partner
Oleg Zhurakousky
Gary Russell
Dave Syer
Josh Long
David Turanski

Spring Integration Reference Manual

by Mark Fisher, Marius Bogoevici, Iwein Fuld, Jonas Partner, Oleg Zhurakousky, Gary Russell, Dave Syer, Josh Long, and David Turanski

2.0.2.RELEASE

Table of Contents

Part I. What's new in Spring Integration 2.0

For those who are already familiar with Spring Integration, this chapter provides a brief overview of the new features of version 2.0.

1. What's new in Spring Integration 2.0?

1.1 Spring 3 support

Spring Integration 2.0 is built on top of Spring 3.0.5 and makes many of its features available to our users.

Support for the Spring Expression Language (SpEL)

You can now use SpEL expressions within the *transformer, router, filter, splitter, aggregator, service-activator, header-enricher*, and many more elements of the Spring Integration core namespace as well as various adapters. There are many samples provided throughout this manual.

ConversionService and Converter

You can now benefit from *Conversion Service* support provided with Spring while configuring many Spring Integration components such as Datatype Channel [http://www.eaipatterns.com/DatatypeChannel.html]. See the section called “Message Channel Implementations” as well the section called “Introduction”. Also, the SpEL support mentioned in the previous point also relies upon the ConversionService. Therefore, you can register Converters once, and take advantage of them anywhere you are using SpEL expressions.

TaskScheduler and Trigger

Spring 3.0 defines two new strategies related to scheduling: *TaskScheduler and Trigger* Spring Integration (which uses a lot of scheduling) now builds upon these. In fact, Spring Integration 1.0 had originally defined some of the components (e.g. CronTrigger) that have now been migrated into Spring 3.0's core API. Now, you can benefit from reusing the same components within the entire Application Context (not just Spring Integration configuration). Configuration of Spring Integration Pollers has been greatly simplified as well by providing attributes for directly configuring rates, delays, cron expressions, and trigger references. See Section 3.2, “Channel Adapter” for sample configurations.

RestTemplate and HttpMessageConverter

Our outbound HTTP adapters now delegate to Spring's RestTemplate for executing the HTTP request and handling its response. This also means that you can reuse any custom HttpMessageConverter implementations. See Section 13.3, “Http Outbound Gateway” for more details.

1.2 Enterprise Integration Pattern Additions

Also in 2.0 we have added support for even more of the patterns described in Hohpe and Woolf's Enterprise Integration Patterns [http://www.eaipatterns.com/] book.

Message History

We now provide support for the Message History [http://www.eaipatterns.com/MessageHistory.html] pattern allowing you to keep track of all traversed components, including the name of each channel and endpoint as well as the timestamp of that traversal. See Section 8.2, “Message History” for more details.

Message Store

We now provide support for the Message Store [http://www.eaipatterns.com/MessageStore.html] pattern. The Message Store provides a strategy for persisting messages on behalf of any process whose scope extends beyond a single transaction, such as the Aggregator and Resequencer. Many sections of this document provide samples on how to use a Message Store as it affects several areas of Spring Integration. See Section 6.3, “Claim Check”, Section 3.1, “Message Channels”, Section 5.4, “Aggregator”, Chapter 17, *JDBC Support*, and Section 5.5, “Resequencer” for more details

Claim Check

We have added an implementation of the Claim Check [http://www.eaipatterns.com/StoreInLibrary.html] pattern. The idea behind the Claim Check pattern is that you can exchange a Message payload for a "claim ticket" and vice-versa. This allows you to reduce bandwidth and/or avoid potential security issues when sending Messages across channels. See Section 6.3, “Claim Check” for more details.

Control Bus

We have provided implementations of the Control Bus [http://www.eaipatterns.com/ControlBus.html] pattern which allows you to use messaging to manage and monitor endpoints and channels. The implementations include both a SpEL-based approach and one that executes Groovy scripts. See Section 8.3, “Control Bus” and the section called “Control Bus” for more details.

1.3 New Channel Adapters and Gateways

We have added several new Channel Adapters and Messaging Gateways in Spring Integration 2.0.

TCP/UDP Adapters

We have added Channel Adapters for receiving and sending messages over the TCP and UDP internet protocols. See Chapter 16, *TCP and UDP Support* for more details. Also, you can checkout the following blog: TCP/UDP support [http://blog.springsource.com/2010/03/29/using-udp-and-tcp-adapters-in-spring-integration-2-0-m3/]

Twitter Adapters

Twitter adapters provides support for sending and receiving Twitter Status updates as well as Direct Messages. You can also perform Twitter Searches with an inbound Channel Adapter. See Chapter 22, *Twitter Adapter* for more details.

XMPP Adapters

The new XMPP adapters support both Chat Messages and Presence events. See Chapter 25, *XMPP Support* for more details.

FTP/FTPS Adapters

Inbound and outbound File transfer support over FTP/FTPS is now available. See Chapter 12, *FTP/FTPS Adapters* for more details.

SFTP Adapters

Inbound and outbound File transfer support over SFTP is now available. See Chapter 20, *SFTP Adapters* for more details.

Feed Adapters

We have also added Channel Adapters for receiving news feeds (ATOM/RSS). See Chapter 10, *Feed Adapter* for more details.

1.4 Other Additions

Groovy Support

With Spring Integration 2.0 we've added Groovy support allowing you to use Groovy scripting language to provide integration and/or business logic. See Section 7.5, “Groovy support” for more details.

Map Transformers

These symmetrical transformers convert payload objects to and from a Map. See Section 6.1, “Transformer” for more details.

JSON Transformers

These symmetrical transformers convert payload objects to and from JSON. See Section 6.1, “Transformer” for more details.

Serialization Transformers

These symmetrical transformers convert payload objects to and from byte arrays. They also support the Serializer and Deserializer strategy interfaces that have been added as of Spring 3.0.5. See Section 6.1, “Transformer” for more details.

1.5 Framework Refactoring

The core API went through some significant refactoring to make it simpler and more usable. Although we anticipate that the impact to the end user should be minimal, please read through this document

to find what was changed. Especially, visit the section called “Dynamic Routers” , Section 7.2, “Inbound Messaging Gateways”, Section 13.3, “Http Outbound Gateway”, Section 4.1, “Message”, and Section 5.4, “Aggregator” for more details. If you are depending directly on some of the core components (Message, MessageHeaders, MessageChannel, MessageBuilder, etc.), you will notice that you need to update any import statements. We restructured some packaging to provide the flexibility we needed for extending the domain model while avoiding any cyclical dependencies (it is a policy of the framework to avoid such "tangles").

1.6 New Source Control Management and Build Infrastructure

With Spring Integration 2.0 we have switched our build environment to use Git for source control. To access our repository simply follow this URL: http://git.springsource.org/spring-integration. We have also switched our build system to Gradle [http://gradle.org/].

1.7 New Spring Integration Samples

With Spring Integration 2.0 we have decoupled the samples from our main release distribution. Please read this blog to get more info New Spring Integration Samples [http://blog.springsource.com/2010/09/29/new-spring-integration-samples/] We have also created many new samples, including samples for every new Adapter.

1.8 SpringSource Tool Suite Visual Editor for Spring Integration

There is an amazing new visual editor for Spring Integration included within the latest version of SpringSource Tool Suite. If you are not already using STS 2.5.1, please download it here: STS [http://www.springsource.com/landing/best-development-tool-enterprise-java]

1.9 Upcoming Spring Integration ROO support

We have started working on Spring Integration ROO support, and plan to have a first milestone release soon. You can follow its development here: Spring Integration Roo Add-on [https://jira.springframework.org/browse/INTROO].

Part II. Overview of Spring Integration Framework

Spring Integration provides an extension of the Spring programming model to support the well-known Enterprise Integration Patterns [http://www.eaipatterns.com/]. It enables lightweight messaging *within* Spring-based applications and supports integration with external systems via declarative adapters. Those adapters provide a higher-level of abstraction over Spring's support for remoting, messaging, and scheduling. Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code.

2. Spring Integration Overview

2.1 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 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 [http://www.eaipatterns.com] 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.

2.2 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:

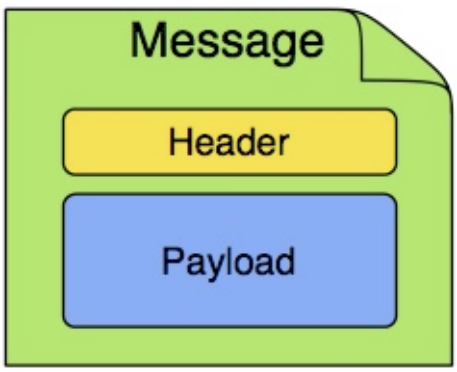
- Components should be *loosely coupled* for modularity and testability.
- The framework should enforce *separation of concerns* between business logic and integration logic.
- Extension points should be abstract in nature but within well-defined boundaries to promote *reuse* and *portability*.

2.3 Main Components

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

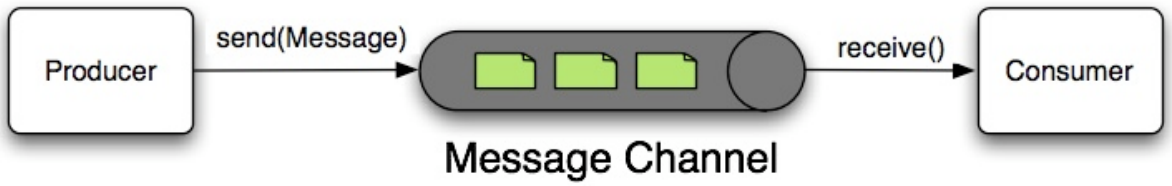
Message

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



Message Channel

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



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

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

Message Endpoint

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

2.4 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 [http://www.eaipatterns.com] book. Here, we provide only a high-level description of the main endpoint types supported by Spring Integration and their roles. The chapters that follow will elaborate and provide sample code as well as configuration examples.

Transformer

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

Filter

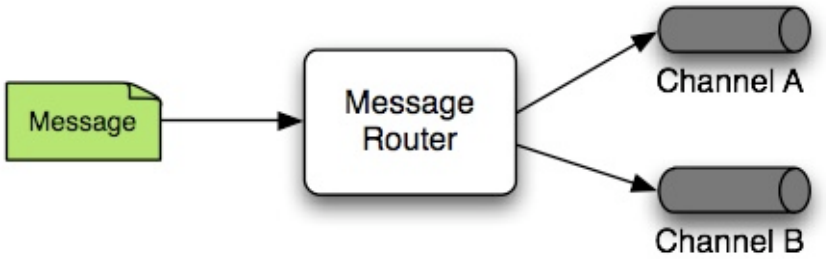
A Message Filter determines whether a Message should be passed to an output channel at all. This simply requires a boolean test method that may check for a particular payload content type, a property value, the presence of a header, etc. If the Message is accepted, it is sent to the output channel, but if not it will be dropped (or for a more severe implementation, an Exception could be thrown). Message Filters are often used in conjunction with a Publish Subscribe channel, where multiple consumers may receive the same Message and use the filter to narrow down the set of Messages to be processed based on some criteria.

Note

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

Router

A Message Router is responsible for deciding what channel or channels should receive the Message next (if any). Typically the decision is based upon the Message's content and/or metadata available in the Message Headers. A Message Router is often used as a dynamic alternative to a statically configured output channel on a Service Activator or other endpoint capable of sending reply Messages. Likewise, a Message Router provides a proactive alternative to the reactive Message Filters used by multiple subscribers as described above.



Splitter

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

Aggregator

Basically a mirror-image of the Splitter, the Aggregator is a type of Message Endpoint that receives multiple Messages and combines them into a single Message. In fact, Aggregators are often downstream consumers in a pipeline that includes a Splitter. Technically, the Aggregator is more complex than a Splitter, because it is required to maintain state (the Messages to-be-aggregated), to decide when the complete group of Messages is available, and to timeout if necessary. Furthermore, in case of a timeout, the Aggregator needs to know whether to send the partial results or to discard them to a separate channel. Spring Integration provides a `CompletionStrategy` as well as configurable settings for timeout, whether to send partial results upon timeout, and the discard channel.

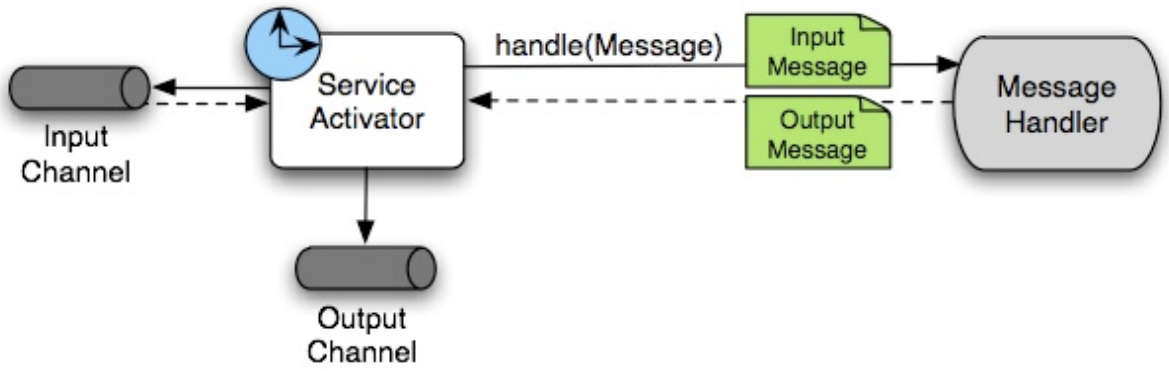
Service Activator

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

Note

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

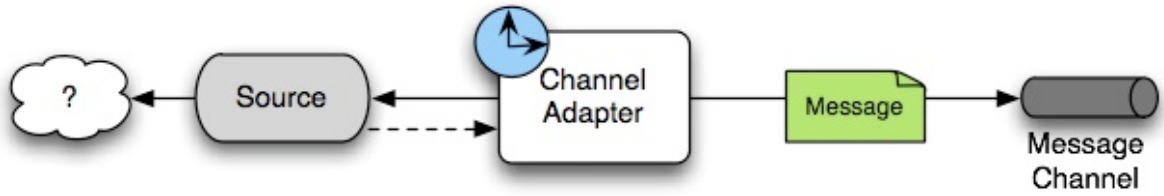
The Service Activator invokes an operation on some service object to process the request Message, extracting the request Message's payload and converting if necessary (if the method does not expect a Message-typed parameter). Whenever the service object's method returns a value, that return value will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



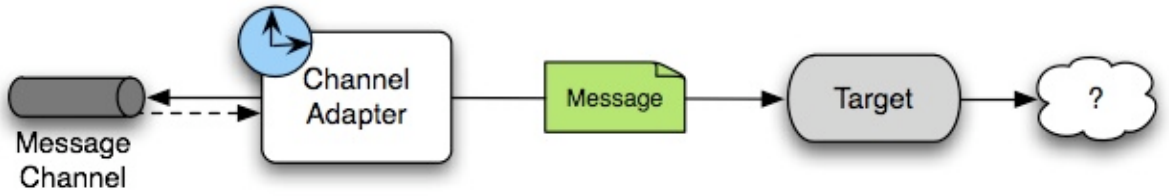
A request-reply "Service Activator" endpoint connects a target object's method to input and output Message Channels.

Channel Adapter

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



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



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

Part III. Core Messaging

This section covers all aspects of the core messaging API in Spring Integration. Here you will learn about Messages, Message Channels, and Message Endpoints. Many of the Enterprise Integration Patterns are covered here as well, such as Filters, Routers, Transformers, Service-Activators, Splitters, and Aggregators. The section also contains material about System Management, including the Control Bus and Message History support.

3. Messaging Channels

3.1 Message Channels

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

The MessageChannel Interface

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

```
public interface MessageChannel {

    boolean send(Message message);

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

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

PollableChannel

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

```
public interface PollableChannel extends MessageChannel {

    Message<?> receive();

    Message<?> receive(long timeout);

}
```

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

SubscribableChannel

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

```
public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);

}
```

Message Channel Implementations

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

PublishSubscribeChannel

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

QueueChannel

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

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

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

PriorityChannel

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

RendezvousChannel

The `RendezvousChannel` enables a "direct-handoff" scenario where a sender will block until another party invokes the channel's `receive()` method or vice-versa. Internally, this implementation is quite similar to the `QueueChannel` except that it uses a `SynchronousQueue` (a zero-capacity

implementation of `BlockingQueue`). This works well in situations where the sender and receiver are operating in different threads but simply dropping the message in a queue asynchronously is not appropriate. In other words, with a `RendezvousChannel` at least the sender knows that some receiver has accepted the message, whereas with a `QueueChannel`, the message would have been stored to the internal queue and potentially never received.

Tip

Keep in mind that all of these queue-based channels are storing messages in-memory only by default. When persistence is required, you can either provide a 'message-store' attribute within the 'queue' element to reference a persistent MessageStore implementation, or you can replace the local channel with one that is backed by a persistent broker, such as a JMS-backed channel or Channel Adapter. The latter option allows you to take advantage of any JMS provider's implementation for message persistence, and it will be discussed in Chapter 18, *JMS Support*. However, when buffering in a queue is not necessary, the simplest approach is to rely upon the `DirectChannel` discussed next.

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

DirectChannel

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

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

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

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.

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.

Scoped Channel

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

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

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

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

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

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 [http://eaipatterns.com/WireTap.html] pattern. It is a simple interceptor that sends the Message to another channel without otherwise altering the existing flow. It can be very useful for debugging and monitoring. An example is shown in the section called “Wire Tap”.

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

```
public class CountingChannelInterceptor extends ChannelInterceptorAdapter {

    private final AtomicInteger sendCount = new AtomicInteger();

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

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.

MessagingTemplate

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

```
MessagingTemplate template = new MessagingTemplate();

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

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

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

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

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

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 the section called “GatewayProxyFactoryBean”.

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 [http://www.eaipatterns.com/DatatypeChannel.html] 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 the section called “Message Channel Implementations”). Examples of each are shown below.

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 the section called “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>
```

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.

Persistent QueueChannel Configuration

Since a `QueueChannel` provides the capability to buffer Messages, but does so in-memory only by default, it also introduces a possibility that Messages could be lost in the event of a system failure. To mitigate this risk, a `QueueChannel` may be backed by a persistent implementation of the `MessageGroupStore` strategy interface. For more details on `MessageGroupStore` and `MessageStore` see Section 17.4, “Message Store”.

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

By default any `QueueChannel` only stores its Messages in an in-memory Queue and can therefore lead to the lost message scenario mentioned above. However Spring Integration provides a `JdbcMessageStore` to allow a `QueueChannel` to be backed by an RDBMS.

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

```
<int:channel id="dbBackedChannel">
    <int:queue message-store="messageStore">
<int:channel id="myChannel">

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

The above example also shows that `JdbcMessageStore` can be configured with the namespace support provided by the Spring Integration JDBC module. All you need to do is inject any `javax.sql.DataSource` instance. The Spring Integration JDBC module also provides schema DDL for most popular databases. These schemas are located in the *org.springframework.integration.jdbc* package of that module (spring-integration-jdbc).

Important

One thing to remember is that with any transactional persistent store (e.g., JdbcMessageStore), a Message removed from the store will only be permanently removed if the transaction completes successfully, otherwise the transaction will roll back and the Message will not be lost.

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

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`.

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 the section called “DirectChannel Configuration”. The same defaults apply as well. So, the channel will have a round-robin load-balancing strategy with failover enabled unless explicit configuration is provided for one or both of those attributes.

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

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>
```

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>
```

Scoped Channel Configuration

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

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

Channel Interceptor Configuration

Message channels may also have interceptors as described in the section called “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.

Global Channel Interceptor Configuration

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

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

Each <channel-interceptor/> element allows you to define a global interceptor which will be applied on all channels that match any patterns defined via the `pattern` attribute. In the above case the global interceptor will be applied on the 'foo' channel and all other channels that begin with 'bar' or 'input'. The *order* attribute allows you to manage where this interceptor will be injected if there are multiple

interceptors on a given channel. For example, channel 'inputChannel' could have individual interceptors configured locally (see below):

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

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

Note

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

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

A little more on Wire Tap

One of the common misconceptions about the wire tap and other similar components (Section 26.1, “Message Publishing Configuration”) is that they are automatically asynchronous in nature. Wire-tap as

a component is not invoked asynchronously be default. Instead, Spring Integration focuses on a single unified approach to configuring asynchronous behavior: the Message Channel. What makes certain parts of the message flow *sync* or *async* is the type of *Message Channel* that has been configured within that flow. That is one of the primary benefits of the Message Channel abstraction. From the inception of the framework, we have always emphasized the need and the value of the *Message Channel* as a first-class citizen of the framework. It is not just an internal, implicit realization of the EIP pattern, it is fully exposed as a configurable component to the end user. So, the Wire-tap component is ONLY responsible for performing the following 3 tasks:

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

It is essentially a variation of the Bridge, but it is encapsulated within a channel definition (and hence easier to enable and disable without disrupting a flow). Also, unlike the bridge, it basically forks another message flow. Is that flow *synchronous* or *asynchronous*? The answer simply depends on the type of *Message Channel* that 'channelB' is. And, now you know that we have: *Direct Channel*, *Pollable Channel*, and *Executor Channel* as options. The last two do break the thread boundary making communication via such channels *asynchronous* simply because the dispatching of the message from that channel to its subscribed handlers happens on a different thread than the one used to send the message to that channel. That is what is going to make your wire-tap flow *sync* or *async*. It is consistent with other components within the framework (e.g., Message Publisher) and actually brings a level of consistency and simplicity by sparing you from worrying in advance (other than writing thread safe code) whether a particular piece of code should be implemented as *sync* or *async*. The actual wiring of two pieces of code (component A and component B) via *Message Channel* is what makes their collaboration *sync* or *async*. You may even want to change from *sync* to *async* in the future and *Message Channel* is what's going to allow you to do it swiftly without ever touching the code.

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

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”.

3.2 Channel Adapter

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

Configuring Inbound Channel Adapter

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

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

<inbound-channel-adapter ref="source2" method="method2" channel="channel2">
    <poller cron="30 * 9-17 * * MON-FRI"/>
</channel-adapter>
```

Note

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

Configuring Outbound Channel Adapter

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

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

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

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

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

</outbound-channel-adapter>
```

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

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

```
<outbound-channel-adapter channel="channel" method="handle">
        <beans:bean class="org.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. Such a configuration will result in an Exception being thrown.

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

3.3 Messaging Bridge

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.

Configuring Bridge

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" fixed-rate="5000"/>
</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.

4. Message Construction

4.1 Message

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.

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.

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 4.1. Pre-defined Message Headers

Header Name	Header Type
ID	java.util.UUID
TIMESTAMP	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
EXPIRATION_DATE	java.lang.Long
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.

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 is also a convenient implementation of `Message` designed to communicate error conditions. THis implementation takes `Throwable` object as its payload:

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

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

Notice that this implementation takes 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.

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
}
```

5. Message Routing

5.1 Router

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 common requirements.

PayloadTypeRouter

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

```
<bean id="payloadTypeRouter" class="org.springframework.integration.router.PayloadTypeRouter">
    <property name="channelIdentifierMap">
        <map>
            <entry key="java.lang.String" value-ref="stringChannel"/>
            <entry key="java.lang.Integer" value-ref="integerChannel"/>
        </map>
    </property>
</bean>
```

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

HeaderValueRouter

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

1. Arbitrary value

2. Channel name

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

Spring Integration provides a simple namespace-based XML configuration to configure a `HeaderValueRouter`. The example below demonstrates two types of namespace-based configuration for the `HeaderValueRouter`.

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>
```

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

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

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

Note

The two router implementations shown above share some common attributes, such as `default-output-channel` and `resolution-required`. If `resolution-required` 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 `default-output-channel` has been specified), then an Exception will be thrown.

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>
```

Spring Integration also provides namespace support for the `RecipientListRouter` configuration (see Section B.2, “Namespace Support”) as the example below demonstrates.

```
<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 effect as it does for a publish-subscribe-channel, and like a publish-subscribe-channel, it is disabled by default on the recipient-list-router. Refer to the section called “PublishSubscribeChannel Configuration” for more information.

Another convenient option when configuring a `RecipientListRouter` is to use Spring Expression Language (SpEL) support as selectors for individual recipient channels. This is similar to using a Filter at the beginning of 'chain' to act as a "Selective Consumer". However, in this case, it's all combined rather concisely into the router's configuration.

```
<int:recipient-list-router id="customRouter" input-channel="routingChannel">
 <int:recipient channel="channel1" selector-expression="payload.equals('foo')"/>
 <int:recipient channel="channel2" selector-expression="headers.contains('bar')"/>
</int:recipient-list-router>
```

In the above configuration a SpEL expression identified by the `selector-expression` attribute will be evaluated to determine if this recipient should be included in the recipient list for a given input Message. The evaluation result of the expression must be a boolean. If this attribute is not defined, the channel will always be among the list of recipients.

Configuring Router

Configuring a Content Based Router with XML

The "router" element provides a simple way to connect a router to an input channel and also accepts the optional `default-output-channel` attribute. The `ref` attribute references 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, `ref` may point to a simple POJO that contains the @Router annotation (see below), or the `ref` may be combined with an explicit `method` name. Specifying a `method` applies the same behavior 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 is referenced in other `<router>` definitions. However if the custom router implementation should be scoped to a single definition of the `<router>`, you may provide an inner bean definition:

```
<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 an Exception will be thrown.

Routers and the Spring Expression Language (SpEL)

Sometimes the routing logic may be simple and writing a separate class for it and configuring it as a bean may seem like overkill. As of Spring Integration 2.0 we offer an alternative where you can now use SpEL [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.htm] to implement simple computations that previously required a custom POJO router.

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

In the above configuration the result channel will be computed by the SpEL expression which simply concatenates the value of the `payload` with the literal String 'Channel'.

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

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

In the above configuration, if the Message includes a header with the name 'channels' the value of which is a `List` of channel names then the Message will be sent to each channel in the list. You may also find *Collection Projection* and *Collection Selection* expressions useful to select multiple channels. See "http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html#d0e12084" [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html#d0e12084]

Configuring a Router with Annotations

When using `@Router` to annotate a method, the method may return either a `MessageChannel` or `String` type. In the latter case, the endpoint will resolve the channel name as it does for the default output channel. Additionally, the method may return either a single value or a collection. If 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 Message may be routed based on metadata available within the message header as either a property or attribute. In this case, a method annotated with `@Router` may include a parameter annotated with `@Header` which is mapped to a header value as illustrated below and 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 24, *XML Support - Dealing with XML Payloads*.

Dynamic Routers

So as you can see, Spring Integration provides quite a few different router configurations for common *content-based routing* use cases as well as the option of implementing custom routers as POJOs. For example `PayloadTypeRouter` provides a simple way to configure a router which computes `channels` based on the `payload type` of the incoming Message while `HeaderValueRouter` provides the same convenience in configuring a router which computes `channels` by evaluating the value of a particular Message Header. There are also *expression-based* (SpEL) routers where the `channel` is determined based on evaluating an expression. Thus, these type of routers exhibit some dynamic characteristics.

However these routers all require *static configuration*. Even in the case of expression-based routers, the expression itself is defined as part of the router configuration which means that *the same expression operating on the same value will always result in the computation of the same channel*. This is acceptable in most cases since such routes are well defined and therefore predictable. But there are times when we need to change router configurations dynamically so message flows may be routed to a different channel.

Example:

You might want to bring down some part of your system for maintenance and temporarily re-reroute messages to a different message flow. Or you may want to introduce more granularity to your message flow by adding another route to handle a more concrete type of java.lang.Number (in the case of `PayloadTypeRouter`).

Unfortunately with static router configuration to accomplish this you would have to bring down your entire application, change the configuration of the router (change routes) and bring it back up. This is obviously not the solution.

The Dynamic Router [http://www.eaipatterns.com/DynamicRouter.html] pattern describes the mechanisms by which one can change/configure routers dynamically without bringing down the system or individual routers.

Before we get into the specifics of how this is accomplished in Spring Integration let's quickly summarize the typical flow of the router, which consists of 3 simple steps:

- *Step 1* - Compute `channel identifier` which is a value calculated by the router once it receives the Message. Typically it is a `String` or and instance of the actual `MessageChannel`.
- *Step 2* - Resolve `channel identifier` to `channel name`. We'll describe specifics of this process in a moment.
- *Step 3* - Resolve `channel name` to the actual `MessageChannel`

There is not much that can be done with regard to dynamic routing if Step 1 results in the actual instance of the `MessageChannel` simply because the `MessageChannel` is the *final product* of any router's job. However, if Step 1 results in a `channel identifier` that is not an instance of `MessageChannel`, then there are quite a few possibilities to influence the process of deriving the `Message Channel`. Lets look at couple of the examples in the context of the 3 steps mentioned above:

Payload Type Router

```
<payload-type-router input-channel="routingChannel">
  <mapping type="java.lang.String" channel="channel1" />
  <mapping type="java.lang.Integer" channel="channel2" />
</payload-type-router>
```

Within the context of the Payload Type Router the 3 steps mentioned above would be realized as:

- *Step 1* - Compute `channel identifier` which is the fully qualified name of the payload type (e.g., java.lang.String).
- *Step 2* - Resolve `channel identifier` to `channel name` where the result of the previous step is used to select the appropriate value from the *payload type mapping* defined via `mapping` element.
- *Step 3* - Resolve `channel name` to the actual instance of the `MessageChannel` where using `ChannelResolver`, the router will obtain a reference to a bean (which is hopefully a `MessageChannel`) identified by the result of the previous step.

In other words each step feeds the next step until the process completes.

Header Value Router

```
<header-value-router input-channel="inputChannel" header-name="testHeader">
  <mapping value="foo" channel="fooChannel" />
  <mapping value="bar" channel="barChannel" />
</header-value-router>
```

Similar to the PayloadTypeRouter:

- *Step 1* - Compute `channel identifier` which is the value of the header identified by the `header-name` attribute.
- *Step 2* - Resolve `channel identifier` to `channel name` where the result of the previous step is used to select the appropriate value from the *general mapping* defined via `mapping` element.

- *Step 3* - Resolve `channel name` to the actual instance of the `MessageChannel` where using `ChannelResolver`, the router will obtain a reference to a bean (which is hopefully a `MessageChannel`) identified by the result of the previous step.

The above two configurations of two different router types look almost identical. However if we look at the alternate configuration of the `HeaderValueRouter` we clearly see that there is no `mapping` sub element:

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

But the configuration is still perfectly valid. So the natural question is what about the mapping in the Step 2?

What this means is that Step 2 is now an optional step. If mapping is not defined then the `channel identifier` value computed in Step 1 will automatically be treated as the `channel name` which will now be resolved to the actual `MessageChannel` in the Step 3. What it also means is that Step 2 is one of the key steps to provide dynamic characteristics to the routers, since it introduces a process which *allows you to change the way 'channel identifier' resolves to 'channel name'*, thus influencing the process of determining the final instance of the `MessageChannel` from the initial `channel identifier`.

For Example:

In the above configuration let's assume that the `testHeader` value is 'kermit' which is now a `channel identifier` (Step 1). Since there is no mapping in this router, resolving this `channel identifier` to a `channel name` (Step 2) is impossible and this `channel identifier` is now treated as `channel name`. However what if there was a mapping but for a different value? The end result would still be the same and that is: *if new value cannot be determined through the process of resolving the 'channel identifier' to a 'channel name', such 'channel identifier' becomes 'channel name'*.

So all that is left is for Step 3 to resolve the `channel name` ('kermit') to an actual instance of the `MessageChannel` identified by this name. That will be done via the default ChannelResolver implementation which is a `BeanFactoryChannelResolver`. It basically does a bean lookup for the name provided. So now all messages which contain the header/value pair as `testHeader=kermit` are going to be routed to a `MessageChannel` whose bean name (id) is 'kermit'.

But what if you want to route these messages to the 'simpson' channel? Obviously changing a static configuration will work, but will also require bringing your system down. However if you had access to the `channel identifier` map, then you could just introduce a new mapping where the header/value pair is now `kermit=simpson`, thus allowing Step 2 to treat 'kermit' as a `channel identifier` while resolving it to 'simpson' as the `channel name` .

The same obviously applies for `PayloadTypeRouter` where you can now remap or remove a particular *payload type mapping*. In fact, it applies to every other router including *expression-based* routers since their computed values will now have a chance to go through Step 2 to be additionally resolved to the actual `channel name`.

In Spring Integration 2.0 the routers hierarchy underwent significant refactoring so that now any router that is a subclass of the `AbstractMessageRouter` (which includes all framework defined

routers) is a Dynamic Router simply because the `channelIdentiferMap` is defined at the `AbstractMessageRouter` level. That map's setter method is exposed as a public method along with 'setChannelMapping' and 'removeChannelMapping' methods. These allow you to change/add/remove router mappings at runtime as long as you have a reference to the router itself. It also means that you could expose these same configuration options via JMX (see Section 8.1, “JMX Support”) or the Spring Integration ControlBus (see Section 8.3, “Control Bus”) functionality.

Control Bus

One way to manage the router mappings is through the Control Bus [http://www.eaipatterns.com/ControlBus.html] pattern which exposes a Control Channel where you can send control messages to manage and monitor Spring Integration components, including routers. For more information about the Control Bus see Section 8.3, “Control Bus”. Typically you would send a control message asking to invoke a particular operation on a particular managed component (e.g., router). The two managed operations (methods) that are specific to changing the router resolution process are:

- `public void setChannelMapping(String channelIdentifier, String channelName)` - will allow you to add a new or modify an existing mapping between `channel identifier` and `channel name`
- `public void removeChannelMapping(String channelIdentifier)` - will allow you to remove a particular channel mapping, thus disconnecting the relationship between `channel identifier` and `channel name`

You can also expose a router instance with Spring's JMS support and then use your favorite JMX client (e.g., JConsole) to manage those operations (methods) for changing the router's configuration. For more information on Spring Integration management and monitoring please visit Section 8.1, “JMX Support”.

5.2 Filter

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 Message content 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 the Message to but only decides *whether* to send.

Note

As you will see momentarily, the Filter also supports 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 an 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.

Configuring Filter

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 method must return a boolean value. If the method returns 'true', the Message *will* be sent 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 handling of the rejected Message. By default (if configured like the example above), rejected Messages will be silently dropped. If rejection should instead result in an error condition, then set the `throw-exception-on-rejection` attribute to `true`:

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

If you want rejected messages to be routed to a specific channel, provide that reference as the `discard-channel`:

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

Note

Message Filters are commonly used 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 to 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 is referenced in other `<filter>` definitions. However if the custom filter implementation is 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 an Exception will be thrown.

With the introduction of SpEL support, Spring Integration 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 with the Message available in the evaluation context. If it is necessary 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 [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html#expressions-beandef].

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

If the Expression itself needs to be dynamic, then an 'expression' sub-element may be used. That provides a level of indirection for resolving the Expression by its key from an ExpressionSource. That is a strategy interface that you can implement directly, or you can rely upon a version available in Spring Integration that loads Expressions from a "resource bundle" and can check for modifications after a given number of seconds. All of this is demonstrated in the following configuration sample where the Expression could be reloaded within one minute if the underlying file had been modified. If the ExpressionSource bean is named "expressionSource", then it is not necessary to provide the `source` attribute on the <expression> element, but in this case it's shown for completeness.

```
<filter input-channel="input" output-channel="output">
    <expression key="filterPatterns.example" source="myExpressions"/>
</filter>

<beans:bean id="myExpressions" id="myExpressions"
        class="org.springframework.integration.expression.ReloadableResourceBundleExpressionSource">
    <beans:property name="basename" value="config/integration/expressions"/>
    <beans:property name="cacheSeconds" value="60"/>
</beans:bean>
```

Then, the 'config/integration/expressions.properties' file (or any more specific version with a locale extension to be resolved in the typical way that resource-bundles are loaded) would contain a key/value pair:

```
filterPatterns.example=payload > 100
```

Note

All of these examples that use `expression` as an attribute or sub-element can also be applied within transformer, router, splitter, service-activator, and header-enricher elements. Of course, the semantics/role of the given component type would affect the interpretation of the evaluation result in the same way that the return value of a method-invocation would be interpreted. For example, an expression can return Strings that are to be treated as Message Channel names by a router component. However, the underlying functionality of evaluating the expression against the Message as the root object, and resolving bean names if prefixed with '@' is consistent across all of the core EIP components within Spring Integration.

5.3 Splitter

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.

Programming model

The API for performing splitting consists of 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 enables tracking 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 [http://www.eaipatterns.com/DistributionAggregate.html] scenario.

An excerpt from `AbstractMessageSplitter` can be seen below:

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

}
```

To implement a specific Splitter in an application, extend `AbstractMessageSplitter` and implement the `splitMessage` method, which contains logic for splitting the messages. The return value may 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 a single message will 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.

Configuring Splitter

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 to the Collection, incorporating each individual element into a Message and sending it to 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 to which 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 implementation may be referenced in other `<splitter>` definitions. However if the custom splitter handler implementation should be scoped to a single definition of the `<splitter>`, 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.

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 item will be wrapped in a Message as its payload. Each message will be sent to the designated output channel for the endpoint on which the `@Splitter` is defined.

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

5.4 Aggregator

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, an Aggregator is often a downstream consumer in a pipeline that includes a Splitter.

Technically, the Aggregator is more complex than a Splitter, because it is stateful as it must hold the Messages to be aggregated and determine when the complete group of Messages is ready to be aggregated. In order to do this it requires a `MessageStore`.

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 the aggregated message as output.

Implementing an Aggregator requires providing the logic to perform the aggregation (i.e., the creation of a single message from many). Two related concepts are correlation and release.

Correlation determines how messages are grouped for aggregation. In Spring Integration correlation is done by default based on the CORRELATION_ID message header. Messages with the same CORRELATION_ID will be grouped together. However, the correlation strategy may be customized to allow other ways of specifying how the messages should be grouped together by implementing a `CorrelationStrategy` (see below).

To determine the point at which a group of messages is ready to be processed, a `ReleaseStrategy` is consulted. The default release strategy for the Aggregator will release a group when all messages

included in a sequence are present, based on the SEQUENCE_SIZE header. This default strategy may be overridden by providing a reference to a custom `ReleaseStrategy` implementation.

Programming model

The Aggregation API consists of a number of classes:

- The interface `MessageGroupProcessor`, and its subclasses: `MethodInvokingAggregatingMessageGroupProcessor` and `ExpressionEvaluatingMessageGroupProcessor`
- The `ReleaseStrategy` interface and its default implementation `SequenceSizeReleaseStrategy`
- The `CorrelationStrategy` interface and its default implementation `HeaderAttributeCorrelationStrategy`

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 can be released
- deciding when the group can be released
- aggregating the released group into a single 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 `aggregatePayloads` method is left to the developer):

```
public abstract class AbstractAggregatingMessageGroupProcessor
              implements MessageGroupProcessor {

    protected Map<String, Object> aggregateHeaders(MessageGroup group) {
        // default implementation exists
    }

   protected abstract Object aggregatePayloads(MessageGroup group, Map<String, Object> defaultHeaders);

}
```

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

```
    public CorrelatingMessageHandler(MessageGroupProcessor processor, MessageGroupStore store,
```

```
                    CorrelationStrategy correlationStrategy, ReleaseStrategy releaseStrategy) {
    ...
     this.correlationStrategy = correlationStrategy == null ?
       new HeaderAttributeCorrelationStrategy(MessageHeaders.CORRELATION_ID) : correlationStrategy;
     this.releaseStrategy = releaseStrategy == null ? new SequenceSizeReleaseStrategy() : releaseStrategy;
     ...
 }
```

As for actual processing of the message group, the default implementation is the `DefaultAggregatingMessageGroupProcessor`. It creates a single Message whose payload is a List of the payloads received for a given group. 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 in which these headers are not necessary.

When implementing a specific aggregator strategy for an application, a developer can extend `AbstractAggregatingMessageGroupProcessor` and implement the `aggregatePayloads` method. However, there are better solutions, less coupled to the API, for implementing the aggregation logic which can be configured easily either through XML or through annotations.

In general, any POJO can implement the aggregation algorithm if it provides a method that accepts a single `java.util.List` as an argument (parameterized lists are supported as well). This method will be invoked for aggregating messages as follows:

- if the argument is a `java.util.List<T>`, and the parameter type T 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-parameterized `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 configuring it in the application.

ReleaseStrategy

The `ReleaseStrategy` interface is defined as follows:

```
public interface ReleaseStrategy {

  boolean canRelease(MessageGroup group);

}
```

In general, any POJO can implement the completion decision logic if it provides a method that accepts a single `java.util.List` as an argument (parameterized lists are supported as well), and returns a boolean value. This method will be invoked after the arrival of each new message, to decide whether the group is complete or not, as follows:

- if the argument is a `java.util.List<T>`, and the parameter type T 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 (potentially a new partial sequence).

Spring Integration provides an out-of-the box implementation for `ReleaseStrategy`, the `SequenceSizeReleaseStrategy`. This implementation consults the SEQUENCE_NUMBER and SEQUENCE_SIZE headers of each arriving message to decide when a message group is complete and ready to be aggregated. As shown above, it is also the default strategy.

CorrelationStrategy

The `CorrelationStrategy` interface is defined as follows:

```
public interface CorrelationStrategy {

  Object getCorrelationKey(Message<?> message);

}
```

The method returns an Object which represents the correlation key used for associating the message with a message group. 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 POJO can implement the correlation logic, 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. If you have a custom header name you would like to use for correlation, then simply configure that on an instance of `HeaderAttributeCorrelationStrategy` and provide that as a reference for the Aggregator's correlation-strategy.

Configuring Aggregator

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.

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


<int:aggregator id="myAggregator" ❶
  auto-startup="true" ❷
  input-channel="inputChannel" ❸
  output-channel="outputChannel" ❹
  discard-channel="throwAwayChannel" ❺
  message-store="persistentMessageStore" ❻
  order="1" ❼
  send-partial-result-on-expiry="false" ❽
  send-timeout="1000" ❾

  correlation-strategy="correlationStrategyBean" ❿
  correlation-strategy-method="correlate" 11

  ref="aggregatorBean" 12
  method="aggregate" 13

  release-strategy="releaseStrategyBean" 14
  release-strategy-method="release"/> 15


<int:channel id="outputChannel"/>

<int:channel id="throwAwayChannel"/>

<bean id="persistentMessageStore" class="org.springframework.integration.jdbc.JdbcMessageStore">
 <constructor-arg ref="dataSource"/>
</bean>

<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 *0ptional*.

❷ Lifecycle attribute signaling if aggregator should be started during Application Context startup. *Optional (default is 'true')*.

❸ The channel from which where aggregator will receive messages. *Required*.

❹ The channel to which the aggregator will send the aggregation results. *Optional (because incoming messages can specify a reply channel themselves via 'replyChannel' Message Header)*.

❺ The channel to which the aggregator will send the messages that timed out (if `send-partial-result-on-expiry` is *false*). *Optional*.

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

❼ Order of this aggregator when more than one handle is subscribed to the same DirectChannel (use for load balancing purposes). *Optional*.

❽ Indicates if partially aggregated messages should be released when their storage time has expired (see `MessageGroupStore.expireMessageGroups(long)`). *Optional*.

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

❿ A reference to a bean that implements the message correlation (grouping) algorithm. 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)* .

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

12 A reference to a bean defined in the application context. The bean must implement the aggregation logic as described above. *Optional (by default the list of aggregated Messages will become a payload of the output message).*

13 A method defined on the bean referenced by `ref`, that implements the message aggregation algorithm. *Optional, depends on `ref` attribute being defined.*

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

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

Using a `ref` attribute is generally recommended if a custom aggregator handler implementation may be referenced in other `<aggregator>` definitions. However if a custom aggregator implementation is only being used by a single definition of the `<aggregator>`, you can use an inner bean definition (starting with version 1.0.3) to configure the aggregation POJO within the `<aggregator>` element:

```
<aggregator input-channel="input" method="sum" output-channel="output">
    <beans:bean class="org.foo.PojoAggregator"/>
</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 provided by the payloads 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).

Aggregators and Spring Expression Language (SpEL)

Since Spring Integration 2.0, the various strategies (correlation, release, and aggregation) may be handled with SpEL [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html] which is recommended if the logic behind such *release strategy* is relatively simple.

Let's say you have a legacy component that was designed to receive an array of objects. We know that the default release strategy will assemble all aggregated messages in the List. So now we have two problems. First we need to extract individual messages from the list, and then we need to extract the payload of each message and assemble the array of objects (see code below).

```
public String[] processRelease(List<Message<String>> messages){
  List<String> stringList = new ArrayList<String>();
  for (Message<String> message : messages) {
   stringList.add(message.getPayload());
  }
  return stringList.toArray(new String[]{});
}
```

However, with SpEL such a requirement could actually be handled relatively easily with a one-line expression, thus sparing you from writing a custom class and configuring it as a bean.

```
<int:aggregator input-channel="aggChannel"
 output-channel="replyChannel"
 expression="#this.![payload].toArray()"/>
```

In the above configuration we are using a Collection Projection [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html#d0e12113] expression to assemble a new collection from the payloads of all messages in the list and then transforming it to an Array, thus achieving the same result as the java code above.

The same expression-based approach can be applied when dealing with custom *Release* and *Correlation* strategies.

Instead of defining a bean for a custom `CorrelationStrategy` via the `correlation-strategy` attribute, you can implement your simple correlation logic via a SpEL expression and configure it via the `correlation-strategy-expression` attribute.

For example:

```
correlation-strategy-expression="payload.person.id"
```

In the above example it is assumed that the payload has an attribute `person` with an `id` which is going to be used to correlate messages.

Likewise, for the `ReleaseStrategy` you can implement your release logic via a SpEL expression and configure it via the `release-strategy-expression` attribute.

For example:

```
release-strategy-expression="payload.size() gt 5"
```

In this example the root object of the SpEL Evaluation Context is the `MessageGroup` itself, and you are simply stating that as soon as there are more than 5 messages in this group, it should be released.

Configuring an Aggregator with Annotations

An aggregator configured using annotations would 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 SequenceSizeReleaseStrategy.

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.

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 (whether defined by the framework or a user) should be able to remain stateless. All state is carried by the `MessageGroup` and its management is delegated to the `MessageGroupStore`.

```
public interface MessageGroupStore {
   int getMessageCountForAllMessageGroups();

   int getMarkedMessageCountForAllMessageGroups();

   int getMessageGroupCount();

   MessageGroup getMessageGroup(Object groupId);

   MessageGroup addMessageToGroup(Object groupId, Message<?> message);

   MessageGroup markMessageGroup(MessageGroup group);

   MessageGroup removeMessageFromGroup(Object key, Message<?> messageToRemove);
```

```
   MessageGroup markMessageFromGroup(Object key, Message<?> messageToMark);

   void removeMessageGroup(Object groupId);

   void registerMessageGroupExpiryCallback(MessageGroupCallback callback);

   int expireMessageGroups(long timeout);
}
```

For more information please refer to the JavaDoc [http://static.springsource.org/spring-integration/api/org/springframework/integration/store/MessageGroupStore.html].

The `MessageGroupStore` accumulates state information in `MessageGroups` while waiting for a release strategy to be triggered, and that event might not ever happen. So to prevent stale messages from lingering, and for volatile stores to provide a hook for cleaning up when the application shuts down, the `MessageGroupStore` allows the user to register callbacks to apply to its `MessageGroups` when they expire. The interface is very straightforward:

```
public interface MessageGroupCallback {

  void execute(MessageGroupStore messageGroupStore, MessageGroup group);

}
```

The callback has direct access 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, on demand, to all messages whose timestamp is earlier than a time supplied as a parameter (see the `registerMessageGroupExpiryCallback(..)` and `expireMessageGroups(..)` methods above).

The `expireMessageGroups` method can be called with a timeout value. Any message older than the current time minus this value will 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="30000"/>
</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 in the example above is that the message group store's expire method is being called once every 10 seconds. The timeout itself is 30 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, any unmarked messages in groups that are not yet released can be sent on to the output channel.

5.5 Resequencer

Introduction

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

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.

Configuring a Resequencer

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 to which the resequencer will send the reordered messages. *Optional*.

❹ The channel to which 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 may 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 the section called “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.

5.6 Message Handler Chain

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 [http://www.eaipatterns.com/MessageSelector.html]. In either case, the chain only requires a single `input-channel` and a single `output-channel` eliminating the need to define 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. If you do indeed want to "drop" the Messages, then the Filter's 'discard-channel' might be useful since it does give you a chance to perform some operation with the dropped message (e.g. send to a JMS queue or simply write to a log).

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 anonymous 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 required to implement the MessageProducer interface (which provides a 'setOutputChannel()' method). 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 as a fallback.

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`.

Configuring Chain

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>
        <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 named "foo" with a value of "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, but the header-enricher is obviously a simpler option.

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

```
<si:chain id="main-chain" input-channel="in" output-channel="out">
  <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="inputA"/>
</si:chain>

<si:chain id="nested-chain-a" input-channel="inputA">
  <si:header-enricher>
    <si:header name="name" value="Moe" />
  </si:header-enricher>
```

```
    <si:gateway request-channel="inputB"/>
    <si:service-activator>
      <bean class="org.foo.SampleService" />
    </si:service-activator>
  </si:chain>

  <si:chain id="nested-chain-b" input-channel="inputB">
    <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 the flow returns to the *main-chain*. When the nested version of a <gateway> element is defined in the chain, it does not require the `service-interface` attribute. Instead, it simple takes the message in its current state and places it on the channel defined via the `request-channel` attribute. When the downstream flow initiated by that gateway completes, a `Message` will be returned to the gateway and continue its journey within the current chain.

6. Message Transformation

6.1 Transformer

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 [http://www.eaipatterns.com/CanonicalDataModel.html], 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 24, *XML Support - Dealing with XML Payloads*.

Configuring Transformer

Configuring Transformer with XML

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).

As of Spring Integration 2.0, a Message Transformer's transformation method can no longer return `null`. Returning `null` will result in an exception since a Message Transformer should always be expected to transform each source Message into a valid target Message. In other words, a Message Transformer should not be used as a Message Filter since there is a dedicated <filter> option for that. However, if you do need this type of behavior (where a component might return NULL and that should not be considered an error), a *service-activator* could be used. Its `requires-reply` value is FALSE by default, but that can be set to TRUE in order to have Exceptions thrown for NULL return values as with the transformer.

Transformers and Spring Expression Language (SpEL)

Just like Routers, Aggregators and other components, as of Spring Integration 2.0 Transformers can also benefit from SpEL support (http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html) whenever transformation logic is relatively simple.

```
<int:transformer input-channel="inChannel"
 output-channel="outChannel"
 expression="payload.toUpperCase() + '- [' + T(java.lang.System).currentTimeMillis() + ']'"/>
```

In the above configuration we are achieving a simple transformation of the *payload* with a simple SpEL expression and without writing a custom transformer. Our *payload* (assuming String) will be upper-cased and concatenated with the current timestamp with some simple formatting.

Common Transformers

There are 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 the section called “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. These will use standard Java serialization by default, but you can provide an implementation of Spring 3.0's Serializer or Deserializer strategies via the 'serializer' and 'deserializer' attributes, respectively.

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

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

Object-to-Map Transformer

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

The 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 the 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 one. Also, if using the 'ref' attribute, you must point to a 'prototype' scoped bean, otherwise a BeanCreationException will be thrown.

Configuring a Transformer with Annotations

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);
}
```

Header Filter

Some times your transformation use case might be as simple as removing a few headers. For such a use case, Spring Integration provides a *Header Filter* which allows you to specify certain header names that should be removed from the output Message (e.g. for security reasons or a value that was only needed temporarily). Basically the *Header Filter* is the opposite of the *Header Enricher*. The latter is discussed in the section called “Header Enricher”

```
<int:header-filter input-channel="inputChannel"
  output-channel="outputChannel" header-names="lastName, state"/>
```

As you can see, configuration of a *Header Filter* is quite simple. It is a typical endpoint with input/output channels and a `header-names` attribute. That attribute accepts the names of the header(s) (delimited by commas if there are multiple) that need to be removed. So, in the above example the headers named 'lastName' and 'state' will not be present on the outbound Message.

6.2 Content Enricher

Introduction

At times you may have a requirement to enhance a request with more information than was provided by the target system. The Content Enricher pattern describes various scenarios as well as the component (Enricher), which allows you to address such requirements.

Header Enricher

If you only need to add headers to a Message, and they are not dynamically determined by the Message content, then referencing a custom implementation of a Transformer may be overkill. For that reason, Spring Integration provides support for the *Header Enricher* pattern. It is exposed via the `<header-enricher>` element.

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

The *Header Enricher* also provides helpful sub-elements to set well-known header names.

```
<int:header-enricher input-channel="in" output-channel="out">
  <int:error-channel ref="applicationErrorChannel"/>
  <int:reply-channel ref="quoteReplyChannel"/>
  <int:correlation-id value="123"/>
  <int:priority value="HIGHEST"/>
  <int:header name="bar" ref="someBean"/>
</int:header-enricher>
```

In the above configuration you can clearly see that for well-known headers such as `errorChannel`, `correlationId`, `priority`, `replyChannel`etc., instead of using generic *<header>* sub-elements where you would have to provide both header 'name' and 'value', you can use convenient sub-elements to set those values directly.

POJO Support

Often a header value cannot be defined statically and has to be determined dynamically based on some content in the Message. That is why *Header Enricher* allows you to also specify a bean 'ref' and 'method' that will calculate the header value. Let's look at the following configuration:

```
<int:header-enricher input-channel="in" output-channel="out">
    <int:header name="foo" method="computeValue" ref="myBean"/>
</int:header-enricher>
```

```
<bean id="myBean" class="foo.bar.MyBean"/>
```

```
public class MyBean {
    public String computeValue(String payload){
        return payload.toUpperCase() + "_US";
    }

}
```

SpEL Support

In Spring Integration 2.0 we have introduced the convenience of the Spring Expression Language (SpEL) to help configure many different components. The *Header Enricher* is one of them. Looking again at the POJO example above, you can see that the computation logic to determine the header value is actually pretty simple. A natural question would be: "is there a simpler way to accomplish this?". That is where SpEL shows its true power.

```
<int:header-enricher input-channel="in" output-channel="out">
 <int:header name="foo" expression="payload.toUpperCase() + '_US'"/>
</int:header-enricher>
```

As you can see, by using SpEL for such simple cases, we no longer have to provide a separate class and configure it in the application context. All we need is the *expression* attribute configured with a valid SpEL expression. The 'payload' and 'headers' variables are bound to the SpEL Evaluation Context, giving you full access to the incoming Message.

Adapter specific Header Enrichers

As you go through the manual, you will see that as an added convenience, Spring Integration also provides adapter specific Header Enrichers (e.g., MAIL, XMPP, etc.)

6.3 Claim Check

Introduction

In the earlier sections we've covered several Content Enricher type components that help you deal with situations where a message is missing a piece of data. We also discussed Content Filtering which lets you remove data items from a message. However there are times when we want to hide data temporarily. For example, in a distributed system we may receive a Message with a very large payload. Some intermittent message processing steps may not need access to this payload and some may only need to access certain headers, so carrying the large Message payload through each processing step may cause performance degradation, may produce a security risk, and may make debugging more difficult.

The Claim Check pattern describes a mechanism that allows you to store data in a well known place while only maintaining a pointer (Claim Check) to where that data is located. You can pass that pointer around as a payload of a new Message thereby allowing any component within the message flow to get the actual data as soon as it needs it. This approach is very similar to the Certified Mail process where you'll get a Claim Check in your mailbox and would have to go to the Post Office to claim your actual package. Of course it's also the same idea as baggage-claim on a flight or in a hotel.

Spring Integration provides two types of Claim Check transformers: *Incoming Claim Check Transformer* and *Outgoing Claim Check Transformer*. Convenient namespace-based mechanisms are available to configure them.

Incoming Claim Check Transformer

An *Incoming Claim Check Transformer* will transform an incoming Message by storing it in the Message Store identified by its `message-store` attribute.

```
<int:claim-check-in id="checkin"
        input-channel="checkinChannel"
        message-store="testMessageStore"
        output-channel="output"/>
```

In the above configuration the Message that is received on the `input-channel` will be persisted to the Message Store identified with the `message-store` attribute and indexed with generated ID. That ID is the Claim Check for that Message. The Claim Check will also become the payload of the new (transformed) Message that will be sent to the `output-channel`.

Now, lets assume that at some point you do need access to the actual Message. You can of course access the Message Store manually and get the contents of the Message, or you can use the same approach as before except now you will be transforming the Claim Check to the actual Message by using an *Outgoing Claim Check Transformer*.

Outgoing Claim Check Transformer

An *Outgoing Claim Check Transformer* allows you to transform a Message with a Claim Check payload into a Message with the original content as its payload.

```
<claim-check-out id="checkout"
        input-channel="checkoutChannel"
        message-store="testMessageStore"
        output-channel="output"/>
```

In the above configuration, the Message that is received on the `input-channel` should have a Claim Check as its payload and the *Outgoing Claim Check Transformer* will transform it into a Message with the original payload by simply querying the Message store for a Message identified by the provided Claim Check. It then sends the newly checked-out Message to the `output-channel`.

Although we rarely care about the details of the claim checks as long as they work, it is still worth knowing that the current implementation of the actual Claim Check (the pointer) in Spring Integration is a UUID to ensure uniqueness.

A word on Message Store

`org.springframework.integration.store.MessageStore` is a strategy interface for storing and retrieving messages. Spring Integration provides two convenient implementations of it. `SimpleMessageStore`: an in-memory, Map-based implementation (the default, good for testing) and `JdbcMessageStore`: an implementation that uses a relational database via JDBC.

7. Messaging Endpoints

7.1 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 the section called “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 Section 3.1, “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 [http://www.eaipatterns.com/PollingConsumer.html] and Event Driven Consumers [http://www.eaipatterns.com/EventDrivenConsumer.html].

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.

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.

Event Driven Consumer

Because it is the simpler of the two, we will cover the Event Driven Consumer endpoint first. You may recall that the `SubscribableChannel` interface provides a `subscribe()` method and that the method accepts a `MessageHandler` parameter (as shown in the section called “SubscribableChannel”):

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

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

```
SubscribableChannel channel = context.getBean("subscribableChannel", SubscribableChannel.class);

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

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 = context.getBean("pollableChannel", PollableChannel.class);

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

PlatformTransactionManager txManager = context.getBean("exampleTxManager", PlatformTransationManager.class)
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.

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 and its attributes. 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 fixed-rate="1000"/>
</transformer>
```

As an alternative to 'fixed-rate' you can also use the 'fixed-delay' attribute.

For a poller based on a Cron expression, use the "cron" attribute instead:

```
<transformer input-channel="pollable"
             ref="transformer"
             output-channel="output">
    <poller cron="*/10 * * * * MON-FRI"/>
</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. The exception will also be thrown if you attempt to configure a poller on the element that is connected to a non-pollable channel.

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

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

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

Note

The "ref" attribute is only allowed on the inner-poller definitions. Defining this attribute on a top-level poller will result in a configuration exception thrown during initialization of the Application Context.

In fact, to simplify the configuration even further, 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" fixed-rate="3000"/>

<!-- 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 fixed-delay="1000">
    <transactional transaction-manager="txManager"
                   propagation="REQUIRED"
                   isolation="REPEATABLE_READ"
                   timeout="10000"
                   read-only="false"/>
</poller>
```

AOP Advice chains

Since Spring transaction support depends on the Proxy mechanism with `TransactionInterceptor` (AOP Advice) handling transactional behavior of the message flow initiated by the poler, some times there is a need to provide extra Advice(s) to handle other cross cutting behavior associated with the poller. For that poller defines an 'advice-chain' element allowing you to add more advices - class that implements `MethodInterceptor` interface..

```
<service-activator id="advicedSa" input-channel="goodInputWithAdvice" ref="testBean"
  method="good" output-channel="output">
 <poller max-messages-per-poll="1" fixed-rate="10000">
  <transactional transaction-manager="txManager" />
   <advice-chain>
   <ref bean="adviceA" />
   <beans:bean class="org.bar.SampleAdvice"/>
  </advice-chain>
 </poller>
</service-activator>
```

For more information on how to implement MethodInterceptor please refer to AOP sections of Spring reference manual (section 7 and 8). Advice chain can also be applied on the poller that does not have any transaction configuration essentially allowing you to enhance the behavior of the message flow initiated by the 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" fixed-rate="1000"/>

<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" fixed-rate="10"/>

</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.

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 a parameter will be mapped to a Message payload or part of the payload or header (when using the 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 the Spring 3.x ConversionService) within its own instance of a conversion service bean named *integrationConversionService*. That bean is automatically created as soon as the first converter is defined using the Spring Integration namespace support. To register a Converter all you need is to

implement `org.springframework.core.convert.converter.Converter` and define it via convenient namespace support:

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

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

or as an inner bean:

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

Asynchronous polling

If you want the polling to be asynchronous, a Poller can optionally specify a 'task-executor' attribute pointing to an existing instance of any `TaskExecutor` bean (Spring 3.0 provides a convenient namespace configuration via the `task` namespace). However, there are certain things you must understand when configuring a Poller with a TaskExecutor.

The problem is that there are two configurations in place. The *Poller* and the *TaskExecutor*, and they both have to be in tune with each other otherwise you might end up creating an artificial memory leak.

Let's look at the following configuration provided by one of the users on the Spring Integration forum (http://forum.springsource.org/showthread.php?t=94519):

```
<int:service-activator input-channel="publishChannel" ref="myService">
  <int:poller receive-timeout="5000" task-executor="taskExecutor" fixed-rate="50"/>
</si:service-activator>

<task:executor id="taskExecutor" pool-size="20" queue-capacity="20"/>
```

The above configuration demonstrates one of those out of tune configurations.

The poller keeps scheduling new tasks even though all the threads are blocked waiting for either a new message to arrive, or the timeout to expire. Given that there are 20 threads executing tasks with a 5 second timeout, they will be executed at a rate of 4 per second (5000/20 = 250ms). But, new tasks are being scheduled at a rate of 20 per second, so the internal queue in the task executor will grow at a rate of 16 per second (while the process is idle), so we essentially have a memory leak.

One of the ways to handle this is to set the `queue-capacity` attribute of the Task Executor to 0. You can also manage it by specifying what to do with messages that can not be queued by setting the `rejection-policy` attribute of the Task Executor (e.g., DISCARD). In other words there are certain details you must understand with regard to configuring the TaskExecutor. Please refer to - *Section 25 - Task Execution and Scheduling* of the Spring reference manual for more detail on the subject.

7.2 Inbound Messaging Gateways

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 below.

```
package org.cafeteria;

public interface Cafe {

  void placeOrder(Order order);

}
```

Namespace support is also provided which allows you to configure such an interface as a service as demonstrated by the following example.

```
<gateway id="cafeService"
         service-interface="org.cafeteria.Cafe"
         default-request-channel="requestChannel"
         default-reply-channel="replyChannel"/>
```

With this configuration defined, the "cafeService" can now be injected into other beans, and the code that invokes the methods on that proxied instance of the Cafe 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).

Important

Typically you don't have to specify the `default-reply-channel` since a Gateway will auto-create a temporary, anonymous reply channel where it will listen for the reply. However, there are some cases which may prompt you to define a `default-reply-channel` (or `reply-channel` with adapter gateways such as HTTP, JMS, etc.). For some background, we'll quickly discuss some of the inner-workings of the Gateway. A Gateway will create a temporary point-to-point reply channel which is anonymous and is added to the Message Headers with the name `replyChannel`. When providing an explicit `default-reply-channel` (`reply-channel` with remote adapter gateways), you have the option to point to a publish-subscribe channel, which is so named because you can add more than one subscriber to it. Internally Spring Integration will create a Bridge between the temporary `replyChannel` and the explicitly defined `default-reply-channel`. So let's say you want your reply to go not only to the gateway, but also to some other consumer. In this case you would want two things: *a) a named channel you can subscribe to and b) that channel is a publish-subscribe-channel.* The default strategy used by the gateway will not satisfy those needs, because the reply channel added to the header is anonymous and point-to-point. This means that no other subscriber can get a handle to it and even if it could, the channel has point-to-point behavior such that only one subscriber would get the Message. So by defining a `default-reply-channel` you can point to a channel of your choosing which in this case would be a `publish-subscribe-channel`. The Gateway would create a bridge from it to the temporary, anonymous reply channel that is stored in the header. Another case where you might want to provide a reply channel explicitly is for monitoring or auditing via an interceptor (e.g., wiretap). You need a named channel in order to configure a Channel Interceptor.

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);

}
```

You may alternatively provide such content in `method` sub-elements if you prefer XML configuration (see the next paragraph).

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 the XML approach of configuring Gateway methods, you can provide *method* sub-elements to the gateway configuration.

```
<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 (the 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 value will be set for the 'RESPONSE_TYPE' header based on the gateway's method.

Of course, the Gateway invocation might result in errors. By default any error that has occurred downstream will be re-thrown as a MessagingException (RuntimeException) upon the Gateway's method invocation. However there are times when you may want to simply log the error rather than propagating it, or you may want to treat an Exception as a valid reply, by mapping it to a Message that will conform to some "error message" contract that the caller understands. To accomplish this, our Gateway provides support for a Message Channel dedicated to the errors via the *error-channel* attribute. In the example below, you can see that a 'transformer' is used to create a reply Message from the Exception.

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

<si:transformer input-channel="exceptionTransformationChannel"
        ref="exceptionTransformer" method="createErrorResponse"/>
```

The *exceptionTransformer* could be a simple POJO that knows how to create the expected error response objects. That would then be the payload that is sent back to the caller. Obviously, you could do many more elaborate things in such an "error flow" if necessary. It might involve routers (including Spring Integration's ErrorMessageExceptionTypeRouter), filters, and so on. Most of the time, a simple 'transformer' should be sufficient, however.

Alternatively, you might want to only log the Exception (or send it somewhere asynchronously). If you provide a one-way flow, then nothing would be sent back to the caller. In the case that you want to completely suppress Exceptions, you can provide a reference to the global "nullChannel" (essentially a /dev/null approach). Finally, as mentioned above, if no "error-channel" is defined at all, then the Exceptions will propagate as usual.

Important

Exposing the messaging system via simple POJI Gateways obviously provides benefits, but "hiding" the reality of the underlying messaging system does come at a price so there are certain things you should consider. We want our Java method to return as quickly as possible and not hang for an indefinite amount of time while the caller is waiting on it to return (void, return value, or a thrown Exception). When regular methods are used as a proxies in front of the Messaging system, we have to take into account the potentially asynchronous nature of the underlying messaging. This means that there might be a chance that a Message that was initiated by a Gateway could be dropped by a Filter, thus never reaching a component that is responsible for producing a reply. Some Service Activator method might result in an Exception, thus providing no reply (as we don't generate Null messages). So as you can see there are multiple scenarios where a reply message might not be coming. That is perfectly natural in messaging systems. However think about the implication on the gateway method. The Gateway's method input arguments were incorporated into a Message and sent downstream. The reply Message would be converted to a return value of the Gateway's method. So you might want to ensure that for each Gateway call there will always be a reply Message. Otherwise, your Gateway method might never return and will hang indefinitely. One of the ways of handling this situation is via an Asynchronous Gateway (explained later in this section). Another way of handling it is to explicitly set the reply-timeout attribute. That way,

the gateway will not hang any longer than the time specified by the reply-timeout and will return 'null' if that timeout does elapse. Finally, you might want to consider setting downstream flags such as 'requires-reply' on a service-activator or 'throw-exceptions-on-rejection' on a filter. These options will be discussed in more detail in the final section of this chapter.

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. As you've seen, the `GatewayProxyFactoryBean` provides a convenient way to expose a Proxy over a service-interface thus giving you 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 replies 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 a `Future`. When `GatewayProxyFactoryBean` sees that the return type of the gateway method is a `Future`, it immediately switches to the async mode by utilizing an `AsyncTaskExecutor`. That is all. The call to such a method always returns immediately with a `Future` instance. Then, you can interact with the `Future` at your own pace to get the result, cancel, etc. And, as with any other use of Future instances, calling get() may reveal a timeout, an execution exception, and so on.

```
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.

Gateway behavior when no response arrives

As it was explained earlier, the Gateway provides a convenient way of interacting with a Messaging system via POJO method invocations, but realizing that a typical method invocation, which is generally expected to always return (even with an Exception), might not always map one-to-one to message exchanges (e.g., a reply message might not arrive - which is equivalent to a method not returning). It is important to go over several scenarios especially in the Sync Gateway case and understand the default behavior of the Gateway and how to deal with these scenarios to make the Sync Gateway behavior more predictable regardless of the outcome of the message flow that was initialed from such Gateway.

There are certain attributes that could be configured to make Sync Gateway behavior more predictable, but some of them might not always work as you might have expected. One of them is *reply-timeout*. So, lets look at the *reply-timeout* attribute and see how it can/can't influence the behavior of the Sync Gateway in various scenarios. We will look at single-threaded scenario (all components downstream are connected via Direct Channel) and multi-threaded scenarios (e.g., somewhere downstream you may have Pollable or Executor Channel which breaks single-thread boundary)

Long running process downstream

Sync Gateway - single-threaded. If a component downstream is still running (e.g., infinite loop or a very slow service), then setting a *reply-timeout* has no effect and the Gateway method call will not return until such downstream service exits (via return or exception). *Sync Gateway - multi-threaded*. If a component downstream is still running (e.g., infinite loop or a very slow service), in a multi-threaded message flow setting the *reply-timeout* will have an effect by allowing gateway method invocation to return once the timeout has been reached, since the `GatewayProxyFactoryBean` will simply poll on the reply channel waiting for a message until the timeout expires. However it could result in a 'null' return from the Gateway method if the timeout has been reached before the actual reply was produced. It is also important to understand that the reply message (if produced) will be sent to a reply channel after the Gateway method invocation might have returned, so you must be aware of that and design your flow with this in mind.

Downstream component returns 'null'

Sync Gateway - single-threaded. If a component downstream returns 'null' and no *reply-timeout* has been configured, the Gateway method call will hang indefinitely unless: a) a *reply-timeout* has been configured or b) the *requires-reply* attribute has been set on the downstream component (e.g., service-activator) that might return 'null'. In this case, an Exception would be thrown and propagated to the Gateway. *Sync Gateway - multi-threaded*. Behavior is the same as above.

Downstream component return signature is 'void' while Gateway method signature is non-void

Sync Gateway - single-threaded. If a component downstream returns 'void' and no *reply-timeout* has been configured, the Gateway method call will hang indefinitely unless a *reply-timeout* has been configured *Sync Gateway - multi-threaded* Behavior is the same as above.

Downstream component results in Runtime Exception (regardless of the method signature)

Sync Gateway - single-threaded. If a component downstream throws a Runtime Exception, such exception will be propagated via an Error Message back to the gateway and re-thrown. *Sync Gateway - multi-threaded* Behavior is the same as above.

Important

It is also important to understand that by default *reply-timeout* is unbounded which means that if not explicitly set there are several scenarios (described above) where your Gateway method invocation might hang indefinitely. So, make sure you analyze your flow and if there is even a remote possibility of one of these scenarios to occur, set the *reply-timeout* attribute to a 'safe' value or, even better, set the *requires-reply* attribute of the downstream component to 'true' to ensure a timely response as produced by the throwing of an Exception as soon as that downstream component does return null internally. But also, realize that there are some scenarios (see the very first one) where *reply-timeout* will not help. That means it is also important to analyze your message flow and decide when to use a Sync Gateway vs an Async Gateway. As you've seen the latter case is simply a matter of defining Gateway methods that return Future instances. Then, you are guaranteed to receive that return value, and you will have more granular control over the results of the invocation.

Also, when dealing with a Router you should remember that setting the *resolution-required* attribute to 'true' will result in an Exception thrown by the router if it can not resolve a particular channel. Likewise, when dealing with a Filter, you can set the *throw-exception-on-rejection* attribute. In both of these cases, the resulting flow will behave like that containing a service-activator with the 'requires-reply' attribute. In other words, it will help to ensure a timely response from the Gateway method invocation.

7.3 Service Activator

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.

Configuring Service Activator

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 `replyChannel` 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. It it can be resolved, the Message will be sent there. This is the technique used for Request Reply messaging in Spring Integration, and it is also an example of the Return Address pattern.

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 or @Headers annotations as described in Section B.5, “Annotation Support”

Note

The service method is not required to have any arguments at all, which means you can 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 is only used within a single definition of the `<service-activator>`, you can provide 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.

Service Activators and the Spring Expression Language (SpEL)

Since Spring Integration 2.0, Service Activators can also benefit from SpEL (http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/expressions.html).

For example, you may now invoke any bean method without pointing to the bean via a `ref` attribute or including it as an inner bean definition. For example:

```
<int:service-activator input-channel="in" output-channel="out"
 expression="@accountService.processAccount(payload, headers.accountId)"/>

 <bean id="accountService" class="foo.bar.Account"/>
```

In the above configuration instead of injecting 'accountService' using a `ref` or as an inner bean, we are simply using SpEL's `@beanId` notation and invoking a method which takes a type compatible with Message payload. We are also passing a header value. As you can see, any valid SpEL expression can be evaluated against any content in the Message. For simple scenarios your *Service Activators* do not even have to reference a bean if all logic can be encapsulated by such an expression.

```
<int:service-activator input-channel="in" output-channel="out" expression="payload * 2"/>
```

In the above configuration our service logic is to simply multiply the payload value by 2, and SpEL lets us handle it relatively easy.

7.4 Delayer

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.

Configuring Delayer

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"/>
```

7.5 Groovy support

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

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 the Groovy script. To enable Groovy support Spring Integration defines a `GroovyScriptExecutingMessageProcessor` which will bind the Message Payload as a `payload` variable and the Message Headers as a `headers` variable within the script execution context. All that is left for you to do is write a script that uses those variables. Below are a 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>
```

Here, you see that the script can be included inline or via the `location` attribute using the groovy namespace support.

Other supported elements are *router, service-activator, transformer, and splitter. The configuration would look identical to that above other than the main element's name.*

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

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

In the above example any invocations that occur within the 5 seconds immediately following the updating of the script would still be using the old script. However, any invocation that occurs after those 5 seconds have elapsed will result in execution of 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 any script modifications as soon as such modification occurs. Basically this is an example of 'real-time' configuration and might not be the most efficient option (but could be useful during development).

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

Any negative number value means the script will never be refreshed after initial initialization of the application context. This is the default behavior. In this case, the "dynamic" aspect of Groovy is not being used, but the syntax might be the primary reason that Groovy has been chosen in the first place.

Important

Inline defined scripts can not be reloaded.

Custom bindings

You already know that by default, 'payload' and 'headers' will be bound as Groovy binding variables. However, some times in order to take the most out of Groovy you may want to customize Groovy bindings (e.g., include extra variables pointing to some scalar values or bind some beans as variables). To support this requirement we have defined a simple strategy: ScriptVariableGenerator.

```
public interface ScriptVariableGenerator {

 Map<String, Object> generateScriptVariables(Message<?> message);

}
```

As you can see the only method to implement is `generateScriptVariables(Message)`. It takes the Message as an argument. That allows you to use data available in the Message payload and/or headers. The return value is the Map of variables that will be bound to the script's evaluation context. This method will be called every time the script is executed, corresponding to the processing of that particular Message. We also provide a default implementation and namespace based configuration for simple bindings via <variable> sub-elements (see below):

```
<groovy:script location="foo/bar/MyScript.groovy">
```

```
 <groovy:variable name="foo" value="foo"/>
 <groovy:variable name="bar" value="bar"/>
 <groovy:variable name="date" ref="date"/>
</groovy:script>
```

As you can see similar to other constructs in Spring, when binding each of these variables you can either provide a scalar value or reference another bean in the Application Context.

If you need more control over how a particular variable is generated, then all you need to do is provide your own implementation of ScriptVariableGenerator and reference it with the `script-variable-generator` attribute:

```
<groovy:script location="foo/bar/MyScript.groovy"
    script-variable-generator="variableGenerator"/>

<bean id="variableGenerator" class="foo.bar.MyScriptVariableGenerator"/>
```

Important

The `script-variable-generator` attribute and <variable> sub-element(s) are mutually exclusive. You can use at most one of them. Also, the `script-variable-generator` and <variable> sub-elements cannot be used with an inline script, but rather only when pointing to the script via the `location` attribute.

If you need to customize the Groovy object itself, beyond setting variables, you can reference a bean that implementats `org.springframework.scripting.groovy.GroovyObjectCustomizer` via the `customizer` attribute. For example, this might be useful if you want to configure a domain-specific language (DSL) by modifying the MetaClass and registering functions to be available within the script.

```
<service-activator input-channel="groovyChannel">
    <groovy:script location="foo/SomeScript.groovy" customizer="groovyCustomizer"/>
</service-activator>

<beans:bean id="groovyCustomizer" class="org.foo.MyGroovyObjectCustomizer"/>
```

Setting a custom GroovyObjectCustomizer is not mutually exclusive with <variable> sub-elements or the `script-variable-generator` attribute. It can also be provided when defining an inline script.

Control Bus

As described in (EIP [http://www.eaipatterns.com/ControlBus.html]), 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. One option for those operations is Groovy scripts.

```
 <groovy:control-bus input-channel="operationChannel"/>
```

The Control Bus has an input channel that can be accessed for invoking operations on the beans in the application context.

The Groovy Control Bus executes messages on the input channel as Groovy scripts. It takes a message, compiles the body to a Script, customizes it with a GroovyObjectCustomizer, and then executes it. The Control Bus' customizer exposes all the beans in the application context that are annotated with @ManagedResource, implement Spring's Lifecycle interface or extend Spring's CustomizableThreadCreator base class (e.g. several of the TaskExecutor and TaskScheduler implementations).

If you need to further customize the Groovy objects, you can also provide a reference to a bean that implements `org.springframework.scripting.groovy.GroovyObjectCustomizer` via the `customizer` attribute.

```
<groovy:control-bus input-channel="input"
        output-channel="output"
        customizer="groovyCustomizer"/>

<beans:bean id="groovyCustomizer" class="org.foo.MyGroovyObjectCustomizer"/>
```

8. System Management

8.1 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.

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-filter="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.

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"/>
```

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" fixed-rate="5000"/>
</jmx:attribute-polling-channel-adapter>
```

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.

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

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 "org.springframework.integration".

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

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

Once the exporter is defined start up your application with

```
-Dcom.sun.management.jmxremote
    -Dcom.sun.management.jmxremote.port=6969
    -Dcom.sun.management.jmxremote.ssl=false
    -Dcom.sun.management.jmxremote.authenticate=false
```

Then start JConsole (free with the JDK), and connect to the local process on `localhost:6969` to get a look at the management endpoints exposed. (The port and client are just examples to get you started quickly, there are other JMX clients available and some offer more sophisticated features than JConsole.)

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, and certain other components in Spring Integration, using the standard `<context:mbean-export/>` tag. The exporter has a couple of useful metrics attached to it, for instance a count of the number of active handlers and the number of queued messages (these would both be important if you wanted to shutdown the context without losing any messages).

MBean ObjectNames

All the MessageChannel, MessageHandler and MessageSource instances in the application are wrapped by the MBean exporter to provide management and monitoring features. For example, MessageChannel send The generated JMX object names for each component type are listed in the table below

Table 8.1.

Component Type	ObjectName
MessageChannel	org.springframework.integration:type=MessageChannel,name=<ch
MessageSource	org.springframework.integration:type=MessageSource,name=<cha
MessageHandler	org.springframework.integration:type=MessageSource,name=<cha

The "bean" attribute in the object names for sources and handlers takes one of the values in the table below

Table 8.2.

Bean Value	Description
endpoint	The bean name of the enclosing endpoint (e.g. <service-activator>) if there is one
anonymous	An indication that the enclosing endpoint didn't have a user-specified bean name, so the JMX name is the input channel name
internal	For well-known Spring Integration default components
handler	None of the above: fallback to the `toString()` of the object being monitored (handler or source)

MessageChannel MBean Features

Message channels report metrics according to their concrete type. If you are looking at a `DirectChannel` you will see statistics for the send operation. If it is a `QueueChannel` you will also see statistics for the receive operation. In both cases there are some metrics that are simple counters (message count and error count), and some that are estimates of averages of interesting quantities. The algorithms used to calculate these estimates are described briefly in the table below:

Table 8.3.

Metric Type	Example	Algorithm
Count	Send Count	Simple incrementer. Increase by one when an event occurs.
Duration	Send Duration (method execution time in milliseconds)	Exponential Moving Average with decay factor 10. Average

Metric Type	Example	Algorithm
		of the method execution time over roughly the last 10 measurements.
Rate	Send Rate (number of operations per second)	Inverse of Exponential Moving Average of the interval between events with decay in time (lapsing over 60 seconds) and per measurement (last 10 events).
Ratio	Send Error Ratio (ratio of errors to total sends)	Estimate the success ratio as the Exponential Moving Average of the series composed of values 1 for success and 0 for failure (decaying as per the rate measurement over time and events). Error ratio is 1 - success ratio.

A feature of the time-based average estimates is that they decay with time if no new measurements arrive. To help interpret the behaviour over time, the time (in seconds) since the last measurement is also exposed as a metric.

There are two basic exponential models: decay per measurement (appropriate for duration and anything where the number of measurements is part of the metric), and decay per time unit (more suitable for rate measurements where the time in between measurements is part of the metric). Both models depend on the fact that

```
S(n) = sum(i=0,i=n) w(i) x(i)
```

has a special form when `w(i) = r^i`, with `r=constant`:

```
S(n) = x(n) + r S(n-1)
```

(so you only have to store `S(n-1)`, not the whole series `x(i)`, to generate a new metric estimate from the last measurement). The algorithms used in the duration metrics use `r=exp(-1/M)` with `M=10`. The net effect is that the estimate `S(n)` is more heavily weighted to recent measurements and is composed roughly of the last `M` measurements. So `M` is the "window" or lapse rate of the estimate In the case of the vanilla moving average, `i` is a counter over the number of measurements. In the case of the rate we interpret `i` as the elapsed time, or a combination of elapsed time and a counter (so the metric estimate contains contributions roughly from the last `M` measurements and the last `T` seconds).

8.2 Message History

The key benefit of a messaging architecture is loose coupling where participating components do not maintain any awareness about one another. This fact alone makes your application extremely flexible,

allowing you to change components without affecting the rest of the flow, change messaging routes, message consuming styles (polling vs event driven), and so on. However, this unassuming style of architecture could prove to be difficult when things go wrong. When debugging, you would probably like to get as much information about the message as you can (its origin, channels it has traversed, etc.)

Message History is one of those patterns that helps 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 the Message History by adding a header to the Message and updating that header every time a message passes through a tracked component.

Message History Configuration

To enable Message History all you need is to define the `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. Its value is very simple - `List<Properties>`.

```
<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"));
```

You might not want to track all of the components. To limit the history to certain components based on their names, all you need is provide the `tracked-components` attribute and specify a comma-delimited list of component names and/or patterns that match the components 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', start with 'sample', or match the name 'foo' exactly.

Note

Remember that by definition the Message History header is immutable (you can't re-write history, although some try). Therefore, when writing Message History values, the components are either creating brand new Messages (when the component is an origin), or they are copying the history from a request Message, modifying it and setting the new list on a reply Message. In either case, the values can be appended even if the Message itself is crossing thread boundaries. That means that the history values can greatly simplify debugging in an asynchronous message flow.

8.3 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.

```
<control-bus input-channel="operationChannel"/>
```

The Control Bus has an input channel that can be accessed for invoking operations on the beans in the application context. It also has all the common properties of a service activating endpoint, e.g. you can specify an output channel if the result of the operation has a return value that you want to send on to a downstream channel.

The Control Bus executes messages on the input channel as Spring Expression Language expressions. It takes a message, compiles the body to an expression, adds some context, and then executes it. The default context supports any method that has been annotated with @ManagedAttribute or @ManagedOperation. It also supports the methods on Spring's Lifecycle interface, and it supports methods that are used to configure several of Spring's TaskExecutor and TaskScheduler implementations. The simplest way to ensure that your own methods are available to the Control Bus is to use the @ManagedAttribute and/or @ManagedOperation annotations. Since those are also used for exposing methods to a JMX MBean registry, it's a convenient by-product (often the same types of operations you want to expose to the Control Bus would be reasonable for exposing via JMS). Resolution of any particular instance within the application context is achieved in the typical SpEL syntax. Simply provide the bean name with the SpEL prefix for beans (@). For example, to execute a method on a Spring Bean a client could send a message to the operation channel as follows:

```
Message operation = MessageBuilder.withPayload("@myServiceBean.shutdown()").build();
operationChannel.send(operation)
```

The root of the context for the expression is the `Message` itself, so you also have access to the 'payload' and 'headers' as variables within your expression. This is consistent with all the other expression support in Spring Integration endpoints.

Part IV. Integration Adapters

This section covers the various Channel Adapters and Messaging Gateways provided by Spring Integration to support Message-based communication with external systems.

9. Spring ApplicationEvent Support

Spring Integration provides support for inbound and outbound `ApplicationEvents` as defined by the underlying Spring Framework. For more information about Spring's support for events and listeners, refer to the Spring Reference Manual [http://static.springsource.org/spring/docs/2.5.x/reference/beans.html#context-functionality-events].

9.1 Receiving Spring ApplicationEvents

To receive events and send them to a channel, simply define an instance of Spring Integration's `ApplicationEventListeningMessageProducer`. 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. If a received event has a Message instance as its 'source', then that will be passed as-is. Otherwise, if a SpEL-based "payloadExpression" has been provided, that will be evaluated against the ApplicationEvent instance. If the event's source is not a Message instance and no "payloadExpression" has been provided, then the ApplicationEvent itself will be passed as the payload.

For convenience namespace support is provided to configure an `ApplicationEventListeningMessageProducer` via the *inbound-channel-adapter* element.

```
<int-event:inbound-channel-adapter channel="eventChannel"
                                   error-channel="eventErrorChannel"
                                   event-types="example.FooEvent, example.BarEvent"/>

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

In the above example, all Application Context events that match one of the types specified by the 'event-types' (optional) attribute will be delivered as Spring Integration Messages to the Message Channel named 'eventChannel'. If a downstream component throws an exception, a MessagingException containing the failed message and exception will be sent to the channel named 'eventErrorChannel'. If no "error-channel" is specified and the downstream channels are synchronous, the Exception will be propagated to the caller.

9.2 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 is provided to configure an `ApplicationEventPublishingMessageHandler` via the *outbound-channel-adapter* element.

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

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

If you are using a PollableChannel (e.g., Queue), you can also provide *poller* as a sub-element of the *outbound-channel-adapter* element. You can also optionally provide a *task-executor* reference for that poller. The following example demonstrates both.

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

<int-event:outbound-channel-adapter channel="eventChannel">
  <int:poller max-messages-per-poll="1" task-executor="executor" fixed-rate="100"/>
</int-event:outbound-channel-adapter>

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

In the above example, all messages sent to the 'eventChannel' channel will be published as ApplicationEvents to any relevant ApplicationListener instances that are registered within the same Spring ApplicationContext. If the payload of the Message is an ApplicationEvent, it will be passed as-is. Otherwise the Message itself will be wrapped in a MessagingEvent instance.

10. Feed Adapter

Spring Integration provides support for Syndication via Feed Adapters

10.1 Introduction

Web syndication is a form of publishing material such as news stories, press releases, blog posts, and other items typically available on a website but also made available in a feed format such as RSS or ATOM.

Spring integration provides support for Web Syndication via its 'feed' adapter and provides convenient namespace-based configuration for it. To configure the 'feed' namespace, include the following elements within the headers of your XML configuration file:

```
xmlns:int-feed="http://www.springframework.org/schema/integration/feed"
xsi:schemaLocation="http://www.springframework.org/schema/integration/feed
 http://www.springframework.org/schema/integration/feed/spring-integration-feed-2.0.xsd"
```

10.2 Feed Inbound Channel Adapter

The only adapter that is really needed to provide support for retrieving feeds is an *inbound channel adapter*. This allows you to subscribe to a particular URL. Below is an example configuration:

```
<int-feed:inbound-channel-adapter id="feedAdapter"
  channel="feedChannel"
  url="http://feeds.bbci.co.uk/news/rss.xml">
 <int:poller fixed-rate="10000" max-messages-per-poll="100" />
</int-feed:inbound-channel-adapter>
```

In the above configuration, we are subscribing to a URL identified by the `url` attribute.

As news items are retrieved they will be converted to Messages and sent to a channel identified by the `channel` attribute. The payload of each message will be a `com.sun.syndication.feed.synd.SyndEntry` instance. That encapsulates various data about a news item (content, dates, authors, etc.).

You can also see that the *Inbound Feed Channel Adapter* is a Polling Consumer. That means you have to provide a poller configuration. However, one important thing you must understand with regard to Feeds is that its inner-workings are slightly different then most other poling consumers. When an Inbound Feed adapter is started, it does the first poll and receives a `com.sun.syndication.feed.synd.SyndEntryFeed` instance. That is an object that contains multiple `SyndEntry` objects. Each entry is stored in the local entry queue and is released based on the value in the `max-messages-per-poll` attribute such that each Message will contain a single entry. If during retrieval of the entries from the entry queue the queue had become empty, the adapter will attempt to update the Feed thereby populating the queue with more entries (SyndEntry instances) if available. Otherwise the next attempt to poll for a feed will be determined by the trigger of the poller (e.g., every 10 seconds in the above configuration).

Duplicate Entries

Polling for a Feed might result in entries that have already been processed ("I already read that news item, why are you showing it to me again?"). Spring Integration provides a convenient mechanism to eliminate the need to worry about duplicate entries. Each feed entry will have a *published date* field. Every time a new Message is generated and sent, Spring Integration will store the value of the latest *published date* in an instance of the `org.springframework.integration.store.MetadataStore` strategy. The MetadataStore interface is designed to store various types of generic meta-data (e.g., published date of the last feed entry that has been processed) to help components such as this Feed adapter deal with duplicates.

The default rule for locating this metadata store is as follows: Spring Integration will look for a bean of type `org.springframework.integration.store.MetadataStore` in the ApplicationContext. If one is found then it will be used, otherwise it will create a new instance of `SimpleMetadataStore` which is an in-memory implementation that will only persist metadata within the lifecycle of the currently running Application Context. This means that upon restart you may end up with duplicate entries. If you need to persist metadata between Application Context restarts, you may use the `PropertiesPersistingMetadataStore` which is backed by a properties file and a properties-persister. Alternatively, you could provide your own implementation of the `MetadataStore` interface (e.g. JdbcMetadataStore) and configure it as bean in the Application Context.

```
<bean id="metadataStore"
    class="org.springframework.integration.store.PropertiesPersistingMetadataStore"/>
```

11. File Support

11.1 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.

11.2 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="${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="${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 filename-pattern or filename-regex 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:${input.directory}"
        p:filter-ref="compositeFilter"/>
<bean id="compositeFilter" class="org.springframework.integration.file.filters.CompositeFileListFilter">
    <constructor-arg>
        <list>
            <bean class="org.springframework.integration.file.filters.AcceptOnceFileListFilter" />
            <bean class="org.springframework.integration.file.filters.RegexPatternFileListFilter">
                <constructor-arg value="^test.*$"/>
            </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-3.0.xsd
      http://www.springframework.org/schema/integration
      http://www.springframework.org/schema/integration/spring-integration-2.0.xsd
      http://www.springframework.org/schema/integration/file
      http://www.springframework.org/schema/integration/file/spring-integration-file-2.0.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="filesIn1"
      directory="file:${input.directory}" prevent-duplicates="true"/>

<file:inbound-channel-adapter id="filesIn2"
       directory="file:${input.directory}"
       filter="customFilterBean" />

<file:inbound-channel-adapter id="filesIn3"
       directory="file:${input.directory}"
       filename-pattern="test*" />

<file:inbound-channel-adapter id="filesIn4"
        directory="file:${input.directory}"
        filename-regex="test[0-9]+\.txt" />
```

The first channel adapter is relying on the default filter that just prevents duplication, the second is using a custom filter, the third is using the *filename-pattern* attribute to add an `AntPathMatcher` based filter, and the fourth is using the *filename-regex* attribute to add a regular expression Pattern based filter to the `FileReadingMessageSource`. The *filename-pattern* and *filename-regex* attributes are each mutually exclusive with the regular *filter* reference attribute. However, you can use the *filter* attribute to reference an instance of `CompositeFileListFilter` that combines any number of filters, including one or more pattern based filters 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.

11.3 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="${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="${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`.

11.4 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).

12. FTP/FTPS Adapters

Spring Integration provides support for file transfer operations via FTP and FTPS.

12.1 Introduction

The File Transfer Protocol (FTP) is a simple network protocol which allows you to transfer files between two computers on the Internet.

There are two actors when it comes to FTP communication: *client* and *server*. To transfer files with FTP/FTPS, you use a *client* which initiates a connection to a remote computer that is running an FTP *server*. After the connection is established, the *client* can choose to send and/or receive copies of files.

Spring Integration supports sending and receiving files over FTP/FTPS by providing two types of *client* side adapters: *Inbound Channel Adapter* and *Outbound Channel Adapter*. It also provides convenient namespace-based configuration options for defining these *client* components.

To use the *FTP* namespace, add the following to the header of your XML file:

```
xmlns:ftp="http://www.springframework.org/schema/integration/ftp"
xsi:schemaLocation="http://www.springframework.org/schema/integration/ftp
 http://www.springframework.org/schema/integration/ftp/spring-integration-ftp-2.0.xsd"
```

12.2 FTP Session Factory

Before configuring FTP adapters you must configure an *FTP Session Factory*. You can configure the *FTP Session Factory* with a regular bean definition where the implementation class is `org.springframework.integration.ftp.session.DefaultFtpSessionFactory`: Below is a basic configuration:

```
<bean id="ftpClientFactory"
 class="org.springframework.integration.ftp.session.DefaultFtpSessionFactory">
  <property name="host" value="localhost"/>
  <property name="port" value="22"/>
  <property name="username" value="kermit"/>
  <property name="password" value="frog"/>
  <property name="clientMode" value="0"/>
  <property name="fileType" value="2"/>
  <property name="bufferSize" value="100000"/>
</bean>
```

For FTPS connections all you need to do is use `org.springframework.integration.ftp.session.DefaultFtpsSessionFactory` instead. Below is the complete configuration sample:

```
<bean id="ftpClientFactory"
 class="org.springframework.integration.ftp.client.DefaultFtpsClientFactory">
  <property name="host" value="localhost"/>
  <property name="port" value="22"/>
  <property name="username" value="oleg"/>
  <property name="password" value="password"/>
  <property name="clientMode" value="1"/>
```

```
  <property name="fileType" value="2"/>
  <property name="useClientMode" value="true"/>
  <property name="cipherSuites" value="a,b.c"/>
  <property name="keyManager" ref="keyManager"/>
  <property name="protocol" value="SSL"/>
  <property name="trustManager" ref="trustManager"/>
  <property name="prot" value="P"/>
  <property name="needClientAuth" value="true"/>
  <property name="authValue" value="oleg"/>
  <property name="sessionCreation" value="true"/>
  <property name="protocols" value="SSL, TLS"/>
  <property name="implicit" value="true"/>
</bean>
```

Every time an adapter requests a session object from its `SessionFactory` the session is returned from a session pool maintained by a caching wrapper around the factory. A Session in the session pool might go stale (if it has been disconnected by the server due to inactivity) so the `SessionFactory` will perform validation to make sure that it never returns a stale session to the adapter. If a stale session was encountered, it will be removed from the pool, and a new one will be created.

Note

If you experience connectivity problems and would like to trace Session creation as well as see which Sessions are polled you may enable it by setting the logger to TRACE level (e.g., log4j.category.org.springframework.integration.file=TRACE)

Now all you need to do is inject these session factories into your adapters. Obviously the protocol (FTP or FTPS) that an adapter will use depends on the type of session factory that has been injected into the adapter.

Note

A more practical way to provide values for *FTP/FTPS Session Factories* is by using Spring's property placeholder support (See: http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-factory-placeholderconfigurer).

12.3 FTP Inbound Channel Adapter

The *FTP Inbound Channel Adapter* is a special listener that will connect to the FTP server and will listen for the remote directory events (e.g., new file created) at which point it will initiate a file transfer.

```
<int-ftp:inbound-channel-adapter id="ftpInbound"
    channel="ftpChannel"
    session-factory="ftpSessionFactory"
    charset="UTF-8"
    auto-create-local-directory="true"
    delete-remote-files="true"
    filename-pattern="*.txt"
    remote-directory="some/remote/path"
    remote-file-separator="/"
    local-directory=".">
   <int:poller fixed-rate="1000"/>
</int-ftp:inbound-channel-adapter>
```

As you can see from the configuration above you can configure an *FTP Inbound Channel Adapter* via the `inbound-channel-adapter` element while also providing values for various attributes such as

`local-directory`, `filename-pattern` (which is based on simple pattern matching, not regular expressions), and of course the reference to a `session-factory`.

Some times file filtering based on the simple pattern specified via `filename-pattern` attribute might not be sufficient. If this is the case, you can use the `filename-regex` attribute to specify a Regular Expression (e.g. `filename-regex=".*\.test$"`). And of course if you need complete control you can use `filter` attribute and provide a reference to any custom implementation of the `org.springframework.integration.file.filters.FileListFilter`, a strategy interface for filtering a list of files.

Note

As of Spring Integration 2.0.2, we have added a 'remote-file-separator' attribute. That allows you to configure a file separator character to use if the default '/' is not applicable for your particular environment.

Please refer to the schema for more details on these attributes.

It is also important to understand that the *FTP Inbound Channel Adapter* is a *Polling Consumer* and therefore you must configure a poller (either via a global default or a local sub-element). Once a file has been transferred, a Message with a `java.io.File` as its payload will be generated and sent to the channel identified by the `channel` attribute.

More on File Filtering and Large Files

Some times the file that just appeared in the monitored (remote) directory is not complete. Typically such a file will be written with temporary extension (e.g., foo.txt.writing) and then renamed after the writing process finished. As a user in most cases you are only interested in files that are complete and would like to filter only files that are complete. To handle these scenarios you can use the filtering support provided by the `filename-pattern`, `filename-regex` and `filter` attributes. Here is an example that uses a custom Filter implementation.

```
<int-ftp:inbound-channel-adapter
   channel="ftpChannel"
   session-factory="ftpSessionFactory"
   filter="customFilter"
   local-directory="file:/my_transfers">
   remote-directory="some/remote/path"
 <int:poller fixed-rate="1000"/>
</int-ftp:inbound-channel-adapter>

<bean id="customFilter" class="org.example.CustomFilter"/>
```

Poller configuration notes for the inbound FTP adapter

The job of the inbound FTP adapter consists of two tasks: *1) Communicate with a remote server in order to transfer files from a remote directory to a local directory. 2) For each transferred file, generate a Message with that file as a payload and send it to the channel identified by the 'channel' attribute.* That is why they are called 'channel-adapters' rather than just 'adapters'. The main job of such an adapter is to generate a Message to be sent to a Message Channel. Essentially, the second task mentioned above takes precedence in such a way that *IF* your local directory already has one or more files it will first

generate Messages from those, and *ONLY* when all local files have been processed, will it initiate the remote communication to retrieve more files.

Also, when configuring a trigger on the poller you should pay close attention to the `max-messages-per-poll` attribute. Its default value is 1 for all `SourcePollingChannelAdapter` instances (including FTP). This means that as soon as one file is processed, it will wait for the next execution time as determined by your trigger configuration. If you happened to have one or more files sitting in the `local-directory`, it would process those files before it would initiate communication with the remote FTP server. And, if the `max-messages-per-poll` were set to 1 (default), then it would be processing only one file at a time with intervals as defined by your trigger, essentially working as *one-poll = one-file*.

For typical file-transfer use cases, you most likely want the opposite behavior: to process all the files you can for each poll and only then wait for the next poll. If that is the case, set `max-messages-per-poll` to -1. Then, on each poll, the adapter will attempt to generate as many Messages as it possibly can. In other words, it will process everything in the local directory, and then it will connect to the remote directory to transfer everything that is available there to be processed locally. Only then is the poll operation considered complete, and the poller will wait for the next execution time.

You can alternatively set the 'max-messages-per-poll' value to a positive value indicating the upward limit of Messages to be created from files with each poll. For example, a value of 10 means that on each poll it will attempt to process no more than 10 files.

12.4 FTP Outbound Channel Adapter

The *FTP Outbound Channel Adapter* relies upon a `MessageHandler` implementation that will connect to the FTP server and initiate an FTP transfer for every file it receives in the payload of incoming Messages. It also supports several representations of the *File* so you are not limited only to java.io.File typed payloads. The *FTP Outbound Channel Adapter* supports the following payloads: 1) `java.io.File` - the actual file object; 2) `byte[]` - a byte array that represents the file contents; and 3) `java.lang.String` - text that represents the file contents.

```
<int-ftp:outbound-channel-adapter id="ftpOutbound"
    channel="ftpChannel"
    session-factory="ftpSessionFactory"
    charset="UTF-8"
    remote-file-separator="/"
    filename-generator="fileNameGenerator"/>
```

As you can see from the configuration above you can configure an *FTP Outbound Channel Adapter* via the `outbound-channel-adapter` element while also providing values for various attributes such as `filename-generator` (an implementation of the `org.springframework.integration.file.FileNameGenerator` strategy interface), a reference to a `client-factory`, as well as other attributes. Please refer to the schema for more details on the available attributes.

Note

By default Spring Integration will use `org.springframework.integration.file.DefaultFileNameGenerator`

if none is specified. `DefaultFileNameGenerator` will determine the file name based on the value of the `file_name` header (if it exists) in the MessageHeaders, or if the payload of the Message is already a `java.io.File`, then it will use the original name of that file.

Important

Defining certain values (e.g., remote-directory) might be platform/ftp server dependent. For example as it was reported on this forum http://forum.springsource.org/showthread.php?p=333478&posted=1#post333478 on some platforms you must add slash to the end of the directory definition (e.g., remote-directory="/foo/bar/" instead of remote-directory="/foo/bar")

13. HTTP Support

13.1 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`.

13.2 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 14.2, “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.inbound.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 [http://static.springsource.org/spring/docs/2.5.x/reference/mvc.html#mvc-multipart].

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.inbound.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>
</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.

13.3 Http Outbound Gateway

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

```
<bean id="httpOutbound"
  class="org.springframework.integration.http.outbound.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.outbound.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.

13.4 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"/>
```

Mapping URI variables

If your URL contains URI variables you can map them using `uri-variable` sub element in *Http Outbound Gateway* configuration.

```
<http:outbound-gateway id="trafficGateway"
```

```
    url="http://local.yahooapis.com/trafficData?appid=YdnDemo&amp;zip={zipCode}"
                request-channel="trafficChannel"
                http-method="GET"
                expected-response-type="java.lang.String">
  <http:uri-variable name="zipCode" expression="payload.getZip()"/>
</http:outbound-gateway>
```

The `uri-variable` defines two attributes `expression` and `value`. You generally use the `value` attribute for literal values, but if the value you are trying to inject is dynamic and requires access to Message data you can use a SpEL expression via the `expression` attribute. In the above configuration the `getZip()` method will be invoked on the payload object of the Message and the result of that method will be used as the value for URI variable named 'zipCode'.

13.5 HTTP Header Mappings

Spring Integration provides support for Http Header mapping for both HTTP Request and HTTP Responses.

By default all standard Http Headers as defined here http://en.wikipedia.org/wiki/List_of_HTTP_header_fields will be mapped from the message to HTTP request/response headers without further configuration. However if you do need further customization you may provide additional configuration via convenient namespace support. You can provide a comma-separated list of header names, and you can also include simple patterns with the '*' character acting as a wildcard. If you do provide such values, it will override the default behavior. Basically, it assumes you are in complete control at that point. However, if you do want to include all of the standard HTTP headers, you can use the shortcut patterns: HTTP_REQUEST_HEADERS and HTTP_RESPONSE_HEADERS. Here are some examples:

```
<int-http:outbound-gateway id="httpGateway"
   url="http://localhost/test2"
   mapped-request-headers="foo, bar"
   mapped-response-headers="X-*, HTTP_RESPONSE_HEADERS"
   channel="someChannel"/>

<int-http:outbound-channel-adapter id="httpAdapter"
   url="http://localhost/test2"
   mapped-request-headers="foo, bar, HTTP_REQUEST_HEADERS"
   channel="someChannel"/>
```

The adapters and gateways will use the `DefaultHttpHeaderMapper` which now provides two static factory methods for "inbound" and "outbound" adapters so that the proper direction can be applied (mapping HTTP requests/responses IN/OUT as appropriate).

If further customization is required you can also configure a `DefaultHttpHeaderMapper` independently and inject it into the adapter via the `header-mapper` attribute.

```
<int-http:outbound-gateway id="httpGateway"
   url="http://localhost/test2"
   header-mapper="headerMapper"
   channel="someChannel"/>

<bean id="headerMapper" class="org.springframework.integration.http.support.DefaultHttpHeaderMapper">
  <property name="inboundHeaderNames" value="foo*, *bar, baz"/>
  <property name="outboundHeaderNames" value="a*b, d"/>
```

```
</bean>
```

Of course, you can even implement the HeaderMapper strategy interface directly and provide a reference to that if you need to do something other than what the `DefaultHttpHeaderMapper` supports.

13.6 HTTP Samples

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 with a Spring Integration HTTP Inbound Adapter. All we are doing is creating a `MultiValueMap` and populating it with multi-part data. The `RestTemplate` will take care of the rest (no pun intended) by converting it to a `MultipartHttpServletRequest` . This particular client will send a multipart HTTP Request which contains the name of the company as well as an image file with the company logo.

```
RestTemplate template = new RestTemplate();
String uri = "http://localhost:8080/multipart-http/inboundAdapter.htm";
Resource s2logo =
   new ClassPathResource("org/springframework/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.MultipartReceiver"/>
</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 that is a `LinkedMultiValueMap`. We then are parsing that in the 'multipartReceiver' service-activator;

```
public void receive(LinkedMultiValueMap<String, Object> multipartRequest){
  System.out.println("### Successfully received 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 received multipart request ###
   company - SpringSource
   company-logo - as UploadedMultipartFile: spring09_logo.png
```

14. HttpInvoker Support

14.1 Introduction

Important

This module is DEPRECATED and will be removed in future releases of Spring Integration. Please use HTTP support (Chapter 13, *HTTP Support*) which should provide all the functionality that was available with Http Invoker plus more.

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.

14.2 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.

14.3 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.

14.4 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-2.0.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"/>
```

15. Mail Support

15.1 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 = context.getBean("mailSender", JavaMailSender.class);

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 the section called “Configuring Transformer with XML” 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`

15.2 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.

15.3 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-3.0.xsd
            http://www.springframework.org/schema/integration/mail
            http://www.springframework.org/schema/integration/mail/spring-integration-mail-2.0.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" fixed-rate="5000"/>
</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"/>
```

... where *javaMailProperties* could be provided by creating and populating a regular `java.utils.Properties` object. For example via *util* namespace provided by Spring.

```
<util:properties id="javaMailProperties">
  <prop key="mail.imap.socketFactory.class">javax.net.ssl.SSLSocketFactory</prop>
  <prop key="mail.imap.socketFactory.fallback">false</prop>
  <prop key="mail.store.protocol">imaps</prop>
  <prop key="mail.debug">false</prop>
</util:properties>
```

IMAP IDLE and lost connection

When using IMAP IDLE channel adapter there might be situations where connection to the server may be lost (e.g., network failure) and since Java Mail documentation explicitly states that the actual IMAP API is EXPERIMENTAL it is important to understand the differences in the API and how to deal with them when configuring IMAP IDLE adapters. Currently Spring Integration Mail adapters was tested with Java Mail 1.4.1 and Java Mail 1.4.3 and depending on which one is used special attention must be payed to some of the java mail properties that needs to be set with regard to auto-reconnect.

The following behavior was observed with GMAIL but should provide you with some tips on how to solve re-connect issue with other providers, however feedback is always welcome. Again, below notes are based on GMAIL.

With Java Mail 1.4.1 if `mail.imaps.timeout` property is set for a relatively short period of time (e.g., ~ 5 min) then `IMAPFolder.idle()` will throw `FolderClosedException` after this timeout. However if this property is not set (should be indefinite) the behavior that was observed is that `IMAPFolder.idle()` method never returns nor it throws an exception. It will however reconnect automatically if connection was lost for a short period of time (e.g., under 10 min), but if connection was lost for a long period of time (e.g., over 10 min), then `IMAPFolder.idle()` will not throw `FolderClosedException` nor it will re-establish connection and will remain in the blocked state indefinitely, thus leaving you no possibility to reconnect without restarting the adapter. So the only way to make re-connect to work with Java Mail 1.4.1 is to set `mail.imaps.timeout` property explicitly to some value, but it also means that such value shoudl be relatively short (under 10 min) and the connection should be re-estabished relatively quickly. Again, it may be different with other providers. With Java Mail 1.4.3 there was significant improvements to the API ensuring that there will always be a condition which will force `IMAPFolder.idle()` method to return via `StoreClosedException` or `FolderClosedException` or simply return, thus allowing us to proceed with auto-reconnect. Currently auto-reconnect will run infinitely making attempts to reconnect every 10 sec.

Important

In both configurations `channel` and `should-delete-messages` are the *REQUIRED* attributes. The important thing to understand is why `should-delete-messages` is required. The issue is with the POP3 protocol, which does NOT have any knowledge of messages that were READ. It can only know what's been read within a single session. This means that when your POP3 mail adapter is running, emails are successfully consumed as as they become available during each poll and no single email message will be delivered more then once. However, as soon as you restart your adapter and begin a new session all the email messages that might have been retrieved in the previous session will be retrieved again. That is the nature of POP3. Some might argue that `should-delete-messages` should be TRUE by default. In other words, there are two valid and mutually exclusive use cases which make it very hard to pick a single "best" default. You may want to configure your adapter as the only email receiver in which case you want to be able to restart such adapter without fear that messages that were delivered before will not be redelivered again.

In this case setting `should-delete-messages` to TRUE would make most sense. However, you may have another use case where you may want to have multiple adapters that simply monitor email servers and their content. In other words you just want to 'peek but not touch'. Then setting `should-delete-messages` to FALSE would be much more appropriate. So since it is hard to choose what should be the right default value for the `should-delete-messages` attribute, we simply made it a required attribute, to be set by the user. Leaving it up to the user also means, you will be less likely to end up with unintended behavior.

Note

When configuring a polling email adapter's *should-mark-messages-as-read* attribute, be aware of the protocol you are configuring to retrieve messages. For example POP3 does not support this flag which means setting it to either value will have no effect as messages will NOT be marked as read.

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"/>
```

Finally, the <imap-idle-channel-adapter/> also accepts the 'error-channel' attribute. If a downstream exception is thrown and an 'error-channel' is specified, a MessagingException message containing the failed message and original exception, will be sent to this channel. Otherwise, if the downstream channels are synchronous, any such exception will simply be logged as a warning by the channel adapter.

16. 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.

16.1 Introduction

Two flavors each of UDP inbound and outbound channel 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 channel adapters are provided `TcpSendingMessageHandler` sends messages over TCP. `TcpReceivingChannelAdapter` receives messages over TCP.

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 a timeout or I/O error occurs.

The TCP and UDP inbound channel adapters, and the TCP inbound gateway, support the "error-channel" attribute. This provides the same basic functionality as described in the section called “GatewayProxyFactoryBean”.

16.2 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.

16.3 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 or gateway (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. For more information refer to Section 16.7, “TCP Message Correlation”.

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 (de)serializers to convert between the message payload and the bits that are sent over TCP. This is accomplished by providing a deserializer and serializer for inbound and outbound messages respectively. Four standard (de)serializers are provided; the first is `ByteArrayCrlfSerializer`, which can convert a byte array to a stream of bytes followed by carriage return and linefeed characters (\r\n). This is the default (de)serializer and can be used with telnet as a client, for example. The second is is `ByteArrayStxEtxSerializer`, which can convert a byte array to a stream of bytes preceded by an STX (0x02) and followed by an ETX (0x03). The third is `ByteArrayLengthHeaderSerializer`, which can convert a byte array to a stream of bytes preceded by a 4 byte binary length in network byte order. Each of these is a subclass of `AbstractByteArraySerializer` which implements both `org.springframework.core.serializer.Serializer` and `org.springframework.core.serializer.Deserializer`. For backwards compatibility, connections using any subclass of `AbstractByteArraySerializer` for serialization will also accept a String which will be converted to a byte array first. Each of these (de)serializers converts an input stream containing the corresponding format to a byte array payload. The fourth standard serializer is `org.springframework.core.serializer.DefaultSerializer` which can be used to convert Serializable objects using java serialization. `org.springframework.core.serializer.DefaultDeserializer` is provided for inbound deserialization of streams containing Serializable objects. To implement a custom (de)serializer

pair, implement the `org.springframework.core.serializer.Deserializer` and `org.springframework.core.serializer.Serializer` interfaces. If you do not wish to use the default (de)serializer (`ByteArrayCrLfSerializer`), you must supply `serializer` and `deserializer` attributes on the connection factory (example below).

```
      <bean id="javaSerializer"
            class="org.springframework.core.serializer.DefaultSerializer" />
      <bean id="javaDeserializer"
            class="org.springframework.core.serializer.DefaultDeserializer" />

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

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

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

16.4 Tcp Connection Interceptors

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. No interceptors are currently provided by the framework but, for an example, see the `InterceptedSharedConnectionTests` in the source repository.

The `HelloWorldInterceptor` used in the test case works as follows:

When configured with a client connection factory, when the first message is sent over a connection that is intercepted, the interceptor sends 'Hello' over the connection, and expects to receive 'world!'. When that occurs, the negotiation is complete and the original message is sent; further messages that use the same connection are sent without any additional negotiation.

When configured with a server connection factory, the interceptor requires the first message to be 'Hello' and, if it is, returns 'world!'. Otherwise it throws an exception causing the connection to be closed.

All `TcpConnection` methods are intercepted. Interceptor instances are created for each connection by an interceptor factory. If an interceptor is stateful, the factory should create a new instance for each connection. Interceptor factories are added to the configuration of an interceptor factory chain, which is provided to a connection factory using the `interceptor-factory` attribute. Interceptors must implement the `TcpConnectionInterceptor` interface; factories must implement the `TcpConnectionInterceptorFactory` interface. A convenience class `AbstractTcpConnectionInterceptor` is provided with passthrough methods; by extending this class, you only need to implement those methods you wish to intercept.

```
<bean id="helloWorldInterceptorFactory"
   class="org.springframework.integration.ip.tcp.connection.TcpConnectionInterceptorFactoryChain">
  <property name="interceptors">
    <array>
      <bean class="org.springframework.integration.ip.tcp.connection.HelloWorldInterceptorFactory"/>
    </array>
  </property>
</bean>

<int-ip:tcp-connection-factory id="server"
  type="server"
  port="12345"
  using-nio="true"
  single-use="true"
  interceptor-factory-chain="helloWorldInterceptorFactory"
/>

<int-ip:tcp-connection-factory id="client"
  type="client"
  host="localhost"
  port="12345"
  single-use="true"
  so-timeout="100000"
  using-nio="true"
  interceptor-factory-chain="helloWorldInterceptorFactory"
/>
```

Configuring a connection interceptor factory chain.

16.5 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 attribute 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="javaSerializer"
              class="org.springframework.core.serializer.DefaultSerializer" />
  <bean id="javaDeserializer"
              class="org.springframework.core.serializer.DefaultDeserializer" />

  <int-ip:tcp-connection-factory id="server"
    type="server"
    port="1234"
    deserializer="javaDeserializer"
    serializer="javaSerializer"
    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"
    deserializer="javaDeserializer"
    serializer="javaSerializer"
  />

  <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'. Java serialization is used on the wire.

16.6 TCP Gateways

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

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

The outbound gateway, after sending a message over the connection, waits for a response and constructs a response message and puts in on the reply channel. 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 connection factory configured with the default (de)serializer 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 outbound TCP gateway.

16.7 TCP Message Correlation

Overview

One goal of the IP Endpoints is to provide communication with systems other than another Spring Integration application. For this reason, only message payloads are sent and received. No message correlation is provided by the framework, except when using the gateways, or collaborating channel adapters on the server side. In the paragraphs below we discuss the various correlation techniques available to applications. In most cases, this requires specific application-level correlation of messages, even when message payloads contain some natural correlation data (such as an order number).

Gateways

The gateways will automatically correlate messages. However, an outbound gateway should only be used for relatively low-volume use. When the connection factory is configured for a single shared connection to be used for all message pairs ('single-use="false"'), only one message can be processed at a time. A new message will have to wait until the reply to the previous message has been received. When a connection factory is configured for each new message to use a new connection ('single-use="true"'), the above restriction does not apply. While this may give higher throughput than a shared connection environment, it comes with the overhead of opening and closing a new connection for each message pair.

Therefore, for high-volume messages, consider using a collaborating pair of channel adapters. However, you will need to provide collaboration logic.

Collaborating Outbound and Inbound Channel Adapters

To achieve high-volume throughput (avoiding the pitfalls of using gateways as mentioned above) you may consider configuring a pair of collaborating outbound and inbound channel adapters. On the server side, message correlation is automatically handled by the adapters because the inbound adapter adds a header allowing the outbound adapter to determine which connection to use to send the reply message. On the client side, however, the application will have to provide its own correlation logic. This can be done in a number of ways.

If the message payload has some natural correlation data, such as a transaction id or an order number, AND there is no need to retain any information (such as a reply channel header) from the original outbound message, the correlation is simple and would done at the application level in any case.

If the message payload has some natural correlation data, such as a transaction id or an order number, but there is a need to retain some information (such as a reply channel header) from the original outbound message, you may need to retain a copy of the original outbound message (perhaps by using a publish-subscribe channel) and use an aggregator to recombine the necessary data.

For either of the previous two paragraphs, if the payload has no natural correlation data, you may need to provide a transformer upstream of the outbound channel adapter to enhance the payload with such data. Such a transformer may transform the original payload to a new object containing both the original payload and some subset of the message headers. Of course, live objects (such as reply channels) from the headers can not be included in the transformed payload.

If such a strategy is chosen you will need to ensure the connection factory has an appropriate serializer/deserializer pair to handle such a payload, such as the `DefaultSerializer/Deserializer` which use java serialization, or a custom serializer and deserializer. The `ByteArray*Serializer` options mentioned in Section 16.3, “TCP Connection Factories”, including the default `ByteArrayCrLfSerializer`, do not support such payloads, unless the transformed payload is a `String` or `byte[]`,

16.8 A Note About NIO

Using NIO (see `using-nio` in Section 16.9, “IP Configuration Attributes”) avoids dedicating a thread to read from each socket. For a small number of sockets, you will likely find that *not* using NIO, together with an async handoff (e.g. to a `QueueChannel`), will perform as well as, or better than, using NIO.

Consider using NIO when handling a large number of connections. However, the use of NIO has some other ramifications. A pool of threads (in the task executor) is shared across all the sockets; each incoming message is assembled and sent to the configured channel as a separate unit of work on a thread selected from that pool. Two sequential messages arriving on the *same* socket *might* be processed by different threads. This means that the order in which the messages are sent to the channel is indeterminate; the strict ordering of the messages on the socket is not maintained.

For some applications, this is not an issue; for others it is. If strict ordering is required, consider setting `using-nio` to false and using async handoff.

16.9 IP Configuration Attributes

Table 16.1. 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.

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
host	Y	N		The host name or ip address of the destination.
port	Y	Y		The port.
serializer	Y	Y		An implementation of `Serializer` used to serialize the payload. Defaults to `ByteArrayCrLfSerializer`
deserializer	Y	Y		An implementation of `Deserializer` used to deserialize the payload. Defaults to `ByteArrayCrLfSerializer`
using-nio	Y	Y	true, false	Whether or not the socket handing uses NIO. Refer to the java.nio package for more information. See Section 16.8, “A Note About NIO”. Default false.
using-direct-buffers	Y	N	true, false	When using NIO, whether or not the tcp 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()`.

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
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 16.2. 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

Attribute Name	Allowed Values	Attribute Description
		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 acknowledgment packets. See `java.net.DatagramSocket` setSendBufferSize() methods for more information.
so-receive-buffer- size		See `java.net.DatagramSocket` setReceiveBufferSize() for more information.
local-address		On a multi-homed system, specifies an IP address for the interface to which the socket will be bound.
error-channel		If an Exception is thrown by a downstream component, the MessagingException message containing the exception and failed message is sent to this channel.

Table 16.3. 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.

Attribute Name	Allowed Values	Attribute Description
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 Transaction (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 acknowledgment. If an acknowledgment 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 acknowledgments 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 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 acknowledgment packets. See `java.net.DatagramSocket` setReceiveBufferSize() methods for more information.
local-address		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

Attribute Name	Allowed Values	Attribute Description
		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 16.4. TCP Inbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description
channel		The channel to which inbound messages will be sent.
connection-factory		If the connection factory has a type 'server', the factory is 'owned' by this adapter. If it has a type 'client', it is 'owned' by an outbound channel adapter and this adapter will receive any incoming messages on the connection created by the outbound adapter.
error-channel		If an Exception is thrown by a downstream component, the MessagingException message containing the exception and failed message is sent to this channel.

Table 16.5. TCP Outbound Channel Adapter Attributes

Attribute Name	Allowed Values	Attribute Description
channel		The channel on which outbound messages arrive.
connection-factory		If the connection factory has a type 'client', the factory is 'owned' by this adapter. If it has a type 'server', it is 'owned' by an inbound channel adapter and this adapter will attempt to correlate messages to the connection on which an original inbound message was received.

Table 16.6. TCP Inbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
connection-factory		The connection factory must be of type server.
request-channel		The channel to which incoming messages will be sent.

Attribute Name	Allowed Values	Attribute Description
reply-channel		The channel on which reply messages may arrive. Usually replies will arrive on a temporary reply channel added to the inbound message header
reply-timeout		The time in milliseconds for which the gateway will wait for a reply.
error-channel		If an Exception is thrown by a downstream component, the MessagingException message containing the exception and failed message is sent to this channel; any reply from that flow will then be returned as a response by the gateway.

Table 16.7. TCP Outbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
connection-factory		The connection factory must be of type client.
request-channel		The channel on which outgoing messages will arrive.
reply-channel		Optional. The channel to which reply messages may be sent if the original outbound message did not contain a reply channel header.
reply-timeout		The time in milliseconds for which the gateway will wait for a reply.
request-timeout		If a single-use connection factory is not being used, The time in milliseconds for which the gateway will wait to get access to the shared connection.

17. JDBC Support

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

17.1 Inbound Channel Adapter

The main function of an inbound Channel Adapter is to execute a SQL `SELECT` query and turn the result set as a message. The message payload is the whole result set, expressed as a `List`, and the types of the items in the list depend on the row-mapping strategy that is used. The default strategy is a generic mapper that just returns a `Map` for each row in the query result. Optionally, this can be changed by adding a reference to a `RowMapper` instance (see the Spring JDBC [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/jdbc.html] 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 a `JdbcTemplate` instance or a `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 that they don't show up in the next poll. The update can be parameterized 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 behavior (the adapter has a `sql-parameter-source-factory` attribute).

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 fixed-rate="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 fail, the transaction rolls back and the input data is reverted to its original state.

17.2 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])"
    data-source="dataSource"
    channel="input"/>
```

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 behavior is part of the `SqlParameterSource` which is the default source created by the outbound adapter. Other behavior 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 therefore the same transaction (if there is one) as the sender of the message.

17.3 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 from the result (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"/>
```

As with the channel 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 the incoming message to the query.

17.4 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_".

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 [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/jdbc.html#d0e24182]. 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).

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 protection 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 when the MessageStore is managing persistent queues backing a Spring Integration Message 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.

18. 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.

18.1 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 fixed-rate="30000"/>
</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 fixed-rate="30000"/>
</jms:inbound-channel-adapter>
```

18.2 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-2.0.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.

Finally, the <message-driven-channel-adapter> also accepts the 'error-channel' attribute. This provides the same basic functionality as described in the section called “GatewayProxyFactoryBean”.

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

When comparing this to the generic gateway configuration, or the JMS 'inbound-gateway' that will be discussed below, the key difference here is that we are in a one-way flow since this is a 'channel-adapter', not a gateway. Therefore, the flow downstream from the 'error-channel' should also be one-way. For example, it could simply send to a logging handler, or it could be connected to a different JMS <outbound-channel-adapter> element.

18.3 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).

18.4 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 that might have occurred on the consumer side and will time out waiting for the reply. However there might be times when you want to communicate an error condition back to the consumer, in other words treat the Exception as a valid reply by mapping it to a Message. To accomplish this JMS Inbound Gateway provides support for a Message Channel to which errors can be sent for processing, potentially resulting in a reply Message payload that conforms to some contract defining what a caller may expect as an "error" reply. Such a channel can be configured via the *error-channel* attribute.

```
<int-jms:inbound-gateway request-destination="requestQueue"
          request-channel="jmsinputchannel"
          error-channel="errorTransformationChannel"/>

<si:transformer input-channel="exceptionTransformationChannel"
        ref="exceptionTransformer" method="createErrorResponse"/>
```

You might notice that this example looks very similar to that included within the section called “GatewayProxyFactoryBean”. The same idea applies here: The *exceptionTransformer* could be a simple POJO that creates error response objects, you could reference the "nullChannel" to suppress the errors, or you could leave 'error-channel' out to let the Exception propagate.

18.5 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.

18.6 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.

18.7 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"/>
```

18.8 JMS Samples

To experiment with these JMS adapters, check out JMS samples available in our new Samples Git repository available here: http://git.springsource.org/+spring-integration/spring-integration/samples . 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.

19. RMI Support

19.1 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.

19.2 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>
```

19.3 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>
```

19.4 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"/>
```

20. SFTP Adapters

Spring Integration provides support for file transfer operations via SFTP.

20.1 Introduction

The Secure File Transfer Protocol (SFTP) is a network protocol which allows you to transfer files between two computers on the Internet over any reliable stream.

The SFTP protocol requires a secure channel, such as SSH, as well as visibility to a client's identity throughout the SFTP session.

Spring Integration supports sending and receiving files over SFTP by providing two types of *client*s - *Inbound Channel Adapters* and *Outbound Channel Adapters* as well as convenient namespace configuration to define these *client*s.

```
xmlns:sftp="http://www.springframework.org/schema/integration/sftp"
xsi:schemaLocation="http://www.springframework.org/schema/integration/sftp
 http://www.springframework.org/schema/integration/sftp/spring-integration-sftp-2.0.xsd"
```

20.2 SFTP Session Factory

Before configuring SFTP adapters you must configure an *SFTP Session Factory*. You can configure the *SFTP Session Factory* via a regular bean definition: Below is a basic configuration:

```
<beans:bean id="sftpSessionFactory" class="org.springframework.integration.sftp.session.DefaultSftpSessionFa
  <beans:property name="host" value="loclahost"/>
  <beans:property name="privateKey" value="classpath:META-INF/keys/sftpTest"/>
  <beans:property name="privateKeyPassphrase" value="springIntegration"/>
  <beans:property name="port" value="22"/>
  <beans:property name="user" value="kermit"/>
</beans:bean>
```

Every time an adapter requests a session object from its `SessionFactory` the session is returned from a session pool maintained by a caching wrapper around the factory. A Session in the session pool might go stale (if it has been disconnected by the server due to inactivity) so the `SessionFactory` will perform validation to make sure that it never returns a stale session to the adapter. If a stale session was encountered, it will be removed from the pool, and a new one will be created.

Note

If you experience connectivity problems and would like to trace Session creation as well as see which Sessions are polled you may enable it by setting the logger to TRACE level (e.g., log4j.category.org.springframework.integration.file=TRACE)

Now all you need to do is inject this *SFTP Session Factory* into your adapters.

Note

A more practical way to provide values for the *SFTP Session Factory* would be via Spring's property placeholder support (http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-factory-placeholderconfigurer)

20.3 SFTP Inbound Channel Adapter

The *SFTP Inbound Channel Adapter* is a special listener that will connect to the server and listen for the remote directory events (e.g., new file created) at which point it will initiate a file transfer.

```
<sftp:inbound-channel-adapter id="sftpAdapterAutoCreate"
     session-factory="sftpSessionFactory"
   channel="requestChannel"
   filename-pattern="*.txt"
   remote-directory="/foo/bar"
   local-directory="file:target/foo"
   auto-create-local-directory="true"
   delete-remote-files="false">
  <poller fixed-rate="1000"/>
</sftp:inbound-channel-adapter>
```

As you can see from the configuration above you can configure the *SFTP Inbound Channel Adapter* via the `inbound-channel-adapter` element while also providing values for various attributes such as `local-directory` - where files are going to be transferred TO and `remote-directory` - the remote source directory where files are going to be transferred FROM - as well as other attributes including a `session-factory` reference to the bean we configured earlier.

Some times file filtering based on the simple pattern specified via `filename-pattern` attribute might not be sufficient. If this is the case, you can use the `filename-regex` attribute to specify a Regular Expression (e.g. `filename-regex=".*\.test$"`). And of course if you need complete control you can use the `filter` attribute to provide a reference to a custom implementation of the `org.springframework.integration.file.filters.FileListFilter` - a strategy interface for filtering a list of files.

Please refer to the schema for more detail on these attributes.

It is also important to understand that *SFTP Inbound Channel Adapter* is a Polling Consumer and therefore you must configure a poller (either a global default or a local sub-element). Once the file has been transferred to a local directory, a Message with `java.io.File` as its payload type will be generated and sent to the channel identified by the `channel` attribute.

More on File Filtering and Large Files

Some times a file that just appeared in the monitored (remote) directory is not complete. Typically such a file will be written with some temporary extension (e.g., foo.txt.writing) and then renamed after the writing process completes. As a user in most cases you are only interested in files that are complete and would like to filter only those files. To handle these scenarios, use filtering support provided via the `filename-pattern`, `filename-regex` and `filter` attributes. If you need a custom filter implementation simply include a reference in your adapter via the `filter` attribute.

```
<int-sftp:inbound-channel-adapter id="sftpInbondAdapter"
```

```
    channel="receiveChannel"
    session-factory="sftpSessionFactory"
    filter="customFilter"
    local-directory="file:/local-test-dir"
    remote-directory="/remote-test-dir">
  <int:poller fixed-rate="1000" max-messages-per-poll="10" task-executor="executor"/>
</int-sftp:inbound-channel-adapter>

<bean id="customFilter" class="org.foo.CustomFilter"/>
```

20.4 SFTP Outbound Channel Adapter

The *SFTP Outbound Channel Adapter*is a special `MessageHandler` that will connect to the remote directory and will initiate a file transfer for every file it will receive as the payload of an incoming `Message`. It also supports several representations of the File so you are not limited to the File object. Similar to the FTP outbound adapter, the *SFTP Outbound Channel Adapter* supports the following payloads: 1) `java.io.File` - the actual file object; 2) `byte[]` - byte array that represents the file contents; 3) `java.lang.String` - text that represents the file contents.

```
<int-sftp:outbound-channel-adapter id="sftpOutboundAdapter"
    session-factory="sftpSessionFactory"
    channel="inputChannel"
    charset="UTF-8"
    remote-directory="foo/bar"
    remote-filename-generator-expression="payload.getName() + '-foo'"/>
```

As you can see from the configuration above you can configure the *SFTP Outbound Channel Adapter* via the `outbound-channel-adapter` element. Please refer to the schema for more detail on these attributes.

SpEL and the SFTP Outbound Adapter

As with many other components in Spring Integration, you can benefit from the Spring Expression Language (SpEL) support when configuring an *SFTP Outbound Channel Adapter*, by specifying two attributes `remote-directory-expression` and `remote-filename-generator-expression` (see above). The expression evaluation context will have the Message as its root object, thus allowing you to provide expressions which can dynamically compute the *file name* or the existing *directory path* based on the data in the Message (either from 'payload' or 'headers'). In the example above we are defining the `remote-filename-generator-expression` attribute with an expression value that computes the *file name* based on its original name while also appending a suffix: '-foo'.

20.5 SFTP/JSCH Logging

Since we use JSch libraries (http://www.jcraft.com/jsch/) to provide SFTP support, at times you may require more information from the JSch API itself, especially if something is not working properly (e.g., Authentication exceptions). Unfortunately JSch does not use commons-logging but instead relies on custom implementations of their `com.jcraft.jsch.Logger` interface. As of Spring Integration 2.0.1, we have implemented this interface. So, now all you need to do to enable JSch logging is to configure your logger the way you usually do. For example, here is valid configuration of a logger using Log4J.

```
log4j.category.com.jcraft.jsch=DEBUG
```

21. Stream Support

21.1 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.

21.2 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>
```

21.3 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 Section 3.2, “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>
```

21.4 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-3.0.xsd
      http://www.springframework.org/schema/integration/stream
      http://www.springframework.org/schema/integration/stream/spring-integration-stream-2.0.xsd">
```

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"/>
```

22. Twitter Adapter

Spring Integration provides support for interacting with Twitter. With the Twitter adapters you can both receive and send Twitter messages. You can also perform a Twitter search based on a schedule and publish the search results within Messages.

22.1 Introduction

Twitter is a social networking and micro-blogging service that enables its users to send and read messages known as tweets. Tweets are text-based posts of up to 140 characters displayed on the author's profile page and delivered to the author's subscribers who are known as followers.

Important

Current Twitter support is based on the Twitter4J API. However, future versions will be changed to use the Spring Social project as it is nearing its first release at the time of writing.

Spring Integration provides a convenient namespace configuration to define Twitter artifacts. You can enable it by adding the following within your XML header.

```
xmlns:twitter="http://www.springframework.org/schema/integration/twitter"
xsi:schemaLocation="http://www.springframework.org/schema/integration/twitter
 http://www.springframework.org/schema/integration/twitter/spring-integration-twitter-2.0.xsd"
```

22.2 Twitter OAuth Configuration

The Twitter API allows for both authenticated and anonymous operations. For authenticated operations Twitter uses OAuth - an authentication protocol that allows users to approve an application to act on their behalf without sharing their password. More information can be found at http://oauth.net/ or in this article http://hueniverse.com/oauth/ from Hueniverse. Please also see OAuth FAQ for more information about OAuth and Twitter.

In order to use OAuth authentication/authorization with Twitter you must create a new Application on the Twitter Developers site. Follow the directions below to create a new application and obtain consumer keys and an access token:

- Go to http://dev.twitter.com/
- Click on the `Register an app` link and fill out all required fields on the form provided; set `Application Type` to `Client` and depending on the nature of your application select `Default Access Type` as *Read & Write* or *Read-only* and Submit the form. If everything is successful you'll be presented with the `Consumer Key` and `Consumer Secret`. Copy both values in a safe place.
- On the same page you should see a `My Access Token` button on the side bar (right). Click on it and you'll be presented with two more values: `Access Token` and `Access Token Secret`. Copy these values in a safe place as well.

22.3 Twitter Template

Spring Integration uses the same familiar template pattern to interact with Twitter. Since current Twitter support is based on the Twitter4J API we provide a simple Twiter4JTemplate. For anonymous operations (e.g., search), you don't have to define `Twitter4JTemplate` explicitly, since a default instance will be created and injected into the endpoint. However, for authenticated operation (update status, send direct message, etc.), you must configure `Twitter4JTemplate` as a bean and inject it explicitly into the endpoint, because the authentication configuration is required. Below is a sample configuration of Twitter4JTemplate:

```
<bean id="twitterTemplate" class="org.springframework.integration.twitter.core.Twitter4jTemplate">
 <constructor-arg value="4XzBPacJQxyBzzzH"/>
 <constructor-arg value="AbRxUAvyCtqQtvxFK8w5ZMtMj20KFhB6o"/>
 <constructor-arg value="21691649-4YZY5iJEOfz2A9qCFd9SjBRGb3HLmIm4HNE"/>
 <constructor-arg value="AbRxUAvyNCtqQtxFK8w5ZMtMj20KFhB6o"/>
</bean>
```

Note

The values above are not real.

As you can see from the configuration above, all we need to do is to provide OAuth `attributes` as constructor arguments. The values would be those you obtained in the previous step. The order of constructor arguments is: 1) `consumerKey`, 2) `consumerSecret`, 3) `accessToken`, and 4) `accessTokenSecret`.

A more practical way to manage OAuth connection attributes would be via Spring's property placeholder support by simply creating a property file (e.g., oauth.properties):

```
twitter.oauth.consumerKey=4XzBPacJQxyBzzzH
twitter.oauth.consumerSecret=AbRxUAvyCtqQtvxFK8w5ZMtMj20KFhB6o
twitter.oauth.accessToken=21691649-4YZY5iJEOfz2A9qCFd9SjBRGb3HLmIm4HNE
twitter.oauth.accessTokenSecret=AbRxUAvyNCtqQtxFK8w5ZMtMj20KFhB6o
```

Then, you can configure a `property-placeholder` to point to the above property file:

```
<context:property-placeholder
            location="classpath:oauth.properties"/>

<bean id="twitterTemplate" class="org.springframework.integration.twitter.core.Twitter4jTemplate">
 <constructor-arg value="${twitter.oauth.consumerKey}"/>
 <constructor-arg value="${twitter.oauth.consumerSecret}"/>
 <constructor-arg value="${twitter.oauth.accessToken}"/>
 <constructor-arg value="${twitter.oauth.accessTokenSecret}"/>
</bean>
```

22.4 Twitter Inbound Adapters

Twitter inbound adapters allow you to receive Twitter Messages. There are several types of twitter messages, or tweets

The current release of Spring Integration provides support for receiving tweets as *Timeline Updates*, *Direct Messages*, *Mention Messages* as well as Search Results.

Every Inbound Twitter Channel Adapter is a *Polling Consumer* which means you have to provide a poller configuration. However, there is one important thing you must understand about Twitter since its inner-workings are slightly different than other polling consumers. Twitter defines a concept of Rate Limiting. You can read more about it here: Rate Limiting. In a nutshell, Rate Limiting is the way Twitter manages how often an application can poll for updates. You should consider this when setting your poller intervals, but we are also doing a few things to limit excessively aggressive polling within our adapters.

Another issue that we need to worry about is handling duplicate Tweets. The same adapter (e.g., Search or Timeline Update) while polling on Twitter may receive the same values more than once. For example if you keep searching on Twitter with the same search criteria you'll end up with the same set of tweets unless some other new tweet that matches your search criteria was posted in between your searches. In that situation you'll get all the tweets you had before plus the new one. But what you really want is only the new tweet(s). Spring Integration provides an elegant mechanism for handling these situations. The latest Tweet timestamp will be stored in an instance of the `org.springframework.integration.store.MetadataStore` which is a strategy interface designed for storing various types of metadata (e.g., last retrieved tweet in this case). That strategy helps components such as these Twitter adapters avoid duplicates. By default, Spring Integration will look for a bean of type `org.springframework.integration.store.MetadataStore` in the ApplicationContext. If one is found then it will be used, otherwise it will create a new instance of `SimpleMetadataStore` which is a simple in-memory implementation that will only persist metadata within the lifecycle of the currently running application context. That means upon restart you may end up with duplicate entries. If you need to persist metadata between Application Context restarts, you may use the `PropertiesPersistingMetadataStore` (which is backed by a properties file, and a persister strategy), or you may create your own custom implementation of the `MetadataStore` interface (e.g., JdbcMetadatStore) and configure it as bean within the Application Context.

```
<bean class="org.springframework.integration.store.PropertiesPersistingMetadataStore"/>
```

The Poller that is configured as part of any Inbound Twitter Adapter (see below) will simply poll from this MetadataStore to determine the latest tweet received.

Inbound Message Channel Adapter

This adapter allows you to receive updates from everyone you follow. It's essentially the "Timeline Update" adapter.

```
<twitter:inbound-channel-adapter
    twitter-template="twitterTemplate"
    channel="inChannel">
    <poller fixed-rate="5000" max-messages-per-poll="3"/>
</twitter:inbound-channel-adapter>
```

Direct Inbound Message Channel Adapter

This adapter allows you to receive Direct Messages that were sent to you from other Twitter users.

```
<twitter:dm-inbound-channel-adapter
```

```
    twitter-template="twiterTemplate"
    channel="inboundDmChannel">
    <poller fixed-rate="5000" max-messages-per-poll="3"/>
</twitter:dm-inbound-channel-adapter>
```

Mentions Inbound Message Channel Adapter

This adapter allows you to receive Twitter Messages that Mention you via @user syntax.

```
<twitter:mentions-inbound-channel-adapter
    twitter-template="twiterTemplate"
  channel="inboundMentionsChannel">
    <poller fixed-rate="5000" max-messages-per-poll="3"/>
</twitter:mentions-inbound-channel-adapter>
```

Search Inbound Message Channel Adapter

This adapter allows you to perform searches. As you can see it is not necessary to define twitter-template since a search can be performed anonymously, however you must define a search query.

```
<twitter:search-inbound-channel-adapter
    query="#springintegration"
  channel="inboundMentionsChannel">
     <poller fixed-rate="5000" max-messages-per-poll="3"/>
</twitter:search-inbound-channel-adapter>
```

Here is a link that will help you learn more about Twitter queries: http://search.twitter.com/operators

As you can see the configuration of all of these adapters is very similar to other inbound adapters with one exception. Some may need to be injected with the `twitter-template`. Once received each Twitter Message would be encapsulated in a Spring Integration Message and sent to the channel specified by the `channel` attribute. Currently the Payload type of any Message is `org.springframework.integration.twitter.core.Tweet` which is very similar to the object with the same name in Spring Social. As we migrate to Spring Social we'll be depending on their API and some of the artifacts that are currently in use will be obsolete, however we've already made sure that the impact of such migration is minimal by aligning our API with the current state (at the time of writing) of Spring Social.

To get the text from the `org.springframework.integration.twitter.core.Tweet` simply invoke the `getText()` method.

22.5 Twitter Outbound Adapter

Twitter outbound channel adapters allow you to send Twitter Messages, or tweets.

The current release of Spring Integration supports sending *Status Update Messages* and *Direct Messages*. Twitter outbound channel adapters will take the Message payload and send it as a Twitter message. Currently the only supported payload type is `String`, so consider adding a *transformer* if the payload of the incoming message is not a String.

Twitter Outbound Update Channel Adapter

This adapter allows you to send regular status updates by simply sending a Message to the channel identified by the `channel` attribute.

```
<twitter:outbound-channel-adapter
    twitter-template="twitterTemplate"
    channel="twitterChannel"/>
```

The only extra configuration that is required for this adapter is the `twitter-template` reference.

Twitter Outbound Direct Message Channel Adapter

This adapter allows you to send Direct Twitter Messages (i.e., @user) by simply sending a Message to the channel identified by the `channel` attribute.

```
<twitter:dm-outbound-channel-adapter
    twitter-template="twitterTemplate"
    channel="twitterChannel"/>
```

The only extra configuration that is required for this adapter is the `twitter-template` reference.

When it comes to Twitter Direct Messages, you must specify who you are sending the message to - the *target userid*. The Twitter Outbound Direct Message Channel Adapter will look for a target userid in the Message headers under the name `twitter_dmTargetUserId` which is also identified by the following constant: `TwitterHeaders.DM_TARGET_USER_ID`. So when creating a Message all you need to do is add a value for that header.

```
Message message = MessageBuilder.withPayload("hello")
        .setHeader(TwitterHeaders.DM_TARGET_USER_ID, "z_oleg").build();
```

The above approach works well if you are creating the Message programmatically. However it's more common to provide the header value within a messaging flow. The value can be provided by an upstream <header-enricher>.

```
<header-enricher input-channel="in" output-channel="out">
    <header name="twitter_dmTargetUserId" value="z_oleg"/>
</header-enricher>
```

It's quite common that the value must be determined dynamically. For those cases you can take advantage of SpEL support within the <header-enricher>.

```
<header-enricher input-channel="in" output-channel="out">
    <header name="twitter_dmTargetUserId" expression="@twitterIdService.lookup(headers.username)"/>
</header-enricher>
```

Important

Twitter does not allow you to post duplicate Messages. This is a common problem during testing when the same code works the first time but does not work the second time. So, make sure to change the content of the Message each time. Another thing that works well for testing is to append a timestamp to the end of each message.

23. Web Services Support

23.1 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 [http://static.springframework.org/spring-ws/sites/1.5/] 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 [http://static.springsource.org/spring-ws/sites/1.5/apidocs/index.html] 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 [http://static.springframework.org/spring-ws/site/reference/html/client.html] as well as the chapter covering Object/XML mapping [http://static.springframework.org/spring-ws/site/reference/html/oxm.html].

23.2 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 [http://static.springframework.org/spring-ws/sites/1.5/reference/html/server.html]. The chapter covering Object/XML mapping [http://static.springframework.org/spring-ws/site/reference/html/oxm.html] is also applicable again.

23.3 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.

24. XML Support - Dealing with XML Payloads

24.1 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.

24.2 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>
```

24.3 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-3.0.xsd
```

```
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-2.0.xsd
           http://www.springframework.org/schema/integration/xml
           http://www.springframework.org/schema/integration/xml/spring-integration-xml-2.0.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 fixed-rate="2000"/>
<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 use `expression` 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.

24.4 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>
```

24.5 Routing xml messages using XPath

Similar to SpEL-based routers, Spring Integration provides support for routing messages based on the XPath expressions allowing you to create a Message Endpoint with an input channel but no output channel since the output channel(s) is determined dynamically.

```
<si-xml:xpath-router id="orderTypeRouter" input-channel="orderChannel">
    <si-xml:xpath-expression expression="/order/type"/>
</si-xml:xpath-router>
```

Internally XPath expression will be evaluated as *NODESET* type and converted to a `List<String>` representing channel names. Typically such list will contain a single channel name. However, based on the result of an XPath Expression XPath router can also take on the characteristics of the *Recipient List Router* if XPath Expression returns more then one value, thus resulting in the `List<String>` containing more then one channel name. In that case Message will be sent to all channels in the list. So assuming that the xml file passed to the router configured below contains many `responder` sub-elements representing channel names, the message will be sent to all of those channels.

```
<!-- route the order to all responders-->
<si-xml:xpath-router id="responderRouter" input-channel="orderChannel">
    <si-xml:xpath-expression expression="/request/responders"/>
</si-xml:xpath-router>
```

If the returned values do not represent the channel names additional mapping could be specified. For example if the `/request/responders` expression results in two values `responderA` and `responderB` but you don't want to couple the responder names to channel names you may provide additional mapping as such:

```
<!-- route the order to all responders-->
<si-xml:xpath-router id="responderRouter" input-channel="orderChannel">
    <si-xml:xpath-expression expression="/request/responders"/>
    <int-xml:mapping value="responderA" channel="channelA"/>
    <int-xml:mapping value="responderB" channel="channelB"/>
</si-xml:xpath-router>
```

As we already said the default evaluation type for XPath expressions is NODESET which is converted to a List<String> of channel names, thus handling single channel scenarios as well as multiple. However certain XPath expressions may evaluate as String type from the very beginning (e.g., 'name(./node())' - which will return the name of the root node) thus resulting in the exception if default evaluation type (NODESET) is used. For these scenarious you may use `evaluate-as-string` attribute which will allow you to manage the evaluation type. It is FALSE by default, however if set to TRUE, the String evaluation type will be used. For example if we want to route based on the name of the root node we can have use the following configuration:

```
<int-xml:xpath-router id="xpathRouterAsString"
  input-channel="xpathStringChannel"
  evaluate-as-string="true">
```

```
 <int-xml:xpath-expression expression="name(./node())"/>
</int-xml:xpath-router>
```

24.6 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>
```

24.7 Transforming xml messages using XPath

When it comes to message transformation XPath is a great way to transform Messages that have XML payloads by defining XPath transformers via <xpath-transformer/> element.

Simple XPath transformation

Let's look at the following transformer configuration:

```
<xpath-transformer input-channel="inputChannel" output-channel="outputChannel"
      xpath-expression="/person/@name" />
```

. . . and Message

```
Message<?> message =
  MessageBuilder.withPayload("<person name='John Doe' age='42' married='true'/>").build();
```

After sending this message to the 'inputChannel' the XPath transformer configured above will transform this XML Message to a simple Message with payload of 'John Doe' all based on the simple XPath Expression specified in the `xpath-expression` attribute.

XPath also has capability to perform simple conversion of extracted elements to a desired type. Valid return types are defined in `XPathConstants` and follows the conversion rules specified by the `XPath`.

The following constants are defined by the `XPathConstants`: *BOOLEAN, DOM_OBJECT_MODEL, NODE, NODESET, NUMBER, STRING*

You can configure the desired type by simply using `evaluation-type` attribute of the <xpath-transformer/> element.

```
<xpath-transformer input-channel="numberInput" xpath-expression="/person/@age"
    evaluation-type="NUMBER_RESULT" output-channel="output"/>

<xpath-transformer input-channel="booleanInput" xpath-expression="/person/@married = 'true'"
    evaluation-type="BOOLEAN_RESULT" output-channel="output"/>
```

Node Mappers

If you need to provide custom mapping for the node extracted by the XPath expression simply provide a reference to the implementation of the `org.springframework.xml.xpath.NodeMapper` - an interface used by `XPathOperations` implementations for mapping Node objects on a per-node basis. To provide a reference to a `NodeMapper` simply use `node-mapper` attribute:

```
<xpath-transformer input-channel="nodeMapperInput" xpath-expression="/person/@age"
    node-mapper="testNodeMapper" output-channel="output"/>
```

. . . and Sample NodeMapper implementation:

```
class TestNodeMapper implements NodeMapper {
  public Object mapNode(Node node, int nodeNum) throws DOMException {
    return node.getTextContent() + "-mapped";
  }
}
```

XML Payload Converter

You can also use implementation of the `org.springframework.integration.xml.XmlPayloadConverter` to provide more granular transformation:

```
<xpath-transformer input-channel="customConverterInput" xpath-expression="/test/@type"
    converter="testXmlPayloadConverter" output-channel="output"/>
```

. . . and Sample XmlPayloadConverter implementation:

```
class TestXmlPayloadConverter implements XmlPayloadConverter {
  public Source convertToSource(Object object) {
    throw new UnsupportedOperationException();
  }
  //
  public Node convertToNode(Object object) {
    try {
      return DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(
          new InputSource(new StringReader("<test type='custom'/>")));
    }
    catch (Exception e) {
```

```
      throw new IllegalStateException(e);
    }
  }
  //
  public Document convertToDocument(Object object) {
    throw new UnsupportedOperationException();
  }
}
```

Combination of SpEL and XPath expressions

You can also combine Spring Expression Language (SpEL) expressions with XPath expression and configure them using `expression` attribute:

```
xpath-expression id="testExpression" expression="/person/@age * 2"/>
```

In the above case the overall result of the expression will be the result of the XPathe expression multiplied by 2.

24.8 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 fixed-rate="2000"/>
</si-xml:xpath-splitter>
```

25. XMPP Support

Spring Integration provides Channel Adapters for XMPP [http://www.xmpp.org].

25.1 Introduction

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 chat 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 (e.g. "presence").

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* [http://www.igniterealtime.org/projects/openfire/] and *ejabberd* [http://www.ejabberd.im]

Spring integration provides support for XMPP via XMPP adapters which support sending and receiving both XMPP chat messages and presence changes from other entries in your roster. As with other adapters, the XMPP adapters come with support for a convenient namespace-based configuration. To configure the XMPP namespace, include the following elements in the headers of your XML configuration file:

```
xmlns:xmpp="http://www.springframework.org/schema/integration/xmpp"
xsi:schemaLocation="http://www.springframework.org/schema/integration/xmpp
 http://www.springframework.org/schema/integration/xmpp/spring-integration-xmpp-2.0.xsd"
```

25.2 XMPP Connection

Before using inbound or outbound XMPP adapters to participate in the XMPP network, an actor must establish its XMPP connection. This connection object could be shared by all XMPP adapters connected to a particular account. Typically this requires - at a minimum - `user`, `password`, and `host`. To create a basic XMPP connection, you can utilize the convenience of the namespace.

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

Note

For added convenience you can rely on the default naming convention and omit the `id` attribute. The default name *xmppConnection* will be used for this connection bean.

If the XMPP Connection goes stale, reconnection attempts will be made with an automatic login as long as the previous connection state was logged (authenticated). We also register a `ConnectionListener` which will log connection events if the DEBUG logging level is enabled.

25.3 XMPP Messages

Inbound Message Channel Adapter

The Spring Integration adapters support receiving chat messages from other users in the system. To do this, the *Inbound Message Channel 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` if you set the `extract-payload` attribute's value to 'true' when configuring an adapter. Configuration support for the XMPP *Inbound Message Channel Adapter* is provided via the `inbound-channel-adapter` element.

```
<xmpp:inbound-channel-adapter id="xmppInboundAdapter"
 channel="xmppInbound"
 xmpp-connection="testConnection"
 extract-payload="false"
 auto-startup="true"/>
```

As you can see amongst the usual attributes this adapter also requires a reference to an XMPP Connection.

It is also important to mention that the XMPP inbound adapter is an *event driven adapter* and a `Lifecycle` implementation. When started it will register a `PacketListener` that will listen for incoming XMPP Chat Messages. It forwards any received messages to the underlying adapter which will convert them to Spring Integration Messages and send them to the specified `channel`. It will unregister the `PacketListener` when it is stopped.

Outbound Message Channel Adapter

You may also send chat messages to other users on XMPP using the *Outbound Message Channel Adapter*. Configuration support for the XMPP *Outbound Message Channel Adapter* is provided via the `outbound-channel-adapter` element.

```
<int-xmpp:outbound-channel-adapter id="outboundEventAdapter"
      channel="outboundEventChannel"
      xmpp-connection="testConnection"/>
```

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

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

Another mechanism of setting the header is by using the XMPP header-enricher support. Here is an example.

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

25.4 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.

Inbound Presence Message Channel Adapter

Spring Integration provides an *Inbound Presence Message Channel Adapter* which supports receiving Presence events from other users in the system who happen to be on your Roster. To do this, the adapter "logs in" as a user on your behalf, registers a `RosterListener` and forwards received Presence update events as Messages to the channel identified by the `channel` attribute. The payload of the Message will be a `org.jivesoftware.smack.packet.Presence` object (see http://www.igniterealtime.org/builds/smack/docs/3.1.0/javadoc/org/jivesoftware/smack/packet/Presence.html).

Configuration support for the XMPP *Inbound Presence Message Channel Adapter* is provided via the `presence-inbound-channel-adapter` element.

```
<int-xmpp:presence-inbound-channel-adapter channel="outChannel"
  xmpp-connection="testConnection" auto-startup="false"/>
```

As you can see amongst the usual attributes this adapter also requires a reference to an XMPP Connection. It is also important to mention that this adapter is an event driven adapter and a `Lifecycle` implementation. It will register a `RosterListener` when started and will unregister that `RosterListener` when stopped.

Outbound Presence Message Channel Adapter

Spring Integration also supports sending Presence events to be seen by other users in the network who happen to have you on their Roster. When you send a Message to the *Outbound Presence Message Channel Adapter* it extracts the payload, which is expected to be of type `org.jivesoftware.smack.packet.Presence` (see http://www.igniterealtime.org/builds/smack/docs/3.1.0/javadoc/org/jivesoftware/smack/packet/Presence.html) and sends it to the XMPP Connection, thus advertising your presence events to the rest of the network.

Configuration support for the XMPP *Outbound Presence Message Channel Adapter* is provided via the `presence-outbound-channel-adapter` element.

```
<int-xmpp:presence-outbound-channel-adapter id="eventOutboundPresenceChannel"
 xmpp-connection="testConnection"/>
```

It can also be a *Polling Consumer* (if it receives Messages from a Pollable Channel) in which case you would need to register a Poller.

```
<int-xmpp:presence-outbound-channel-adapter id="pollingOutboundPresenceAdapter"
  xmpp-connection="testConnection"
  channel="pollingChannel">
 <int:poller fixed-rate="1000" max-messages-per-poll="1"/>
</int-xmpp:presence-outbound-channel-adapter>
```

Like its inbound counterpart, it requires a reference to an XMPP Connection.

Note

If you are relying on the default naming convention for an XMPP Connection bean (described earlier), and you have only one XMPP Connection bean configured in your Application Context, you may omit the `xmpp-connection` attribute. In that case, the bean with the name *xmppConnection* will be located and injected into the adapter.

25.5 Appendices

Since Spring Integration XMPP support is based on the Smack 3.1 API (http://www.igniterealtime.org/downloads/index.jsp), it is important to know a few details related to more complex configuration of the XMPP Connection object.

As stated earlier the `xmpp-connection` namespace support is designed to simplify basic connection configuration and only supports a few common configuration attributes. However, the `org.jivesoftware.smack.ConnectionConfiguration` object defines about 20 attributes, and there is no real value of adding namespace support for all of them. So, for more complex connection configurations, simply configure an instance of our `XmppConnectionFactoryBean` as a regular bean, and inject a `org.jivesoftware.smack.ConnectionConfiguration` as a constructor argument to that FactoryBean. Every property you need, can be specified directly on that ConnectionConfiguration instance (a bean definition with the 'p' namespace would work well). This way SSL, or any other attributes, could be set directly. Here's an example:

```
<bean id="xmppConnection" class="org.springframework.integration.xmpp.XmppConnectionFactoryBean">
 <constructor-arg>
  <bean class="org.jivesoftware.smack.ConnectionConfiguration">
   <constructor-arg value="myServiceName"/>
   <property name="truststorePath" value="..."/>
   <property name="socketFactory" ref="..."/>
  </bean>
 </constructor-arg>
</bean>
<int:channel id="outboundEventChannel"/>

<int-xmpp:outbound-channel-adapter id="outboundEventAdapter"
 channel="outboundEventChannel"
 xmpp-connection="xmppConnection"/>
```

Another important aspect of the Smack API is static initializers. For more complex cases (e.g., registering a SASL Mechanism), you may need to execute certain static initializers. One of those static initializers is `SASLAuthentication`, which allows you to register supported SASL mechanisms. For that level of complexity, we would recommend Spring JavaConfig-style of the XMPP Connection

configuration. Then, you can configure the entire component through Java code and execute all other necessary Java code including static initializers at the appropriate time.

```
@Configuration
public class CustomConnectionConfiguration {
  @Bean
  public XMPPConnection xmppConnection() {
 SASLAuthentication.supportSASLMechanism("EXTERNAL", 0); // static initializer

 ConnectionConfiguration config = new ConnectionConfiguration("localhost", 5223);
 config.setTrustorePath("path_to_truststore.jks");
 config.setSecurityEnabled(true);
 config.setSocketFactory(SSLSocketFactory.getDefault());
 return new XMPPConnection(config);
  }
}
```

For more information on the JavaConfig style of Application Context configuration, refer to the following section in the Spring Reference Manual: http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/beans.html#beans-java

Part V. Appendices

Advanced Topics and Additional Resources

26. Message Publishing

The AOP Message Publishing feature allows you to construct and send a message as a by-product of a 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.

26.1 Message Publishing Configuration

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

Annotation-driven approach via @Publisher annotation

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

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

The `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 a 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 an 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, except that we are not using a default publishing channel. Instead we are specifying the publishing channel via the 'channel' attribute of the @Publisher annotation. We are also adding a @Header annotation which results in the Message header named 'last' having the same value as the 'lname' method parameter. That header will 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 a @Payload annotation on the method, thus explicitly specifying that the return value of the method should be used as the 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 expanding on the previous configuration by using the Spring Expression Language in the @Payload annotation to further instruct the framework 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 the 'lname' input argument. The Message header named 'x' will have its value determined by the 'num' input argument. That header 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 the @Payload annotation. Here we are annotating a method argument which will become the payload of the 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 instead use namespace support for a more concise configuration:

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

Similar to other Spring annotations (@Component, @Scheduled, etc.), `@Publisher` can also be used as a meta-annotation. That means you can define your own annotations that will be treated in the same way as the `@Publisher` itself.

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

Here we defined the `@Audit` annotation which itself is annotated with `@Publisher`. Also note that you can define a `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 the `test()` method will result in a Message with a payload created from its return value. Each Message will be sent to the channel named *auditChannel*. One of the benefits of this technique is that you can avoid the duplication of the same channel name across multiple annotations. You also can provide a level of indirection between your own, potentially domain-specific annotations and those provided by the framework.

You can also annotate the class which would mean that the properties of this annotation will be applied on every public method of that class.

```
@Audit
static class BankingOperationsImpl implements BankingOperations {

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

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

XML-based approach via the <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 looks rather similar to the 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 with the content "Echoing: [value]" where `value` is the value returned by an executed method.
- The Message will have a header with the name "foo" and 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 a header named "foo" whose value 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*.

Async Publishing

One important thing to understand is that publishing occurs in the same thread as your component's execution. So by default in is synchronous. This means that the entire message flow would have to wait until the publisher's flow completes. However, quite often you want the complete opposite and that is to use this Message publishing feature to initiate asynchronous sub-flows. For example, you might host a service (HTTP, WS etc.) which receives a remote request.You may want to send this request internally into a process that might take a while. However you may also want to reply to the user right away. So, instead of sending inbound requests for processing via the output channel (the conventional way), you can simply use 'output-channel' or a 'replyChannel' header to send a simple acknowledgment-like reply back to the caller while using the Message publisher feature to initiate a complex flow.

EXAMPLE: Here is the simple service that receives a complex payload, which needs to be sent further for processing, but it also needs to reply to the caller with a simple acknowledgment.

```
public String echo(Object complexPayload) {
     return "ACK";
}
```

So instead of hooking up the complex flow to the output channel we use the Message publishing feature instead. We configure it to create a new Message using the input argument of the service method (above) and send that to the 'localProcessChannel'. And to make sure this sub-flow is asynchronous all we need to do is send it to any type of asynchronous channel (ExecutorChannel in this example).

```
<int:service-activator  input-channel="inputChannel" output-channel="outputChannel" ref="sampleservice"/>

<bean id="sampleservice" class="test.SampleService"/>

<aop:config>
  <aop:advisor advice-ref="interceptor" pointcut="bean(sampleservice)" />
</aop:config>

<int:publishing-interceptor id="interceptor" >
  <int:method pattern="echo" payload="#args[0]" channel="localProcessChannel">
    <int:header name="sample_header" expression="'some sample value'"/>
  </int:method>
</int:publishing-interceptor>

<int:channel id="localProcessChannel">
  <int:dispatcher task-executor="executor"/>
</int:channel>

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

Another way of handling this type of scenario is with a wire-tap.

Producing and publishing messages based on a scheduled trigger

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 in those cases, you are still responsible for invoking the method. In Spring Integration 2.0 we've added another related useful feature: support for scheduled Message producers/publishers via the new "expression" attribute on the 'inbound-channel-adapter' element. Scheduling could be based on several triggers, any one of which may be configured on the 'poller' sub-element. Currently we support `cron`, `fixed-rate`, `fixed-delay` as well as any custom trigger implemented by you and referenced by the 'trigger' attribute value.

As mentioned above, support for scheduled producers/publishers is provided via the <*inbound-channel-adapter*> xml element. Let's look at couple of examples:

```
<inbound-channel-adapter id="fixedDelayProducer"
       expression="'fixedDelayTest'"
       channel="fixedDelayChannel">
    <poller fixed-delay="1000"/>
</inbound-channel-adapter>
```

In the above example an inbound Channel Adapter will be created which will construct a Message with its payload being the result of the expression defined in the `expression` attribute. Such messages will be created and sent every time the delay specified by the `fixed-delay` attribute occurs.

```
<inbound-channel-adapter id="fixedRateProducer"
       expression="'fixedRateTest'"
       channel="fixedRateChannel">
    <poller fixed-rate="1000"/>
</inbound-channel-adapter>
```

This example is very similar to the previous one, except that we are using the `fixed-rate` attribute which will allow us to send messages at a fixed rate (measuring from the start time of each task).

```
<inbound-channel-adapter id="cronProducer"
       expression="'cronTest'"
       channel="cronChannel">
    <poller cron="7 6 5 4 3 ?"/>
</inbound-channel-adapter>
```

This example demonstrates how you can apply a Cron trigger with a value specified in the `cron` attribute.

```
<inbound-channel-adapter id="headerExpressionsProducer"
       expression="'headerExpressionsTest'"
       channel="headerExpressionsChannel"
       auto-startup="false">
    <poller fixed-delay="5000"/>
    <header name="foo" expression="6 * 7"/>
    <header name="bar" value="x"/>
</inbound-channel-adapter>
```

Here you can see that in a way very similar to the Message publishing feature we are enriching a newly constructed Message with extra Message headers which can take scalar values or the results of evaluating Spring expressions.

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

```
<inbound-channel-adapter id="triggerRefProducer"
       expression="'triggerRefTest'" channel="triggerRefChannel">
    <poller trigger="customTrigger"/>
</inbound-channel-adapter>

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

27. Transaction Support

27.1 Understanding Transactions in Message flows

Spring Integration exposes several hooks to address transactional needs of you message flows. But to better understand these hooks and how you can benefit from them we must first revisit the 6 mechanisms that could be used to initiate Message flows and see how transactional needs of these flows could be addressed within each of these mechanisms.

Here are the 6 mechanisms to initiate a Message flow and their short summary (details for each are provided throughout this manual):

- *Gateway Proxy* - Your basic Messaging Gateway
- *MessageChannel* - Direct interactions with MessageChannel methods (e.g., channel.send(message))
- *Message Publisher* - the way to initiate message flow as the by-product of method invocations on Spring beans
- *Inbound Channel Adapters/Gateways* - the way to initiate message flow based on connecting third-party system with Spring Integration messaging system(e.g., [JmsMessage] -> Jms Inbound Adapter[SI Message] -> SI Channel)
- *Scheduler* - the way to initiate message flow based on scheduling events distributed by a pre-configured Scheduler
- *Poller* - similar to the Scheduler and is the way to initiate message flow based on scheduling or interval-based events distributed by a pre-configured Poller

These 6 could be split in 2 general categories:

- *Message flows initiated by a USER process* - Example scenarios in this category would be invoking a Gateway method or explicitly sending a Message to a MessageChannel. In other words, these message flows depend on a third party process (e.g., some code that we wrote) to be initiated.
- *Message flows initiated by a DAEMON process* - Example scenarios in this category would be a Poller polling a Message queue to initiate a new Message flow with the polled Message or a Scheduler scheduling the process by creating a new Message and initiating a message flow at a predefined time.

Clearly the *Gateway Proxy*, *MessageChannel.send(..)* and *MessagePublisher* all belong to the 1st category and *Inbound Adapters/Gateways*, *Scheduler* and *Poller* belong to the 2nd.

So, how do we address transactional needs in various scenarios within each category and is there a need for Spring Integration to provide something explicitly with regard to transactions for a particular scenario? Or, can Spring's Transaction Support be leveraged instead?.

The first and most obvious goal is NOT to re-invent something that has already been invented unless you can provide a better solution. In our case Spring itself provides first class support for transaction

management. So our goal here is not to provide something new but rather delegate/use Spring to benefit from the existing support for transactions. In other words as a framework we must expose hooks to the Transaction management functionality provided by Spring. But since Spring Integration configuration is based on Spring Configuration it is not always necessary to expose these hooks as they are already exposed via Spring natively. Remember every Spring Integration component is a Spring Bean after all.

With this goal in mind let's look at the two scenarios.

If you think about it, Message flows that are initiated by the *USER process* (Category 1) and obviously configured in a Spring Application Context, are subject to transactional configuration of such processes and therefore don't need to be explicitly configured by Spring Integration to support transactions. The transaction could and should be initiated through standard Transaction support provided by Spring. The Spring Integration message flow will honor the transactional semantics of the components naturally because it is Spring configured. For example, a Gateway or ServiceActivator method could be annotated with `@Transactional` or `TransactionInterceptor` could be defined in an XML configuration with a point-cut expression pointing to specific methods that should be transactional. The bottom line is that you have full control over transaction configuration and boundaries in these scenarios.

However, things are a bit different when it comes to Message flows initiated by the *DAEMON process* (Category 2). Although configured by the developer these flows do not directly involve a human or some other process to be initiated. These are trigger-based flows that are initiated by a trigger process (DAEMON process) based on the configuration of such process. For example, we could have a Scheduler initiating a message flow every Friday night of every week. We can also configure a trigger that initiates a Message flow every second, etc. So, we obviously need a way to let these trigger-based processes know of our intention to make the resulting Message flows transactional so that a Transaction context could be created whenever a new Message flow is initiated. In other words we need to expose some Transaction configuration, but ONLY enough to delegate to Transaction support already provided by Spring (as we do in other scenarios).

Spring Integration provides transactional support for Pollers. Pollers are a special type of component because we can call receive() within that poller task against a resource that is itself transactional thus including *receive()* call in the the boundaries of the Transaction allowing it to be rolled back in case of a task failure. If we were to add the same support for channels, the added transactions would affect all downstream components starting with that *send()* call. That is providing a rather wide scope for transaction demarcation without any strong reason especially when Spring already provides several ways to address the transactional needs of any component downstream. However the *receive()* method being included in a transaction boundary is the "strong reason" for pollers.

Poller Transaction Support

Any time you configure a Poller you can provide transactional configuration via the *transactional* sub-element and its attributes:

```
<poller max-messages-per-poll="1" fixed-rate="1000">
    <transactional transaction-manager="txManager"
                   isolation="DEFAULT"
                   propagation="REQUIRED"
                   read-only="true"
```

```
                    timeout="1000"/>
</poller>
```

As you can see this configuration looks very similar to native Spring transaction configuration. You must still provide a reference to a Transaction manager and specify transaction attributes or rely on defaults (e.g., if the 'transaction-manager' attribute is not specified, it will default to the bean with the name 'transactionManager'). Internally the process would be wrapped in Spring's native Transaction where `TransactionInterceptor` is responsible for handling transactions. For more information on how to configure a Transaction Manager, the types of Transaction Managers (e.g., JTA, Datasource etc.) and other details related to transaction configuration please refer to Spring's Reference manual (Chapter 10 - Transaction Management).

With the above configuration all Message flows initiated by this poller will be transactional. For more information and details on a Poller's transactional configuration please refer to section - *21.1.1. Polling and Transactions*.

Along with transactions, several more cross cutting concerns might need to be addressed when running a Poller. To help with that, the Poller element accepts an *<advice-chain>* sub-element which allows you to define a custom chain of Advice instances to be applied on the Poller. (see section 4.4 for more details) In Spring Integration 2.0, the Poller went through the a refactoring effort and is now using a proxy mechanism to address transactional concerns as well as other cross cutting concerns. One of the significant changes evolving from this effort is that we made *<transactional>* and *<advice-chain>* elements mutually exclusive. The rationale behind this is that if you need more than one advice, and one of them is Transaction advice, then you can simply include it in the *<advice-chain>* with the same convenience as before but with much more control since you now have an option to position any advice in the desired order.

```
<poller max-messages-per-poll="1" fixed-rate="10000">
  <advice-chain>
    <ref bean="txAdvice"/>
    <ref bean="someAotherAdviceBean" />
    <beans:bean class="foo.bar.SampleAdvice"/>
  </advice-chain>
</poller>

<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

As yo can see from the example above, we have provided a very basic XML-based configuration of Spring Transaction advice - "txAdvice" and included it within the *<advice-chain>* defined by the Poller. If you only need to address transactional concerns of the Poller, then you can still use the *<transactional>* element as a convinience.

27.2 Transaction Boundaries

Another important factor is the boundaries of Transactions within a Message flow. When a transaction is started, the transaction context is bound to the current thread. So regardless of how many endpoints and channels you have in your Message flow your transaction context will be preserved as long as you are

ensuring that the flow continues on the same thread. As soon as you break it by introducing a *Pollable Channel* or *Executor Channel* or initiate a new thread manually in some service, the Transactional boundary will be broken as well. Essentially the Transaction will END right there, and if a successful handoff has transpired between the threads, the flow would be considered a success and a COMMIT signal would be sent even though the flow will continue and might still result in an Exception somewhere downstream. If such a flow were synchronous, that Exception could be thrown back to the initiator of the Message flow who is also the initiator of the transactional context and the transaction would result in a ROLLBACK. The middle ground is to use transactional channels at any point where a thread boundary is being broken. For example, you can use a Queue-backed Channel that delegates to a transactional MessageStore strategy, or you could use a JMS-backed channel.

28. Security in Spring Integration

28.1 Introduction

Spring Integration builds upon the Spring Security project [http://static.springframework.org/spring-security/site/] to enable role based security checks to be applied to channel send and receive invocations.

28.2 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 a `ChannelSecurityMetadataSource` which provides the metadata describing the send and receive access policies for certain channels. 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-2.0.xsd
      http://www.springframework.org/schema/integration/security
      http://www.springframework.org/schema/integration/security/spring-integration-security-2.0.xsd">

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

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

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

Appendix A. Spring Integration Samples

A.1 Introduction

As of Spring Integration 2.0, the *samples* are no longer included with the Spring Integration distribution. Instead we have switched to a much simpler collaborative model that should promote better community participation and, ideally, more contributions. Samples now have a dedicated Git repository and a dedicated JIRA Issue Tracking system. Sample development will also have its own lifecycle which is not dependent on the lifecycle of the framework releases although the repository will still be tagged with each major release for compatibility reasons.

The great benefit to the community is that we can now add more samples and make them available to you right away without waiting for the next release. Having its own JIRA that is not tied to the the actual framework is also a great benefit. You now have a dedicated place to suggest samples as well as report issues with existing samples. Or, *you may want to submit a sample to us* as an attachment through the JIRA or, better, through the collaborative model that Git promotes. If we believe your sample adds value, we would be more then glad to add it to the 'samples' repository, properly crediting you as the author.

A.2 Where to get Samples

To monitor samples development and to get more information on the repository you can visit the following URL: http://git.springsource.org/spring-integration/samples Since we are using Git as the SCM, we should use the proper terminology as well when it comes to the tasks you need to perform to make *samples* available locally on your machine. For more information on Git SCM please visit their website: http://git-scm.com/

CLONE *samples* repository. (For those unfamiliar with Git, this is somewhat the equivalent of a checkout.)

This is the first step you should go through. You must have Git installed on your machine. There are many GUI-based products available for many platforms. A simple Google search will help you find them. To clone the Spring Integration samples repository, issue the following at the command line:

```
> mkdir spring-integration-samples
> cd spring-integration-samples
> git clone git://git.springsource.org/spring-integration/samples.git
```

That is all you need to do. Now you have cloned the entire samples repository. Since the samples repository is a live repository, you might want to perform periodic "pulls" to get new samples as well as updates to the existing samples. To get the updates use the git PULL command:

```
> git pull
```

Submit samples or sample requests

As mentioned earlier, Spring Integration *samples* have a dedicated JIRA Issue tracking system. To submit new sample requests or to submit an actual sample (as an attachment), please visit our JIRA Issue Tracking system: https://jira.springsource.org/browse/INTSAMPLES

A.3 Samples Structure

The structure of the *samples* changed as well. With plans for more samples we realized that some samples have different goals than others. While they all share the common goal of showing you how to apply and work with the Spring Integration framework, they also differ in areas where some samples are meant to concentrate on a technical use case while others focus on a business use case, and some samples are all about showcasing various techniques that could be applied to address certain scenarios (both technical and business). The new categorization of samples will allow us to better organize them based on the problem each sample addresses while giving you a simpler way of finding the right sample for your needs.

Currently there are 4 categories. Within the samples repository each category has its own directory which is named after the category name:

BASIC (samples/basic)

This is a good place to get started. The samples here are technically motivated and demonstrate the bare minimum with regard to configuration and code. These should help you to get started quickly by introducing you to the basic concepts, API and configuration of Spring Integration as well as Enterprise Integration Patterns (EIP). For example, if you are looking for an answer on how to implement and wire a *Service Activator* to a *Message Channel* or how to use a *Messaging Gateway* as a facade to your message exchange, or how to get started with using MAIL or TCP/UDP modules etc., this would be the right place to find a good sample. The bottom line is this is a good place to get started.

INTERMEDIATE (samples/intermediate)

This category targets developers who are already familiar with the Spring Integration framework (past getting started), but need some more guidance while resolving the more advanced technical problems one might deal with after switching to a Messaging architecture. For example, if you are looking for an answer on how to handle errors in various message exchange scenarios or how to properly configure the *Aggregator* for the situations where some messages might not ever arrive for aggregation, or any other issue that goes beyond a basic implementation and configuration of a particular component and addresses *what else* types of problems, this would be the right place to find these type of samples.

ADVANCED (samples/advanced)

This category targets developers who are very familiar with the Spring Integration framework but are looking to extend it to address a specific custom need by using Spring Integration's public API. For example, if you are looking for samples showing you how to implement a custom *Channel* or *Consumer* (event-based or polling-based), or you are trying to figure out what is the most appropriate way to implement a custom Bean parser on top of the Spring Integration Bean parser hierarchy when implementing your own namespace and schema for a custom component, this would be the right place to look. Here you can also find samples that will help you with *Adapter* development. Spring Integration

comes with an extensive library of adapters to allow you to connect remote systems with the Spring Integration messaging framework. However you might have a need to integrate with a system for which the core framework does not provide an adapter. So, you may decide to implement your own (and potentially contribute it). This category would include samples showing you how.

APPLICATIONS (samples/applications)

This category targets developers and architects who have a good understanding of Message-driven architecture and EIP, and an above average understanding of Spring and Spring Integration who are looking for samples that address a particular *business problem*. In other words the emphasis of samples in this category is *business use cases* and how they can be solved with a Message-Driven Architecture and Spring Integration in particular. For example, if you are interested to see how a *Loan Broker* or *Travel Agent* process could be implemented and automated via Spring Integration, this would be the right place to find these types of samples.

Important

Remember: Spring Integration is a community driven framework, therefore community participation is IMPORTANT. That includes Samples; so, if you can't find what you are looking for, let us know!

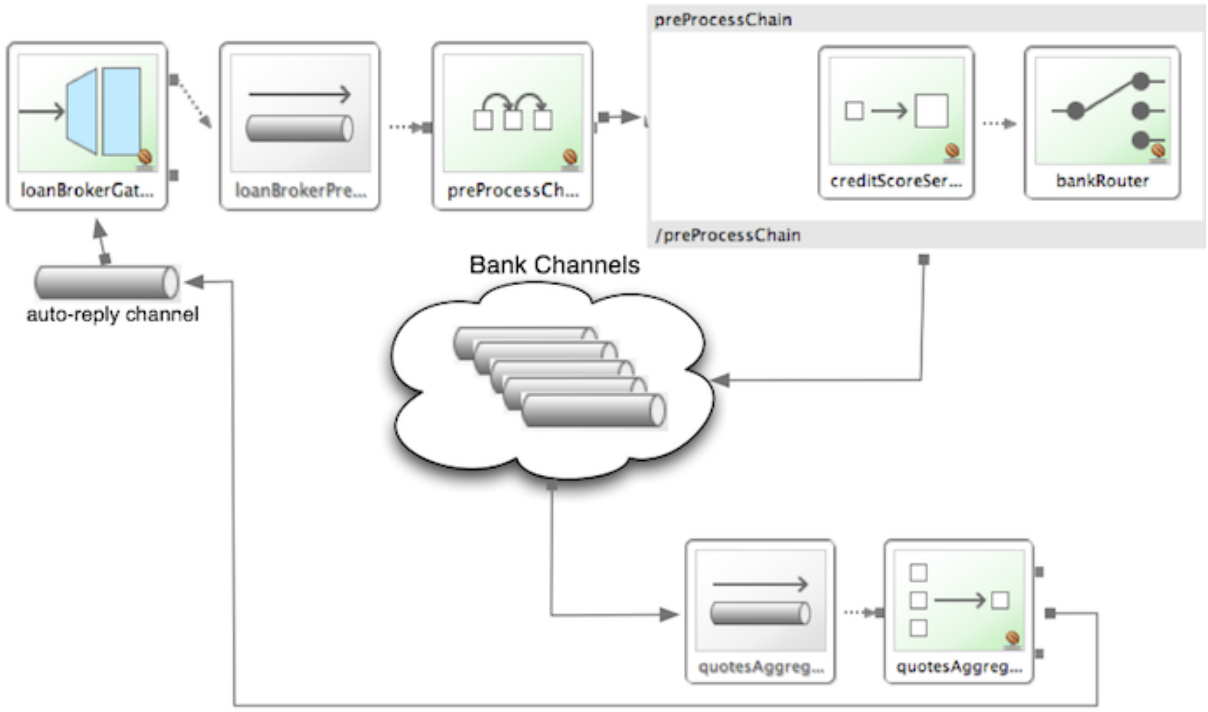
A.4 Samples

Currently Spring Integration comes with quite a few samples and you can only expect more. To help you better navigate through them, each sample comes with its own `readme.txt` file which covers several details about the sample (e.g., what EIP patterns it addresses, what problem it is trying to solve, how to run sample etc.). However, certain samples require a more detailed and sometimes graphical explanation. In this section you'll find details on samples that we believe require special attention.

Loan Broker

In this section, we will review the *Loan Broker* sample application that is included in the Spring Integration samples. This sample is inspired by one of the samples featured in Gregor Hohpe and Bobby Woolf's book, Enterprise Integration Patterns [http://www.eaipatterns.com].

The diagram below represents the entire process



Now lets look at this process in more detail

At the core of an EIP architecture are the very simple yet powerful concepts of Pipes and Filters, and of course: Messages. Endpoints (Filters) are connected with one another via Channels (Pipes). The producing endpoint sends Message to the Channel, and the Message is retrieved by the Consuming endpoint. This architecture is meant to define various mechanisms that describe HOW information is exchanged between the endpoints, without any awareness of WHAT those endpoints are or what information they are exchanging. Thus, it provides for a very loosely coupled and flexible collaboration model while also decoupling Integration concerns from Business concerns. EIP extends this architecture by further defining:

- The types of pipes (Point-to-Point Channel, Publish-Subscribe Channel, Channel Adapter, etc.)
- The core filters and patterns around how filters collaborate with pipes (Message Router, Splitters and Aggregators, various Message Transformation patterns, etc.)

The details and variations of this use case are very nicely described in Chapter 9 of the EIP Book, but here is the brief summary; A Consumer while shopping for the best Loan Quote(s) subscribes to the services of a Loan Broker, which handles details such as:

- Consumer pre-screening (e.g., obtain and review the consumer's Credit history)
- Determine the most appropriate Banks (e.g., based on consumer's credit history/score)
- Send a Loan quote request to each selected Bank
- Collect responses from each Bank
- Filter responses and determine the best quote(s), based on consumer's requirements.

- Pass the Loan quote(s) back to the consumer.

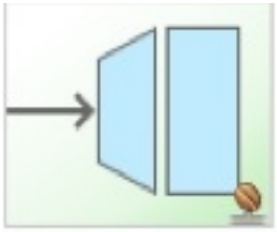
Obviously the real process of obtaining a loan quote is a bit more complex, but since our goal here is to demonstrate how Enterprise Integration Patterns are realized and implemented within SI, the use case has been simplified to concentrate only on the Integration aspects of the process. It is not an attempt to give you an advice in consumer finances.

As you can see, by hiring a Loan Broker, the consumer is isolated from the details of the Loan Broker's operations, and each Loan Broker's operations may defer from one another to maintain competitive advantage, so whatever we assemble/implement must be flexible so any changes could be introduced quickly and painlessly. Speaking of change, the Loan Broker sample does not actually talk to any 'imaginary' Banks or Credit bureaus. Those services are stubbed out. Our goal here is to assemble, orchestrate and test the integration aspect of the process as a whole. Only then can we start thinking about wiring such process to the real services. At that time the assembled process and its configuration will not change regardless of the number of Banks a particular Loan Broker is dealing with, or the type of communication media (or protocols) used (JMS, WS, TCP, etc.) to communicate with these Banks.

DESIGN

As you analyze the 6 requirements above you'll quickly see that they all fall into the category of Integration concerns. For example, in the consumer pre-screening step we need to gather additional information about the consumer and the consumer's desires and enrich the loan request with additional meta information. We then have to filter such information to select the most appropriate list of Banks, and so on. Enrich, filter, select – these are all integration concerns for which EIP defines a solution in the form of patterns. SI provides an implementation of these patterns.

Messaging Gateway



The *Messaging Gateway* pattern provides a simple mechanism to access messaging systems, including our Loan Broker. In SI you define the *Gateway* as a Plain Old Java Interface (no need to provide an implementation), configure it via the XML *<gateway>* element or via annotation and use it as any other Spring bean. SI will take care of delegating and mapping method invocations to the Messaging infrastructure by generating a *Message* (payload is mapped to an input parameter of the method) and sending it to the designated channel.

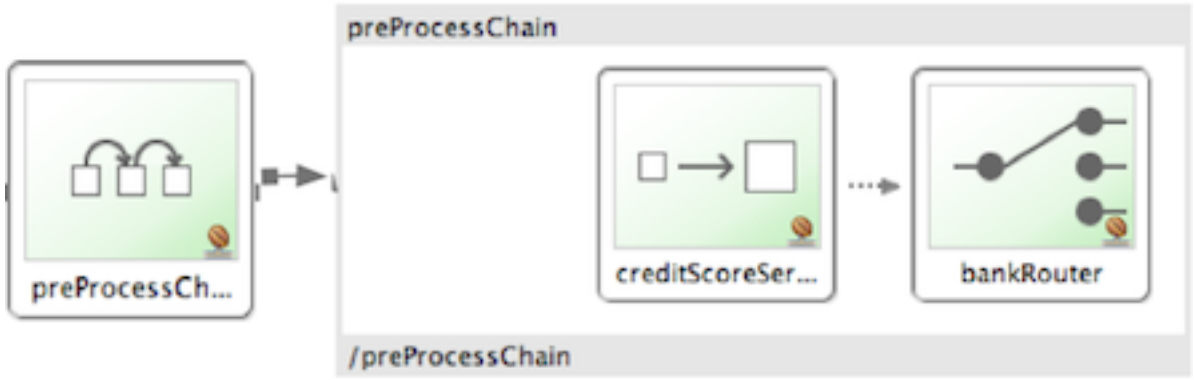
```
<gateway id="loanBrokerGateway"
  default-request-channel="loanBrokerPreProcessingChannel"
  service-interface="org.springframework.integration.samples.loanbroker.LoanBrokerGateway">
  <method name="getBestLoanQuote">
    <header name="RESPONSE_TYPE" value="BEST"/>
  </method>
</gateway>
```

Our current *Gateway* provides two methods that could be invoked. One that will return the best single quote and another one that will return all quotes. Somehow downstream we need to know what type of reply the caller is looking for. The best way to achieve this in Messaging architecture is to enrich the content of the message with some meta-data describing your intentions. *Content Enricher* is one of the patterns that addresses this and although Spring Integration does provide a separate configuration element to enrich Message Headers with arbitrary data (we'll see it later), as a convenience, since *Gateway* element is responsible to construct the initial *Message* it provides embedded capability to enrich the newly created *Message* with arbitrary *Message Headers*. In our example we are adding header RESPONSE_TYPE with value 'BEST" whenever the getBestQuote() method is invoked. For other method we are not adding any header. Now we can check downstream for an existence of this header and based on its presence and its value we can determine what type of reply the caller is looking for.

Based on the use case we also know there are some pre-screening steps that needs to be performed such as getting and evaluating the consumer's credit score, simply because some premiere Banks will only typically accept quote requests from consumers that meet a minimum credit score requirement. So it would be nice if the *Message* would be enriched with such information before it is forwarded to the Banks. It would also be nice if when several processes needs to be completed to provide such meta-information, those processes could be grouped in a single unit. In our use case we need to determine credit score and based on the credit score and some rule select a list of *Message Channels* (Bank Channels) we will sent quote request to.

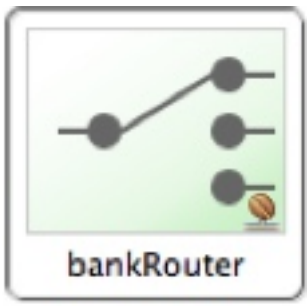
Composed Message Processor

The *Composed Message Processor* pattern describes rules around building endpoints that maintain control over message flow which consists of multiple message processors. In Sprig Integration *Composed Message Processor* pattern is implemented via <*chain*> element.



As you can see from the above configuration we have a chain with inner header-enricher element which will further enrich the content of the *Message* with the header CREDIT_SCORE and value that will be determined by the call to a credit service (simple POJO spring bean identified by 'creditBureau' name) and then it will delegate to the *Message Router*

Message Router



There are several implementations of the *Message Routing* pattern available in Spring Integration. Here we are using a router that will determine a list of channels based on evaluating an expression (Spring Expression Language) which will look at the credit score that was determined is the previous step and will select the list of channels from the Map bean with id 'banks' whose values are 'premier' or 'secondary' based o the value of credit score. Once the list of *Channels* is selected, the *Message* will be routed to those *Channels*.

Now, one last thing the Loan Broker needs to to is to receive the loan quotes form the banks, aggregate them by consumer (we don't want to show quotes from one consumer to another), assemble the response based on the consumer's selection criteria (single best quote or all quotes) and reply back to the consumer.

Message Aggregator

An *Aggregator* pattern describes an endpoint which groups related *Messages* into a single *Message*. Criteria and rules can be provided to determine an aggregation and correlation strategy. SI provides several implementations of the *Aggregator* pattern as well as a convenient name-space based configuration.

```
<aggregator id="quotesAggregator"
      input-channel="quotesAggregationChannel"
      method="aggregateQuotes">
  <beans:bean class="org.springframework.integration.samples.loanbroker.LoanQuoteAggregator"/>
</aggregator>
```

Our Loan Broker defines a 'quotesAggregator' bean via the <*aggregator*> element which provides a default aggregation and correlation strategy. The default correlation strategy correlates messages based on the `correlationId` header (see *Correlation Identifier* pattern). What's interesting is that we never provided the value for this header. It was set earlier by the router automatically, when it generated a separate *Message* for each Bank channel.

Once the *Messages* are correlated they are released to the actual *Aggregator* implementation. Although default *Aggregator* is provided by SI, its strategy (gather the list of payloads from all *Messages* and construct a new *Message* with this List as payload) does not satisfy our requirement. The reason is that our consumer might require a single best quote or all quotes. To communicate the consumer's intention, earlier in the process we set the RESPONSE_TYPE header. Now we have to evaluate this header and return either all the quotes (the default aggregation strategy would work) or the best quote (the default aggregation strategy will not work because we have to determine which loan quote is the best).

Obviously selecting the best quote could be based on complex criteria and would influence the complexity of the aggregator implementation and configuration, but for now we are making it simple. If consumer wants the best quote we will select a quote with the lowest interest rate. To accomplish that the LoanQuoteAggregator.java will sort all the quotes and return the first one. The `LoanQuote.java` implements `Comparable` which compares quotes based on the rate attribute. Once the response *Message* is created it is sent to the default-reply-channel of the *Messaging Gateway* (thus the consumer) which started the process. Our consumer got the Loan Quote!

Conclusion

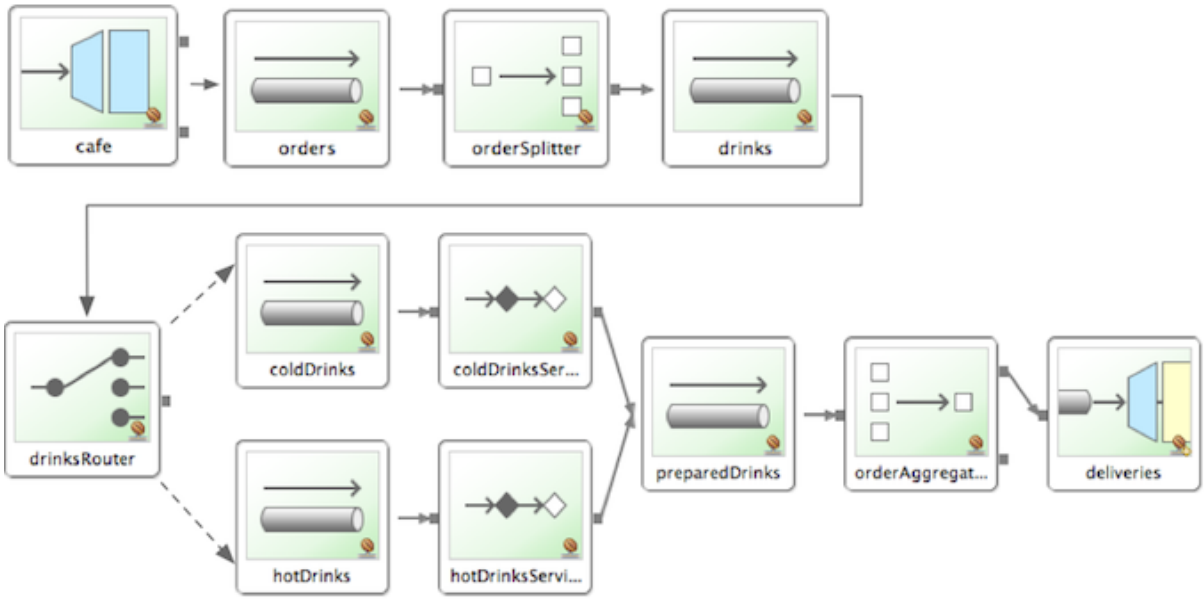
As you can see a rather complex process was assembled based on POJO (read existing, legacy), light weight, embeddable messaging framework (Spring Integration) with a loosely coupled programming model intended to simplify integration of heterogeneous systems without requiring a heavy-weight ESB-like engine or proprietary development and deployment environment, because as a developer you should not be porting your Swing or console-based application to an ESB-like server or implementing proprietary interfaces just because you have an integration concern.

This and other samples in this section are build on top of Enterprise Integration Patterns that meant to describe "building blocks" for YOUR solution but not to be solutions in of themselves. Integration concerns exist in all types of applications (server based and not) and should not require change in design, testing and deployment strategy if such applications need to integrate with one another.

The Cafe Sample

In this section, we will review a *Cafe* sample application that is included in the Spring Integration samples. This sample is inspired by another sample featured in Gregor Hohpe's Ramblings [http://www.eaipatterns.com/ramblings.html].

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-3.0.xsd
              http://www.springframework.org/schema/integration
              http://www.springframework.org/schema/integration/spring-integration-2.0.xsd
              http://www.springframework.org/schema/integration/stream
              http://www.springframework.org/schema/integration/stream/spring-integration-stream-2.0.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" fixed-rate="1000"/>

  </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 = context.getBean("cafe", Cafe.class);
        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" fixed-rate="1000"/>

  </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.

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="orderNamespaceMa
  </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"
```

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 [http://www.eaipatterns.com].

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-3.0.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-2.0.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-3.0.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-2.0.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-3.0.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/spring-integration-2.0.xsd
            http://www.springframework.org/schema/integration/file
            http://www.springframework.org/schema/integration/file/spring-integration-file-2.0.xsd
            http://www.springframework.org/schema/integration/jms
            http://www.springframework.org/schema/integration/jms/spring-integration-jms-2.0.xsd
            http://www.springframework.org/schema/integration/mail
            http://www.springframework.org/schema/integration/mail/spring-integration-mail-2.0.xsd
            http://www.springframework.org/schema/integration/rmi
            http://www.springframework.org/schema/integration/rmi/spring-integration-rmi-2.0.xsd
            http://www.springframework.org/schema/integration/ws
            http://www.springframework.org/schema/integration/ws/spring-integration-ws-2.0.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) {
        ...
    }

}
```

Note

The value of the annotation can also be a SpEL expression (e.g., 'payload.getCustomerId()') which is quite useful when the name of the header has to be dynamically computed. It also provides an optional 'required' property which specifies whether the attribute value must be available within the header. The default value for 'required' is `true`.

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 [http://eaipatterns.com/ReturnAddress.html].

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 Section 3.3, “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.

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 explicit way of mapping Messages to method. As you'll see later on, without an annotation this signature would result in an ambiguous condition. However by explicitly mapping the first argument to a Message Payload and the second argument to a value of the 'foo' Message Header, we have avoided any ambiguity.

```
public String foo(@Payload String s, @RequestParam("foo") String b)
```

Looks almost identical to the previous example, however @RequestMapping or any other non-Spring Integration mapping annotation is irrelevant and therefore will be ignored leaving the second parameter unmapped. Although the second parameter could easily be mapped to a Payload, there can only be one Payload. Therefore this method mapping is 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 the 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 without annotations because it has more than 2 arguments. Furthermore, two of them are Maps. However, with annotation-based mapping, the ambiguity is easily avoided. In this example the first argument is mapped to all the Message Headers, while the second and third argument map to the values of Message Headers 'foo' and 'bar'. The payload is not being mapped to any argument.

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 an Exception being raised.

```
public String foo(String s, int i)
```

- the two parameters are equal in weight, therefore there is no way to determine which one is a payload.

```
public String foo(String s, Map m, String b)
```

- almost the same as above. Although the Map could be easily mapped to Message Headers, there is no way to determine what to do with the 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 than one method argument which is not (Map, <T>), and those parameters are not annotated, will result in an 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 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 methods will trigger an Exception. However if the method names were different you could influence the mapping with a '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 the '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 [http://www.springsource.org/spring-integration] 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.