Spring Integration Reference Manual

3.0.0.M3

Mark Fisher , Marius Bogoevici , Iwein Fuld , Jonas Partner , Oleg Zhurakousky , Gary Russell , Dave Syer , Josh Long , David Turanski , Gunnar Hillert , Artem Bilan , Amol Nayak

Table of Contents

Preface

1 Requirements

This section details the compatible Java and Spring Framework versions.

Compatible Java Versions

For *Spring Integration* **3.0.x**, the **minimum** compatible Java version is **Java SE 6**. Older versions of Java will not be supported any longer.

Spring Integration **3.0.x** is also compatible with **Java SE 7** as well as **Java SE 8** (once released).

Note

Spring Integration **2.2.x** is the last version that is compatible with *Java 5* (J2SE 5.0).

Compatible Versions of the Spring Framework

The default dependency used by *Spring Integration* **3.0.0.RELEASE** is *Spring Framework* **3.1.4.RELEASE**.

Generally, *Spring Integration* **3.0.x** is compatible with the following *Spring Framework* releases:

- *Spring Framework* **3.1.x**
- *Spring Framework* **3.2.x**
- *Spring Framework* **4.0.x**

Note

Spring Integration **2.2.x** is the last version that is compatible with *Spring Framework* **3.0.x**.

2 Code Conventions

The Spring Framework 2.0 introduced support for namespaces, which simplifies the Xml configuration of the application context, and consequently Spring Integration provides broad namespace support. This reference guide applies the following conventions for all code examples that use namespace support:

The **int** namespace prefix will be used for Spring Integration's core namespace support. Each Spring Integration adapter type (module) will provide its own namespace, which is configured using the following convention:

int- followed by the name of the module, e.g. **int-twitter**, **int-stream**, …

```
<?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:int="http://www.springframework.org/schema/integration"
  xmlns:int-twitter="http://www.springframework.org/schema/integration/twitter"
  xmlns:int-stream="http://www.springframework.org/schema/integration/stream"
  xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/integration
   http://www.springframework.org/schema/integration/spring-integration.xsd
   http://www.springframework.org/schema/integration/twitter
   http://www.springframework.org/schema/integration/twitter/spring-integration-
twitter.xsd
   http://www.springframework.org/schema/integration/stream
   http://www.springframework.org/schema/integration/stream/spring-integration-
stream.xsd">
…
</beans>
```

For a detailed explanation regarding Spring Integration's namespace support see *Section F.2, "Namespace Support"*.

Note

Please note that the namespace prefix can be freely chosen. You may even choose not to use any namespace prefixes at all. Therefore, apply the convention that suits your application needs best. Be aware, though, that SpringSource Tool Suite™ (STS) uses the same namespace conventions for Spring Integration as used in this reference guide.

Part I. What's new?

For those who are already familiar with Spring Integration, this chapter provides a brief overview of the new features of version 2.2. If you are interested in the changes and features, that were introduced in earlier versions, please take a look at chapter: Appendix H, *Change History*

1. What's new in Spring Integration 3.0?

This chapter provides an overview of the new features and improvements that have been introduced with Spring Integration 3.0 If you are interested in even more detail, please take a look at the Issue Tracker tickets that were resolved as part of the 3.0 development process.

1.1 New Components

TCP/IP Connection Events and Connection Management

The (supplied) `TcpConnections` now emit `ApplicationEvents` (specifically `TcpConnectionEvents`) when connections are opened, closed, or an exception occurs. This allows applications to be informed of changes to TCP connections using the normal Spring `ApplicationListener` mechanism.

`AbstractTcpConnection` has been renamed `TcpConnectionSupport`; custom connections that are subclasses of this class, can use its methods to publish events. Similarly, `AbstractTcpConnectionInterceptor` has been renamed to `TcpConnectionInterceptorSupport`.

In addition, a new `<int-ip:tcp-connection-event-inbound-channel-adapter/>` is provided; by default, this adapter sends all `TcpConnectionEvents` to a `Channel`.

Further, the TCP Connection Factories, now provide a new method `getOpenConnectionIds()`, which returns a list of identifiers for all open connections; this allows applications, for example, to broadcast to all open connections.

Finally, the connection factories also provide a new method `closeConnection(String connectionId)` which allows applications to explicitly close a connection using its ID.

For more information see Section 16.5, “TCP Connection Events”.

Syslog Support

Building on the 2.2 `SyslogToMapTransformer` Spring Integration 3.0 now introduces `UDP` and `TCP` inbound channel adapters especially tailored for receiving SYSLOG messages. For more information, see Chapter 27, *Syslog Support*.

JMX Support

A new `<int-jmx:tree-polling-channel-adapter/>` is provided; this adapter queries the JMX MBean tree and sends a message with a payload that is the graph of objects that matches the query. By default the MBeans are mapped to primitives and simple Objects like Map, List and arrays - permitting simple transformation, for example, to JSON Section 8.1, “JMX Support”.

'Tail' Support

File 'tail'ing inbound channel adapters are now provided to generate messages when lines are added to the end of text files. the section called “'Tail'ing Files”.

Spring Expression Language (SpEL) Configuration

A new `IntegrationEvaluationContextFactoryBean` is provided to allow configuration of custom `PropertyAccessors` and functions for use in SpEL expressions throughout the framework. For more information see Appendix A, *Spring Expression Language (SpEL)*.

SpEL Functions Support

To customize the SpEL `EvaluationContext` with static `Method` functions the new `<spel-function/>` component is introduced. For more information see Section A.3, “SpEL Functions”.

1.2 General Changes

Aggregator 'empty-group-min-timeout' property

`AbstractCorrelatingMessageHandler` provides a new property `empty-group-min-timeout` to allow empty group expiry to run on a longer schedule than expiring partial groups. Empty groups will not be removed from the `MessageStore` until they have not been modified for at least this number of milliseconds. For more information see the section called “Configuring an Aggregator”.

Advising Filters

Previously, when a <filter/> had a <request-handler-advice-chain/>, the discard action was all performed within the scope of the advice chain (including any downstream flow on the `discard-channel`). The filter element now has an attribute `discard-within-advice` (default `true`), to allow the discard action to be performed after the advice chain completes. See the section called “Advising Filters”.

Advising Endpoints using Annotations

Request Handler Advice Chains can now be configured using annotations. See the section called “Advising Endpoints Using Annotations”.

ObjectToStringTransformer Improvements

This transformer now correctly transforms `byte[]` and `char[]` payloads to `String`. For more information see Section 6.1, “Transformer”.

Web Service Outbound URI Configuration

Web Service Outbound Gateway 'uri' attribute now supports `<uri-variable/>` substitution for all URI-schemes supported by Spring Web Services. For more information see Section 29.4, “Outbound URI Configuration”.

FTP, SFTP and FTPS Cached Sessions

The FTP, SFTP and FTPS endpoints no longer cache sessions by default.

The deprecated `cached-sessions` attribute has been removed from all endpoints. Previously, the embedded caching mechanism controlled by this attribute's value didn't provide a way to limit the size of the cache, which could grow indefinitely. The `CachingConnectionFactory` was introduced in release 2.1 and it became the preferred (and is now the only) way to cache sessions. For more information, see Section 13.6, “FTP Session Caching” and Section 25.3, “SFTP Session Caching”.

FTP, SFTP and FTPS Inbound Adapters

Previously, there was no way to override the default filter used to process files retrieved from a remote server. The `filter` attribute determines which files are retrieved but the `FileReadingMessageSource` uses an `AcceptOnceFileListFilter`. This means that if a new copy of a file is retrieved, with the same name as a previously copied file, no message was sent from the adapter.

With this release, a new attribute `local-filter` allows you to override the default filter, for example with an `AcceptAllFileListFilter`, or some other custom filter.

For users that wish the behavior of the `AcceptOnceFileListFilter` to be maintained across JVM executions, a custom filter that retains state, perhaps on the file system, can now be configured.

FTP, SFTP and FTPS Gateways

The gateways now support the `mv` command, enabling the renaming of remote files.

JDBC Message Store Improvements

Spring Integration 3.0 adds a new set of DDL scripts for *MySQL* version 5.6.4 and higher. Now *MySQL* supports *fractional seconds* and is thus improving the FIFO ordering when polling from a MySQL-based Message Store. For more information, please see the section called “The Generic JDBC Message Store”.

Jackson Support (JSON)

A new abstraction for JSON conversion has been introduced. Implementations for Jackson 1.x and Jackson 2 are currently provided, with the version being determined by presence on the classpath. Previously, only Jackson 1.x was supported. For more information, see 'JSON Transformers' in Section 6.1, “Transformer”.

HTTP Outbound Endpoint 'encode-uri' property

`<http:outbound-gateway/>` and `<http:outbound-channel-adapter/>` now provide an `encode-uri` attribute to allow disabling the encoding of the URI object before sending the request. For more information see Chapter 15, *HTTP Support*.

AMQP Outbound Gateway Header Mapping

Previously, the <int-amqp:outbound-gateway/> mapped headers before invoking the message converter, and the converter could overwrite headers such as `content-type`. The outbound adapter maps the headers after the conversion, which means headers like `content-type` from the outbound `Message` (if present) are used.

Starting with this release, the gateway now maps the headers after the message conversion, consistent with the adapter. If your application relies on the previous behavior (where the converter's headers overrode the mapped headers), you either need to filter those headers (before the message reaches the gateway) or set them appropriately. The headers affected by the `SimpleMessageConverter` are `content-type` and `content-encoding`. Custom message converters may set other headers.

Chain Elements 'id' Attribute

Previously, the *id* attribute for elements within a `<chain>` was ignored and, in some cases, disallowed. Now, the *id* attribute is allowed for all elements within a `<chain>`. The bean names of chain elements

is a combination of the surrounding chain's *id* and the *id* of the element itself. For example: 'fooChain $child.fooTransformer.handler'. For more information see Section 5.6, “Message Handler Chain”.

JMS Message Driven Channel Adapter

Previously, when configuring a `<message-driven-channel-adapter/>`, if you wished to use a specific `TaskExecutor`, it was necessary to declare a container bean and provide it to the adapter using the `container` attribute. The `task-executor` is now provided, allowing it to be set directly on the adapter. This is in addition to several other container attributes that were already available.

RMI Inbound Gateway

The RMI Inbound Gateway now supports an `error-channel` attribute. See Section 24.3, “Inbound RMI”.

SqlReturnType support for Stored Procedure components

For more complex database-specific types, not supported by the standard `CallableStatement.getObject` method, 2 new additional attributes were introduced to the `<sql-parameter-definition/>` element with OUT-direction:

- *type-name*
- *return-type*

For more information see Section 17.5, “Stored Procedures”.

IMAP Idle Connection Exceptions

Previously, if an IMAP idle connection failed, it was logged but there was no mechanism to inform an application. Such exceptions now generate `ApplicationEvents`. Applications can obtain these events using an `<int-event:inbound-channel-adapter>` or any `ApplicationListener` configured to receive an `ImapIdleExceptionEvent` or one of its super classes.

Message ID Generation

Previously, message ids were generated using the JDK `UUID.randomUUID()` method. With this release, the default mechanism has been changed to use the `com.eaio.uuid` package which generates Type 1 UUIDs, and is significantly faster. In addition, the ability to change the strategy used to generate message ids has been added. For more information see the section called “Message ID Generation”.

Message Headers and TCP

The TCP connection factories now enable the configuration of a flexible mechanism to transfer selected headers (as well as the payload) over TCP. A new `TcpMessageMapper` enables the selection of the headers, and an appropriate (de)serializer needs to be configured to write the resulting `Map` to the TCP stream. A `MapJsonSerializer` is provided as a convenient mechanism to transfer headers and payload over TCP. For more information see the section called “Transferring Headers”.

'requires-reply' Attribute for Outbound Gateways

All Outbound Gateways (e.g. `<jdbc:outbound-gateway/>` or `<jms:outbound-gateway/>`) are designed for 'request-reply' scenarios. A response is expected from the external service and will be

published to the `reply-channel`, or the `replyChannel` message header. However, there are some cases where the external system might not always return a result, e.g. a `<jdbc:outbound-gateway/>`, when a SELECT ends with an empty `ResultSet` or, say, a Web Service is One-Way. An option is therefore needed to configure whether or not a *reply* is required. For this purpose, the *requires-reply* attribute has been introduced for Outbound Gateway components. In most cases, the default value for *requires-reply* is `true` and, if there is not any result, a `ReplyRequiredException` will be thrown. Changing the value to `false` means that, if an external service doesn't return anything, the message-flow will end at that point, similar to an Outbound Channel Adapter.

Note

The WebService outbound gateway has an additional attribute `ignore-empty-responses`; this is used to treat an empty String response as if no response was received. It is true by default but can be set to false to allow the application to receive an empty String in the reply message payload. When the attribute is true an empty string is treated as no response for the purposes of the *requires-reply* attribute. *requires-reply* is false by default for the WebService outbound gateway.

Note, the `requiresReply` property was previously present in the `AbstractReplyProducingMessageHandler` but set to `false`, and there wasn't any way to configure it on Outbound Gateways using the XML namespace.

Important

Previously, a gateway receiving no reply would silently end the flow (with a DEBUG log message); with this change an exception will now be thrown by default by most gateways. To revert to the previous behavior, set `requires-reply` to false.

Delayer: delay expression

Previously, the `<delayer>` provided a `delay-header-name` attribute to determine the *delay* value at runtime. In complex cases it was necessary to precede the `<delayer>` with a `<header-enricher>`. Spring Integration 3.0 introduced the `expression` attribute and `expression` sub-element for dynamic delay determination. The `delay-header-name` attribute is now deprecated because the header evaluation can be specified in the `expression`. In addition, the `ignore-expression-failures` was introduced to control the behavior when an expression evaluation fails. For more information see Section 7.4, “Delayer”.

DefaultHttpHeaderMapper and 'If-(Un)Modified-Since' HTTP headers

Previously, 'If-Modified-Since' and 'If-Unmodified-Since' HTTP headers were incorrectly processed within from/to HTTP headers mapping in the `DefaultHttpHeaderMapper`. Now, in addition to the fix of that issue, `DefaultHttpHeaderMapper` provides date parsing from formatted strings for HTTP headers, which accept date-time values. For more information see Chapter 15, *HTTP Support*.

PublishSubscribeChannel Behavior

Previously, sending to a <publish-subscribe-channel/> that had no subscribers would return a `false` result. If used in conjunction with a `MessagingTemplate`, this would result in an exception being thrown. Now, the `PublishSubscribeChannel` has a property `minSubscribers` (default 0). If the message is sent to at least the minimum number of subscribers, the send is deemed to be successful (even if zero). If an application is expecting to get an exception under these conditions, set the minimum subscribers to at least 1.

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. 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 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.

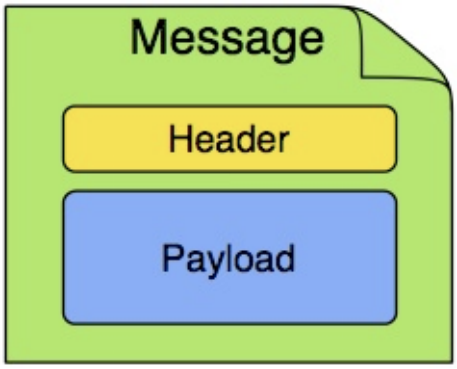
- Extension points should be abstract in nature but within well-defined boundaries to promote *reuse* and *portability*.

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.

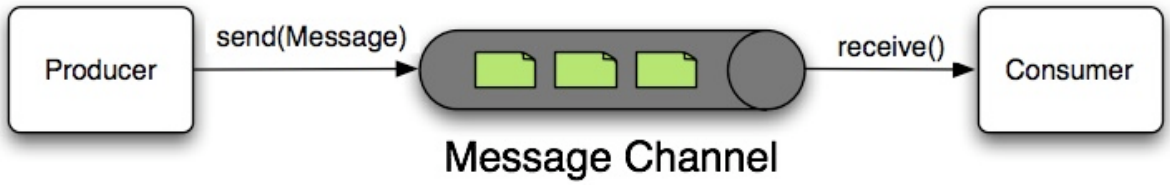
Message

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 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

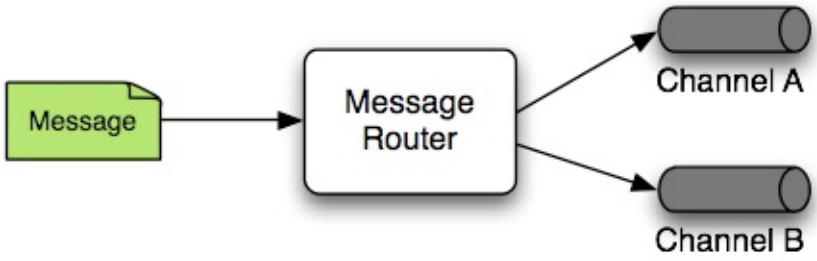
A Message Filter determines whether a Message should be passed to an output channel at all. This simply requires a boolean test method that may check for a particular payload content type, a property value, the presence of a header, etc. If the Message is accepted, it is sent to the output channel, but 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.

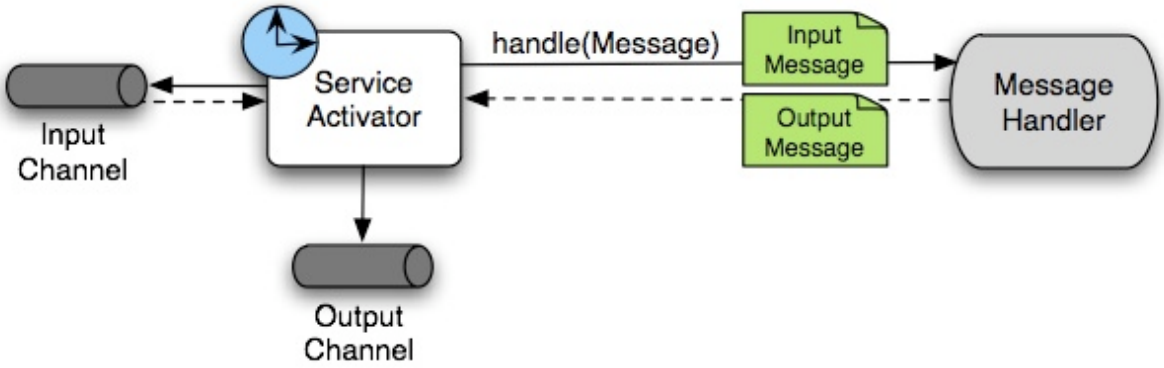
Service Activator

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.

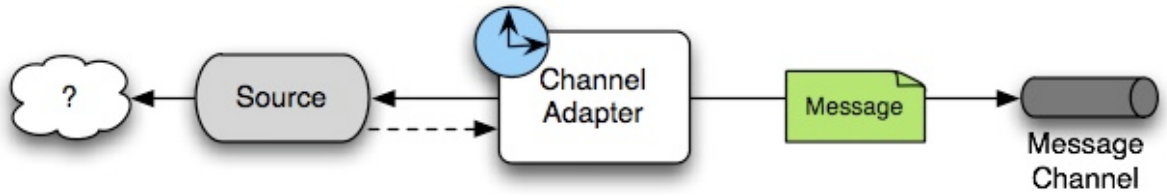
The Service Activator invokes an operation on some service object to process the request Message, extracting the request Message's payload and converting if necessary (if the method does not expect a Message-typed parameter). Whenever the service object's method returns a value, that return value will 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.



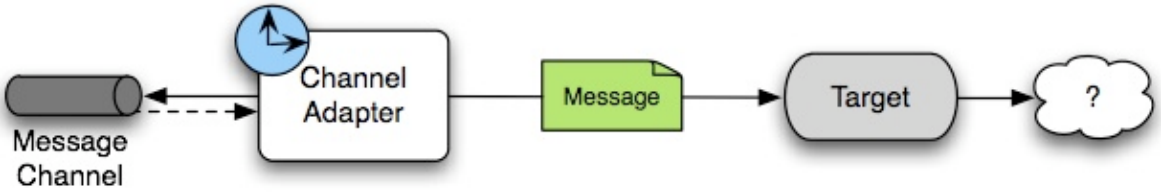
A request-reply "Service Activator" endpoint connects a target object's method to input and output Message Channels.

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 `MessageHandlers`. Therefore, they do not provide receive methods for polling, but instead define methods for managing those subscribers:

```
public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);

}
```

Message Channel Implementations

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

PublishSubscribeChannel

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

Prior to version 3.0, invoking the send method on a `PublishSubscribeChannel` that had no subscribers returned `false`. When used in conjunction with a `MessagingTemplate`, a `MessageDeliveryException` was thrown. Starting with version 3.0, the behavior has changed such that a send is always considered successful if at least the minimum subscribers are present (and successfully handle the message). This behavior can be modified by setting the `minSubscribers` property, which defaults to `0`.

Note

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

QueueChannel

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

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

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

PriorityChannel

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

RendezvousChannel

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

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 19, *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 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 “Enter the GatewayProxyFactoryBean”.

Configuring Message Channels

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

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

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

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

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

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

DirectChannel Configuration

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

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

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

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

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

Datatype Channel Configuration

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

To create a Datatype Channel that only accepts messages containing a certain payload type, provide the fully-qualified class name in the channel element's `datatype` attribute:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note

For more information regarding Payload Type Conversion, please read the section called “Payload Type Conversion”.

QueueChannel Configuration

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

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

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

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

PublishSubscribeChannel Configuration

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

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

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

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

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).

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

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

PriorityChannel Configuration

To create a `PriorityChannel`, use the <priority-queue/> sub-element:

```
<int:channel id="priorityChannel">
    <int:priority-queue capacity="20"/>
</int:channel>
```

By default, the channel will consult the `priority` header of the message. However, a custom `Comparator` reference may be provided instead. Also, note that the `PriorityChannel` (like the other types) does support the `datatype` attribute. As with the QueueChannel, it also supports a `capacity` attribute. The following example demonstrates all of these:

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

RendezvousChannel Configuration

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

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

Scoped Channel Configuration

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

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

Channel Interceptor Configuration

Message channels may also have interceptors as described in 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:

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

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

Global Channel Interceptor Configuration

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

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

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

<int: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 B.1, “Message Publishing Configuration”) is that they are automatically asynchronous in nature. Wire-tap as a component is not invoked asynchronously be default. Instead, Spring Integration focuses on a single unified approach to configuring asynchronous behavior: the Message Channel. What makes certain parts of the message flow *sync* or *async* is the type of *Message Channel* that has been configured within that flow. That is one of the primary benefits of the Message Channel abstraction. From the inception of the framework, we have always emphasized the need and the value of the *Message Channel* as a first-class citizen of the framework. It is not just an internal, implicit realization of the EIP pattern, it is fully exposed as a configurable component to the end user. So, the Wire-tap component is ONLY responsible for performing the following 3 tasks:

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

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

sync or *async*. You may even want to change from *sync* to *async* in the future and *Message Channel* is what's going to allow you to do it swiftly without ever touching the code.

One final point regarding the Wire Tap is that, despite the rationale provided above for not being async 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.

Global Wire Tap Configuration

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

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

Tip

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

Special Channels

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

3.2 Poller (Polling Consumer)

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

- PollingConsumer
- EventDrivenConsumer

The actual implementation depends on which type of channel these Endpoints are connected to. A channel adapter connected to a channel that implements the `org.springframework.integration.core.SubscribableChannel` interface will produce an instance of `EventDrivenConsumer`. On the other hand, a channel adapter connected to a channel that

implements the `org.springframework.integration.core.PollableChannel` interface (e.g. a QueueChannel) will produce an instance of `PollingConsumer`.

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

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

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

Furthermore, in Spring Integration a second variation of the Polling Consumer pattern exists. When Inbound Channel Adapters are being used, these adapters are often wrapped by a `SourcePollingChannelAdapter`. For example, when retrieving messages from a remote FTP Server location, the adapter described in Section 13.3, “FTP Inbound Channel Adapter” is configured with a *poller* to retrieve messages periodically. So, when components are configured with Pollers, the resulting instances are of one of the following types:

- PollingConsumer
- SourcePollingChannelAdapter

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

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

This chapter is meant to only give a high-level overview regarding Polling Consumers and how they fit into the concept of message channels - Section 3.1, “Message Channels” and channel adapters - Section 3.3, “Channel Adapter”. For more in-depth information regarding Messaging Endpoints in general and Polling Consumers in particular, please see Section 7.1, “Message Endpoints”.

3.3 Channel Adapter

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

Configuring An Inbound Channel Adapter

An "inbound-channel-adapter" element can invoke any method on a Spring-managed Object and send a non-null return value to a `MessageChannel` after converting it to a `Message`. When the adapter's

subscription is activated, a poller will attempt to receive messages from the source. The poller will be scheduled with the `TaskScheduler` according to the provided configuration. To configure the polling interval or cron expression for an individual channel-adapter, provide a 'poller' element with one of the scheduling attributes, such as 'fixed-rate' or 'cron'.

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

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

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.

Important

Poller Configuration

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

For example:

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

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

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

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

Note there is no `max-messages-per-poll` specified. As you'll learn later the identical poller configuration in the `PollingConsumer` (e.g., service-activator, filter, router etc.) would have a default value of -1 for `max-messages-per-poll` which means "execute poling task non-stop unless polling method returns null (e.g., no more Messages in the QueueChannel)" and then sleep for 1 second.

However in the SourcePollingChannelAdapter it is a bit different. The default value for `max-messages-per-poll` will be set to 1 by default unless you explicitly set it to a negative value (e.g., -1). It is done so to make sure that poller can react to a LifeCycle events (e.g., start/stop) and prevent it from potentially spinning in the infinite loop if the implementation of the custom method of the `MessageSource` has a potential to never return null and happened to be non-interruptible.

However if you are sure that your method can return null and you need the behavior where you want to poll for as many sources as available per each poll, then you should explicitly set `max-messages-per-poll` to negative value.

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

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.

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

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

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

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

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

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

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

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.

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

3.4 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:

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

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

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

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

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

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

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

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

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	java.lang.Integer

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

Message ID Generation

When a message transitions through an application, each time it is mutated (e.g. by a transformer) a new message id is assigned. The message id is a `UUID`. Beginning with Spring Integration 3.0, the default strategy used for id generation is to use the `com.eaio.uuid` package to generate Type 1 UUIDs. This is much more efficient than the previous `java.util.UUID.randomUUID()` implementation.

A different UUID generation strategy can be selected by declaring a bean that implements `MessageHeaders.IdGenerator` in the application context.

Important

Only one UUID generation strategy can be used in a classloader. This means that if two or more application contexts are running in the same classloader, they will share the same strategy. If one of the contexts changes the strategy, it will be used by all contexts. If two or more contexts in the same classloader declare a bean of type `MessageHeaders.IdGenerator`, they must all be an instance of the same class, otherwise the context attempting to replace a custom strategy will fail to initialize. If the strategy is the same, but parameterized, the strategy in the first context to initialize will be used.

In addition to the default strategy, two additional `IdGenerators` are provided; `MessageHeaders.JdkIdGenerator` uses the previous `UUID.randomUUID()` mechanism;

`MessageHeaders.SimpleIncrementingIdGenerator` can be used in cases where a UUID is not really needed and a simple incrementing value is sufficient.

Important

The default strategy of creating Type 1 UUIDs may present security concerns for some users because the UUID contains the MAC address of a network interface on the platform. For these users, an alternate strategy should be selected.

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(5)
        .build();

assertEquals(5, importantMessage.getHeaders().getPriority());

Message<Integer> lessImportantMessage = MessageBuilder.fromMessage(importantMessage)
        .setHeaderIfAbsent(MessageHeaders.PRIORITY, 2)
        .build();

assertEquals(2, lessImportantMessage.getHeaders().getPriority());
```

The `priority` header is only considered when using a `PriorityChannel` (as described in the next chapter). It is defined as *java.lang.Integer*.

5. Message Routing

5.1 Routers

Overview

Routers are a crucial element in many messaging architectures. They consume Messages from a Message Channel and forward each consumed message to one or more different Message Channel depending on a set of conditions.

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

- *Payload Type Router*
- *Header Value Router*
- *Recipient List Router*
- *XPath Router (Part of the XML Module)*
- *Error Message Exception Type Router*
- *(Generic) Router*

Router implementations share many configuration parameters. Yet, certain differences exist between routers. Furthermore, the availability of configuration parameters depends on whether Routers are used inside or outside of a chain. In order to provide a quick overview, all available attributes are listed in the 2 tables below.

Table 5.1. Routers Outside of a Chain

Attribute	router	header value router	xpath router	payload type router	recipient list router	exception type router
apply-sequence	✓	✓	✓	✓	✓	✓
default-output-channel	✓	✓	✓	✓	✓	✓
resolution-required	✓	✓	✓	✓	✓	✓
ignore-send-failures	✓	✓	✓	✓	✓	✓
timeout	✓	✓	✓	✓	✓	✓
id	✓	✓	✓	✓	✓	✓
auto-startup	✓	✓	✓	✓	✓	✓
input-channel	✓	✓	✓	✓	✓	✓
order	✓	✓	✓	✓	✓	✓

Attribute	router	header value router	xpath router	payload type router	recipient list router	exception type router
method	✓					
ref	✓					
expression	✓					
header-name		✓				
evaluate-as-string			✓			
xpath-expression-ref			✓			
converter			✓			

Table 5.2. Routers Inside of a Chain

Attribute	router	header value router	xpath router	payload type router	recipient list router	exception type router
apply-sequence	✓	✓	✓	✓	✓	✓
default-output-channel	✓	✓	✓	✓	✓	✓
resolution-required	✓	✓	✓	✓	✓	✓
ignore-send-failures	✓	✓	✓	✓	✓	✓
timeout	✓	✓	✓	✓	✓	✓
id						
auto-startup						
input-channel						
order						
method	✓					
ref	✓					
expression	✓					
header-name		✓				
evaluate-as-string			✓			

Attribute	router	header value router	xpath router	payload type router	recipient list router	exception type router
xpath-expression-ref			✓			
converter			✓			

Important

Router parameters have been more standardized across all router implementations with Spring Integration 2.1. Consequently, there are a few minor changes that leave the possibility of breaking older Spring Integration based applications.

Since Spring Integration 2.1 the `ignore-channel-name-resolution-failures` attribute is removed in favor of consolidating its behavior with the `resolution-required` attribute. Also, the `resolution-required` attribute now defaults to `true`.

Prior to these changes, the `resolution-required` attribute defaulted to `false` causing messages to be silently dropped when no channel was resolved and no `default-output-channel` was set. The new behavior will require at least one resolved channel and by default will throw an `MessageDeliveryException` if no channel was determined (or an attempt to send was not successful).

If you do desire to drop messages silently simply set `default-output-channel="nullChannel"`.

Common Router Parameters

Inside and Outside of a Chain

The following parameters are valid for all routers inside and outside of chains.

apply-sequence

This attribute specifies whether sequence number and size headers should be added to each Message. This *optional* attribute defaults to *false*.

default-output-channel

If set, this attribute provides a reference to the channel, where Messages should be sent, if channel resolution fails to return any channels. If no default output channel is provided, the router will throw an Exception. If you would like to silently drop those messages instead, add the `nullChannel` as the default output channel attribute value.

resolution-required

If *true* this attribute specifies that channel names must always be successfully resolved to channel instances that exist. If set to *true*, a `MessagingException` will be raised, in case the channel cannot be resolved. Setting this attribute to *false*, will cause any unresovable channels to be ignored. This *optional* attribute will, if not explicitly set, default to *true*.

ignore-send-failures

If set to *true*, failures to send to a message channel will be ignored. If set to *false*, a `MessageDeliveryException` will be thrown instead, and if the router resolves more than one channel, any subsequent channels will not receive the message.

The exact behavior of this attribute depends on the type of the `Channel` messages are sent to. For example, when using direct channels (single threaded), send-failures can be caused by exceptions thrown by components much further down-stream. However, when sending messages to a simple queue channel (asynchronous) the likelihood of an exception to be thrown is rather remote.

Note

While most routers will route to a single channel, they are allowed to return more than one channel name. The `recipient-list-router`, for instance, does exactly that. If you set this attribute to *true* on a router that only routes to a single channel, any caused exception is simply swallowed, which usually makes little sense to do. In that case it would be better to catch the exception in an error flow at the flow entry point. Therefore, setting the `ignore-send-failures` attribute to *true* usually makes more sense when the router implementation returns more than one channel name, because the other channel(s) following the one that fails would still receive the Message.

This attribute defaults to *false*.

timeout

The `timeout` attribute specifies the maximum amount of time in milliseconds to wait, when sending Messages to the target Message Channels. By default the send operation will block indefinitely.

Top-Level (Outside of a Chain)

The following parameters are valid only across all top-level routers that are ourside of chains.

id

Identifies the underlying Spring bean definition which in case of Routers is an instance of EventDrivenConsumer or PollingConsumer depending on whether the Router's *input-channel* is a *SubscribableChannel* or *PollableChannel*, respectively. This is an *optional* attribute.

auto-startup

This `Lifecycle` attribute signaled if this component should be started during startup of the Application Context. This *optional* attribute defaults to *true*.

input-channel

The receiving Message channel of this endpoint.

order

This attribute defines the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a *failover* dispatching strategy. It has no effect when this endpoint itself is a Polling Consumer for a channel with a queue.

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 F.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:

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

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

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

During the resolution process this router may encounter channel resolution failures, causing an exception. If you want to suppress such exceptions and send unresolved messages to the default output channel (identified with the `default-output-channel` attribute) set `resolution-required` to `false`.

Normally, messages for which the header value is not explicitly mapped to a channel will be sent to the `default-output-channel`. However, in cases where the header value is mapped to a channel name but the channel cannot be resolved, setting the `resolution-required` attribute to `false` will result in routing such messages to the `default-output-channel`.

Important

With Spring Integration 2.1 the attribute was changed from `ignore-channel-name-resolution-failures` to `resolution-required`. Attribute `resolution-required` will default to `true`.

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

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

Note

Since Spring Integration 2.1 the behavior of resolving channels is more explicit. For example, if you ommit the `default-output-channel` attribute and the Router was unable to resolve at least one valid channel, and any channel name resolution failures were ignored by setting `resolution-required` to `false`, then a `MessageDeliveryException` is thrown.

Basically, by default the Router must be able to route messages successfully to at least one channel. If you really want to drop messages, you must also have `default-output-channel` set to `nullChannel`.

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 F.2, “Namespace Support”) as the example below demonstrates.

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

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.containsKey('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.

XPath Router

The XPath Router is part of the XML Module. As such, please read chapter *Routing XML Messages Using XPath*

Routing and Error handling

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

Below is a sample configuration for `ErrorMessageExceptionTypeRouter`.

```
<int:exception-type-router input-channel="inputChannel"
                           default-output-channel="defaultChannel">
    <int:mapping exception-type="java.lang.IllegalArgumentException"
                 channel="illegalChannel"/>
    <int:mapping exception-type="java.lang.NullPointerException"
                 channel="npeChannel"/>
</int:exception-type-router>

<int:channel id="illegalChannel" />
<int:channel id="npeChannel" />
```

Configuring (Generic) Router

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`):

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

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

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

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

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.

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

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

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

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 to implement simple computations that previously required a custom POJO router.

Note

For more information about the Spring Expression Language, please refer to the respective chapter in the Spring Framework Reference Documentation at:

http://static.springsource.org/spring/docs/current/spring-framework-reference/html/expressions.html

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

```
<int:router input-channel="inChannel" expression="payload.paymentType">
    <int:mapping value="CASH" channel="cashPaymentChannel"/>
    <int:mapping value="CREDIT" channel="authorizePaymentChannel"/>
    <int:mapping value="DEBIT" channel="authorizePaymentChannel"/>
</int:router>
```

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

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

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. For further information, please see:

- Collection Projection
- Collection Selection

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 F.5, “Annotation Support”.

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

Note

For routing of XML-based Messages, including XPath support, see Chapter 30, *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 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

```
<int:payload-type-router input-channel="routingChannel">
    <int:mapping type="java.lang.String"  channel="channel1" />
    <int:mapping type="java.lang.Integer" channel="channel2" />
</int: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` as a reference to a bean within the Application Context (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

```
<int:header-value-router input-channel="inputChannel" header-name="testHeader">
    <int:mapping value="foo" channel="fooChannel" />
    <int:mapping value="bar" channel="barChannel" />
</int: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` as a reference to a bean within the Application Context (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:

```
<int: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` as in 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 a 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 basically involves 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 router 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 8.4, “Control Bus”) functionality.

Manage Router Mappings using the Control Bus

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

Note

For more information about the Control Bus, please see chapter *Section 8.4, “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`

Manage Router Mappings using JMX

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

Note

For more information about Spring Integration's JMX suppor, please see chapter *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

Configuring a Filter with XML

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:

```
<int: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.

```
<int: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`:

```
<int: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`:

```
<int: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:

```
<int: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.

```
<int: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.

```
<int: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.

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

<beans:bean id="myExpressions" id="myExpressions"
    class="o.s.i.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.

Configuring a Filter with Annotations

A filter configured using annotations would look like this.

```
public class PetFilter {
    ...
    @Filter ❶
    public boolean dogsOnly(String input) {
        ...
    }
}
```

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

All of the configuration options provided by the xml element are also available for the `@Filter` annotation.

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

Also see the section called “Advising Endpoints Using Annotations”.

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 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:

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

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

<int: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:

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

Note

Using both a `ref` attribute and an inner handler definition in the same `<int: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()
}
```

Also see the section called “Advising Endpoints Using Annotations”.

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 `MessageHeaders.CORRELATION_ID` message header. Messages with the same `MessageHeaders.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 `MessageHeaders.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`

AggregatingMessageHandler

The `AggregatingMessageHandler` (subclass of `AbstractCorrelatingMessageHandler`) 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 `AbstractCorrelatingMessageHandler` and it has a default value based on the `MessageHeaders.CORRELATION_ID` message header:

```
public AbstractCorrelatingMessageHandler(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.

For example:

```
public class MyReleaseStrategy {

    @ReleaseStrategy
    public boolean canMessagesBeReleased(List<Message<?>>) {...}
}
```

```
public class MyReleaseStrategy {

    @ReleaseStrategy
    public boolean canMessagesBeReleased(List<String>) {...}
}
```

As you can see based on the above signatures, the POJO-based Release Strategy will be passed a `Collection` of not-yet-released Messages (if you need access to the whole `Message`) or a `Collection` of payload objects (if the type parameter is anything other than `Message`). Typically this would satisfy the majority of use cases. However if, for some reason, you need to access the full `MessageGroup` then you should simply provide an implementation of the `ReleaseStrategy` interface.

When the group is released for aggregation, all its not-yet-released messages are processed and removed from the group. 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 marked as complete. Any new messages for this group will be sent to the discard channel (if defined). Setting `expire-groups-upon-completion` to `true` (default is `false`) removes the entire group and any new messages, with the same correlation id as the removed group, will form a new group. Partial sequences can be released by using a `MessageGroupStoreReaper` together with `send-partial-result-on-expiry` being set to `true`.

Important

To facilitate discarding of late-arriving messages, the aggregator must maintain state about the group after it has been released. This can eventually cause out of memory conditions. To avoid such situations, you should consider configuring a `MessageGroupStoreReaper` to remove the group metadata; the expiry parameters should be set to expire groups after it is not expected that late messages will arrive. For information about configuring a reaper, see the section called “Managing State in an Aggregator: MessageGroupStore”.

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 an 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
  correlation-strategy-expression="headers['foo']" 12

  ref="aggregatorBean" 13
  method="aggregate" 14

  release-strategy="releaseStrategyBean" 15
  release-strategy-method="release" 16
  release-strategy-expression="size() == 5" 17

  expire-groups-upon-completion="false" 18
  empty-group-min-timeout="60000" /> 19

<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 that expired messages should be aggregated and sent to the 'output-channel' or 'replyChannel' once their containing `MessageGroup` is expired (see `MessageGroupStore.expireMessageGroups(long)`). One way of expiring `MessageGroups` is by configuring a `MessageGroupStoreReaper`. However `MessageGroups` can alternatively be expired by simply calling `MessageGroupStore.expireMessageGroup(groupId)`. That could be accomplished via a Control Bus operation or by simply invoking that method if you have a reference to the `MessageGroupStore` instance. Otherwise by itself this attribute has no behavior. It only serves as an indicator of what to do (discard or send to the output/reply channel) with Messages that are still in the `MessageGroup` that is about to be expired. *Optional*.

Default - 'false'.

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 `MessageHeaders.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 SpEL expression representing the correlation strategy. Example: `"headers['foo']"`. Only one of `correlation-strategy` or `correlation-strategy-expression` is allowed.

13 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).*

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

15 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 `MessageHeaders.SEQUENCE_SIZE` header attribute)*.

16 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).*

17 A SpEL expression representing the release strategy; the root object for the expression is a `Collection` of `Messages`. Example: `"size() == 5"`. Only one of `release-strategy` or `release-strategy-expression` is allowed.

18 When set to true (default false), completed groups are removed from the message store, allowing subsequent messages with the same correlation to form a new group. The default behavior is to send messages with the same correlation as a completed group to the *discard-channel*.

19 Only applies if a `MessageGroupStoreReaper` is configured for the `<aggregator>`'s `MessageStore`. By default, when a `MessageGroupStoreReaper` is configured to expire partial groups, empty groups are also removed. Empty groups exist after a group is released normally. This is to enable the detection and discarding of late-arriving messages. If you wish to expire empty groups on a longer schedule than expiring partial groups, set this property. Empty groups will then not be removed from the `MessageStore` until they have not been modified for at least this number of milliseconds. Note that the actual time to expire an empty group will also be affected by the reaper's *timeout* property and it could be as much as this value plus the timeout.

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 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 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 as a SpEL expression and configure it via the `release-strategy-expression` attribute. The only difference is that since ReleaseStrategy is passed the List of Messages, the root object in the SpEL evaluation context is the List itself. That List can be referenced as `#this` within the expression.

For example:

```
release-strategy-expression="#this.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.

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.

Note

It is important to understand that the 'timeout' property of the `MessageGroupStoreReaper` is an approximate value and is impacted by the the rate of the task scheduler since this property will only be checked on the next scheduled execution of the `MessageGroupStoreReaper` task. For example if the timeout is set for 10 min, but the `MessageGroupStoreReaper` task is scheduled to run every 60 min and the last execution of the `MessageGroupStoreReaper` task happened 1 min before the timeout, the `MessageGroup` will not expire for the next 59 min. So it is recommended to set the rate at least equal to the value of the timeout or shorter.

In addition to the reaper, the expiry callbacks are invoked when the application shuts down via a lifecycle callback in the `AbstractCorrelatingMessageHandler`.

The `AbstractCorrelatingMessageHandler` 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.

Important

When using a `MessageGroupStoreReaper`, it is generally recommended to use a separate `MessageStore` for each correlating endpoint. Otherwise, unexpected results may occur because one endpoint may remove another endpoint's groups.

Some `MessageStore` implementations allow using the same physical resources, by partitioning the data; for example, the `JdbcMessageStore` has a `region` property; the `MongoDbMessageStore` has a `collectionName` property.

For more information about `MessageStore` interface and its implementations, please read Section 8.3, “Message Store”.

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.

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

<int:channel id="outputChannel"/>

<int: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
  correlation-strategy="correlationStrategyBean" ❾
  correlation-strategy-method="correlate" ❿
  correlation-strategy-expression="headers['foo']" 11

  release-strategy="releaseStrategyBean" 12
  release-strategy-method="release" 13
  release-strategy-expression="size() == 10" 14
  empty-group-min-timeout="60000" />15
```

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*.

❾ 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 `MessageHeaders.CORRELATION_ID` header)* .

10 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).*

11 A SpEL expression representing the correlation strategy. Example: `"headers['foo']"`. Only one of `correlation-strategy` or `correlation-strategy-expression` is allowed.

12 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 `MessageHeaders.SEQUENCE_SIZE` header attribute)*.

13 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).*

14 A SpEL expression representing the release strategy; the root object for the expression is a `Collection` of `Messages`. Example: `"size() == 5"`. Only one of `release-strategy` or `release-strategy-expression` is allowed.

15 Only applies if a `MessageGroupStoreReaper` is configured for the `<resequencer>`'s `MessageStore`. By default, when a `MessageGroupStoreReaper` is configured to expire partial groups, empty groups are also removed. Empty groups exist after a group is released normally. This is to enable the detection and discarding of late-arriving messages. If you wish to expire empty groups on a longer schedule than expiring partial groups, set this property. Empty groups will then not be removed from the `MessageStore` until they have not been modified for at least this number of milliseconds. Note that the actual time to expire an empty group will also be affected by the reaper's *timeout* property and it could be as much as this value plus the timeout.

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. 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 a 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 or an outbound-channel-adapter.

```
<int:chain input-channel="input" output-channel="output">
    <int:filter ref="someSelector" throw-exception-on-rejection="true"/>
    <int:header-enricher>
        <int:header name="foo" value="bar"/>
    </int:header-enricher>
    <int:service-activator ref="someService" method="someMethod"/>
</int: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.

The <chain> can be configured as the last 'black-box' consumer of the message flow. For this solution it is enough to put at the end of the <chain> some <outbound-channel-adapter>:

```
<int:chain input-channel="input">
    <si-xml:marshalling-transformer marshaller="marshaller" result-type="StringResult" />
    <int:service-activator ref="someService" method="someMethod"/>
    <int:header-enricher>
        <int:header name="foo" value="bar"/>
    </int:header-enricher>
    <int:logging-channel-adapter level="INFO" log-full-message="true"/>
</int:chain>
```

Disallowed Attributes and Elements

It is important to note that certain attributes, such as **order** and **input-channel** are not allowed to be specified on components used within a *chain*. The same is true for the **poller** sub-element.

Important

For the *Spring Integration* core components, the XML Schema itself will enforce some of these constraints. However, for non-core components or your own custom components, these constraints are enforced by the XML namespace parser, not by the XML Schema.

These XML namespace parser constraints were added with *Spring Integration 2.2*. The XML namespace parser will throw an `BeanDefinitionParsingException` if you try to use disallowed attributes and elements.

'id' Attribute

Beginning with Spring Integration 3.0, if a chain element is given an *id*, the bean name for the element is a combination of the chain's *id* and the *id* of the element itself. Elements without an *id* are not registered as beans, but they are given `componentNames` that include the chain id. For example:

```
<int:chain id="fooChain" input-channel="input">
    <int:service-activator id="fooService" ref="someService" method="someMethod"/>
    <int:object-to-json-transformer/>
</int:chain>
```

- The `<chain>` root element has an *id* 'fooChain'. So, the `AbstractEndpoint` implementation (`PollingConsumer` or `EventDrivenConsumer`, depending on the *input-channel* type) bean takes this value as it's bean name.
- The `MessageHandlerChain` bean acquires a bean alias 'fooChain.handler', which allows direct access to this bean from the `BeanFactory`.
- The `<service-activator>` is not a fully-fledged Messaging Endpoint (`PollingConsumer` or `EventDrivenConsumer`) - it is simply a `MessageHandler` within the `<chain>`. In this case, the bean name registered with the `BeanFactory` is 'fooChain$child.fooService.handler'.
- The *componentName* of this `ServiceActivatingHandler` takes the same value, but without the '.handler' suffix - 'fooChain$child.fooService'.
- The last `<chain>` sub-component, `<object-to-json-transformer>`, doesn't have an *id* attribute. Its *componentName* is based on its position in the `<chain>`. In this case, it is 'fooChain $child#1'. (The final element of the name is the order within the chain, beginning with '#0'). Note, this transformer isn't registered as a bean within the application context, so, it doesn't get a *beanName*, however its *componentName* has a value which is useful for logging etc.

The *id* attribute for `<chain>` elements allows them to be eligible for JMX export and they are trackable via Message History. They can also be accessed from the `BeanFactory` using the appropriate bean name as discussed above.

Tip

It is useful to provide an explicit *id* attribute on `<chain>`s to simplify the identification of sub-components in logs, and to provide access to them from the `BeanFactory` etc.

Calling a Chain from within a Chain

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:

```
<int:chain id="main-chain" input-channel="in" output-channel="out">
    <int:header-enricher>
      <int:header name="name" value="Many" />
    </int:header-enricher>
    <int:service-activator>
      <bean class="org.foo.SampleService" />
    </int:service-activator>
    <int:gateway request-channel="inputA"/>
</int:chain>

<int:chain id="nested-chain-a" input-channel="inputA">
    <int:header-enricher>
        <int:header name="name" value="Moe" />
    </int:header-enricher>
    <int:gateway request-channel="inputB"/>
    <int:service-activator>
        <bean class="org.foo.SampleService" />
    </int:service-activator>
</int:chain>

<int:chain id="nested-chain-b" input-channel="inputB">
    <int:header-enricher>
        <int:header name="name" value="Jack" />
    </int:header-enricher>
    <int:service-activator>
        <bean class="org.foo.SampleService" />
    </int:service-activator>
</int: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, 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 30, *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.

```
<int: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:

```
<int: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.

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.

```
<int: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.

Note

The *object-to-string-transformer* is very simple; it invokes `toString()` on the inbound payload. There are two exceptions to this (since 3.0): if the payload is a `char[]`, it invokes

`new String(payload)`; if the payload is a `byte[]`, it invokes `new String(payload, charset)`, where `charset` is "UTF-8" by default. The `charset` can be modified by supplying the *charset* attribute on the transformer.

For more sophistication (such as selection of the charset dynamically, at runtime), you can use a SpEL expression-based transformer instead; for example:

```
<int:transformer input-channel="in" output-channel="out"
   expression="new java.lang.String(payload, headers['myCharset']" />
```

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.

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

<int: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:

```
<int: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.

JSON Transformers

Object to JSON and *JSON to Object* transformers are provided.

```
<int:object-to-json-transformer input-channel="objectMapperInput"/>
```

```
<int:json-to-object-transformer input-channel="objectMapperInput"
    type="foo.MyDomainObject"/>
```

These use a vanilla Jackson ObjectMapper by default. If you wish to customize the ObjectMapper (for example, to configure the 'ALLOW_COMMENTS' feature when parsing JSON), you can supply a reference to your custom ObjectMapper bean using the object-mapper attribute.

```
<int:json-to-object-transformer input-channel="objectMapperInput"
    type="foo.MyDomainObject" object-mapper="customObjectMapper"/>
```

Note

Beginning with version 3.0, the `object-mapper` attribute references an instance of a new strategy interface `JsonObjectMapper`. This abstraction allows multiple implementations of json mappers to be used. Implementations that wrap Jackson 1.x and Jackson 2 are provided, with

the version being detected on the classpath. These classes are `JacksonJsonObjectMapper` and `Jackson2JsonObjectMapper`.

For backward compatibility, a simple Jackson 1.x `ObjectMapper` can be provided instead of a `JsonObjectMapper`. This will be removed in a future release.

You may wish to consider using a `FactoryBean` or simple factory method to create the `JsonObjectMapper` with the required characteristics.

```
public class ObjectMapperFactory {

    public static Jackson2JsonObjectMapper getMapper() {
        ObjectMapper mapper = new ObjectMapper();
        mapper.configure(JsonParser.Feature.ALLOW_COMMENTS, true);
        return new Jackson2JsonObjectMapper(mapper);
    }
}
```

```
<bean id="customObjectMapper" class="foo.ObjectMapperFactory"
            factory-method="getMapper"/>
```

Important

Beginning with version 2.2, the `object-to-json-transformer` sets the *content-type* header to `application/json`, by default, if the input message does not already have that header present.

It you wish to set the *content type* header to some other value, or explicitly overwrite any existing header with some value (including `application/json`), use the `content-type` attribute. If you wish to suppress the setting of the header, set the `content-type` attribute to an empty string (`""`). This will result in a message with no `content-type` header, unless such a header was present on the input message.

The behavior of adding the default header has a side affect - causing applications with the following sequence to fail:

```
->object-to-json-transformer->amqp-outbound-adapter---->
```

```
---->amqp-inbound-adapter->json-to-object-transformer->
```

This is because the default `SimpleMessageConverter` used by the inbound adapter doesn't recognize this content type and the adapter emits a message with a `byte[]` payload instead of `String`, which was the case with earlier versions.

If you are using this pattern, there are a number of ways to configure the environment so that JSON conversion will be performed correctly.

One solution is to set the content type to a text type, so the inbound converter will convert the JSON to String. This solution requires a change to just the outbound application.

```
<object-to-json-transformer ... content-type="text/x-json"/>
```

The second solution is to eliminate the json transformers altogether and use an `org.springframework.amqp.support.converter.JsonMessageConverter` on both

the outbound and inbound adapters. This configures the adapters to perform the JSON conversion and the transformers are not necessary. The converter on the outbound adapter adds type information to the message properties; the inbound converter uses this type information for the conversion. The converter is provided to the adapters using the *message-converter* attribute. This solution requires a change to both the inbound and outbound applications.

The third solution is to eliminate the *json-to-object-transformer* in just the inbound application and use an `org.springframework.amqp.support.converter.JsonMessageConverter` on the inbound adapter. The converter is provided to the adapter using the *message-converter* attribute. However, because there will be no type information in the message properties, this also requires adding the *defaultType* to the converter, using the same type as currently configured on the *json-to-object-transformer*. This solution requires a change to just the inbound application. The configuration below shows how to configure the message converter; it requires `spring-amqp` 1.1.3 or above.

```
<bean id="jsonConverterWithPOType"
    class="org.springframework.amqp.support.converter.JsonMessageConverter">
    <property name="classMapper">
        <bean class="org.springframework.amqp.support.converter.DefaultClassMapper">
            <property name="defaultType"
                value="foo.PurchaseOrder" />
        </bean>
    </property>
</bean>

<int-amqp:inbound-channel-adapter ... message-
converter="jsonConverterWithPOType" ... />
```

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 F.5, “Annotation Support”

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

Also see the section called “Advising Endpoints Using Annotations”.

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.

The Spring Integration `Core` module includes 2 enrichers:

- Header Enricher
- Payload Enricher

Furthermore, several *Adapter specific Header Enrichers* are included as well:

- XPath Header Enricher (XML Module)
- Mail Header Enricher (Mail Module)
- XMPP Header Enricher (XMPP Module)

Please go to the adapter specific sections of this reference manual to learn more about those adapters.

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 reference using the `ref` and `method` attribute. The specified method 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";
    }

}
```

You can also configure your POJO as inner bean:

```
<int:header-enricher  input-channel="inputChannel" output-channel="outputChannel">
 <int:header name="some_header">
            <bean class="org.MyEnricher"/>
        </int:header>
</int:header-enricher>
```

as well as point to a Groovy script:

```
<int:header-enricher  input-channel="inputChannel" output-channel="outputChannel">
  <int:header name="some_header">
            <int-groovy:script location="org/SampleGroovyHeaderEnricher.groovy"/>
        </int:header>
</int:header-enricher>
```

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.

Tip

For more examples for configuring header enrichers, see Header Enricher Advanced Configuration.

Payload Enricher

In certain situations the Header Enricher, as discussed above, may not be sufficient and payloads themselves may have to be enriched with additional information. For example, order messages that enter the Spring Integration messaging system have to look up the order's customer based on the provided customer number and then enrich the original payload with that information.

Since Spring Integration 2.1, the Payload Enricher is provided. A Payload Enricher defines an endpoint that passes a `Message` to the exposed request channel and then expects a reply message. The reply message then becomes the root object for evaluation of expressions to enrich the target payload.

The Payload Enricher provides full XML namespace support via the `enricher` element. In order to send request messages, the payload enricher has a `request-channel` attribute that allows you to dispatch messages to a request channel.

Basically by defining the request channel, the Payload Enricher acts as a Gateway, waiting for the message that were sent to the request channel to return, and the Enricher then augments the message's payload with the data provided by the reply message.

When sending messages to the request channel you also have the option to only send a subset of the original payload using the `request-payload-expression` attribute.

The enriching of payloads is configured through SpEL expressions, providing users with a maximum degree of flexibility. Therefore, users are not only able to enrich payloads with direct values from the reply channel's `Message`, but they can use SpEL expressions to extract a subset from that Message, only, or to apply addtional inline transformations, allowing them to further manipulate the data.

If you only need to enrich payloads with static values, you don't have to provide the `request-channel` attribute.

Note

Enrichers are a variant of Transformers and in many cases you could use a Payload Enricher or a generic Transformer implementation to add additional data to your messages payloads. Thus, familiarize yourself with all transformation-capable components that are provided by Spring Integration and carefully select the implementation that semantically fits your business case best.

Configuration

Below, please find an overview of all available configuration options that are available for the payload enricher:

```
<int:enricher request-channel=""                          ❶
              auto-startup="true"                         ❷
              id=""                                       ❸
              order=""                                    ❹
              output-channel=""                           ❺
              request-payload-expression=""               ❻
              reply-channel=""                            ❼
              send-timeout=""                             ❽
              should-clone-payload="false">               ❾
    <int:poller></int:poller>                             ❿
    <int:property name="" expression=""/>  11
    <int:property name="" value=""/>
</int:enricher>
```

❶ Channel to which a Message will be sent to get the data to use for enrichment. *Optional*.

❷ Lifecycle attribute signaling if this component should be started during Application Context startup. Defaults to true. *Optional*.

❸ Id of the underlying bean definition, which is either an `EventDrivenConsumer` or a `PollingConsumer`. *Optional*.

❹ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a "failover" dispatching strategy. It has no effect when this endpoint itself is a Polling Consumer for a channel with a queue. *Optional*.

❺ Identifies the Message channel where a Message will be sent after it is being processed by this endpoint. *Optional*.

❻ By default the original message's payload will be used as payload that will be send to the `request-channel`. By specifying a SpEL expression as value for the `request-payload-expression` attribute, a subset of the original payload, a header value or any other resolvable SpEL expression can be used as the basis for the payload, that will be sent to the request-channel.

For the Expression evaluation the full message is available as the 'root object'.

For instance the following SpEL expressions (among others) are possible:

- payload.foo
- headers.foobar
- new java.util.Date()
- 'foo' + 'bar'

If more sophisticated logic is required (e.g. changing the message headers etc.) please use additional downstream transformers. *Optional*.

❼ Channel where a reply Message is expected. This is optional; typically the auto-generated temporary reply channel is sufficient. *Optional*.

❽ Maximum amount of time in milliseconds to wait when sending a message to the channel, if such channel may block.

For example, a Queue Channel can block until space is available, if its maximum capacity has been reached. Internally the send timeout is set on the `MessagingTemplate` and ultimately applied when invoking the send operation on the `MessageChannel`.

By default the send timeout is set to '-1', which may cause the send operation on the `MessageChannel`, depending on the implementation, to block indefinitely. *Optional*.

❾ Boolean value indicating whether any payload that implements `Cloneable` should be cloned prior to sending the Message to the request chanenl for acquiring the enriching data. The cloned version would be used as the target payload for the ultimate reply. Default is `false`. *Optional*.

❿ Allows you to configure a Message Poller if this endpoint is a Polling Consumer. *Optional*.

11 Each `property` sub-element provides the name of a property (via the mandatory `name` attribute). That property should be settable on the target payload instance. Exactly one of the `value` or `expression` attributes must be provided as well. The former for a literal value to set, and the latter for a SpEL expression to be evaluated. The root object of the evaluation context is the Message that was returned from the flow initiated by this enricher, the input Message if there is no request channel, or the application context (using the '@<beanName>.<beanProperty>' SpEL syntax).

Examples

Below, please find several examples of using a Payload Enricher in various situations.

In the following example, a `User` object is passed as the payload of the `Message`. The `User` has several properties but only the `username` is set initially. The Enricher's `request-channel` attribute below is configured to pass the `User` on to the `findUserServiceChannel`.

Through the implicitly set `reply-channel` a `User` object is returned and using the `property` sub-element, properties from the reply are extracted and used to enrich the original payload.

```
<int:enricher id="findUserEnricher"
              input-channel="findUserEnricherChannel"
              request-channel="findUserServiceChannel">
    <int:property name="email"    expression="payload.email"/>
    <int:property name="password" expression="payload.password"/>
</int:enricher>
```

Note

The code samples shown here, are part of the *Spring Integration Samples* project. Please feel free to check it out at: https://github.com/SpringSource/spring-integration-samples

How do I pass only a subset of data to the request channel?

Using a `request-payload-expression` attribute a single property of the payload can be passed on to the request channel instead of the full message. In the example below on the username property is passed on to the request channel. Keep in mind, that alwhough only the username is passed on, the resulting message send to the request channel will contain the full set of `MessageHeaders`.

```
<int:enricher id="findUserByUsernameEnricher"
              input-channel="findUserByUsernameEnricherChannel"
              request-channel="findUserByUsernameServiceChannel"
              request-payload-expression="payload.username">
    <int:property name="email"    expression="payload.email"/>
    <int:property name="password" expression="payload.password"/>
</int:enricher>
```

How can I enrich payloads that consist of Collection data?

In the following example, instead of a `User` object, a `Map` is passed in. The `Map` contains the username under the map key `username`. Only the `username` is passed on to the request channel. The reply contains a full `User` object, which is ultimately added to the `Map` under the `user` key.

```
<int:enricher id="findUserWithMapEnricher"
              input-channel="findUserWithMapEnricherChannel"
              request-channel="findUserByUsernameServiceChannel"
              request-payload-expression="payload.username">
    <int:property name="user"    expression="payload"/>
</int:enricher>
```

How can I enrich payloads with static information without using a request channel?

Here is an example that does not use a request channel at all, but solely enriches the message's payload with static values. But please be aware that the word 'static' is used loosly here. You can still use SpEL expressions for setting those values.

```
<int:enricher id="userEnricher"
              input-channel="input">
    <int:property name="user.updateDate" expression="new java.util.Date()"/>
    <int:property name="user.firstName" value="foo"/>
    <int:property name="user.lastName"  value="bar"/>
    <int:property name="user.age"       value="42"/>
</int:enricher>
```

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*
- *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*.

Here is an overview of all available parameters of an Incoming Claim Check Transformer:

```
<int:claim-check-in auto-startup="true" ❶
                    id=""                           ❷
                    input-channel=""                ❸
                    message-store="messageStore"    ❹
                    order=""                        ❺
                    output-channel=""               ❻
                    send-timeout="">                ❼
    <int:poller></int:poller>                       ❽
</int:claim-check-in>
```

❶ Lifecycle attribute signaling if this component should be started during Application Context startup. Defaults to true. Attribute is not available inside a `Chain` element. *Optional*.

❷ Id identifying the underlying bean definition (`MessageTransformingHandler`). Attribute is not available inside a `Chain` element. *Optional*.

❸ The receiving Message channel of this endpoint. Attribute is not available inside a `Chain` element. *Optional*.

❹ Reference to the MessageStore to be used by this Claim Check transformer. If not specified, the default reference will be to a bean named *messageStore*. *Optional*.

❺ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a *failover* dispatching strategy. It has no effect when this endpoint itself is a Polling Consumer for a channel with a queue. Attribute is not available inside a `Chain` element. *Optional*.

❻ Identifies the Message channel where Message will be sent after its being processed by this endpoint. Attribute is not available inside a `Chain` element. *Optional*.

❼ Specify the maximum amount of time in milliseconds to wait when sending a reply Message to the output channel. By default the send will block for one second. Attribute is not available inside a `Chain` element. *Optional*.

❽ Defines a poller. Element is not available inside a `Chain` element. *Optional*.

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.

```
<int: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`.

Here is an overview of all available parameters of an Outgoing Claim Check Transformer:

```
<int:claim-check-out auto-startup="true" ❶
                     id=""                          ❷
                     input-channel=""               ❸
                     message-store="messageStore"   ❹
                     order=""                       ❺
                     output-channel=""              ❻
                     remove-message="false"         ❼
                     send-timeout="">               ❽
    <int:poller></int:poller>                       ❾
</int:claim-check-out>
```

❶ Lifecycle attribute signaling if this component should be started during Application Context startup. Defaults to true. Attribute is not available inside a `Chain` element. *Optional*.

❷ Id identifying the underlying bean definition (`MessageTransformingHandler`). Attribute is not available inside a `Chain` element. *Optional*.

❸ The receiving Message channel of this endpoint. Attribute is not available inside a `Chain` element. *Optional*.

❹ Reference to the MessageStore to be used by this Claim Check transformer. If not specified, the default reference will be to a bean named *messageStore*. *Optional*.

❺ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a *failover* dispatching strategy. It has no effect when this endpoint itself is a Polling Consumer for a channel with a queue. Attribute is not available inside a `Chain` element. *Optional*.

❻ Identifies the Message channel where Message will be sent after its being processed by this endpoint. Attribute is not available inside a `Chain` element. *Optional*.

❼ If set to `true` the Message will be removed from the MessageStore by this transformer. Useful when Message can be "claimed" only once. Defaults to `false`. *Optional*.

❽ Specify the maximum amount of time in milliseconds to wait when sending a reply Message to the output channel. By default the send will block for one second. Attribute is not available inside a `Chain` element. *Optional*.

❾ Defines a poller. Element is not available inside a `Chain` element. *Optional*.

Claim Once

There are scenarios when a particular message must be claimed only once. As an analogy, consider the airplane luggage check-in/out process. Checking-in your luggage on departure and and then claiming it on arrival is a classic example of such a scenario. Once the luggage has been claimed, it can not be claimed again without first checking it back in. To accommodate such cases, we introduced a `remove-message` boolean attribute on the `claim-check-out` transformer. This attribute is set to `false` by default. However, if set to `true`, the claimed Message will be removed from the MessageStore, so that it can no longer be claimed again.

This is also something to consider in terms of storage space, especially in the case of the in-memory Map-based `SimpleMessageStore`, where failing to remove the Messages could ultimately lead to an `OutOfMemoryException`. Therefore, if you don't expect multiple claims to be made, it's recommended that you set the `remove-message` attribute's value to `true`.

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

A word on Message Store

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.

`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 and Event Driven Consumers.

Of the two, Event Driven Consumers are much simpler. Without any need to manage and schedule a separate poller thread, they are essentially just listeners with a callback method. When connecting to one of Spring Integration's subscribable Message Channels, this simple option works great. However, when connecting to a buffering, pollable Message Channel, some component has to schedule and manage the polling thread(s). Spring Integration provides two different endpoint implementations to accommodate these two types of consumers. Therefore, the consumers themselves can simply implement the callback interface. When polling is required, the endpoint acts as a *container* for the consumer instance. The benefit is similar to that of using a container for hosting Message Driven Beans, but since these consumers are simply Spring-managed Objects running within an ApplicationContext, it more closely resembles Spring's own MessageListener containers.

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);
```

Note

For more information regarding Polling Consumers, please also read Section 3.2, “Poller (Polling Consumer)” as well as Section 3.3, “Channel Adapter”.

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`, as illustrated in the following example:

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

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

Furthermore, a `PollingConsumer` has a property called *adviceChain*. This property allows you to specify a `List` of AOP Advices for handling additional cross cutting concerns including transactions. These advices are applied around the `doPoll()` method. For more in-depth information, please see the sections *AOP Advice chains* and *Transaction Support* under the section called “Namespace Support”.

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` called `ConsumerEndpointFactoryBean` that creates the appropriate consumer type based on the type of channel, and there is full XML namespace support to even further hide those details. 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.

Configuration

Below you find a *poller* with all available configuration options:

```
<int:poller cron=""                                    ❶
            default="false"                            ❷
            error-channel=""                           ❸
            fixed-delay=""                             ❹
            fixed-rate=""                              ❺
            id=""                                      ❻
            max-messages-per-poll=""                   ❼
            receive-timeout=""                         ❽
            ref=""                                     ❾
            task-executor=""                           ❿
            time-unit="MILLISECONDS"                   11
            trigger="">                                12
            <int:advice-chain />                       13
            <int:transactional />                      14
</int:poller>
```

❶ Provides the ability to configure Pollers using Cron expressions. The underlying implementation uses a `org.springframework.scheduling.support.CronTrigger`. If this attribute is set, none of the following attributes must be specified: `fixed-delay`, `trigger`, `fixed-rate`, `ref`.

❷ By setting this attribute to *true*, it is possible to define exactly one (1) global default poller. An exception is raised if more than one default poller is defined in the application context. Any endpoints connected to a PollableChannel (PollingConsumer) or any SourcePollingChannelAdapter that does not have any explicitly configured poller will then use the global default Poller. *Optional*. Defaults to `false`.

❸ Identifies the channel which error messages will be sent to if a failure occurs in this poller's invocation. To completely suppress Exceptions, provide a reference to the `nullChannel`. *Optional*.

❹ The fixed delay trigger uses a `PeriodicTrigger` under the covers. If the `time-unit` attribute is not used, the specified value is represented in milliseconds. If this attribute is set, none of the following attributes must be specified: `fixed-rate`, `trigger`, `cron`, `ref`.

❺ The fixed rate trigger uses a `PeriodicTrigger` under the covers. If the `time-unit` attribute is not used the specified value is represented in milliseconds. If this attribute is set, none of the following attributes must be specified: `fixed-delay`, `trigger`, `cron`, `ref`.

❻ The Id referring to the Poller's underlying bean-definition, which is of type `org.springframework.integration.scheduling.PollerMetadata`. The *id* attribute is required for a top-level poller element unless it is the default poller (`default="true"`).

❼ Please see the section called “Configuring An Inbound Channel Adapter” for more information. *Optional*. If not specified the default values used depends on the context. If a `PollingConsumer` is used, this atribute will default to *-1*. However, if a `SourcePollingChannelAdapter` is used, then the `max-messages-per-poll` attribute defaults to *1*.

❽ Value is set on the underlying class `PollerMetadata` *Optional*. If not specified it defaults to 1000 (milliseconds).

❾ Bean reference to another top-level poller. The `ref` attribute must not be present on the top-level `poller` element. However, if this attribute is set, none of the following attributes must be specified: `fixed-rate`, `trigger`, `cron`, `fixed-deleay`.

❿ Provides the ability to reference a custom *task executor*. Please see the section below titled *TaskExecutor Support* for further information. *Optional*.

11 This attribute specifies the `java.util.concurrent.TimeUnit` enum value on the underlying `org.springframework.scheduling.support.PeriodicTrigger`. Therefore, this attribute can *ONLY* be used in combination with the `fixed-delay` or `fixed-rate` attributes. If combined with either `cron` or a `trigger` reference attribute, it will cause a failure.

The minimal supported granularity for a `PeriodicTrigger` is MILLISECONDS. Therefore, the only available options are MILLISECONDS and SECONDS. If this value is not provided, then any `fixed-delay` or `fixed-rate` value will be interpreted as MILLISECONDS by default.

Basically this enum provides a convenience for SECONDS-based interval trigger values. For hourly, daily, and monthly settings, consider using a `cron` trigger instead.

12 Reference to any spring configured bean which implements the `org.springframework.scheduling.Trigger` interface. *Optional*. However, if this attribute is set, none of the following attributes must be specified: `fixed-delay`, `fixed-rate`, `cron`, `ref`.

13 Allows to specify extra AOP Advices to handle additional cross cutting concerns. Please see the section below titled *Transaction Support* for further information. *Optional*.

14 Pollers can be made transactional. Please see the section below titled *AOP Advice chains* for further information. *Optional*.

Examples

For example, a simple interval-based poller with a 1-second interval would be configured like this:

```
<int:transformer input-channel="pollable"
    ref="transformer"
    output-channel="output">
    <int:poller fixed-rate="1000"/>
</int: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:

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

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

<int:transformer input-channel="pollable"
    ref="transformer"
    output-channel="output">
    <int:poller ref="weekdayPoller"/>
</int: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.

Global Default Pollers

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.

```
<int:poller id="defaultPoller" default="true" max-messages-per-poll="5" fixed-rate="3000"/
>

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

Transaction Support

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:

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

For more information please refer to the section called “Poller Transaction Support”.

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 poller, 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..

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

For more information on how to implement MethodInterceptor please refer to AOP sections of Spring reference manual (section 8 and 9). 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.

TaskExecutor Support

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:

```
<int: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 F.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.

```
<int:service-activator input-channel="someQueueChannel"
    output-channel="output">
    <int:poller receive-timeout="30000" fixed-rate="10"/>

</int: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.

Change Polling Rate at Runtime

When configuring Pollers with a `fixed-delay` or `fixed-rate` attribute, the default implementation will use a `PeriodicTrigger` instance. The `PeriodicTrigger` is part of the Core Spring Framework and it accepts the *interval* as a constructor argument, only. Therefore it cannot be changed at runtime.

However, you can define your own implementation of the `org.springframework.scheduling.Trigger` interface. You could even use the PeriodicTrigger as a starting point. Then, you can add a setter for the interval (period), or you could even embed your own throttling logic within the trigger itself if desired. The *period* property will be used with each call to *nextExecutionTime* to schedule the next poll. To use this custom trigger within pollers, declare the bean defintion of the custom Trigger in your application context and inject the dependency into your Poller configuration using the `trigger` attribute, which references the custom Trigger bean instance. You can now obtain a reference to the Trigger bean and the polling interval can be changed between polls.

For an example, please see the Spring Integration Samples project. It contains a sample called *dynamic-poller*, which uses a custom Trigger and demonstrates the ability to change the polling interval at runtime.

- https://github.com/SpringSource/spring-integration-samples/tree/master/intermediate

The sample provides a custom Trigger which implements the *org.springframework.scheduling.Trigger* interface. The sample's Trigger is based on Spring's PeriodicTrigger implementation. However, the fields of the custom trigger are not final and the properties have explicit getters and setters, allowing to dynamically change the polling period at runtime.

Note

It is important to note, though, that because the Trigger method is *nextExecutionTime()*, any changes to a dynamic trigger will not take effect until the next poll, based on the existing configuration. It is not possible to force a trigger to fire before it's currently configured next execution time.

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="o.s.i.config.xml.ConverterParserTests$TestConverter3"/>
</int:converter>
```

Important

When configuring an *Application Context*, the Spring Framework allows you to add a *conversionService* bean (see Configuring a ConversionService chapter). This service is used, when needed, to perform appropriate conversions during bean creation and configuration.

In contrast, the *integrationConversionService* is used for runtime conversions. These uses are quite different; converters that are intended for use when wiring bean constructor-args and properties may produce unintended results if used at runtime for Spring Integration expression evaluation against Messages within Datatype Channels, Payload Type transformers etc.

However, if you do want to use the Spring *conversionService* as the Spring Integration *integrationConversionService*, you can configure an *alias* in the Application Context:

```
<alias name="conversionService" alias="integrationConversionService"/>
```

In this case the *conversionService*'s Converters will be available for Spring Integration runtime conversion.

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"/>
</int: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 Messaging Gateways

The primary purpose of a Gateway is to hide the messaging API provided by Spring Integration. It allows your application's business logic to be completely unaware of the Spring Integration API and using a generic Gateway, your code interacts instead with a simple interface, only.

Enter the GatewayProxyFactoryBean

As mentioned above, it would be great to have no dependency on the Spring Integration API at all - including the gateway class. For that reason, Spring Integration provides the `GatewayProxyFactoryBean` that generates a proxy for any interface and internally invokes the gateway methods shown below. Using dependency injection you can then expose the interface to your business methods.

Here is an example of an interface that can be used to interact with Spring Integration:

```
package org.cafeteria;

public interface Cafe {

    void placeOrder(Order order);

}
```

Gateway XML Namespace Support

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

```
<int: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).

Setting the Default Reply Channel

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.

Gateway Configuration with Annotations and/or XML

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.

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

Invoking No-Argument Methods

When invoking methods on a Gateway interface that do not have any arguments, the default behavior is to *receive* a `Message` from a `PollableChannel`.

At times however, you may want to trigger no-argument methods so that you can in fact interact with other components downstream that do not require user-provided parameters, e.g. triggering no-argument SQL calls or Stored Procedures.

In order to achieve *send-and-receive* semantics, you must provide a payload. In order to generate a payload, method parameters on the interface are not necessary. You can either use the `@Payload` annotation or the `payload-expression` attribute in XML on the `method` sub-element. Below please find a few examples of what the payloads could be:

- a literal string
- #method (for the method name)
- new java.util.Date()
- @someBean.someMethod()'s return value

Here is an example using the `@Payload` annotation:

```
public interface Cafe {

    @Payload("new java.util.Date()")
    List<Order> retrieveOpenOrders();

}
```

If a method has no argument and no return value, but does contain a payload expression, it will be treated as a *send-only* operation.

Error Handling

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.

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

<int: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.

Asynchronous Gateway and AsyncTaskExecutor

By default `GatewayProxyFactoryBean` uses `org.springframework.core.task.SimpleAsyncTaskExecutor` when submitting internal `AsyncInvocationTask` instances for any gateway method whose return type is `Future.class`. However the `async-executor` attribute in the `<gateway/>` element's configuration allows you to provide a reference to any implementation of `java.util.concurrent.Executor` available within the Spring application context.

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.

Note

* *reply-timeout* is unbounded for *<gateway/>* elements (created by the GatewayProxyFactoryBean). Inbound gateways for external integration (ws, http, etc.) share many characteristics and attributes with these gateways. However, for those inbound gateways, the default *reply-timeout* is 1000 milliseconds (1 second). If a downstream async handoff is made to another thread, you may need to increase this attribute to allow enough time for the flow to complete before the gateway times out.

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:

```
<int: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.

```
<int: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:

```
<int: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 F.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:

```
<int:service-activator id="exampleServiceActivator" input-channel="inChannel"
            output-channel = "outChannel" method="foo">
    <beans:bean class="org.foo.ExampleServiceActivator"/>
</int: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 `org.springframework.scheduling.TaskScheduler` 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 has 'default-delay' and 'expression' attributes (and 'expression' sub-element) that are used to determine the number of milliseconds that each Message should be delayed. The following delays all messages by 3 seconds:

```
<int:delayer id="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 SpEL expression using the 'expression' attribute:

```
<int:delayer id="delayer" input-channel="input" output-channel="output"
             default-delay="3000" expression="headers['delay']"/>
```

In the example above, the 3 second delay would only apply when the expression evaluates to *null* for a given inbound Message. If you only want to apply a delay to Messages that have a valid result of the expression evaluation, then you can use a 'default-delay' of 0 (the default). For any Message that has a delay of 0 (or less), the Message will be sent immediately, on the calling Thread.

Tip

The delay handler supports expression evaluation results 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 first 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). With a Date instance, the Message will not be released until the time represented by that Date object. 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 on the original sender's Thread. If the expression evaluation result is not a Date, and can not be parsed as a Long, the default delay (if any) will be applied.

Important

The expression evaluation may throw an evaluation Exception for various reasons, including an invalid expression, or other conditions. By default, such exceptions are ignored (logged at DEBUG level) and the delayer falls back to the default delay (if any). You can modify this behavior by setting the `ignore-expression-failures` attribute. By default this attribute is set to `true` and the Delayer behavior is as described above. However, if you wish to not ignore expression evaluation exceptions, and throw them to the delayer's caller, set the `ignore-expression-failures` attribute to `false`.

Tip

Notice in the example above that the delay expression is specified as `headers['delay']`. This is the SpEL `Indexer` syntax to access a `Map` element (`MessageHeaders` implements `Map`), it invokes: `headers.get("delay")`. For simple map element names (that do not contain '.') you can also use the SpEL *dot accessor* syntax, where the above header expression can be specified as `headers.delay`. But, different results are achieved if the header is missing. In the first case, the expression will evaluate to `null`; the second will result in something like:

```
org.springframework.expression.spel.SpelEvaluationException: EL1008E:(pos 8):
    Field or property 'delay' cannot be found on object of
type 'org.springframework.integration.MessageHeaders'
```

So, if there is a possibility of the header being omitted, and you want to fall back to the default delay, it is generally more efficient (and recommended) to use the *Indexer* syntax instead of *dot property accessor* syntax, because detecting the null is faster than catching an exception.

The delayer delegates to an instance of Spring's `TaskScheduler` abstraction. The default scheduler used by the delayer is the `ThreadPoolTaskScheduler` instance provided by Spring Integration on startup: Section F.3, “Configuring the Task Scheduler”. If you want to delegate to a different scheduler, you can provide a reference through the delayer element's 'scheduler' attribute:

```
<int:delayer id="delayer" input-channel="input" output-channel="output"
    expression="headers.delay"
    scheduler="exampleTaskScheduler"/>

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

Tip

If you configure an external `ThreadPoolTaskScheduler` you can set on this scheduler property `waitForTasksToCompleteOnShutdown = true`. It allows successful completion of 'delay' tasks, which already in the execution state (releasing the Message), when the application is shutdown. Before Spring Integration 2.2 this property was available on the `<delayer>` element, because `DelayHandler` could create its own scheduler on the background. Since 2.2 delayer requires an external scheduler instance and `waitForTasksToCompleteOnShutdown` was deleted; you should use the scheduler's own configuration.

Tip

Also keep in mind `ThreadPoolTaskScheduler` has a property `errorHandler` which can be injected with some implementation of `org.springframework.util.ErrorHandler`. This handler allows to process an `Exception` from the thread of the scheduled task sending the delayed message. By default it uses an `org.springframework.scheduling.support.TaskUtils$LoggingErrorHandler` and you will see a stack trace in the logs. You might want to consider using an `org.springframework.integration.channel.MessagePublishingErrorHandler`, which sends an `ErrorMessage` into an `error-channel`, either from the failed Message's header or into the default `error-channel`.

Delayer and Message Store

The `DelayHandler` persists delayed Messages into the Message Group in the provided `MessageStore`. (The 'groupId' is based on required 'id' attribute of `<delayer>` element.) A delayed message is removed from the `MessageStore` by the scheduled task just before the `DelayHandler` sends the Message to the `output-channel`. If the provided `MessageStore` is persistent (e.g. `JdbcMessageStore`) it provides the ability to not lose Messages on the application shutdown. After application startup, the `DelayHandler` reads Messages from its Message Group in the `MessageStore` and reschedules them with a delay based on the original arrival time of the Message (if the delay is numeric). For messages where the delay header was a `Date`, that is used when rescheduling. If a delayed Message remained in the `MessageStore` more than its 'delay', it will be sent immediately after startup.

The `<delayer>` can be enriched with mutually exclusive sub-elements `<transactional>` or `<advice-chain>`. The List of these AOP Advices is applied to the proxied internal `DelayHandler.ReleaseMessageHandler`, which has the responsibility to release the Message, after the delay, on a `Thread` of the scheduled task. It might be used, for example, when the downstream message flow throws an Exception and the `ReleaseMessageHandler`'s transaction will be rolled back. In this case the delayed Message will remain in the persistent `MessageStore`. You can use any custom `org.aopalliance.aop.Advice` implementation within the `<advice-chain>`. A sample configuration of the `<delayer>` may look like this:

```
<int:delayer id="delayer" input-channel="input" output-channel="output"
    expression="headers.delay"
    message-store="jdbcMessageStore">
    <int:advice-chain>
        <beans:ref bean="customAdviceBean"/>
        <tx:advice>
            <tx:attributes>
                <tx:method name="*" read-only="true"/>
            </tx:attributes>
        </tx:advice>
    </int:advice-chain>
</int:delayer>
```

The `DelayHandler` can be exported as a JMX `MBean` with managed operations `getDelayedMessageCount` and `reschedulePersistedMessages`, which allows the rescheduling of delayed persisted Messages at runtime, for example, if the `TaskScheduler` has previously been stopped. These operations can be invoked via a `Control Bus` command:

```
Message<String> delayerReschedulingMessage =

 MessageBuilder.withPayload("@'delayer.handler'.reschedulePersistedMessages()").build();
    controlBusChannel.send(delayerReschedulingMessage);
```

Note

For more information regarding the Message Store, JMX and the Control Bus, please read Chapter 8, *System Management*.

7.5 Scripting support

With Spring Integration 2.1 we've added support for the JSR223 Scripting for Java specification, introduced in Java version 6. This allows you to use scripts written in any supported language including Ruby/JRuby, Javascript and Groovy to provide the logic for various integration components similar to the way the Spring Expression Language (SpEL) is used in Spring Integration. For more information about JSR223 please refer to the documentation

Important

Note that this feature requires Java 6 or higher. Sun developed a JSR223 reference implementation which works with Java 5 but it is not officially supported and we have not tested it with Spring Integration.

In order to use a JVM scripting language, a JSR223 implementation for that language must be included in your class path. Java 6 natively supports Javascript. The Groovy and JRuby projects provide JSR233 support in their standard distribution. Other language implementations may be available or under development. Please refer to the appropriate project website for more information.

Important

Various JSR223 language implementations have been developed by third parties. A particular implementation's compatibility with Spring Integration depends on how well it conforms to the specification and/or the implementer's interpretation of the specification.

Tip

If you plan to use Groovy as your scripting language, we recommended you use Section 7.6, “Groovy support” as it offers additional features specific to Groovy. *However you will find this section relevant as well*.

Script configuration

Depending on the complexity of your integration requirements scripts may be provided inline as CDATA in XML configuration or as a reference to a Spring resource containing the script. To enable scripting support Spring Integration defines a `ScriptExecutingMessageProcessor` which will bind the Message Payload to a variable named `payload` and the Message Headers to a `headers` variable, both accessible within the script execution context. All that is left for you to do is write a script that uses these variables. Below are a couple of sample configurations:

Filter

```
<int:filter input-channel="referencedScriptInput">
   <int-script:script lang="ruby" location="some/path/to/ruby/script/RubyFilterTests.rb"/>
</int:filter>

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

Here, you see that the script can be included inline or can reference a resource location via the `location` attribute. Additionally the `lang` attribute corresponds to the language name (or JSR223 alias)

Other Spring Integration endpoint elements which support scripting include *router*, *service-activator*, *transformer*, and *splitter*. The scripting configuration in each case would be identical to the above (besides the endpoint element).

Another useful feature of Scripting support is the ability to update (reload) scripts without having to restart the Application Context. To accomplish this, specify the `refresh-check-delay` attribute on the *script* element:

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

In the above example, the script location will be checked for updates every 5 seconds. If the script is updated, any invocation that occurs later than 5 seconds since the update will result in execution of the new script.

```
<int-script: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, providing a simple mechanism for 'real-time' configuration. Any negative number value means the script will not be reloaded after initialization of the application context. This is the default behavior.

Important

Inline scripts can not be reloaded.

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

Script variable bindings

Variable bindings are required to enable the script to reference variables externally provided to the script's execution context. As we have seen, `payload` and `headers` are used as binding variables by default. You can bind additional variables to a script via `<variable>` sub-elements:

```
<script:script lang="js" location="foo/bar/MyScript.js">
 <script:variable name="foo" value="foo"/>
 <script:variable name="bar" value="bar"/>
 <script:variable name="date" ref="date"/>
</script:script>
```

As shown in the above example, you can bind a script variable either to a scalar value or a Spring bean reference. Note that `payload` and `headers` will still be included as binding variables.

If you need more control over how variables are generated, you can implement your own Java class using the `ScriptVariableGenerator` strategy:

```
public interface ScriptVariableGenerator {

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

}
```

This interface requires you to implement the method `generateScriptVariables(Message)`. The Message argument allows you to access any data available in the Message payload and headers and the return value is the Map of bound variables. This method will be called every time the script is executed for a Message. All you need to do is provide an implementation of `ScriptVariableGenerator` and reference it with the `script-variable-generator` attribute:

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

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

Important

You cannot provide both the `script-variable-generator` attribute and `<variable>` sub-element(s) as they are mutually exclusive. Also, custom variable bindings cannot be used with an inline script.

7.6 Groovy support

In Spring Integration 2.0 we 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

Groovy configuration

With Spring Integration 2.1, Groovy Support's configuration namespace is an extension of Spring Integration's Scripting Support and shares the core configuration and behavior described in detail in the Section 7.5, “Scripting support” section. Even though Groovy scripts are well supported by generic Scripting Support, Groovy Support provides the *Groovy* configuration namespace which is backed by the Spring Framework's `org.springframework.scripting.groovy.GroovyScriptFactory` and related components, offering extended capabilities for using Groovy. Below are a couple of sample configurations:

Filter

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

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

As the above examples show, the configuration looks identical to the general Scripting Support configuration. The only difference is the use of the Groovy namespace as indicated in the examples by the *int-groovy* namespace prefix. Also note that the `lang` attribute on the `<script>` tag is not valid in this namespace.

Groovy object customization

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

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

```
<int-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' `MessageProcessor` exposes all 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).

Important

Be careful about using managed beans with custom scopes (e.g. 'request') in the Control Bus' command scripts, especially inside an *async* message flow. If The Control Bus' `MessageProcessor` can't expose a bean from the application context, you may end up with some `BeansException` during *command script's* executing. For example, if a custom scope's context is not established, the attempt to get a bean within that scope will trigger a `BeanCreationException`.

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.

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

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

7.7 Adding Behavior to Endpoints

Prior to Spring Integration 2.2, you could add behavior to an entire Integration flow by adding an AOP Advice to a poller's <advice-chain /> element. However, let's say you want to retry, say, just a ReST Web Service call, and not any downstream endpoints.

For example, consider the following flow:

inbound-adapter->poller->http-gateway1->http-gateway2->jdbc-outbound-adapter

If you configure some retry-logic into an advice chain on the poller, and, the call to *http-gateway2* failed because of a network glitch, the retry would cause both *http-gateway1* and *http-gateway2* to be called a second time. Similarly, after a transient failure in the *jdbc-outbound-adapter*, both http-gateways would be called a second time before again calling the *jdbc-outbound-adapter*.

Spring Integration 2.2 adds the ability to add behavior to individual endpoints. This is achieved by the addition of the <request-handler-advice-chain /> element to many endpoints. For example:

```
<int-http:outbound-gateway id="withAdvice"
    url-expression="'http://localhost/test1'"
    request-channel="requests"
    reply-channel="nextChannel">
    <int:request-handler-advice-chain>
        <ref bean="myRetryAdvice" />
    </request-handler-advice-chain>
</int-http:outbound-gateway>
```

In this case, *myRetryAdvice* will only be applied locally to this gateway and will not apply to further actions taken downstream after the reply is sent to the *nextChannel*. The scope of the advice is limited to the endpoint itself.

Important

At this time, you cannot advise an entire <chain/> of endpoints. The schema does not allow a <request-handler-advice-chain/> as a child element of the chain itself.

However, a <request-handler-advice-chain/> can be added to individual reply-producing endpoints *within* a <chain/> element. An exception is that, in a chain that produces no reply, because the last element in the chain is an *outbound-channel-adapter*, that *last* element cannot be advised. If you need to advise such an element, it must be moved outside of the chain (with the *output-channel* of the chain being the *input-channel* of the adapter. The adapter can then be advised as normal. For chains that produce a reply, every child element can be advised.

Provided Advice Classes

In addition to providing the general mechanism to apply AOP Advice classes in this way, three standard Advices are provided:

- RequestHandlerRetryAdvice
- RequestHandlerCircuitBreakerAdvice
- ExpressionEvaluatingRequestHandlerAdvice

These are each described in detail in the following sections.

Retry Advice

The retry advice (`o.s.i.handler.advice.RequestHandlerRetryAdvice`) leverages the rich retry mechanisms provided by the spring-retry project. The core component of *spring-retry* is the `RetryTemplate`, which allows configuration of sophisticated retry scenarios, including `RetryPolicy` and `BackoffPolicy` strategies, with a number of implementations, as well as a `RecoveryCallback` strategy to determine the action to take when retries are exhausted.

Stateless Retry

Stateless retry is the case where the retry activity is handled entirely within the advice, where the thread pauses (if so configured) and retries the action.

Stateful Retry

Stateful retry is the case where the retry state is managed within the advice, but where an exception is thrown and the caller resubmits the request. An example for stateful retry is when we want the message originator (e.g. JMS) to be responsible for resubmitting, rather than performing it on the current thread. Stateful retry needs some mechanism to detect a retried submission.

Further Information

For more information on *spring-retry*, refer to the project's javadocs, as well as the reference documentation for Spring Batch, where *spring-retry* originated.

Caution

The default back off behavior is no back off - retries are attempted immediately. Using a back off policy that causes threads to pause between attempts may cause performance issues, including excessive memory use and thread starvation. In high volume environments, back off policies should be used with caution.

Configuring the Retry Advice

The following examples use a simple <service-activator />> that always throws an exception:

```
public class FailingService {

    public void service(String message) {
        throw new RuntimeException("foo");
    }
}
```

Simple Stateless Retry

This example uses the default *RetryTemplate* which has a *SimpleRetryPolicy* which tries 3 times. There is no *BackoffPolicy* so the 3 attempts are made back-to-back-to-back with no delay between attempts. There is no *RecoveryCallback* so, the result is to throw the exception to the caller after the final failed retry occurs. In a *Spring Integration* environment, this final exception might be handled using an *error-channel* on the inbound endpoint.

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice"/>
    </request-handler-advice-chain>
</int:service-activator>

DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
DEBUG [task-scheduler-2]Retry: count=0
DEBUG [task-scheduler-2]Checking for rethrow: count=1
DEBUG [task-scheduler-2]Retry: count=1
DEBUG [task-scheduler-2]Checking for rethrow: count=2
DEBUG [task-scheduler-2]Retry: count=2
DEBUG [task-scheduler-2]Checking for rethrow: count=3
DEBUG [task-scheduler-2]Retry failed last attempt: count=3
```

Simple Stateless Retry with Recovery

This example adds a `RecoveryCallback` to the above example; it uses a `ErrorMessageSendingRecoverer` to send an *ErrorMessage* to a channel.

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice">
            <property name="recoveryCallback">
                <bean class="o.s.i.handler.advice.ErrorMessageSendingRecoverer">
                    <constructor-arg ref="myErrorChannel" />
                </bean>
            </property>
        </bean>
    </request-handler-advice-chain>
</int:int:service-activator>

DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
DEBUG [task-scheduler-2]Retry: count=0
DEBUG [task-scheduler-2]Checking for rethrow: count=1
DEBUG [task-scheduler-2]Retry: count=1
DEBUG [task-scheduler-2]Checking for rethrow: count=2
DEBUG [task-scheduler-2]Retry: count=2
DEBUG [task-scheduler-2]Checking for rethrow: count=3
DEBUG [task-scheduler-2]Retry failed last attempt: count=3
DEBUG [task-scheduler-2]Sending ErrorMessage :failedMessage:[Payload=...]
```

Stateless Retry with Customized Policies, and Recovery

For more sophistication, we can provide the advice with a customized *RetryTemplate*. This example continues to use the `SimpleRetryPolicy` but it increases the attempts to 4. It also adds an `ExponentialBackoffPolicy` where the first retry waits 1 second, the second waits 5 seconds and the third waits 25 (for 4 attempts in all).

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice">
            <property name="recoveryCallback">
                <bean class="o.s.i.handler.advice.ErrorMessageSendingRecoverer">
                    <constructor-arg ref="myErrorChannel" />
                </bean>
            </property>
            <property name="retryTemplate" ref="retryTemplate" />
        </bean>
    </request-handler-advice-chain>
</int:service-activator>

<bean id="retryTemplate" class="org.springframework.retry.support.RetryTemplate">
 <property name="retryPolicy">
  <bean class="org.springframework.retry.policy.SimpleRetryPolicy">
   <property name="maxAttempts" value="4" />
  </bean>
 </property>
 <property name="backOffPolicy">
  <bean class="org.springframework.retry.backoff.ExponentialBackOffPolicy">
   <property name="initialInterval" value="1000" />
   <property name="multiplier" value="5" />
  </bean>
 </property>
</bean>

27.058 DEBUG [task-scheduler-1]preSend on channel 'input', message: [Payload=...]
27.071 DEBUG [task-scheduler-1]Retry: count=0
27.080 DEBUG [task-scheduler-1]Sleeping for 1000
28.081 DEBUG [task-scheduler-1]Checking for rethrow: count=1
28.081 DEBUG [task-scheduler-1]Retry: count=1
28.081 DEBUG [task-scheduler-1]Sleeping for 5000
33.082 DEBUG [task-scheduler-1]Checking for rethrow: count=2
33.082 DEBUG [task-scheduler-1]Retry: count=2
33.083 DEBUG [task-scheduler-1]Sleeping for 25000
58.083 DEBUG [task-scheduler-1]Checking for rethrow: count=3
58.083 DEBUG [task-scheduler-1]Retry: count=3
58.084 DEBUG [task-scheduler-1]Checking for rethrow: count=4
58.084 DEBUG [task-scheduler-1]Retry failed last attempt: count=4
58.086 DEBUG [task-scheduler-1]Sending ErrorMessage :failedMessage:[Payload=...]
```

Simple Stateful Retry with Recovery

To make retry stateful, we need to provide the Advice with a RetryStateGenerator implementation. This class is used to identify a message as being a resubmission so that the *RetryTemplate* can determine the current state of retry for this message. The framework provides a `SpelExpressionRetryStateGenerator` which determines the message identifier using a SpEL expression. This is shown below; this example again uses the default policies (3 attempts with no back off); of course, as with stateless retry, these policies can be customized.

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerRetryAdvice">
            <property name="retryStateGenerator">
                <bean class="o.s.i.handler.advice.SpelExpressionRetryStateGenerator">
                    <constructor-arg value="headers['jms_messageId']" />
                </bean>
            </property>
            <property name="recoveryCallback">
                <bean class="o.s.i.handler.advice.ErrorMessageSendingRecoverer">
                    <constructor-arg ref="myErrorChannel" />
                </bean>
            </property>
        </bean>
    </int:request-handler-advice-chain>
</int:service-activator>

24.351 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
24.368 DEBUG [Container#0-1]Retry: count=0
24.387 DEBUG [Container#0-1]Checking for rethrow: count=1
24.387 DEBUG [Container#0-1]Rethrow in retry for policy: count=1
24.387 WARN  [Container#0-1]failure occurred in gateway sendAndReceive
org.springframework.integration.MessagingException: Failed to invoke handler
...
Caused by: java.lang.RuntimeException: foo
...
24.391 DEBUG [Container#0-1]Initiating transaction rollback on application exception
...
25.412 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
25.412 DEBUG [Container#0-1]Retry: count=1
25.413 DEBUG [Container#0-1]Checking for rethrow: count=2
25.413 DEBUG [Container#0-1]Rethrow in retry for policy: count=2
25.413 WARN  [Container#0-1]failure occurred in gateway sendAndReceive
org.springframework.integration.MessagingException: Failed to invoke handler
...
Caused by: java.lang.RuntimeException: foo
...
25.414 DEBUG [Container#0-1]Initiating transaction rollback on application exception
...
26.418 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
26.418 DEBUG [Container#0-1]Retry: count=2
26.419 DEBUG [Container#0-1]Checking for rethrow: count=3
26.419 DEBUG [Container#0-1]Rethrow in retry for policy: count=3
26.419 WARN  [Container#0-1]failure occurred in gateway sendAndReceive
org.springframework.integration.MessagingException: Failed to invoke handler
...
Caused by: java.lang.RuntimeException: foo
...
26.420 DEBUG [Container#0-1]Initiating transaction rollback on application exception
...
27.425 DEBUG [Container#0-1]preSend on channel 'input', message: [Payload=...]
27.426 DEBUG [Container#0-1]Retry failed last attempt: count=3
27.426 DEBUG [Container#0-1]Sending ErrorMessage :failedMessage:[Payload=...]
```

Comparing with the stateless examples, you can see that with stateful retry, the exception is thrown to the caller on each failure.

Circuit Breaker Advice

The general idea of the Circuit Breaker Pattern is that, if a service is not currently available, then don't waste time (and resources) trying to use it. The

`o.s.i.handler.advice.RequestHandlerCircuitBreakerAdvice` implements this pattern. When the circuit breaker is in the *closed* state, the endpoint will attempt to invoke the service. The circuit breaker goes to the *open* state if a certain number of consecutive attempts fail; when it is in the *open* state, new requests will "fail fast" and no attempt will be made to invoke the service until some time has expired.

When that time has expired, the circuit breaker is set to the *half-open* state. When in this state, if even a single attempt fails, the breaker will immediately go to the *open* state; if the attempt succeeds, the breaker will go to the *closed* state, in which case, it won't go to the *open* state again until the configured number of consecutive failures again occur. Any successful attempt resets the state to zero failures for the purpose of determining when the breaker might go to the *open* state again.

Typically, this Advice might be used for external services, where it might take some time to fail (such as a timeout attempting to make a network connection).

The `RequestHandlerCircuitBreakerAdvice` has two properties: `threshold` and `halfOpenAfter`. The *threshold* property represents the number of consecutive failures that need to occur before the breaker goes *open*. It defaults to 5. The *halfOpenAfter* property represents the time after the last failure that the breaker will wait before attempting another request. Default is 1000 milliseconds.

Example:

```
<int:service-activator input-channel="input" ref="failer" method="service">
    <int:request-handler-advice-chain>
        <bean class="o.s.i.handler.advice.RequestHandlerCircuitBreakerAdvice">
            <property name="threshold" value="2" />
            <property name="halfOpenAfter" value="12000" />
        </bean>
    </int:request-handler-advice-chain>
</int:service-activator>

05.617 DEBUG [task-scheduler-1]preSend on channel 'input', message: [Payload=...]
05.638 ERROR [task-scheduler-1]org.springframework.integration.MessageHandlingException:
 java.lang.RuntimeException: foo
...
10.598 DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
10.600 ERROR [task-scheduler-2]org.springframework.integration.MessageHandlingException:
 java.lang.RuntimeException: foo
...
15.598 DEBUG [task-scheduler-3]preSend on channel 'input', message: [Payload=...]
15.599 ERROR [task-scheduler-3]org.springframework.integration.MessagingException: Circuit
 Breaker is Open for ServiceActivator
...
20.598 DEBUG [task-scheduler-2]preSend on channel 'input', message: [Payload=...]
20.598 ERROR [task-scheduler-2]org.springframework.integration.MessagingException: Circuit
 Breaker is Open for ServiceActivator
...
25.598 DEBUG [task-scheduler-5]preSend on channel 'input', message: [Payload=...]
25.601 ERROR [task-scheduler-5]org.springframework.integration.MessageHandlingException:
 java.lang.RuntimeException: foo
...
30.598 DEBUG [task-scheduler-1]preSend on channel 'input', message: [Payload=foo...]
30.599 ERROR [task-scheduler-1]org.springframework.integration.MessagingException: Circuit
 Breaker is Open for ServiceActivator
```

In the above example, the threshold is set to 2 and halfOpenAfter is set to 12 seconds; a new request arrives every 5 seconds. You can see that the first two attempts invoked the service; the third and fourth failed with an exception indicating the circuit breaker is open. The fifth request was attempted because

the request was 15 seconds after the last failure; the sixth attempt fails immediately because the breaker immediately went to *open*.

Expression Evaluating Advice

The final supplied advice class is the `o.s.i.handler.advice.ExpressionEvaluatingRequestHandlerAdvice`. This advice is more general than the other two advices. It provides a mechanism to evaluate an expression on the original inbound message sent to the endpoint. Separate expressions are available to be evaluated, either after success, or failure. Optionally, a message containing the evaluation result, together with the input message, can be sent to a message channel.

A typical use case for this advice might be with an <ftp:outbound-channel-adapter />, perhaps to move the file to one directory if the transfer was successful, or to another directory if it fails:

The Advice has properties to set an expression when successful, an expression for failures, and corresponding channels for each. For the successful case, the message sent to the *successChannel* is an `AdviceMessage`, with the payload being the result of the expression evaluation, and an additional property `inputMessage` which contains the original message sent to the handler. A message sent to the *failureChannel* (when the handler throws an excecption) is an ErrorMessage with a payload of `MessageHandlingExpressionEvaluatingAdviceException`. Like all `MessagingExceptions`, this payload has `failedMessage` and `cause` properties, as well as an additional property `evaluationResult`, containing the result of the expression evaluation.

Custom Advice Classes

In addition to the provided Advice classes above, you can implement your own Advice classes. While you can provide any implementation of `org.aopalliance.aop.Advice`, it is generally recommended that you subclass `o.s.i.handler.advice.AbstractRequestHandlerAdvice`. This has the benefit of avoiding writing low-level *Aspect Oriented Programming* code as well as providing a starting point that is specifically tailored for use in this environment.

Subclasses need to implement the doInvoke() method:

```
/**
 * Subclasses implement this method to apply behavior to the {@link MessageHandler}
 callback.execute()
 * invokes the handler method and returns its result, or null).
 * @param callback Subclasses invoke the execute() method on this interface to invoke the
 handler method.
 * @param target The target handler.
 * @param message The message that will be sent to the handler.
 * @return the result after invoking the {@link MessageHandler}.
 * @throws Exception
 */
protected abstract Object doInvoke(ExecutionCallback callback, Object target, Message<?>
 message) throws Exception;
```

The *callback* parameter is simply a convenience to avoid subclasses dealing with AOP directly; invoking the `callback.execute()` method invokes the message handler.

The *target* parameter is provided for those subclasses that need to maintain state for a specific handler, perhaps by maintaining that state in a `Map`, keyed by the target. This allows the same advice to be applied to multiple handlers. The `RequestHandlerCircuitBreakerAdvice` uses this to keep circuit breaker state for each handler.

The *message* parameter is the message that will be sent to the handler. While the advice cannot modify the message before invoking the handler, it can modify the payload (if it has mutable properties). Typically, an advice would use the message for logging and/or to send a copy of the message somewhere before or after invoking the handler.

The return value would normally be the value returned by `callback.execute()`; but the advice does have the ability to modify the return value. Note that only `AbstractReplyProducingMessageHandlers` return a value.

```
public class MyAdvice extends AbstractRequestHandlerAdvice {

    @Override
    protected Object doInvoke(ExecutionCallback callback, Object target, Message<?>
 message) throws Exception {
        // add code before the invocation
        Object result = callback.execute();
        // add code after the invocation
        return result;
    }
}
```

Note

In addition to the `execute()` method, the `ExecutionCallback` provides an additional method `cloneAndExecute()`. This method must be used in cases where the invocation might be called multiple times within a single execution of `doInvoke()`, such as in the `RequestHandlerRetryAdvice`. This is required because the Spring AOP `org.springframework.aop.framework.ReflectiveMethodInvocation` object maintains state of which advice in a chain was last invoked; this state must be reset for each call.

For more information, see the ReflectiveMethodInvocation JavaDocs.

Other Advice Chain Elements

While the abstract class mentioned above is provided as a convenience, you can add any `Advice` to the chain, including a transaction advice.

Advising Filters

There is an additional consideration when advising `Filters`. By default, any discard actions (when the filter returns false) are performed *within* the scope of the advice chain. This could include all the flow downstream of the *discard channel*. So, for example if an element downstream of the *discard-channel* throws an exception, and there is a retry advice, the process will be retried. This is also the case if *throwExceptionOnRejection* is set to true (the exception is thrown within the scope of the advice).

Setting *discard-within-advice* to "false" modifies this behavior and the discard (or exception) occurs after the advice chain is called.

Advising Endpoints Using Annotations

When configuring certain endpoints using annotations (`@Filter`, `@ServiceActivator`, `@Splitter`, and `@Transformer`), you can supply a bean name for the advice chain in the `adviceChain` attribute. In addition, the `@Filter` annotation also has the `discardWithinAdvice` attribute, which can be used

to configure the discard behavior as discussed in the section called “Advising Filters”. An example with the discard being performed after the advice is shown below.

```
@MessageEndpoint
public class MyAdvisedFilter {

 @Filter(inputChannel="input", outputChannel="output",
   adviceChain="adviceChain", discardWithinAdvice="false")
 public boolean filter(String s) {
  return s.contains("good");
 }
}
```

Ordering Advices within an Advice Chain

Advice classes are "around" advices and are applied in a nested fashion. The first advice is the outermost, the last advice the innermost (closest to the handler being advised). It is important to put the advice classes in the correct order to achieve the functionality you desire.

For example, let's say you want to add a retry advice and a transaction advice. You may want to place the retry advice advice first, followed by the transaction advice. Then, each retry will be performed in a new transaction. On the other hand, if you want all the attempts, and any recovery operations (in the retry `RecoveryCallback`), to be scoped within the transaction, you would put the transaction advice first.

7.8 Logging Channel Adapter

The `<logging-channel-adapter/>` is often used in conjunction with a Wire Tap, as discussed in the section called “Wire Tap”. However, it can also be used as the ultimate consumer of any flow. For example, consider a flow that ends with a `<service-activator/>` that returns a result, but you wish to discard that result. To do that, you could send the result to `NullChannel`. Alternatively, you can route it to an `INFO` level `<logging-channel-adapter/>`; that way, you can see the discarded message when logging at `INFO` level, but not see it when logging at, say, `WARN` level. With a `NullChannel`, you would only see the discarded message when logging at `DEBUG` level.

```
<int:logging-channel-adapter
 channel=""❶
 level="INFO"❷
 expression=""❸
 log-full-message="false"❹
 logger-name=""❺/>
```

❶ The channel connecting the logging adapter to an upstream component.

❷ The logging level at which messages sent to this adapter will be logged. Default: `INFO`.

❸ A SpEL expression representing exactly what part(s) of the message will be logged. Default: `payload` - just the payload will be logged. This attribute cannot be specified if `log-full-message` is specified.

❹ When `true`, the entire message will be logged (including headers). Default: `false` - just the payload will be logged. This attribute cannot be specified if `expression` is specified.

❺ Specifies the *name* of the logger (known as `category` in `log4j`) used for log messages created by this adapter. This enables setting the log name (in the logging subsystem) for individual adapters. By default, all adapters will log under the name `org.springframework.integration.handler.LoggingHandler`.

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:

```
<int-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:

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

The *Notification-listening Channel Adapter* is event-driven and registered with the `MBeanServer` directly. It does not require any poller configuration.

Note

For this component only, the *object-name* attribute can contain an ObjectName pattern (e.g. "org.foo:type=Bar,name=*") and the adapter will receive notifications from all MBeans with ObjectNames that match the pattern. In addition, the *object-name* attribute can contain a SpEL reference to a <util:list/> of ObjectName patterns:

```
<jmx:notification-listening-channel-adapter id="manyNotificationsAdapter"
    channel="manyNotificationsChannel"
    object-name="#{patterns}"/>

<util:list id="patterns">
    <value>org.foo:type=Foo,name=*</value>
    <value>org.foo:type=Bar,name=*</value>
</util:list>
```

The names of the located MBean(s) will be logged when DEBUG level logging is enabled.

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/>

<int-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/>

<int-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 Channel 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.

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

Tree Polling Channel Adapter

The *Tree Polling Channel Adapter* queries the JMX MBean tree and sends a message with a payload that is the graph of objects that matches the query. By default the MBeans are mapped to primitives and simple Objects like Map, List and arrays - permitting simple transformation, for example, to JSON. 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. A basic configuration would be:

```
<int-jmx:tree-polling-channel-adapter id="adapter"
    channel="channel"
    query-name="example.domain:type=*">
        <int:poller max-messages-per-poll="1" fixed-rate="5000"/>
</int-jmx:tree-polling-channel-adapter>
```

This will include all attributes on the MBeans selected. You can filter the attributes by providing an `MBeanObjectConverter` that has an appropriate filter configured. The converter can be provided as a reference to a bean definition using the `converter` attribute, or as an inner <bean/> definition. A `DefaultMBeanObjectConverter` is provided which can take a `MBeanAttributeFilter` in its constructor argument.

Two standard filters are provided; the `NamedFieldsMBeanAttributeFilter` allows you to specify a list of attributes to include and the `NotNamedFieldsMBeanAttributeFilter` allows you to specify a list of attributes to exclude. You can also implement your own filter

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:

```
<int-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 the *operation-invoking-channel-adapter* Spring Integration also provides a *operation-invoking-outbound-gateway*, which could be used when dealing with non-void operations and a return value is required. Such return value will be sent as message payload to the *reply-channel* specified by this Gateway.

```
<int-jmx:operation-invoking-outbound-gateway request-channel="requestChannel"
   reply-channel="replyChannel"
   object-name="o.s.i.jmx.config:type=TestBean,name=testBeanGateway"
   operation-name="testWithReturn"/>
```

If the *reply-channel* attribute is not provided, the reply message will be sent to the channel that is identified by the `MessageHeaders.REPLY_CHANNEL` header. That header is typically auto-created by the entry point into a message flow, such as any *Gateway* component. However, if the message flow

was started by manually creating a Spring Integration Message and sending it directly to a *Channel*, then you must specify the message header explicitly or use the provided *reply-channel* attribute.

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*.

```
<int-jmx:mbean-export 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. The generated JMX object names for each component type are listed in the table below:

Table 8.1.

Component Type	ObjectName
MessageChannel	o.s.i:type=MessageChannel,name=<channelName>
MessageSource	o.s.i:type=MessageSource,name=<channelName>,bean=<source>
MessageHandler	o.s.i:type=MessageSource,name=<channelName>,bean=<source>

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, as well as the count of messages that are currently buffered by this `QueueChannel`. 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 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).

Orderly Shutdown Managed Operation

The MBean exporter provides a JMX operation to shut down the application in an orderly manner, intended for use before terminating the JVM.

```
public void stopActiveComponents(boolean force, long howLong)
```

Its use and operation are described in Section 8.5, “Orderly Shutdown”.

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 Message Store

Enterprise Integration Patterns (EIP) identifies several patterns that have the capability to buffer messages. For example, an *Aggregator* buffers messages until they can be released and a *QueueChannel* buffers messages until consumers explicitly receive those messages from that channel. Because of the failures that can occur at any point within your message flow, EIP components that buffer messages also introduce a point where messages could be lost.

To mitigate the risk of losing Messages, EIP defines the Message Store pattern which allows EIP components to store *Messages* typically in some type of persistent store (e.g. RDBMS).

Spring Integration provides support for the *Message Store* pattern by a) defining a `org.springframework.integration.store.MessageStore` strategy interface, b) providing several implementations of this interface, and c) exposing a `message-store` attribute on all components that have the capability to buffer messages so that you can inject any instance that implements the `MessageStore` interface.

Details on how to configure a specific *Message Store* implementation and/or how to inject a `MessageStore` implementation into a specific buffering component are described throughout the manual (see the specific component, such as *QueueChannel*, *Aggregator*, *Resequencer* etc.), but here are a couple of samples to give you an idea:

QueueChannel

```
<int:channel id="myQueueChannel">
    <int:queue message-store="refToMessageStore"/>
<int:channel>
```

Aggregator

```
<int:aggregator … message-store="refToMessageStore"/>
```

By default *Messages* are stored in-memory using `org.springframework.integration.store.SimpleMessageStore`, an implementation of `MessageStore`. That might be fine for development or simple low-volume environments where the potential loss of non-persistent messages is not a concern. However, the typical production application will need a more robust option, not only to mitigate the risk of message loss but also to avoid potential out-of-memory errors. Therefore, we also provide MessageStore implementations for a variety of data-stores. Below is a complete list of supported implementations:

- Section 17.4, “JDBC Message Store” - uses RDBMS to store Messages
- Section 22.4, “Redis Message Store” - uses Redis key/value datastore to store Messages
- Section 21.3, “MongoDB Message Store” - uses MongoDB document store to store Messages
- Section 14.5, “Gemfire Message Store” - uses Gemfire distributed cache to store Messages

Important

However be aware of some limitations while using persistent implementations of the `MessageStore`.

The Message data (payload and headers) is *serialized* and *deserialized* using different serialization strategies depending on the implementation of the `MessageStore`. For example, when using `JdbcMessageStore`, only `Serializable` data is persisted by default. In this case non-Serializable headers are removed before serialization occurs. Also be aware of the protocol specific headers that are injected by transport adapters (e.g., FTP, HTTP, JMS etc.). For example, `<http:inbound-channel-adapter/>` maps HTTP-headers into Message Headers and one of them is an `ArrayList` of non-Serializable `org.springframework.http.MediaType` instances. However you are able to inject your own implementation of the `Serializer` and/or `Deserializer` strategy interfaces into some `MessageStore` implementations (such as JdbcMessageStore) to change the behaviour of serialization and deserialization.

Special attention must be paid to the headers that represent certain types of data. For example, if one of the headers contains an instance of some *Spring Bean*, upon deserialization you may end up with a different instance of that bean, which directly affects some of the implicit headers created by the framework (e.g., REPLY_CHANNEL or ERROR_CHANNEL). Currently they are not serializable, but even if they were the deserialized channel would not represent the expected instance. As a workaround we suggest to remove bean-ref headers via a `<header-filter/>` before sending a message to an endpoint backed by a persistent `MessageStore`. Also, we recommend using channel names instead of channel instances when setting those types of headers, thus allowing it to be resolved in real time by the `ChannelResolver`.

Also avoid configuration of a message-flow like this: *gateway -> queue-channel (backed by a persistent Message Store) -> service-activator* That gateway creates a *Temporary Reply Channel* in the background, and it will be lost by the time the service-activator's poller reads from the queue, because it has been deserialized by another thread on the sending side.

Nevertheless we are constantly thinking about potential improvements to the framework, such as a way to provide some robust default serialization strategy for messages in these cases.

8.4 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.

```
<int: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.

8.5 Orderly Shutdown

As described in the section called “MBean Exporter”, the MBean exporter provides a JMX operation *stopActiveComponents*, which is used to stop the application in an orderly manner. The operation has two parameters, a boolean and a long. The boolean indicates whether attempts will be made to stop (interrupt) active threads; in most cases this will be set to *false* for orderly shutdown. The long parameter indicates how long (in milliseconds) the operation will wait to allow in-flight messages to complete. The operation works as follows:

The first step calls `beforeShutdown()` on all beans that implement `OrderlyShutdownCapable`. This allows such components to prepare for shutdown. Examples of components that implement this interface, and what they do with this call include: JMS and AMQP message-driven adapters stop their listener containers; TCP server connection factories stop accepting new connections (while keeping existing connections open); TCP inbound endpoints drop (log) any new messages received; http inbound endpoints return *503 - Service Unavailable* for any new requests.

The second step stops any active channels, such as JMS- or AMQP-backed channels.

The third step stops all `TaskSchedulers`, preventing any new scheduled operations (polling etc).

The fourth step stops all `TaskExecutors`, preventing any new tasks from running.

Note

If the shutdown is running from a Spring-managed `TaskExecutor`, shutting down that executor would cause all the timeout time to be consumed by this step, because the thread won't terminate). For this reason, either use a dedicated executor (via the shutdownExecutor property on the MBean exporter), or do not use a Spring-managed executor to invoke this operation.

The fifth step stops all `MessageSources`.

The sixth step waits for any remaining time left, as defined by the value of the long parameter passed in to the operation. This is intended to allow any in-flight messages to complete their journeys. It is therefore important to select an appropriate timeout when invoking this operation.

The seventh step calls `afterShutdown()` on all OrderlyShutdownCapable components. This allows such components to perform final shutdown tasks (closing all open sockets, for example).

Note

If no time is left when we get to step 6, it probably means some thread is hung; in which case, the operation attempts a forced shutdown on all schedulers and executors before exiting.

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. AMQP Support

9.1 Introduction

Spring Integration provides Channel Adapters for receiving and sending messages using the Advanced Message Queuing Protocol (AMQP). The following adapters are available:

- Inbound Channel Adapter
- Outbound Channel Adapter
- Inbound Gateway
- Outbound Gateway

Spring Integration also provides a point-to-point Message Channel as well as a publish/subscribe Message Channel backed by AMQP Exchanges and Queues.

In order to provide AMQP support, Spring Integration relies on Spring AMQP (http://www.springsource.org/spring-amqp) which "applies core Spring concepts to the development of AMQP-based messaging solutions". Spring AMQP provides similar semantics as Spring JMS (http://static.springsource.org/spring/docs/current/spring-framework-reference/html/jms.html).

Whereas the provided AMQP Channel Adapters are intended for unidirectional Messaging (send or receive) only, Spring Integration also provides inbound and outbound AMQP Gateways for request/reply operations.

Tip

Please familiarize yourself with the reference documentation of the Spring AMQP project as well. It provides much more in-depth information regarding Spring's integration with AMQP in general and RabbitMQ in particular.

You can find the documentation at: http://static.springsource.org/spring-amqp/reference/html/

9.2 Inbound Channel Adapter

A configuration sample for an AMQP Inbound Channel Adapter is shown below with all available parameters.

```
<int-amqp:inbound-channel-adapter id="inboundAmqp"❶
                                  channel="inboundChannel"❷
                                  queue-names="si.test.queue"❸
                                  acknowledge-mode="AUTO"❹
                                  advice-chain=""❺
                                  channel-transacted=""❻
                                  concurrent-consumers=""❼
                                  connection-factory=""❽
                                  error-channel=""❾
                                  expose-listener-channel=""❿
                                  header-mapper=""11
                                  mapped-request-headers=""12
                                  mapped-reply-headers=""13
                                  listener-container=""14
                                  message-converter=""15
                                  message-properties-converter=""16
                                  phase=""17
                                  prefetch-count=""18
                                  receive-timeout=""19
                                  recovery-interval=""20
                                  shutdown-timeout=""21
                                  task-executor=""22
                                  transaction-attribute=""23
                                  transaction-manager=""24
                                  tx-size=""25/>
```

❶ Unique ID for this adapter. *Optional*.

❷ Message Channel to which converted Messages should be sent. *Required*.

❸ Names of the AMQP Queues from which Messages should be consumed (comma-separated list). *Required*.

❹ Acknowledge Mode for the MessageListenerContainer. *Optional (Defaults to AUTO)*.

❺ Extra AOP Advice(s) to handle cross cutting behavior associated with this Inbound Channel Adapter. *Optional*.

❻ Flag to indicate that channels created by this component will be transactional. Ff true, tells the framework to use a transactional channel and to end all operations (send or receive) with a commit or rollback depending on the outcome, with an exception signalling a rollback. *Optional (Defaults to false)*.

❼ Specify the number of concurrent consumers to create. Default is 1. Raising the number of concurrent consumers is recommended in order to scale the consumption of messages coming in from a queue. However, note that any ordering guarantees are lost once multiple consumers are registered. In general, stick with 1 consumer for low-volume queues. *Optional*.

❽ Bean reference to the RabbitMQ ConnectionFactory. *Optional (Defaults to 'connectionFactory')*.

❾ Message Channel to which error Messages should be sent. *Optional*.

❿ Shall the listener channel (com.rabbitmq.client.Channel) be exposed to a registered ChannelAwareMessageListener. *Optional (Defaults to true)*.

11 `AmqpHeaderMapper` to use when receiving AMQP Messages. *Optional*. By default only standard AMQP properties (e.g. contentType) will be copied to and from Spring Integration MessageHeaders. Any user-defined headers within the AMQP MessageProperties will NOT be copied to or from an AMQP Message unless explicitly identified via 'requestHeaderNames' and/or 'replyHeaderNames' properties of this `DefaultAmqpHeaderMapper`. If you need to copy all user-defined headers simply use wild-card character '*'.

12 Comma-separated list of names of AMQP Headers to be mapped from the AMQP request into the MessageHeaders. This can only be provided if the 'header-mapper' reference is not being set directly. The values in this list can also be simple patterns to be matched against the header names (e.g. "*" or "foo*, bar" or "*foo").

13 Comma-separated list of names of MessageHeaders to be mapped into the AMQP Message Properties of the AMQP reply message. All standard Headers (e.g., contentType) will be mapped to AMQP Message Properties while user-defined headers will be mapped to 'headers' property which itself is a Map. This can only be provided if the 'header-mapper' reference is not being set directly. The values in this list can also be simple patterns to be matched against the header names (e.g. "*" or "foo*, bar" or "*foo").

14 Reference to the `SimpleMessageListenerContainer` to use for receiving AMQP Messages. If this attribute is provided, then no other attribute related to the listener container configuration should be provided. In other words, by setting this reference, you must take full responsibility of the listener container configuration. The only exception is the MessageListener itself. Since that is actually the core responsibility of this Channel Adapter implementation, the referenced listener container must NOT already have its own MessageListener configured. *Optional*.

15 The MessageConverter to use when receiving AMQP Messages. *Optional*.

16 The MessagePropertiesConverter to use when receiving AMQP Messages. *Optional*.

17 Specify the phase in which the underlying SimpleMessageListenerContainer should be started and stopped. The startup order proceeds from lowest to highest, and the shutdown order is the reverse of that. By default this value is Integer.MAX_VALUE meaning that this container starts as late as possible and stops as soon as possible. *Optional*.

18 Tells the AMQP broker how many messages to send to each consumer in a single request. Often this can be set quite high to improve throughput. It should be greater than or equal to the transaction size (see attribute "tx-size"). *Optional (Defaults to 1)*.

19 Receive timeout in milliseconds. *Optional (Defaults to 1000)*.

20 Specifies the interval between recovery attempts of the underlying SimpleMessageListenerContainer (in milliseconds). *Optional (Defaults to 5000)*.

21 The time to wait for workers in milliseconds after the underlying SimpleMessageListenerContainer is stopped, and before the AMQP connection is forced closed. If any workers are active when the shutdown signal comes they will be allowed to finish processing as long as they can finish within this timeout. Otherwise the connection is closed and messages remain unacked (if the channel is transactional). Defaults to 5000 milliseconds. *Optional (Defaults to 5000)*.

22 By default, the underlying SimpleMessageListenerContainer uses a SimpleAsyncTaskExecutor implementation, that fires up a new Thread for each task, executing it asynchronously. By default, the number of concurrent threads is unlimited. NOTE: This implementation does not reuse threads. Consider a thread-pooling TaskExecutor implementation as an alternative. *Optional (Defaults to SimpleAsyncTaskExecutor)*.

23 By default the underlying SimpleMessageListenerContainer creates a new instance of the DefaultTransactionAttribute (takes the EJB approach to rolling back on runtime, but not checked exceptions. *Optional (Defaults to DefaultTransactionAttribute)*.

24 Sets a Bean reference to an external PlatformTransactionManager on the underlying SimpleMessageListenerContainer. The transaction manager works in conjunction with the "channel-transacted" attribute. If there is already a transaction in progress when the framework is sending or receiving a message, and the channelTransacted flag is true, then the commit or rollback of the messaging transaction will be deferred until the end of the current transaction. If the channelTransacted flag is false, then no transaction semantics apply to the messaging operation (it is auto-acked). For further information see chapter 1.9 of the Spring AMQP reference guide: http://static.springsource.org/spring-amqp/docs/1.0.x/reference/html/#d0e525 *Optional*.

25 Tells the SimpleMessageListenerContainer how many messages to process in a single transaction (if the channel is transactional). For best results it should be less than or equal to the set "prefetch-count". *Optional (Defaults to 1)*.

Important

Even though the Spring Integration JMS and AMQP support is very similar, important differences exist. The JMS Inbound Channel Adapter is using a JmsDestinationPollingSource under the covers and expects a configured Poller. The AMQP Inbound Channel Adapter on the other side uses a SimpleMessageListenerContainer and is message driven. In that regard it is more similar to the JMS Message Driven Channel Adapter.

9.3 Outbound Channel Adapter

A configuration sample for an AMQP Outbound Channel Adapter is shown below with all available parameters.

```
<int-amqp:outbound-channel-adapter id="outboundAmqp"❶
                                   channel="outboundChannel"❷
                                   amqp-template="myAmqpTemplate"❸
                                   exchange-name=""❹
                                   order="1"❺
                                   routing-key=""❻
                                   routing-key-expression=""❼
                                   confirm-correlation-expression=""❽
                                   confirm-ack-channel=""❾
                                   confirm-nack-channel=""❿
                                   return-channel=""11/>
```

❶ Unique ID for this adapter. *Optional*.

❷ Message Channel to which Messages should be sent in order to have them converted and published to an AMQP Exchange. *Required*.

❸ Bean Reference to the configured AMQP Template *Optional (Defaults to "amqpTemplate")*.

❹ The name of the AMQP Exchange to which Messages should be sent. If not provided, Messages will be sent to the default, no-name Exchange. *Optional*.

❺ The order for this consumer when multiple consumers are registered thereby enabling load- balancing and/or failover. *Optional (Defaults to Ordered.LOWEST_PRECEDENCE [=Integer.MAX_VALUE])*.

❻ The fixed routing-key to use when sending Messages. By default, this will be an empty String. *Optional*.

❼ The routing-key to use when sending Messages evaluated as an expression on the message (e.g. 'payload.key'). By default, this will be an empty String. *Optional*.

❽ An expression defining correlation data. When provided, this configures the underlying amqp template to receive publisher confirms. Requires a `RabbitTemplate` and a `CachingConnectionFactory` with the `publisherConfirms` property set to `true`. When a publisher confirm is received, it is written to either the confirm-ack-channel, or the confirm-nack-channel, depending on the confirmation type. The payload of the confirm is the correlation data as defined by this expression and the message will have a header 'amqp_publishConfirm' set to true (ack) or false (nack). Examples: "headers['myCorrelationData']", "payload". *Optional*.

❾ The channel to which positive (ack) publisher confirms are sent; payload is the correlation data defined by the *confirm-correlation-expression*. *Optional, default=nullChannel*.

❿ The channel to which negative (nack) publisher confirms are sent; payload is the correlation data defined by the *confirm-correlation-expression*. *Optional, default=nullChannel*.

11 The channel to which returned messages are sent. When provided, the underlying amqp template is configured to return undeliverable messages to the adapter. The message will be constructed from the data received from amqp, with the following additional headers: *amqp_returnReplyCode, amqp_returnReplyText, amqp_returnExchange, amqp_returnRoutingKey*. *Optional*.

Important

Using a `return-channel` requires a `RabbitTemplate` with either the `mandatory` or `immediate` properties set to `true`, and a `CachingConnectionFactory` with the `publisherReturns` property set to `true`. When using multiple outbound endpoints with returns, a separate `RabbitTemplate` is needed for each endpoint.

9.4 Inbound Gateway

A configuration sample for an AMQP Inbound Gateway is shown below with all available parameters.

```
<int-amqp:inbound-gateway id="inboundGateway"❶
                          request-channel="myRequestChannel"❷
                          queue-names="si.test.queue"❸
                          advice-chain=""❹
                          concurrent-consumers="1"❺
                          connection-factory="connectionFactory"❻
                          reply-channel="myReplyChannel"❼/>
```

❶ Unique ID for this adapter. *Optional*.

❷ Message Channel to which converted Messages should be sent. *Required*.

❸ Names of the AMQP Queues from which Messages should be consumed (comma-separated list). *Required*.

❹ Extra AOP Advice(s) to handle cross cutting behavior associated with this Inbound Gateway. *Optional*.

❺ Specify the number of concurrent consumers to create. Default is 1. Raising the number of concurrent consumers is recommended in order to scale the consumption of messages coming in from a queue. However, note that any ordering guarantees are lost once multiple consumers are registered. In general, stick with 1 consumer for low-volume queues. *Optional (Defaults to 1)*.

❻ Bean reference to the RabbitMQ ConnectionFactory. *Optional (Defaults to 'connectionFactory')*.

❼ Message Channel where reply Messages will be expected. *Optional*.

9.5 Outbound Gateway

A configuration sample for an AMQP Outbound Gateway is shown below with all available parameters.

```
<int-amqp:outbound-gateway id="inboundGateway"❶
                           request-channel="myRequestChannel"❷
                           amqp-template=""❸
                           exchange-name=""❹
                           order="1"❺
                           reply-channel=""❻
                           routing-key=""❼
                           routing-key-expression=""❽
                           return-channel=""❾/>
```

❶ Unique ID for this adapter. *Optional*.

❷ Message Channel to which Messages should be sent in order to have them converted and published to an AMQP Exchange. *Required*.

❸ Bean Reference to the configured AMQP Template *Optional (Defaults to "amqpTemplate")*.

❹ The name of the AMQP Exchange to which Messages should be sent. If not provided, Messages will be sent to the default, no-name Exchange. *Optional*.

❺ The order for this consumer when multiple consumers are registered thereby enabling load- balancing and/or failover. *Optional (Defaults to Ordered.LOWEST_PRECEDENCE [=Integer.MAX_VALUE])*.

❻ Message Channel to which replies should be sent after being received from an AQMP Queue and converted. *Optional*.

❼ The routing-key to use when sending Messages. By default, this will be an empty String. *Optional*.

❽ The routing-key to use when sending Messages evealuated as an expression on the message (e.g. 'payload.key'). By default, this will be an empty String. *Optional*.

❾ The channel to which returned messages are sent. When provided, the underlying amqp template is configured to return undeliverable messages to the gateway. The message will be constructed from the data received from amqp, with the following additional headers: *amqp_returnReplyCode, amqp_returnReplyText, amqp_returnExchange, amqp_returnRoutingKey*. *Optional*.

Important

Using a `return-channel` requires a `RabbitTemplate` with either the `mandatory` or `immediate` properties set to `true`, and a `CachingConnectionFactory` with the `publisherReturns` property set to `true`. When using multiple outbound endpoints with returns, a separate `RabbitTemplate` is needed for each endpoint.

Note

Prior to Spring Integration 2.2, and Spring AMQP 1.1, the outbound gateway used a new, temporary, reply queue for each request. This is still the default, but now the RabbitTemplate can be configured with a specific queue for replies; headers are added to the outbound message for request/reply correlation. It is important that the consuming application returns these headers unchanged. The headers are `spring_reply_correlation` and `spring_reply_to`. If the consuming application is a Spring Integration application, these headers will be managed automatically, including the case where that application might send a request/reply to a third application using an outbound gateway.

9.6 AMQP Backed Message Channels

There are two Message Channel implementations available. One is point-to-point, and the other is publish/subscribe. Both of these channels provide a wide range of configuration attributes for the underlying AmqpTemplate and SimpleMessageListenerContainer as you have seen on the Channel Adapters and Gateways. However, the examples we'll show here are going to have minimal configuration. Explore the XML schema to view the available attributes.

A point-to-point channel would look like this:

```
<int-amqp:channel id="p2pChannel"/>
```

Under the covers a Queue named "si.p2pChannel" would be declared, and this channel will send to that Queue (technically by sending to the no-name Direct Exchange with a routing key that matches this Queue's name). This channel will also register a consumer on that Queue. If for some reason, you want the Queue to be "pollable" instead of message-driven, then simply provide the "message-driven" flag with a value of false:

```
<int-amqp:channel id="p2pPollableChannel"  message-driven="false"/>
```

A publish/subscribe channel would look like this:

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

Under the covers a Fanout Exchange named "si.fanout.pubSubChannel" would be declared, and this channel will send to that Fanout Exchange. This channel will also declare a server-named exclusive, autodelete, non-durable Queue and bind that to the Fanout Exchange while registering a consumer on that Queue to receive Messages. There is no "pollable" option for a publish-subscribe-channel; it must be message-driven.

9.7 AMQP Message Headers

The Spring Integration AMPQ Adapters will map standard AMQP properties automatically. These properties will be copied by default to and from Spring Integration `MessageHeaders` using the `DefaultAmqpHeaderMapper`.

Of course, you can pass in your own implementation of AMQP specific header mappers, as the adapters have respective properties to support that.

Any user-defined headers within the AMQP `MessageProperties` will NOT be copied to or from an AMQP Message, unless explicitly specified by the *requestHeaderNames* and/or *replyHeaderNames* properties of the `DefaultAmqpHeaderMapper`.

Tip

When mapping user-defined headers, the values can also contain simple wildcard patterns (e.g. "foo*" or "*foo") to be matched. For example, if you need to copy all user-defined headers simply use the wild-card character '*'.

Class `AmqpHeaders` identifies the default headers that will be used by the `DefaultAmqpHeaderMapper`:

- amqp_appId
- amqp_clusterId

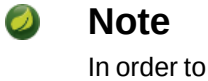
- amqp_contentEncoding
- amqp_contentLength
- content-type
- amqp_correlationId
- amqp_deliveryMode
- amqp_deliveryTag
- amqp_expiration
- amqp_messageCount
- amqp_messageId
- amqp_receivedExchange
- amqp_receivedRoutingKey
- amqp_redelivered
- amqp_replyTo
- amqp_timestamp
- amqp_type
- amqp_userId
- amqp_springReplyCorrelation
- amqp_springReplyToStack
- amqp_publishConfirm
- amqp_returnReplyCode
- amqp_returnReplyText
- amqp_returnExchange
- amqp_returnRoutingKey

9.8 AMQP Samples

To experiment with the AMQP adapters, check out the samples available in the Spring Integration Samples Git repository at:

- https://github.com/SpringSource/spring-integration-samples

Currently there is one sample available that demonstrates the basic functionality of the Spring Integration AMQP Adapter using an Outbound Channel Adapter and an Inbound Channel Adapter. As AMQP Broker implementation the sample uses RabbitMQ (http://www.rabbitmq.com/).

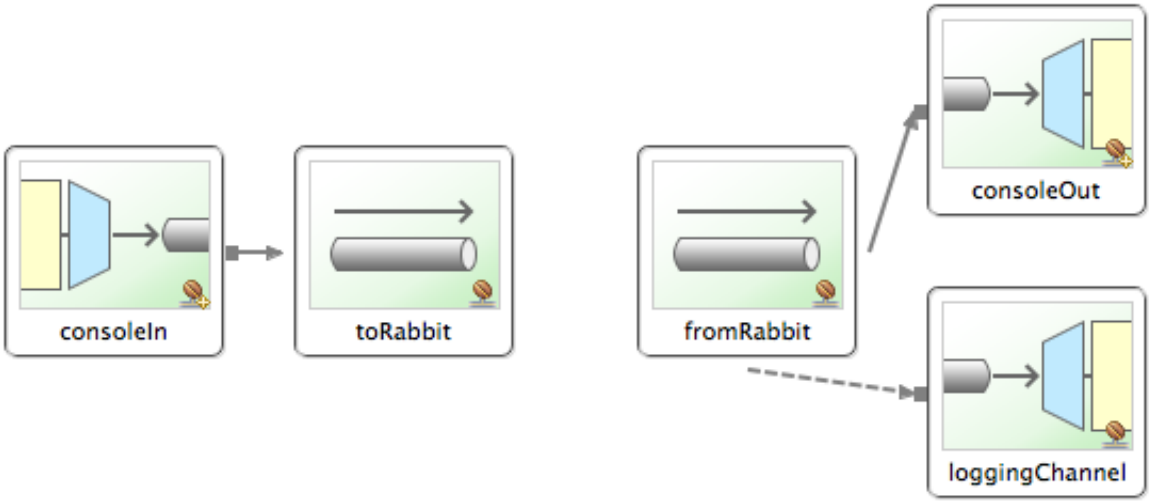


Note

In order to run the example you will need a running instance of RabbitMQ. A local installation with just the basic defaults will be sufficient. For detailed RabbitMQ installation procedures please visit: http://www.rabbitmq.com/install.html

Once the sample application is started, you enter some text on the command prompt and a message containing that entered text is dispatched to the AMQP queue. In return that message is retrieved via Spring Integration and then printed to the console.

The image belows illustrates the basic set of Spring Integration components used in this sample.



The Spring Integration graph of the AMQP sample

10. 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.

10.1 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.

10.2 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.

11. Feed Adapter

Spring Integration provides support for Syndication via Feed Adapters

11.1 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.xsd"
```

11.2 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"/>
```

12. File Support

12.1 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.

12.2 Reading Files

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

```
<bean id="pollableFileSource"
    class="org.springframework.integration.file.FileReadingMessageSource"
    p:inputDirectory="${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="${input.directory}"
    p:filter-ref="compositeFilter"/>
<bean id="compositeFilter"
    class="org.springframework.integration.file.filters.CompositeFileListFilter">
    <constructor-arg>
        <list>
            <bean class="o.s.i.file.filters.AcceptOnceFileListFilter"/>
            <bean class="o.s.i.file.filters.RegexPatternFileListFilter">
                <constructor-arg value="^test.*$"/>
            </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:int="http://www.springframework.org/schema/integration"
  xmlns:int-file="http://www.springframework.org/schema/integration/file"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/file
    http://www.springframework.org/schema/integration/file/spring-integration-file.xsd">
</beans>
```

Within this namespace you can reduce the FileReadingMessageSource and wrap it in an inbound Channel Adapter like this:

```
<int-file:inbound-channel-adapter id="filesIn1"
    directory="file:${input.directory}" prevent-duplicates="true"/>

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

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

<int-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

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

A custom locker you can configure like this:

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

Note

When a file inbound adapter is configured with a locker, it will take the responsibility to acquire a lock before the file is allowed to be received. **It will not assume the responsibility to unlock the file.** If you have processed the file and keeping the locks hanging around you have a memory leak. If this is a problem in your case you should call FileLocker.unlock(File file) yourself at the appropriate time.

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 <int-file:inbound-channel-adapter/> on the `scanner` attribute.

```
<int-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.

Important

It is important to understand that filters (including patterns, regex, prevent-duplicates etc) and lockers, are actually used by the scanner. Any of these attributes set on the adapter are subsequently injected into the scanner. For this reason, if you need to provide a custom scanner and you have multiple file inbound adapters in the same application context, each adapter must be provided with its own instance of the scanner, either by declaring separate beans, or declaring `scope="prototype"` on the scanner bean so that the context will create a new instance for each use.

'Tail'ing Files

Another popular use case is to get 'lines' from the end (or tail) of a file. Two implementations are provided; the first, `OSDelegatingFileTailingMessageProducer`, uses the native `tail` command (on operating systems that have one). This is likely the most efficient implementation on those platforms. For operating systems that do not have a `tail` command, the second implementation `ApacheCommonsFileTailingMessageProducer` which uses the Apache `commons-io Tailer` class.

In both cases, file system events, such as files being unavailable etc, are published as `ApplicationEvents` using the normal Spring event publishing mechanism. Examples of such events are:

```
[message=tail: cannot open `/tmp/foo' for reading: No such file or directory,
file=/tmp/foo]
```

```
[message=tail: `/tmp/foo' has become accessible, file=/tmp/foo]
```

```
[message=tail: `/tmp/foo' has become inaccessible: No such file or directory,
file=/tmp/foo]
```

```
[message=tail: `/tmp/foo' has appeared; following end of new file, file=/
tmp/foo]
```

This sequence of events might occur, for example, when a file is rotated.

Note

Not all platforms supporting a `tail` command provide these status messages.

Example configurations:

```
<int-file:tail-inbound-channel-adapter id="native"
 channel="input"
 task-executor="exec"
 file="/tmp/foo"/>
```

This creates a native adapter with default '-F -n 0' options (follow the file name from the current end).

```
<int-file:tail-inbound-channel-adapter id="native"
 channel="input"
 native-options="-F -n 6"
 task-executor="exec"
 file-delay=10000
 file="/tmp/foo"/>
```

This creates a native adapter with '-F -n 6' options (follow the file name, emit up to 6 lines before the current end). If the tail command fails (on some platforms, a missing file causes the `tail` to fail, even with `-F` specified), the command will be retried every 10 seconds.

```
<int-file:tail-inbound-channel-adapter id="apache"
 channel="input"
 task-executor="exec"
 file="/tmp/bar"
 delay="2000"
 end="false"
 reopen="true"
 file-delay="10000"/>
```

This creates a commons-io `Tailer` adapter that examines the file for new lines every 2 seconds, and checks for existence of a missing file every 10 seconds. The file will be tailed from the beginning (`end="false"`) instead of the end (which is the default). The file will be reopened for each chunk (the default is to keep the file open).

12.3 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.

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* or *Outbound Gateway* using the provided XML namespace support.

Generating Filenames

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`.

Alternatively, you can specify an expression to be evaluated against the Message in order to generate a file name, e.g.: *headers['myCustomHeader'] + 'foo'*. The expression must evaluate to a `String`. For convenience, the `DefaultFileNameGenerator` also provides the *setHeaderName* method, allowing you to explicitly specify the Message header whose value shall be used as the filename.

Once setup, the `DefaultFileNameGenerator` will employ the following resolution steps to determine the filename for a given Message payload:

1. Evaluate the expression against the Message and, if the result is a non-empty `String`, use it as the filename.
2. Otherwise, if the payload is a `java.io.File`, use the file's filename.
3. Otherwise, use the Message ID appended with ".msg" as the filename.

When using the XML namespace support, both, the *File Oubound Channel Adapter* and the *File Outbound Gateway* support the following two mutually exclusive configuration attributes:

- `filename-generator` (a reference to a `FileNameGenerator`) implementation)
- `filename-generator-expression` (an expression evaluating to a `String`)

While writing files, a temporary file suffix will be used (default: ".writing"). It is appended to the filename while the file is being written. To customize the suffix, you can set the *temporary-file-suffix* attribute on both, the *File Oubound Channel Adapter* and the *File Outbound Gateway*.

Note

When using the *APPEND* file *mode*, the *temporary-file-suffix* attribute is ignored, since the data is appended to the file directly.

Specifying the Output Directory

Both, the *File Oubound Channel Adapter* and the *File Outbound Gateway* provide two configuration attributes for specifying the output directory:

- *directory*
- *directory-expression*

Note

The *directory-expression* attribute is available since Spring Integration 2.2.

Using the directory attribute

When using the *directory* attribute, the output directory will be set to a fixed value, that is set at intialization time of the `FileWritingMessageHandler`. If you don't specify this attribute, then you must use the *directory-expression* attribute.

Using the directory-expression attribute

If you want to have full SpEL support you would choose the *directory-expression* attribute. This attribute accepts a SpEL expression that is evaluated for each message being processed. Thus, you have full access to a Message's payload and its headers to dynamically specify the output file directory.

The SpEL expression must resolve to either a `String` or to `java.io.File`. Furthermore the resulting `String` or `File` must point to a directory. If you don't specify the *directory-expression* attribute, then you must set the *directory* attribute.

Using the auto-create-directory attribute

If the destination directory does not exists, yet, by default the respective destination directory and any non-existing parent directories are being created automatically. You can set the *auto-create-directory* attribute to *false* in order to prevent that. This attribute applies to both, the *directory* and the *directory-expression* attribute.

Note

When using the *directory* attribute and *auto-create-directory* is `false`, the following change was made starting with Spring Integration 2.2:

Instead of checking for the existence of the destination directory at initialization time of the adapter, this check is now performed for each message being processed.

Furthermore, if *auto-create-directory* is `true` and the directory was deleted between the processing of messages, the directory will be re-created for each message being processed.

Dealing with Existing Destination Files

When writing files and the destination file already exists, the default behavior is to overwrite that target file. This behavior, though, can be changed by setting the *mode* attribute on the respective File Outbound components. The following options exist:

- REPLACE (Default)
- APPEND
- FAIL
- IGNORE

Note

The *mode* attribute and the options *APPEND*, *FAIL* and *IGNORE*, are available since *Spring Integration 2.2*.

REPLACE

If the target file already exists, it will be overwritten. If the *mode* attribute is not specified, then this is the default behavior when writing files.

APPEND

This mode allows you to append Message content to the existing file instead of creating a new file each time. Note that this attribute is mutually exclusive with *temporary-file-suffix* attribute since when appending content to the existing file, the adapter no longer uses a temporary file.

FAIL

If the target file exists, a MessageHandlingException is thrown.

IGNORE

If the target file exists, the message payload is silently ignored.

Note

When using a temporary file suffix (default: `.writing`), the *IGNORE* mode will apply if the final file name exists, or the temporary file name exists.

File Outbound Channel Adapter

```
<int-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 the deletion of the original source files after writing to a destination. The default value for that flag is `false`.

```
<int-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.

Outbound Gateway

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.

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

As mentioned earlier, you can also specify the *mode* attribute, which defines the behavior of how to deal with situations where the destination file already exists. Please see the section called “Dealing with Existing Destination Files” for further details. Generally, when using the *File Outbound Gateway*, the result file is returned as the Message payload on the reply channel.

This also applies when specifying the *IGNORE* mode. In that case the pre-existing destination file is returned. If the payload of the request message was a file, you still have access to that original file through the Message Header *FileHeaders.ORIGINAL_FILE*.

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`.

12.4 File Transformers

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

`FileToByteArrayTransformer` transforms Files into byte[]s using 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.

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

<int-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).

13. FTP/FTPS Adapters

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

13.1 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 three *client* side endpoints: *Inbound Channel Adapter*, *Outbound Channel Adapter*, and *Outbound Gateway*. 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:int-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.xsd"
```

13.2 FTP Session Factory

Important

Starting with version 3.0, sessions are no longer cached by default. See Section 13.6, “FTP Session Caching”.

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).

Advanced Configuration

`DefaultFtpSessionFactory` provides an abstraction over the underlying client API which, in the current release of Spring Integration, is Apache Commons Net. This spares you from the low level configuration details of the `org.apache.commons.net.ftp.FTPClient`. However there are times when access to lower level `FTPClient` details is necessary to achieve more advanced configuration (e.g., setting data timeout, default timeout etc.). For that purpose, `AbstractFtpSessionFactory` (the base class for all FTP Session Factories) exposes hooks, in the form of the two post-processing methods below.

```
/**
 * Will handle additional initialization after client.connect() method was invoked,
 * but before any action on the client has been taken
 */
protected void postProcessClientAfterConnect(T t) throws IOException {
    // NOOP
}
/**
 * Will handle additional initialization before client.connect() method was invoked.
 */
protected void postProcessClientBeforeConnect(T client) throws IOException {
    // NOOP
}
```

As you can see, there is no default implementation for these two methods. However, by extending `DefaultFtpSessionFactory` you can override these methods to provide more advanced configuration of the `FTPClient`. For example:

```
public class AdvancedFtpSessionFactory extends DefaultFtpSessionFactory {

    protected void postProcessClientBeforeConnect(FTPClient ftpClient) throws IOException
 {
       ftpClient.setDataTimeout(5000);
       ftpClient.setDefaultTimeout(5000);
    }
}
```

13.3 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-filename-generator-expression="#this.toUpperCase() + '.a'"
    local-filter="myFilter"
    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`.

By default the transferred file will carry the same name as the original file. If you want to override this behavior you can set the `local-filename-generator-expression` attribute which allows you to provide a SpEL Expression to generate the name of the local file. Unlike outbound gateways and adapters where the root object of the SpEL Evaluation Context is a `Message`, this inbound adapter does not yet have the Message at the time of evaluation since that's what it ultimately generates with the

transferred file as its payload. So, the root object of the SpEL Evaluation Context is the original name of the remote file (String).

Sometimes 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. This filter determines which remote files are retrieved.

Note

Beginning with 3.0, you can also specify a filter used to filter the files locally, once they have been retrieved. The default filter is an `AcceptOnceFileListFilter` which prevents processing files with the same name multiple times in the same JVM execution; this can now be overridden (for example with an `AcceptAllFileListFilter`), using the `local-filter` attribute. Previously, the default `AcceptOnceFileListFilter` could not be overridden.

The 'remote-file-separator' attribute 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

Sometimes 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.

13.4 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="/"
    auto-create-directory="true"
    remote-directory-expression="headers.['remote_dir']"
    temporary-remote-directory-expression="headers.['temp_remote_dir']"
    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 `session-factory`, as well as other attributes. You can also see some examples of `*expression` attributes which allow you to use SpEL to configure things like `remote-directory-expression`, `temporary-remote-directory-expression` and `remote-filename-generator-expression` (a SpEL alternative to `filename-generator` shown above). As with any component that allows the usage of SpEL, access to Payload and Message Headers is available via 'payload' and 'headers' variables. Please refer to the schema for more details on the available attributes.

Note

By default Spring Integration will use `o.s.i.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")

Avoiding Partially Written Files

One of the common problems, when dealing with file transfers, is the possibility of processing a *partial file* - a file might appear in the file system before its transfer is actually complete.

To deal with this issue, Spring Integration FTP adapters use a very common algorithm where files are transferred under a temporary name and then renamed once they are fully transferred.

By default, every file that is in the process of being transferred will appear in the file system with an additional suffix which, by default, is `.writing`; this can be changed using the `temporary-file-suffix` attribute.

However, there may be situations where you don't want to use this technique (for example, if the server does not permit renaming files). For situations like this, you can disable this feature by setting `use-temporary-file-name` to `false` (default is `true`). When this attribute is `false`, the file is written with its final name and the consuming application will need some other mechanism to detect that the file is completely uploaded before accessing it.

13.5 FTP Outbound Gateway

The *FTP Outbound Gateway* provides a limited set of commands to interact with a remote FTP/FTPS server.

Commands supported are:

- ls (list files)
- get (retrieve file)
- mget (retrieve file(s))
- rm (remove file(s))
- mv (move/rename file)

ls

ls lists remote file(s) and supports the following options:

- -1 - just retrieve a list of filenames, default is to retrieve a list of `FileInfo` objects.
- -a - include all files (including those starting with '.')

- -f - do not sort the list
- -dirs - include directories (excluded by default)
- -links - include symbolic links (excluded by default)

In addition, filename filtering is provided, in the same manner as the `inbound-channel-adapter`.

The message payload resulting from an *ls* operation is a list of file names, or a list of `FileInfo` objects. These objects provide information such as modified time, permissions etc.

The remote directory that the *ls* command acted on is provided in the `file_remoteDirectory` header.

get

get retrieves a remote file and supports the following option:

- -P - preserve the timestamp of the remote file

The message payload resulting from a *get* operation is a `File` object representing the retrieved file.

The remote directory is provided in the `file_remoteDirectory` header, and the filename is provided in the `file_remoteFile` header.

mget

mget retrieves multiple remote files based on a pattern and supports the following option:

- -x - Throw an exception if no files match the pattern (otherwise an empty list is returned)

The message payload resulting from an *mget* operation is a `List<File>` object - a List of File objects, each representing a retrieved file.

The remote directory is provided in the `file_remoteDirectory` header, and the pattern for the filenames is provided in the `file_remoteFile` header.

rm

The *rm* command has no options.

The message payload resulting from an *rm* operation is Boolean.TRUE if the remove was successful, Boolean.FALSE otherwise. The remote directory is provided in the `file_remoteDirectory` header, and the filename is provided in the `file_remoteFile` header.

mv

The *mv* command has no options.

The *expression* attribute defines the "from" path and the *rename-expression* attribute defines the "to" path. By default, the *rename-expression* is `headers['file_renameTo']`. This expression must not evaluate to null, or an empty `String`. If necessary, any remote directories needed will be created. The payload of the result message is `Boolean.TRUE`. The original remote directory is provided in the `file_remoteDirectory` header, and the filename is provided in the `file_remoteFile` header. The new path is in the `file_renameTo` header.

For all commands, the PATH that the command acts on is provided by the 'expression' property of the gateway. For the mget command, the expression might evaluate to '*', meaning retrieve all files, or 'somedirectory/*' etc.

Here is an example of a gateway configured for an ls command...

```
<int-ftp:outbound-gateway id="gateway1"
    session-factory="ftpSessionFactory"
    request-channel="inbound1"
    command="ls"
    command-options="-1"
    expression="payload"
    reply-channel="toSplitter"/>
```

The payload of the message sent to the toSplitter channel is a list of String objects containing the filename of each file. If the `command-options` was omitted, it would be a list of `FileInfo` objects. Options are provided space-delimited, e.g. `command-options="-1 -dirs -links"`.

13.6 FTP Session Caching

Important

Starting with version 3.0, sessions are no longer cached by default; the `cache-sessions` attribute is no longer supported on endpoints. You must now use a `CachingSessionFactory` (see below) if you wish to cache sessions.

In versions prior to 3.0, the sessions were cached automatically by default. A `cache-sessions` attribute was available for disabling the auto caching, but that solution did not provide a way to configure other session caching attributes. For example, you could not limit on the number of sessions created. To support that requirement and other configuration options, a `CachingSessionFactory` was provided. It provides `sessionCacheSize` and `sessionWaitTimeout` properties. As its name suggests, the `sessionCacheSize` property controls how many active sessions the factory will maintain in its cache (the DEFAULT is unbounded). If the `sessionCacheSize` threshold has been reached, any attempt to acquire another session will block until either one of the cached sessions becomes available or until the wait time for a Session expires (the DEFAULT wait time is Integer.MAX_VALUE). The `sessionWaitTimeout` property enables configuration of that value.

If you want your Sessions to be cached, simply configure your default Session Factory as described above and then wrap it in an instance of `CachingSessionFactory` where you may provide those additional properties.

```
<bean id="ftpSessionFactory" class="o.s.i.ftp.session.DefaultFtpSessionFactory">
    <property name="host" value="localhost"/>
</bean>

<bean id="cachingSessionFactory" class="o.s.i.file.remote.session.CachingSessionFactory">
    <constructor-arg ref="ftpSessionFactory"/>
    <constructor-arg value="10"/>
    <property name="sessionWaitTimeout" value="1000"/>
</bean>
```

In the above example you see a `CachingSessionFactory` created with the `sessionCacheSize` set to 10 and the `sessionWaitTimeout` set to 1 second (its value is in millliseconds).

14. GemFire Support

Spring Integration provides support for VMWare vFabric GemFire

14.1 Introduction

VMWare vFabric GemFire (GemFire) is a distributed data management platform providing a key-value data grid along with advanced distributed system features such as event processing, continuous querying, and remote function execution. This guide assumes some familiarity with GemFire and its API.

Spring integration provides support for GemFire by providing inbound adapters for entry and continuous query events, an outbound adapter to write entries to the cache, and `MessageStore` and `MessageGroupStore` implementations. Spring integration leverages the Spring Gemfire project, providing a thin wrapper over its components.

To configure the 'int-gfe' namespace, include the following elements within the headers of your XML configuration file:

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

14.2 Inbound Channel Adapter

The *inbound-channel-adapter* produces messages on a channel triggered by a GemFire `EntryEvent`. GemFire generates events whenever an entry is CREATED, UPDATED, DESTROYED, or INVALIDATED in the associated region. The inbound channel adapter allows you to filter on a subset of these events. For example, you may want to only produce messages in response to an entry being CREATED. In addition, the inbound channel adapter can evaluate a SpEL expression if, for example, you want your message payload to contain an event property such as the new entry value.

```
<gfe:cache/>
<gfe:replicated-region id="region"/>
<int-gfe:inbound-channel-adapter id="inputChannel" region="region"
    cache-events="CREATED" expression="newValue"/>
```

In the above configuration, we are creating a GemFire `Cache` and `Region` using Spring GemFire's 'gfe' namespace. The inbound-channel-adapter requires a reference to the GemFire region for which the adapter will be listening for events. Optional attributes include `cache-events` which can contain a comma separated list of event types for which a message will be produced on the input channel. By default CREATED and UPDATED are enabled. Note that this adapter conforms to Spring integration conventions. If no `channel` attribute is provided, the channel will be created from the `id` attribute. This adapter also supports an `error-channel`. If `expression` is not provided the message payload will be a GemFire `EntryEvent`

14.3 Continuous Query Inbound Channel Adapter

The *cq-inbound-channel-adapter* produces messages a channel triggered by a GemFire continuous query or `CqEvent` event. Spring GemFire introduced continuous query support in release 1.1, including a `ContinuousQueryListenerContainer` which provides a nice abstraction over the GemFire native API. This adapter requires a reference to a ContinuousQueryListenerContainer, and creates a listener

for a given `query` and executes the query. The continuous query acts as an event source that will fire whenever its result set changes state.

Note

GemFire queries are written in OQL and are scoped to the entire cache (not just one region). Additionally, continuous queries require a remote (i.e., running in a separate process or remote host) cache server. Please consult the GemFire documentation for more information on implementing continuous queries.

```
<gfe:client-cache id="client-cache" pool-name="client-pool"/>

<gfe:pool id="client-pool" subscription-enabled="true" >
    <!--configure server or locator here required to address the cache server -->
</gfe:pool>

<gfe:client-region id="test" cache-ref="client-cache" pool-name="client-pool"/>

<gfe:cq-listener-container id="queryListenerContainer" cache="client-cache"
    pool-name="client-pool"/>

<int-gfe:cq-inbound-channel-adapter id="inputChannel"
    cq-listener-container="queryListenerContainer"
    query="select * from /test"/>
```

In the above configuration, we are creating a GemFire client cache (recall a remote cache server is required for this implementation and its address is configured as a sub-element of the pool), a client region and a `ContinuousQueryListenerContainer` using Spring GemFire. The continuous query inbound channel adapter requires a `cq-listener-container` attribute which contains a reference to the `ContinuousQueryListenerContainer`. Optionally, it accepts an `expression` attribute which uses SpEL to transform the `CqEvent` or extract an individual property as needed. The cq-inbound-channel-adapter provides a `query-events` attribute, containing a comma separated list of event types for which a message will be produced on the input channel. Available event types are CREATED, UPDATED, DESTROYED, REGION_DESTROYED, REGION_INVALIDATED. CREATED and UPDATED are enabled by default. Additional optional attributes include, `query-name` which provides an optional query name, and `expression` which works as described in the above section, and `durable` - a boolean value indicating if the query is durable (false by default). Note that this adapter conforms to Spring integration conventions. If no `channel` attribute is provided, the channel will be created from the `id` attribute. This adapter also supports an `error-channel`

14.4 Outbound Channel Adapter

The *outbound-channel-adapter* writes cache entries mapped from the message payload. In its simplest form, it expects a payload of type `java.util.Map` and puts the map entries into its configured region.

```
<int-gfe:outbound-channel-adapter id="cacheChannel" region="region"/>
```

Given the above configuration, an exception will be thrown if the payload is not a Map. Additionally, the outbound channel adapter can be configured to create a map of cache entries using SpEL of course.

```
<int-gfe:outbound-channel-adapter id="cacheChannel" region="region">
    <int-gfe:cache-entries>
        <entry key="payload.toUpperCase()" value="payload.toLowerCase()"/>
        <entry key="'foo'" value="'bar'"/>
    </int-gfe:cache-entries>
</int-gfe:outbound-channel-adapter>
```

In the above configuration, the inner element `cache-entries` is semantically equivalent to Spring 'map' element. The adapter interprets the `key` and `value` attributes as SpEL expressions with the message as the evaluation context. Note that this contain arbitrary cache entries (not only those derived from the message) and that literal values must be enclosed in single quotes. In the above example, if the message sent to `cacheChannel` has a String payload with a value "Hello", two entries `[HELLO:hello, foo:bar]` will be written (created or updated) in the cache region. This adapter also supports the `order` attribute which may be useful if it is bound to a PublishSubscribeChannel.

14.5 Gemfire Message Store

As described in EIP, a Message Store allows you to persist Messages. This can be very useful when dealing with components that have a capability to buffer messages (*QueueChannel, Aggregator, Resequencer*, etc.) if reliability is a concern. In Spring Integration, the MessageStore strategy also provides the foundation for the ClaimCheck pattern, which is described in EIP as well.

Spring Integration's Gemfire module provides the `GemfireMessageStore` which is an implementation of both the the `MessageStore` strategy (mainly used by the *QueueChannel* and *ClaimCheck* patterns) and the `MessageGroupStore` strategy (mainly used by the *Aggregator* and *Resequencer* patterns).

```
<bean id="gemfireMessageStore" class="o.s.i.gemfire.store.GemfireMessageStore">
    <constructor-arg ref="myCache"/>
</bean>

<bean id="myCache" class="org.springframework.data.gemfire.CacheFactoryBean"/>

<int:channel id="somePersistentQueueChannel">
    <int:queue message-store="gemfireMessageStore"/>
<int:channel>

<int:aggregator input-channel="inputChannel" output-channel="outputChannel"
    message-store="gemfireMessageStore"/>
```

Above is a sample `GemfireMessageStore` configuration that shows its usage by a *QueueChannel* and an *Aggregator*. As you can see it is a normal Spring bean configuration. The simplest configuration requires a reference to a `GemFireCache` (created by `CacheFactoryBean`) as a constructor argument. If the cache is standalone, i.e., embedded in the same JVM, the MessageStore will create a message store region named "messageStoreRegion". If your application requires customization of the messageStore region, for example, multiple Gemfire message stores each with its own region, you can configure a region for each message store instance and use the `Region` as the constructor argument:

```
<bean id="gemfireMessageStore" class="o.s.i.gemfire.store.GemfireMessageStore">
    <constructor-arg ref="myRegion"/>
</bean>

<gfe:cache/>

<gfe:replicated-region id="myRegion"/>
```

In the above examle, the cache and region are configured using the spring-gemfire namespace (not to be confused with the spring-integration-gemfire namespace). Often it is desirable for the message store to be maintained in one or more remote cache servers in a client-server configuration (See the GemFire product documentation for more details). In this case, you configure a client cache, client region, and client pool and inject the region into the MessageStore. Here is an example:

```
<bean id="gemfireMessageStore"
    class="org.springframework.integration.gemfire.store.GemfireMessageStore">
    <constructor-arg ref="myRegion"/>
</bean>

<gfe:client-cache/>

<gfe:client-region id="myRegion" shortcut="PROXY" pool-name="messageStorePool"/>

<gfe:pool id="messageStorePool">
    <gfe:server host="localhost" port="40404" />
</gfe:pool>
```

Note the *pool* element is configured with the address of a cache server (a locator may be substituted here). The region is configured as a 'PROXY' so that no data will be stored locally. The region's id corresponds to a region with the same name configured in the cache server.

15. HTTP Support

15.1 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`.

15.2 Http Inbound Gateway

To receive messages over HTTP, you need to use an *HTTP Inbound Channel Adapter* or *Gateway*. To support the *HTTP Inbound Adapters*, they need to be deployed within a servlet container such as Apache Tomcat or Jetty. The easiest way to do this is to use Spring's `HttpRequestHandlerServlet`, by providing the following servlet definition in the *web.xml* file:

```
<servlet>
    <servlet-name>inboundGateway</servlet-name>
    <servlet-class>o.s.web.context.support.HttpRequestHandlerServlet</servlet-class>
</servlet>
```

Notice that the servlet name matches the bean name. For more information on using the `HttpRequestHandlerServlet`, see chapter "Remoting and web services using Spring", which is part of the Spring Framework Reference documentation.

If you are running within a Spring MVC application, then the aforementioned explicit 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. For more information, please see the chapter "Web MVC framework", which is part of the Spring Framework Reference documentation.

Tip

For a sample application and the corresponding configuration, please see the Spring Integration Samples repository. It contains the Http Sample application demonstrating Spring Integration's HTTP support.

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.

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.

15.3 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.

Note

In the case of the Outbound Gateway, the reply message produced by the gateway will contain all Message Headers present in the request message.

Cookies

Basic cookie support is provided by the *transfer-cookies* attribute on the outbound gateway. When set to true (default is false), a *Set-Cookie* header received from the server in a response will be converted to *Cookie* in the reply message. This header will then be used on subsequent sends. This enables simple stateful interactions, such as...

...->logonGateway->...->doWorkGateway->...->logoffGateway->...

If *transfer-cookies* is false, any *Set-Cookie* header received will remain as *Set-Cookie* in the reply message, and will be dropped on subsequent sends.

15.4 HTTP Namespace Support

Spring Integration provides an *http* namespace and the corresponding schema definition. To include it in your configuration, simply provide the following namespace declaration in your application context configuration file:

```
<?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:int="http://www.springframework.org/schema/integration"
  xmlns:int-http="http://www.springframework.org/schema/integration/http"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/http
    http://www.springframework.org/schema/integration/http/spring-integration-http.xsd">
    ...
</beans>
```

Inbound

The XML Namespace provides two components for handling HTTP Inbound requests. In order to process requests without returning a dedicated response, use the *inbound-channel-adapter*:

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

To process requests that do expect a response, use an *inbound-gateway*:

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

Important

Beginning with *Spring Integration 2.1* the *HTTP Inbound Gateway* and the *HTTP Inbound Channel Adapter* should use the *path* attribute instead of the *name* attribute for specifying the request path. The *name* attribute for those 2 components has been deprecated.

If you simply want to identify component itself within your application context, please use the *id* attribute.

Defining the UriPathHandlerMapping

In order to use the *HTTP Inbound Gateway* or the *HTTP Inbound Channel Adapter* you must define a `UriPathHandlerMapping`. This particular implementation of the `HandlerMapping` matches against the value of the *path* attribute.

```
<bean class="org.springframework.integration.http.inbound.UriPathHandlerMapping"/>
```

For more information regarding *Handler Mappings*, please see:

- http://static.springsource.org/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-handlermapping

URI Template Variables and Expressions

By Using the *path* attribute in conjunction with the *payload-expression* attribute as well as the *header* sub-element, you have a high degree of flexiblity for mapping inbound request data.

In the following example configuration, an Inbound Channel Adapter is configured to accept requests using the following URI: */first-name/{firstName}/last-name/{lastName}*

Using the *payload-expression* attribute, the URI template variable *{firstName}* is mapped to be the Message payload, while the *{lastName}* URI template variable will map to the *lname* Message header.

```
<int-http:inbound-channel-adapter id="inboundAdapterWithExpressions"
    path="/first-name/{firstName}/last-name/{lastName}"
    channel="requests"
    payload-expression="#pathVariables.firstName">
    <int-http:header name="lname" expression="#pathVariables.lastName"/>
</int-http:inbound-channel-adapter>
```

For more information about *URI template variables*, please see the Spring Reference Manual:

- http://static.springsource.org/spring/docs/current/spring-framework-reference/htmlsingle/spring-framework-reference.html#mvc-ann-requestmapping

Outbound

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 contain the ResponseEntity as long as it's http status is a success (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.

Important

Beginning with Spring Integration 2.1 the *request-timeout* attribute of the HTTP Outbound Gateway was renamed to *reply-timeout* to better reflect the intent.

```
<int-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"
    reply-timeout="1234"
    reply-channel="replies"/>
```

Important

Since *Spring Integration 2.2*, Java serialization over HTTP is no longer enabled by default. Previously, when setting the `expected-response-type` attribute to a `Serializable` object, the `Accept` header was not properly set up. Since *Spring Integration 2.2*, the `SerializingHttpMessageConverter` has now been updated to set the `Accept` header to `application/x-java-serialized-object`.

However, because this could cause incompatibility with existing applications, it was decided to no longer automatically add this converter to the HTTP endpoints. If you wish to use Java serialization, you will need to add the `SerializingHttpMessageConverter` to the appropriate endpoints, using the `message-converters` attribute, when using XML configuration, or using the `setMessageConverters()` method. Alternatively, you may wish to consider using JSON instead which is enabled by simply having `Jackson` on the classpath.

Beginning with Spring Integration 2.2 you can also determine the HTTP Method dynamically using SpEL and the *http-method-expression* attribute. Note that this attribute is obviously murually exclusive with *http-method* You can also use `expected-response-type-expression` attribute instead of `expected-response-type` and provide any valid SpEL expression that determines the type of the response.

```
<int-http:outbound-gateway id="example"
    request-channel="requests"
    url="http://localhost/test"
    http-method-expression="headers.httpMethod"
    extract-request-payload="false"
    expected-response-type-expression="payload"
    charset="UTF-8"
    request-factory="requestFactory"
    reply-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:

```
<int-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"/>
```

Note

To specify the URL; you can use either the 'url' attribute or the 'url-expression' attribute. The 'url' is a simple string (with placedholders for URI variables, as described below); the 'url-expression' is a SpEL expression, with the Message as the root object, enabling dynamic urls. The url resulting from the expression evaluation can still have placeholders for URI variables.

In previous releases, some users used the place holders to replace the entire URL with a URI variable. Changes in Spring 3.1 can cause some issues with escaped characters, such as '?'. For this reason, it is recommended that if you wish to generate the URL entirely at runtime, you use the 'url-expression' attribute.

Mapping URI Variables

If your URL contains URI variables, you can map them using the `uri-variable` sub-element. This sub-element is available for the *Http Outbound Gateway* and the *Http Outbound Channel Adapter*.

```
<int-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">
    <int-http:uri-variable name="zipCode" expression="payload.getZip()"/>
</int-http:outbound-gateway>
```

The `uri-variable` sub-element defines two attributes: `name` and `expression`. The `name` attribute identifies the name of the URI variable, while the `expression` attribute is used to set the actual value. Using the `expression` attribute, you can leverage the full power of the Spring Expression Language (SpEL) which gives you full dynamic access to the message payload and the message headers. For example, 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 the URI variable named 'zipCode'.

Controlling URI Encoding

By default, the URL string is encoded (see UriComponentsBuilder) to the URI object before sending the request. In some scenarios with a non-standard URI (e.g. the RabbitMQ Rest API) it is undesirable to perform the encoding. The `<http:outbound-gateway/>` and `<http:outbound-channel-adapter/>` provide an `encode-uri` attribute. To disable encoding the URL, this attribute should be set to `false` (by default it is `true`). If you wish to partially encode some of the URL, this can be achieved using an `expression` within a `<uri-variable/>`:

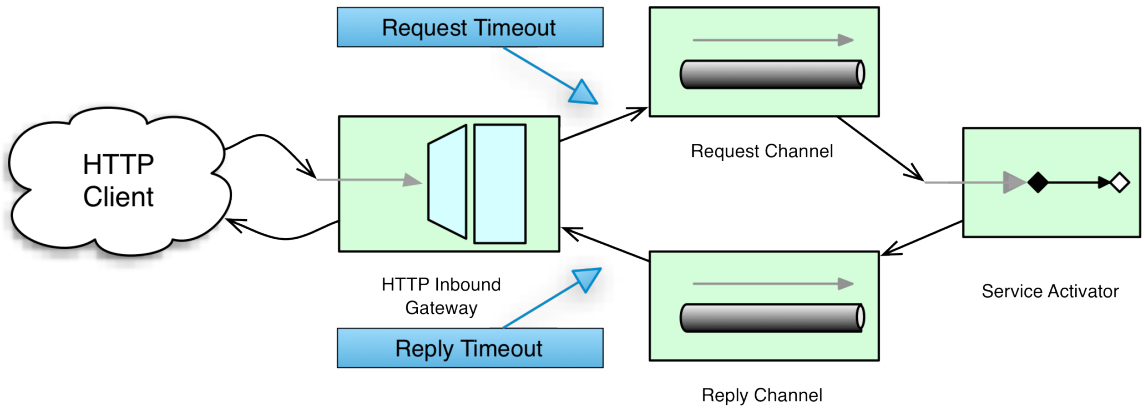
```
<http:outbound-gateway url="http://somehost/%2f/fooApps?bar={param}" encode-uri="false">
          <http:uri-variable name="param"
            expression="T(org.apache.commons.httpclient.util.URIUtil)
                                          .encodeWithinQuery('Hellow World!')"/>
</http:outbound-gateway>
```

15.5 Timeout Handling

In the context of HTTP components, there are two timing areas that have to be considered.

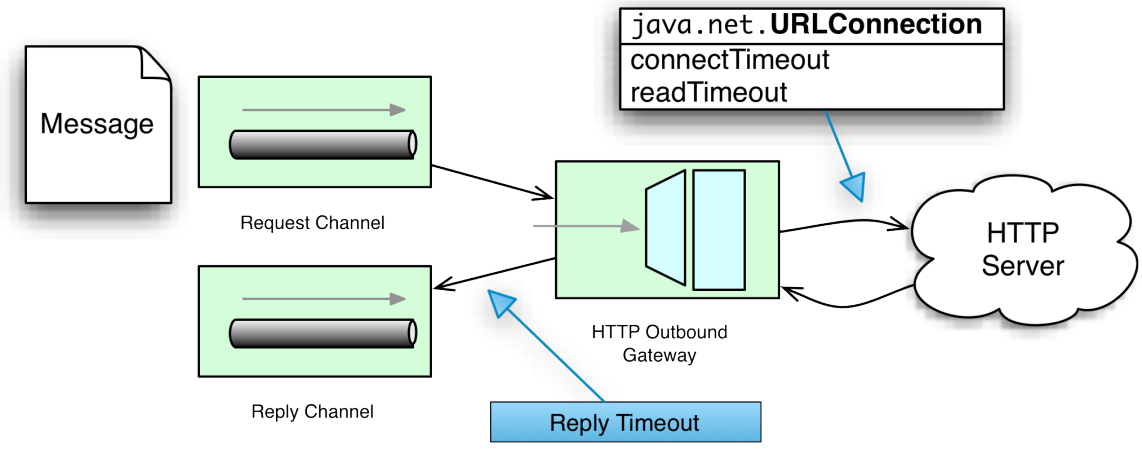
- Timeouts when interacting with Spring Integration Channels
- Timeouts when interacting with a remote HTTP server

First, the components interact with Message Channels, for which timeouts can be specified. For example, an HTTP Inbound Gateway will forward messages received from connected HTTP Clients to a Message Channel (Request Timeout) and consequently the HTTP Inbound Gateway will receive a reply Message from the Reply Channel (Reply Timeout) that will be used to generate the HTTP Response. Please see the figure below for an illustration.



How timeout settings apply to an HTTP Inbound Gateway

For outbound endpoints, the second thing to consider is timing while interacting with the remote server.



How timeout settings apply to an HTTP Outbound Gateway

You may want to configure the HTTP related timeout behavior, when making active HTTP requests using the *HTTP Oubound Gateway* or the *HTTP Outbound Channel Adapter*. In those instances, these two components use Spring's `RestTemplate` support to execute HTTP requests.

In order to configure timeouts for the *HTTP Oubound Gateway* and the *HTTP Outbound Channel Adapter*, you can either reference a `RestTemplate` bean directly, using the *rest-template* attribute, or you can provide a reference to a *ClientHttpRequestFactory* bean using the *request-factory* attribute. Spring provides the following implementations of the `ClientHttpRequestFactory` interface:

`SimpleClientHttpRequestFactory` - Uses standard J2SE facilities for making HTTP Requests

`HttpComponentsClientHttpRequestFactory` - Uses Apache HttpComponents HttpClient (Since Spring 3.1)

`ClientHttpRequestFactory` - Uses Jakarta Commons HttpClient (Deprecated as of Spring 3.1)

If you don't explicitly configure the *request-factory* or *rest-template* attribute respectively, then a default RestTemplate which uses a `SimpleClientHttpRequestFactory` will be instantiated.

Note

With some JVM implementations, the handling of timeouts using the *URLConnection* class may not be consistent.

E.g. from the *Java™ Platform, Standard Edition 6 API Specification* on *setConnectTimeout*: “Some non-standard implmentation of this method may ignore the specified timeout. To see the connect timeout set, please call getConnectTimeout().”

Please test your timeouts if you have specific needs. Consider using the `HttpComponentsClientHttpRequestFactory` which, in turn, uses *Apache HttpComponents HttpClient* instead.

Here is an example of how to configure an *HTTP Outbound Gateway* using a `SimpleClientHttpRequestFactory`, configured with connect and read timeouts of 5 seconds respectively:

```
<int-http:outbound-gateway url="http://www.google.com/ig/api?weather={city}"
                           http-method="GET"
                           expected-response-type="java.lang.String"
                           request-factory="requestFactory"
                           request-channel="requestChannel"
                           reply-channel="replyChannel">
    <int-http:uri-variable name="city" expression="payload"/>
</int-http:outbound-gateway>

<bean id="requestFactory"
      class="org.springframework.http.client.SimpleClientHttpRequestFactory">
    <property name="connectTimeout" value="5000"/>
    <property name="readTimeout"    value="5000"/>
</bean>
```

HTTP Outbound Gateway

For the *HTTP Outbound Gateway*, the XML Schema defines only the *reply-timeout*. The *reply-timeout* maps to the *sendTimeout* property of the

org.springframework.integration.http.outbound.HttpRequestExecutingMessageHandler class. More precisely, the property is set on the extended `AbstractReplyProducingMessageHandler` class, which ultimatelly sets the property on the *MessagingTemplate*.

The value of the *sendTimeout* property defaults to "-1" and will be applied to the connected `MessageChannel`. This means, that depending on the implementation, the Message Channel's *send* method may block indefinitely. Furthermore, the *sendTimeout* property is only used, when the actual MessageChannel implementation has a blocking send (such as 'full' bounded QueueChannel).

HTTP Inbound Gateway

For the *HTTP Inbound Gateway*, the XML Schema defines the *request-timeout* attribute, which will be used to set the *requestTimeout* property on the `HttpRequestHandlingMessagingGateway` class (on the extended MessagingGatewaySupport class). Secondly, the *reply-timeout* attribute exists and it maps to the *replyTimeout* property on the same class.

The default for both timeout properties is "1000ms". Ultimately, the *request-timeout* property will be used to set the *sendTimeout* on the used `MessagingTemplate` instance. The *replyTimeout* property on the other hand, will be used to set the *receiveTimeout* property on the used `MessagingTemplate` instance.

Tip

In order to simulate connection timeouts, connect to a non-routable IP address, for example 10.255.255.10.

15.6 HTTP Proxy configuration

If you are behind a proxy and need to configure proxy settings for HTTP outbound adapters and/or gateways, you can apply one of two approaches. In most cases, you can rely on the standard Java System Properties that control the proxy settings. Otherwise, you can explicitly configure a Spring bean for the HTTP client request factory instance.

Standard Java Proxy configuration

There are 3 System Properties you can set to configure the proxy settings that will be used by the HTTP protocol handler:

- *http.proxyHost* - the host name of the proxy server.
- *http.proxyPort* - the port number, the default value being 80.
- *http.nonProxyHosts* - a list of hosts that should be reached directly, bypassing the proxy. This is a list of patterns separated by '|'. The patterns may start or end with a '*' for wildcards. Any host matching one of these patterns will be reached through a direct connection instead of through a proxy.

And for HTTPS:

- *https.proxyHost* - the host name of the proxy server.
- *https.proxyPort* - the port number, the default value being 80.

For more information please refer to this document: http://download.oracle.com/javase/6/docs/technotes/guides/net/proxies.html

Spring's SimpleClientHttpRequestFactory

If for any reason, you need more explicit control over the proxy configuration, you can use Spring's `SimpleClientHttpRequestFactory` and configure its 'proxy' property as such:

```
<bean id="requestFactory"
    class="org.springframework.http.client.SimpleClientHttpRequestFactory">
    <property name="proxy">
        <bean id="proxy" class="java.net.Proxy">
            <constructor-arg>
                <util:constant static-field="java.net.Proxy.Type.HTTP"/>
            </constructor-arg>
            <constructor-arg>
                <bean class="java.net.InetSocketAddress">
                    <constructor-arg value="123.0.0.1"/>
                    <constructor-arg value="8080"/>
                </bean>
            </constructor-arg>
        </bean>
    </property>
</bean>
```

15.7 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="o.s.i.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.

15.8 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
```

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 “Enter the GatewayProxyFactoryBean”.

16.2 UDP Adapters

```
<int-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.

```
<int-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.

```
<int-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.

```
<int-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.

```
<int-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.

By default, reverse DNS lookups are done on inbound packets to convert IP addresses to hostnames for use in message headers. In environments where DNS is not configured, this can cause delays. This default behavior can be overridden by setting the `lookup-host` attribute to "false".

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. By default, only payload information is transferred using TCP; therefore any message correlation must be performed by downstream components such as aggregators or other endpoints. Support for transferring selected headers was introduced in version 3.0. For more information refer to Section 16.8, “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.

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

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

```
<int-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.

```
<int-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.

```
<int-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. A number of standard (de)serializers are provided.

The `ByteArrayCrlfSerializer`, converts 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 `ByteArraySingleTerminatorSerializer`, converts a byte array to a stream of bytes followed by a single termination character (default 0x00).

The `ByteArrayLfSerializer`, converts a byte array to a stream of bytes followed by a single linefeed character (0x0a).

The `ByteArrayStxEtxSerializer`, converts a byte array to a stream of bytes preceded by an STX (0x02) and followed by an ETX (0x03).

The `ByteArrayLengthHeaderSerializer`, converts a byte array to a stream of bytes preceded by a binary length in network byte order (big endian). This a very efficient deserializer because it does not have to parse every byte looking for a termination character sequence. It can also be used for payloads containing binary data; the above serializers only support text in the payload. The default size of the length header is 4 bytes (Integer), allowing for messages up to 2**31-1 bytes. However, the length header can be a single byte (unsigned) for messages up to 255 bytes, or an unsigned short (2 bytes) for messages up to 2**16 bytes. If you need any other format for the header, you can subclass this class and provide implementations for the readHeader and writeHeader methods. The absolute maximum data size supported is 2**31-1 bytes.

The `ByteArrayRawSerializer`, converts a byte array to a stream of bytes and adds no additional message demarcation data; with this (de)serializer, the end of a message is indicated by the client closing the socket in an orderly fashion. When using this serializer, message reception will hang until the client closes the socket, or a timeout occurs; a timeout will NOT result in a message. When this serializer is being used, and the client is a Spring Integration application, the client must use a connection factory

that is configured with single-use=true - this causes the adapter to close the socket after sending the message; the serializer will not, itself, close the connection. This serializer should only be used with connection factories used by channel adapters (not gateways), and the connection factories should be used by either an inbound or outbound adapter, and not both.

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.

To avoid memory exhaustion due to a badly behaved client (one that does not adhere to the protocol of the configured serializer), these serializers impose a maximum message size. If the size is exceeded by an incoming message, an exception will be thrown. The default maximum message size is 2048 bytes, and can be increased by setting the `maxMessageSize` property. If you are using the default (de)serializer and wish to increase the maximum message size, you must declare it as an explicit bean with the property set and configure the connection factory to use that bean.

The `MapJsonSerializer` uses a Jackson `ObjectMapper` to convert between a `Map` and JSON. This can be used in conjunction with a `MessageConvertingTcpMessageMapper` and a `MapMessageConverter` to transfer selected headers and the payload in a JSON format.

Note

The Jackson `ObjectMapper` cannot demarcate messages in the stream. Therefore, the `MapJsonSerializer` needs to delegate to another (de)serializer to handle message demarcation. By default, a `ByteArrayLfSerializer` is used, resulting in messages with the format `<json><LF>` on the wire, but you can configure it to use others instead.

The final 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" />

<int-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.

By default, reverse DNS lookups are done on inbound packets to convert IP addresses to hostnames for use in message headers. In environments where DNS is not configured, this can cause connection delays. This default behavior can be overridden by setting the `lookup-host` attribute to "false".

Note

It is possible to modify the creation of and/or attributes of sockets - see Section 16.10, “SSL/TLS Support”. As is noted there, such modifications are possible whether or not SSL is being used.

TCP Caching Client Connection Factory

As noted above, TCP sockets cam be 'single-use' (one request/response) or shared. Shared sockets do not perform well with outbound gateways, in high-volume environments, because the socket can only process one request/response at a time.

To improve performance, users could use collaborating channel adapters instead of gateways, but that requires application-level message correlation. See Section 16.8, “TCP Message Correlation”for more information.

Spring Integration 2.2 introduced a caching client connection factory, where a pool of shared sockets is used, allowing a gateway to process multiple concurrent requests with a pool of shared connections.

TCP Failover Client Connection Factory

It is now possible to configure a TCP connection factory that supports failover to one or more other servers. When sending a message, the factory will iterate over all its configured factories until either the message can be sent, or no connection can be found. Initially, the first factory in the configured list is used; if a connection subsequently fails the next factory will become the current factory.

```
<bean id="failCF" class="o.s.i.ip.tcp.connection.FailoverClientConnectionFactory">
    <constructor-arg>
        <list>
            <ref bean="clientFactory1"/>
            <ref bean="clientFactory2"/>
        </list>
    </constructor-arg>
</bean>
```

Note

When using the failover connection factory, the singleUse property must be consistent between the factory itself and the list of factories it is configured to use.

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="o.s.i.ip.tcp.connection.TcpConnectionInterceptorFactoryChain">
    <property name="interceptors">
        <array>
            <bean class="o.s.i.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 Connection Events

Beginning with version 3.0, changes to `TcpConnections` are reported by `TcpConnectionEvents`. `TcpConnectionEvent` is a subclass of `ApplicationEvent` and thus can be received by any `ApplicationListener` defined in the `ApplicationContext`.

For convenience, a `<int-ip:tcp-connection-event-inbound-channel-adapter/>` is provided. This adapter will receive all `TcpConnectionEvents` (by default), and send them to its `channel`. The adapter accepts an `event-type` attribute, which is a list of class names for events that should be sent. This can be used if an application subclasses `TcpConnectionEvent` for some reason, and wishes to only receive those events. Omitting this attribute will mean that all `TcpConnectionEvents` will be sent. You can also use this

to limit which `TcpConnectionEvents` you are interested in (`TcpConnectionOpenEvent`, `TcpConnectionCloseEvent`, or `TcpConnectionExceptionEvent`).

`TcpConnectionEvents` contain:

- Connection Id (which can be used in a message header to send data to the connection)
- Connection Factory Name (the bean name of the connection factory the connection belongs to)
- The `Throwable` (for `TcpConnectionExceptionEvent` events only)

16.6 TCP Adapters

TCP inbound and outbound channel adapters that utilize the above connection factories are provided. These adapters have 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.

Normally, inbound adapters use a type="server" connection factory, which listens for incoming connection requests. In some cases, it is desireable to establish the connection in reverse, whereby the inbound adapter connects to an external server and then waits for inbound messages on that connection.

This topology is supported by using *client-mode="true"* on the inbound adapter. In this case, the connection factory must be of type 'client' and must have *single-use* set to false.

Two additional attributes are used to support this mechanism: *retry-interval* specifies (in milliseconds) how often the framework will attempt to reconnect after a connection failure. *scheduler* is used to supply

a `TaskScheduler` used to schedule the connection attempts, and to test that the connection is still active.

For an outbound adapter, the connection is normally established when the first message is sent. *client-mode="true"* on an outbound adapter will cause the connection to be established when the adapter is started. Adapters are automatically started by default. Again, the connection factory must be of type client and have *single-use* set to false and *retry-interval* and *scheduler* are also supported. If a connection fails, it will be re-established either by the scheduler or when the next message is sent.

For both inbound and outbound, if the adapter is started, you may force the adapter to establish a connection by sending a <control-bus /> command: `@adapter_id.retryConnection()` and examine the current state with `@adapter_id.isConnected()`.

16.7 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.

Note

For the inbound gateway, care must be taken to retain, or populate, the *ip_connnection_id* header because it is used to correlate the message to a connection. Messages that originate at the gateway will automatically have the header set. If the reply is constructed as a new message, you will need to set the header. The header value can be captured from the incoming message.

As with inbound adapters, inbound gateways normally use a type="server" connection factory, which listens for incoming connection requests. In some cases, it is desireable to establish the connection in reverse, whereby the inbound gateway connects to an external server and then waits for, and replies to, inbound messages on that connection.

This topology is supported by using *client-mode="true"* on the inbound gateway. In this case, the connection factory must be of type 'client' and must have *single-use* set to false.

Two additional attributes are used to support this mechanism: *retry-interval* specifies (in milliseconds) how often the framework will attempt to reconnect after a connection failure. *scheduler* is used to supply a `TaskScheduler` used to schedule the connection attempts, and to test that the connection is still active.

If the gateway is started, you may force the gateway to establish a connection by sending a <control-bus /> command: `@adapter_id.retryConnection()` and examine the current state with `@adapter_id.isConnected()`.

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.

```
<int-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.

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

A simple outbound TCP gateway.

16.8 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, by default. Since 3.0, headers can be transferred, using JSON, Java serialization, or with custom `Serializers` and `Deserializers`; see the section called “Transferring Headers” for more information. 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.

Another solution, introduced in Spring Integration 2.2, is to use a `CachingClientConnectionFactory`, which allows the use of a pool of shared connections.

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. Collaborating adapters can also be used (server-side or client-side) for totally asynchronous communication (rather than with request/reply semantics). 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.

Note

On the server side, care must be taken to populate the *ip_connnection_id* header because it is used to correlate the message to a connection. Messages that originate at the inbound adapter will automatically have the header set. If you wish to construct other messages to send, you will need to set the header. The header value can be captured from an incoming message.

On the client side, the application will have to provide its own correlation logic, if needed. 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[]`,

Note

Before the 2.2 release, when a *client* connection factory was used by collaborating channel adapters, the *so-timeout* attribute defaulted to the default reply timeout (10 seconds). This meant that if no data were received by the inbound adapter for this period of time, the socket was closed.

This default behavior was not appropriate in a truly asynchronous environment, so it now defaults to an infinite timeout. You can reinstate the previous default behavior by setting the *so-timeout* attribute on the client connection factory to 10000 milliseconds.

Transferring Headers

TCP is a streaming protocol; `Serializers` and `Deserializers` are used to demarcate messages within the stream. Prior to 3.0, only message payloads (String or byte[]) could be transferred over TCP. Beginning with 3.0, you can now transfer selected headers as well as the payload. It is important to understand, though, that "live" objects, such as the `replyChannel` header cannot be serialized.

Sending header information over TCP requires some additional configuration.

The first step is to provide the `ConnectionFactory` with a `MessageConvertingTcpMessageMapper` using the `mapper` attribute. This mapper delegates to any `MessageConverter` implementation to convert the message to/from some object that can be (de)serialized by the configured `serializer` and `deserializer`.

A `MapMessageConverter` is provided, which allows the specification of a list of headers that will be added to a `Map` object, along with the payload. The generated Map has two entries: `payload` and `headers`. The `headers` entry is itself a `Map` containing the selected headers.

The second step is to provide a (de)serializer that can convert between a `Map` and some wire format. This can be a custom `(de)Serializer`, which would typically be needed if the peer system is not a Spring Integration application.

A `MapJsonSerializer` is provided that will convert a Map to/from JSON. This uses a Spring Integration `JsonObjectMapper` to perform this function. You can provide a custom `JsonObjectMapper` if needed. By default, the serializer inserts a linefeed `0x0a` character between objects. See the JavaDocs for more information.

Note

At the time of writing, the `JsonObjectMapper` uses whichever version of `Jackson` is on the classpath.

You can also use standard Java serialization of the Map, using the `DefaultSerializer` and `DefaultDeserializer`.

The following example shows the configuration of a connection factory that transfers the `correlationId`, `sequenceNumber`, and `sequenceSize` headers using JSON.

```
<int-ip:tcp-connection-factory id="client"
    type="client"
    host="localhost"
    port="12345"
    mapper="mapper"
    serializer="jsonSerializer"
    deserializer="jsonSerializer"/>

<bean id="mapper"
      class="o.sf.integration.ip.tcp.connection.MessageConvertingTcpMessageMapper">
    <constructor-arg name="messageConverter">
        <bean class="o.sf.integration.support.converter.MapMessageConverter">
            <property name="headerNames">
                <list>
                    <value>correlationId</value>
                    <value>sequenceNumber</value>
                    <value>sequenceSize</value>
                </list>
            </property>
        </bean>
    </constructor-arg>
</bean>

<bean id="jsonSerializer" class="o.sf.integration.ip.tcp.serializer.MapJsonSerializer" />
```

A message sent with the above configuration, with payload 'foo' would appear on the wire like so:

```
{"headers":{"correlationId":"bar","sequenceSize":5,"sequenceNumber":1},"payload":"foo"}
```

16.9 A Note About NIO

Using NIO (see `using-nio` in Section 16.11, “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 arriving 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.

Alternatively, you may choose to insert a resequencer downstream of the inbound endpoint to return the messages to their proper sequence. Set *apply-sequence* to true on the connection factory, and messages arriving on a TCP connection will have *sequenceNumber* and *correlationId* headers set. The resequencer uses these headers to return the messages to their proper sequence.

Pool Size

The pool size attribute is no longer used; previously, it specified the size of the default thread pool when a task-executor was not specified. It was also used to set the connection backlog on server sockets. The first function is no longer needed (see below); the second function is replaced by the *backlog* attribute.

Previously, when using a fixed thread pool task executor (which was the default), with NIO, it was possible to get a deadlock and processing would stop. The problem occurred when a buffer was full, a thread reading from the socket was trying to add more data to the buffer, and there were no threads available to make space in the buffer. This only occurred with a very small pool size, but it could be possible under extreme conditions. Since 2.2, two changes have eliminated this problem. First, the default task executor is a cached thread pool executor. Second, deadlock detection logic has been added such that if thread starvation occurs, instead of deadlocking, an exception is thrown, thus releasing the deadlocked resources.

Note

Now that the default task executor is unbounded, it is possible that an out of memory condition might occur with high rates of incoming messages, if message processing takes extended time. If your application exhibits this type of behavior, you are advised to use a pooled task executor with an appropriate pool size.

16.10 SSL/TLS Support

Overview

Secure Sockets Layer/Transport Layer Security is supported. When using NIO, the JDK 5+ `SSLEngine` feature is used to handle handshaking after the connection is established. When not using NIO, standard `SSLSocketFactory` and `SSLServerSocketFactory` objects are used to create connections. A number of strategy interfaces are provided to allow significant customization; default implementations of these interfaces provide for the simplest way to get started with secure communications.

Getting Started

Regardless of whether NIO is being used, you need to configure the `ssl-context-support` attribute on the connection factory. This attribute references a <bean/> definition that describes the location and passwords for the required key stores.

SSL/TLS peers require two keystores each; a keystore containing private/public key pairs identifying the peer; a truststore, containing the public keys for peers that are trusted. See the documentation for the `keytool` utility provided with the JDK. The essential steps are

1. Create a new key pair and store in a keystore.

2. Export the public key.

3. Import the public key into the peer's truststore.

Repeat for the other peer.

Note

It is common in test cases to use the same key stores on both peers, but this should be avoided for production.

After establishing the key stores, the next step is to indicate their locations to the `TcpSSLContextSupport` bean, and provide a reference to that bean to the connection factory.

```
<bean id="sslContextSupport"
    class="o.sf.integration.ip.tcp.connection.support.DefaultTcpSSLContextSupport">
    <constructor-arg value="client.ks"/>
    <constructor-arg value="client.truststore.ks"/>
    <constructor-arg value="secret"/>
    <constructor-arg value="secret"/>
</bean>

<ip:tcp-connection-factory id="clientFactory"
    type="client"
    host="localhost"
    port="1234"
    ssl-context-support="sslContextSupport"
```

The `DefaultTcpSSLContextSupport` class also has an optional 'protocol' property, which can be 'SSL' or 'TLS' (default).

The keystore file names (first two constructor arguments) use the Spring `Resource` abstraction; by default the files will be located on the classpath, but this can be overridden by using the `file:` prefix, to find the files on the filesystem instead.

Advanced Techniques

In many cases, the configuration described above is all that is needed to enable secure communication over TCP/IP. However, a number of strategy interfaces are provided to allow customization and modification of socket factories and sockets.

- `TcpSSLContextSupport`
- `TcpSocketFactorySupport`
- `TcpSocketSupport`

```
public interface TcpSSLContextSupport {

 SSLContext getSSLContext() throws Exception;

}
```

Implementations of this interface are responsible for creating an SSLContext. The sole implementation provided by the framework is the `DefaultTcpSSLContextSupport` described above. If you require different behavior, implement this interface and provide the connection factory with a reference to a bean of your class' implementation.

```
public interface TcpSocketFactorySupport {

    ServerSocketFactory getServerSocketFactory();

    SocketFactory getSocketFactory();

}
```

Implementations of this interface are responsible for obtaining references to `ServerSocketFactory` and `SocketFactory`. Two implementations are provided; the first is `DefaultTcpNetSocketFactorySupport` for non-SSL sockets (when no 'ssl-context-support' attribute is defined); this simply uses the JDK's default factories. The second implementation is

`DefaultTcpNetSSLSocketFactorySupport`; this is used, by default, when an 'ssl-context-support' attribute is defined; it uses the `SSLContext` created by that bean to create the socket factories.

Note

This interface only applies if `using-nio` is "false"; socket factories are not used by NIO.

```
public interface TcpSocketSupport {

    void postProcessServerSocket(ServerSocket serverSocket);

    void postProcessSocket(Socket socket);

}
```

Implementations of this interface can modify sockets after they are created, and after all configured attributes have been applied, but before the sockets are used. This applies whether or not NIO is being used. For example, you could use an implementation of this interface to modify the supported cipher suites on an SSL socket, or you could add a listener that gets notified after SSL handshaking is complete. The sole implementation provided by the framework is the `DefaultTcpSocketSupport` which does not modify the sockets in any way

To supply your own implementation of `TcpSocketFactorySupport` or `TcpSocketSupport`, provide the connection factory with references to beans of your custom type using the `socket-factory-support` and `socket-support` attributes, respectively.

16.11 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.
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 connection uses NIO. Refer to the java.nio package for more information. See Section 16.9, “A Note About NIO”. Default false.
using-direct-buffers	Y	N	true, false	When using NIO, whether or not the connection uses direct buffers. Refer to `java.nio.ByteBuffer`

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
				documentation for more information. Must be false if using-nio is false.
apply-sequence	Y	Y	true, false	When using NIO, it may be necessary to resequence messages. When this attribute is set to true, *correlationId* and *sequenceNumber* headers will be added to received messages. See Section 16.9, “A Note About NIO”. Default false.
so-timeout	Y	Y		Defaults to 0 (infinity), except for server connection factories with single-use="true". In that case, it defaults to the default reply timeout (10 seconds).
so-send-buffer-size	Y	Y		See `java.net.Socket.setSendBufferSize()`.
so-receive-buffer-size	Y	Y		See `java.net.Socket.setReceiveBufferSize()`.
so-keep-alive	Y	Y	true, false	See `java.net.Socket.setKeepAlive()`.
so-linger	Y	Y		Sets linger to true with supplied value. See `java.net.Socket.setSoLinger()`.
so-tcp-no-delay	Y	Y	true, false	See `java.net.Socket.setTcpNoDelay()`.
so-traffic-class	Y	Y		See `java.net.Socket.setTrafficClass()`.
local-address	N	Y		On a multi-homed system, specifies an IP address for the interface to which the socket will be bound.
task-executor	Y	Y		Specifies a specific Executor to be used for socket handling. If not supplied, an internal cached thread executor will be used. Needed on some platforms that require the use of specific task executors such as a WorkManagerTaskExecutor.
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	N	N		This attribute is no longer used. For backward compatibility, it sets the backlog but users should use backlog to

Attribute Name	Client?	Server?	Allowed Values	Attribute Description
				specify the connection backlog in server factories
backlog	N	Y		Sets the connection backlog for server factories.
lookup-host	Y	Y	true, false	Specifies whether reverse lookups are done on IP addresses to convert to host names for use in message headers. If false, the IP address is used instead. Defaults to true.
interceptor-factory-chain	Y	Y		See Section 16.4, “TCP Connection Interceptors”
ssl-context-support	Y	Y		See Section 16.10, “SSL/TLS Support”
socket-factory-support	Y	Y		See Section 16.10, “SSL/TLS Support”
socket-support	Y	Y		See Section 16.10, “SSL/TLS Support”

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 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.

Attribute Name	Allowed Values	Attribute Description
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.
lookup-host	true, false	Specifies whether reverse lookups are done on IP addresses to convert to host names for use in message headers. If false, the IP address is used instead. Defaults to true.

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.
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

Attribute Name	Allowed Values	Attribute Description
		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 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

Attribute Name	Allowed Values	Attribute Description
		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.
client-mode	true, false	When true, the inbound adapter will act as a client, with respect to establishing the connection and then receive incoming messages on that connection. Default = false. Also see *retry-interval* and *scheduler*. The connection factory must be of type 'client' and have *single-use* set to false.
retry-interval		When in *client-mode*, specifies the number of milliseconds to wait between connection attempts, or after a connection failure. Default 60,000 (60 seconds).
scheduler	true, false	Specifies a `TaskScheduler` to use for managing the *client-mode* connection. Defaults to a `ThreadPoolTaskScheduler` with a pool size of `.

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.
client-mode	true, false	When true, the outbound adapter will attempt to establish the connection as soon as it is started. When false, the connection is established when the first message is sent. Default = false. Also see *retry-interval* and *scheduler*. The connection factory must be of type 'client' and have *single-use* set to false.
retry-interval		When in *client-mode*, specifies the number of milliseconds to wait between connection attempts, or after a connection failure. Default 60,000 (60 seconds).
scheduler	true, false	Specifies a `TaskScheduler` to use for managing the *client-mode* connection. Defaults to a `ThreadPoolTaskScheduler` with a pool size of `.

Table 16.6. TCP Inbound Gateway Attributes

Attribute Name	Allowed Values	Attribute Description
connection-factory		The connection factory must be of type server.

Attribute Name	Allowed Values	Attribute Description
request-channel		The channel to which incoming messages will be sent.
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. Default 1000 (1 second).
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.
client-mode	true, false	When true, the inbound gateway will act as a client, with respect to establishing the connection and then receive (and reply to) incoming messages on that connection. Default = false. Also see *retry-interval* and *scheduler*. The connection factory must be of type 'client' and have *single-use* set to false.
retry-interval		When in *client-mode*, specifies the number of milliseconds to wait between connection attempts, or after a connection failure. Default 60,000 (60 seconds).
scheduler	true, false	Specifies a `TaskScheduler` to use for managing the *client-mode* connection. Defaults to a `ThreadPoolTaskScheduler` with a pool size of `.

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.
remote-timeout		The time in milliseconds for which the gateway will wait for a reply from the remote system. Default: Same value as reply-timeout, if specified, or 10000 (10 seconds) otherwise.
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.
reply-timeout		The time in milliseconds for which the gateway will wait when sending the reply to the reply-channel. Only applies if the reply-channel might block, such as a bounded QueueChannel that is currently full.

17. JDBC Support

Spring Integration provides Channel Adapters for receiving and sending messages via database queries. Through those adapters Spring Integration supports not only plain JDBC SQL Queries, but also Stored Procedure and Stored Function calls.

The following JDBC components are available by default:

- *Inbound Channel Adapter*
- *Outbound Channel Adapter*
- *Outbound Gateway*
- *Stored Procedure Inbound Channel Adapter*
- *Stored Procedure Outbound Channel Adapter*
- *Stored Procedure Outbound Gateway*

Furthermore, the Spring Integration JDBC Module also provides a *JDBC Message Store*

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 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.

```
<int-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:

```
<int-jdbc:inbound-channel-adapter query="..."
        channel="target" data-source="dataSource" update="...">
    <int:poller fixed-rate="1000">
        <int:transactional/>
    </int:poller>
</int-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.

Max-rows-per-poll versus Max-messages-per-poll

The *JDBC Inbound Channel Adapter* defines an attribute *max-rows-per-poll*. When you specify the adapter's *Poller*, you can also define a property called *max-messages-per-poll*. While these two attributes look similar, their meaning is quite different.

max-messages-per-poll specifies the number of times the query is executed per polling interval, whereas *max-rows-per-poll* specifies the number of rows returned for each execution.

Under normal circumstances, you would likely not want to set the Poller's *max-messages-per-poll* property when using the *JDBC Inbound Channel Adapter*. Its default value is *1*, which means that the *JDBC Inbound Channel Adapter's* *receive()* method is executed exactly once for each poll interval.

Setting the *max-messages-per-poll* attribute to a larger value means that the query is executed that many times back to back. For more information regarding the *max-messages-per-poll* attribute, please see the section called “Configuring An Inbound Channel Adapter”.

In contrast, the *max-rows-per-poll* attribute, if greater than *0*, specifies the maximum number of rows that will be used from the query result set, per execution of the *receive()* method. If the attribute is set to *0*, then all rows will be included in the resulting message. If not explicitly set, the attribute defaults to *0*.

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:

```
<int-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 labelled *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 each incoming message to a 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.

Passing Parameters using SpEL Expressions

A common requirement for most JDBC Channel Adapters is to pass parameters as part of Sql queries or Stored Procedures/Functions. As mentioned above, these parameters are by default bean property expressions, not SpEL expressions. However, if you need to pass SpEL expression as parameters, you must inject a `SqlParameterSourceFactory` explicitly.

The following example uses a `ExpressionEvaluatingSqlParameterSourceFactory` to achieve that requirement.

```
<jdbc:outbound-channel-adapter data-source="dataSource" channel="input"
    query="insert into MESSAGES (MESSAGE_ID,PAYLOAD,CREATED_DATE)     \
    values (:id, :payload, :createdDate)"
    sql-parameter-source-factory="spelSource"/>

<bean id="spelSource"
      class="o.s.integration.jdbc.ExpressionEvaluatingSqlParameterSourceFactory">
    <property name="parameterExpressions">
        <map>
            <entry key="id"          value="headers['id'].toString()"/>
            <entry key="createdDate" value="new java.util.Date()"/>
            <entry key="payload"     value="payload"/>
        </map>
    </property>
</bean>
```

For further information, please also see the section called “Defining Parameter Sources”

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:

```
<int-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:

```
<int-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:

```
<int-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"/>
```

Since *Spring Integration 2.2* the update SQL query is no longer mandatory. You can now solely provide a select query, using either the *query attribute* or the *query sub-element*. This is extremely useful if you need to actively retrieve data using e.g. a generic Gateway or a Payload Enricher. The reply message is then generated from the result, like the inbound adapter, and passed to the reply channel.

```
<int-jdbc:outbound-gateway
    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 JDBC Message Store

Spring Integration provides 2 JDBC specifc Message Store implementations. The first one, is the `JdbcMessageStore` which is suitable to be used in conjunction with *Aggregators* and the *Claimcheck*

pattern. While it can be used for backing *Message Channels* as well, you may want to consider using the `JdbcChannelMessageStore` implementation instead, as it provides a more targeted and scalable implementation.

The Generic JDBC 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:

```
<int-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:

```
<int-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_".

Note

If you plan on using **MySQL**, please use MySQL version *5.6.4* or higher, if possible. Prior versions do not support *fractional seconds* for temporal data types. Because of that, messages may not arrive in the precise FIFO order when polling from such a MySQL Message Store.

Therefore, starting with *Spring Integration 3.0*, we provide an additional set of DDL scripts for MySQL version *5.6.4* or higher:

- schema-drop-mysql-5_6_4.sql
- schema-mysql-5_6_4.sql

For more information, please see:

http://dev.mysql.com/doc/refman/5.6/en/fractional-seconds.html

Also important, please ensure that you use an up-to-date version of the JDBC driver for MySQL (Connector/J), e.g. version *5.1.24* or higher.

Backing Message Channels

If you intent backing *Message Channels* using JDBC, it is recommended to use the provided `JdbcChannelMessageStore` implementation instead. It can only be used in conjuntion with *Message Channels*.

Note

The provided `JdbcChannelMessageStore` implementation is available since *Spring Integration 2.2.*.

Supported Database

The `JdbcChannelMessageStore` uses database specific SQL queries to retrieve messages from the database. Therefore, users must set the `ChannelMessageStoreQueryProvider` property on the `JdbcChannelMessageStore`. This `channelMessageStoreQueryProvider` provides the SQL queries and Spring Integration provides support for the following relational databases:

- PostgreSQL
- HSQLDB
- MySQL
- Oracle
- Derby

If your database is not listed, you can easily extend the `AbstractChannelMessageStoreQueryProvider` class and provide your own custom queries.

Important

Generally it is not recommened to use a relational database for the purpose of queuing. Instead, if possible, consider using either JMS or AMQP, for which message store implementation are provided as well. For further reference please see the following resources:

- 5 subtle ways you're using MySQL as a queue, and why it'll bite you
- The Database As Queue Anti-Pattern

Concurrent Polling

When polling a *Message Channel*, you have the option to configure the associated `Poller` with a `TaskExecutor` reference.

Keep in mind, though, that if you use a JDBC backed *Message Channel* and you are planning on polling the channel and consequently the message store transactionally with multiple threads, you should ensure that you use a relational database that supports Multiversion Concurrency Control (MVCC). Otherwise, locking may be an issue and the performance, when using multiple threads, may not materialize as expected. For example Apache Derby is problematic in that regard.

```
…
<bean id="queryProvider"
    class="o.s.i.jdbc.store.channel.PostgresChannelMessageStoreQueryProvider"/>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="@store.removeFromIdCache(headers.id.toString())" />
    <int:after-rollback expression="@store.removeFromIdCache(headers.id.toString())"/>
</int:transaction-synchronization-factory>

<task:executor id="pool" pool-size="10"
    queue-capacity="10" rejection-policy="CALLER_RUNS" />

<bean id="store" class="o.s.i.jdbc.store.JdbcChannelMessageStore">
    <property name="dataSource" ref="dataSource"/>
    <property name="channelMessageStoreQueryProvider" ref="queryProvider"/>
    <property name="region" value="TX_TIMEOUT"/>
    <property name="usingIdCache" value="true"/>
</bean>

<int:channel id="inputChannel">
    <int:queue message-store="store"/>
</int:channel>

<int:bridge input-channel="inputChannel" output-channel="outputChannel">
    <int:poller fixed-delay="500" receive-timeout="500"
        max-messages-per-poll="1" task-executor="pool">
        <int:transactional propagation="REQUIRED" synchronization-factory="syncFactory"
        isolation="READ_COMMITTED" transaction-manager="transactionManager" />
    </int:poller>
</int:bridge>

<int:channel id="outputChannel" />
…
```

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` and in the `org.springframework.integration.jdbc.store.channel` package: there is a create and a drop script example for a range of common database platforms. A common way to use these scripts is to reference them in a Spring JDBC data source initializer. Note that the scripts are provided as samples or specifications of the the required table and column names. You may find that you need to enhance them for production use (e.g. with index declarations).

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.

17.5 Stored Procedures

In certain situations plain JDBC support is not sufficient. Maybe you deal with legacy relational database schemas or you have complex data processing needs, but ultimately you have to use Stored Procedures or Stored Functions. Since Spring Integration 2.1, we provide three components in order to execute Stored Procedures or Stored Functions:

- Stored Procedures Inbound Channel Adapter
- Stored Procedures Outbound Channel Adapter
- Stored Procedures Outbound Gateway

Supported Databases

In order to enable calls to *Stored Procedures* and *Stored Functions*, the Stored Procedure components use the `org.springframework.jdbc.core.simple.SimpleJdbcCall` class. Consequently, the following databases are fully supported for executing Stored Procedures:

- Apache Derby
- DB2
- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL
- Sybase

If you want to exute Stored Functions instead, the following databases are fully supported:

- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL

Note

Even though your particular database may not be fully supported, chances are, that you can use the Stored Procedure Spring Integration components quite successfully anyway, provided your RDBMS supports Stored Procedures or Functions.

As a matter of fact, some of the provided integration tests use the H2 database. Nevertheless, it is very important to thoroughly test those usage scenarios.

Configuration

The Stored Procedure components provide full XML Namespace support and configuring the components is similar as for the general purpose JDBC components discussed earlier.

Common Configuration Attributes

Certain configuration parameters are shared among all Stored Procedure components and are described below:

auto-startup

Lifecycle attribute signaling if this component should be started during Application Context startup. Defaults to `true`. *Optional*.

data-source

Reference to a `javax.sql.DataSource`, which is used to access the database. *Required*.

id

Identifies the underlying Spring bean definition, which is an instance of either `EventDrivenConsumer` or `PollingConsumer`, depending on whether the Outbound Channel Adapter's `channel` attribute references a `SubscribableChannel` or a `PollableChannel`. *Optional*.

ignore-column-meta-data

For fully supported databases, the underlying `SimpleJdbcCall` class can automatically retrieve the parameter information for the to be invoked Stored Procedure or Function from the JDBC Meta-data.

However, if the used database does not support meta data lookups or if you like to provide customized parameter definitions, this flag can be set to `true`. It defaults to `false`. *Optional*.

is-function

If `true`, a SQL Function is called. In that case the `stored-procedure-name` or `stored-procedure-name-expression` attributes define the name of the called function. Defaults to `false`. *Optional*.

stored-procedure-name

The attribute specifies the name of the stored procedure. If the `is-function` attribute is set to `true`, this attribute specifies the function name instead. Either this property or *stored-procedure-name-expression* must be specified.

stored-procedure-name-expression

This attribute specifies the name of the stored procedure using a SpEL expression. Using SpEL you have access to the full message (if available), including its headers and payload. You can use this attribute to invoke different Stored Procedures at runtime. For example, you can provide Stored Procedure names that you would like to execute as a Message Header. The expression must resolve to a String.

If the `is-function` attribute is set to `true`, this attribute specifies a Stored Function. Either this property or *stored-procedure-name* must be specified.

jdbc-call-operations-cache-size

Defines the maximum number of cached `SimpleJdbcCallOperations` instances. Basically, for each Stored Procedure Name a new `SimpleJdbcCallOperations` instance is created that in return is being cached.

Note

The *stored-procedure-name-expression* attribute and the *jdbc-call-operations-cache-size* were added with Spring Integration 2.2.

The default cache size is *10*. A value of *0* disables caching. Negative values are not permitted.

If you enable JMX, statistical information about the *jdbc-call-operations-cache* is exposed as MBean. Please see the section called “MBean Exporter” for more information.

sql-parameter-source-factory (Not available for the Stored Procedure Inbound Channel Adapter.)

Reference to a `SqlParameterSourceFactory`. By default bean properties of the passed in `Message` payload will be used as a source for the Stored Procedure's input parameters using a `BeanPropertySqlParameterSourceFactory`.

This may be sufficient for basic use cases. For more sophisticated options, consider passing in one or more `ProcedureParameter`. Please also refer to the section called “Defining Parameter Sources”. *Optional*.

use-payload-as-parameter-source (Not available for the Stored Procedure Inbound Channel Adapter.)

If set to `true`, the payload of the Message will be used as a source for providing parameters. If false, however, the entire Message will be available as a source for parameters.

If no Procedure Parameters are passed in, this property will default to `true`. This means that using a default `BeanPropertySqlParameterSourceFactory` the bean properties of the payload will be used as a source for parameter values for the to-be-executed Stored Procedure or Stored Function.

However, if Procedure Parameters are passed in, then this property will by default evaluate to `false`. `ProcedureParameter` allow for SpEL Expressions to be provided and therefore it is highly beneficial to have access to the entire Message. The property is set on the underlying `StoredProcExecutor`. *Optional*.

Common Configuration Sub-Elements

The Stored Procedure components share a common set of sub-elements to define and pass parameters to Stored Procedures or Functions. The following elements are available:

- parameter
- returning-resultset
- sql-parameter-definition
- poller

parameter

Provides a mechanism to provide Stored Procedure parameters. Parameters can be either static or provided using a SpEL Expressions. *Optional*.

```
<int-jdbc:parameter name=""    ❶
                    type=""    ❷
                    value=""/> ❸

<int-jdbc:parameter name=""
                    expression=""/>❹
```

❶ The name of the parameter to be passed into the Stored Procedure or Stored Function. *Required*.

❷ This attribute specifies the type of the value. If nothing is provided this attribute will default to `java.lang.String`. This attribute is only used when the `value` attribute is used. *Optional*.

❸ The value of the parameter. You have to provider either this attribute or the `expression` attribute must be provided instead. *Optional*.

❹ Instead of the `value` attribute, you can also specify a SpEL expression for passing the value of the parameter. If you specify the `expression` the `value` attribute is not allowed. *Optional*.

returning-resultset

Stored Procedures may return multiple resultsets. By setting one or more `returning-resultset` elements, you can specify `RowMappers` in order to convert each returned `ResultSet` to meaningful objects. *Optional*.

```
<int-jdbc:returning-resultset name="" row-mapper="" />
```

sql-parameter-definition

If you are using a database that is fully supported, you typically don't have to specify the Stored Procedure parameter definitions. Instead, those parameters can be automatically derived from the JDBC Meta-data. However, if you are using databases that are not fully supported, you must set those parameters explicitly using the `sql-parameter-definition` sub-element.

You can also choose to turn off any processing of parameter meta data information obtained via JDBC using the `ignore-column-meta-data` attribute.

```
<int-jdbc:sql-parameter-definition
                                name=""                          ❶
                                direction="IN"                   ❷
                                type="STRING"                    ❸
                                scale="5"                        ❹
                                type-name="FOO_STRUCT"           ❺
                                return-type="fooSqlReturnType"/> ❻
```

❶ Specifies the name of the SQL parameter. *Required*.

❷ Specifies the direction of the SQL parameter definition. Defaults to `IN`. Valid values are: `IN`, `OUT` and `INOUT`. If your procedure is returning ResultSets, please use the `returning-resultset` element. *Optional*.

❸ The SQL type used for this SQL parameter definition. Will translate into the integer value as defined by java.sql.Types. Alternatively you can provide the integer value as well. If this attribute is not explicitly set, then it will default to 'VARCHAR'. *Optional*.

❹ The scale of the SQL parameter. Only used for numeric and decimal parameters. *Optional*.

❺ The typeName for types that are user-named like: STRUCT, DISTINCT, JAVA_OBJECT, named array types. This attribute is mutually exclusive with the *scale* attribute. *Optional*.

❻ The reference to a custom value handler for complex types. An implementation of SqlReturnType. This attribute is mutually exclusive with the *scale* attribute and is applicable for OUT(INOUT)-parameters only. *Optional*.

poller

Allows you to configure a Message Poller if this endpoint is a `PollingConsumer`. *Optional*.

Defining Parameter Sources

Parameter Sources govern the techniques of retrieving and mapping the Spring Integration Message properties to the relevant Stored Procedure input parameters. The Stored Procedure components follow certain rules.

By default bean properties of the passed in `Message` payload will be used as a source for the Stored Procedure's input parameters. In that case a `BeanPropertySqlParameterSourceFactory` will be used. This may be sufficient for basic use cases. The following example illustrates that default behavior.

Important

Please be aware that for the "automatic" lookup of bean properties using the `BeanPropertySqlParameterSourceFactory` to work, your bean properties must be defined in lower case. This is due to the fact that in `org.springframework.jdbc.core.metadata.CallMetaDataContext` (method matchInParameterValuesWithCallParameters()), the retrieved Stored Procedure parameter declarations are converted to lower case. As a result, if you have camel-case bean properties such as "lastName", the lookup will fail. In that case, please provide an explicit `ProcedureParameter`.

Let's assume we have a payload that consists of a simple bean with the following three properties: *id*, *name* and *description*. Furthermore, we have a simplistic Stored Procedure called *INSERT_COFFEE* that accepts three input parameters: *id*, *name* and *description*. We also use a fully supported database. In that case the following configuration for a Stored Procedure Oubound Adapter will be sufficient:

```
<int-jdbc:stored-proc-outbound-channel-adapter data-source="dataSource"
    channel="insertCoffeeProcedureRequestChannel"
    stored-procedure-name="INSERT_COFFEE"/>
```

For more sophisticated options consider passing in one or more `ProcedureParameter`.

If you do provide `ProcedureParameter` explicitly, then as default an `ExpressionEvaluatingSqlParameterSourceFactory` will be used for parameter processing in order to enable the full power of SpEL expressions.

Furthermore, if you need even more control over how parameters are retrieved, consider passing in a custom implementation of a `SqlParameterSourceFactory` using the `sql-parameter-source-factory` attribute.

Stored Procedure Inbound Channel Adapter

```
<int-jdbc:stored-proc-inbound-channel-adapter
                                    channel=""                                  ❶
                                    stored-procedure-name=""
                                    data-source=""
                                    auto-startup="true"
                                    id=""
                                    ignore-column-meta-data="false"
                                    is-function="false"
                                    max-rows-per-poll=""                        ❷
                                    skip-undeclared-results=""                  ❸
                                    return-value-required="false"               ❹
    <int:poller/>
    <int-jdbc:sql-parameter-definition name="" direction="IN"
                                               type="STRING"
                                               scale=""/>
    <int-jdbc:parameter name="" type="" value=""/>
    <int-jdbc:parameter name="" expression=""/>
    <int-jdbc:returning-resultset name="" row-mapper="" />
</int-jdbc:stored-proc-inbound-channel-adapter>
```

❶ Channel to which polled messages will be sent. If the stored procedure or function does not return any data, the payload of the Message will be Null. *Required*.

❷ Limits the number of rows extracted per query. Otherwise all rows are extracted into the outgoing message. *Optional*.

❸ If this attribute is set to `true`, then all results from a stored procedure call that don't have a corresponding `SqlOutParameter` declaration will be bypassed.

E.g. Stored Procedures may return an update count value, even though your Stored Procedure only declared a single result parameter. The exact behavior depends on the used database. The value is set on the underlying `JdbcTemplate`.

Few developers will probably ever want to process update counts, thus the value defaults to `true`. *Optional*.

❹ Indicates whether this procedure's return value should be included. Since *Spring Integration 3.0. Optional*.

Note

When you declare a Poller, you may notice the Poller's *max-messages-per-poll* attribute. For information about how it relates to the *max-rows-per-poll* attribute of the *Stored Procedure Inbound Channel Adapter*, please see the section called “Max-rows-per-poll versus Max-messages-per-poll” for a thourough discussion. The meaning of the attributes is the same as for the *JDBC Inbound Channel Adapter*.

Stored Procedure Outbound Channel Adapter

```
<int-jdbc:stored-proc-outbound-channel-adapter channel=""                        ❶
                                               stored-procedure-name=""
                                               data-source=""
                                               auto-startup="true"
                                               id=""
                                               ignore-column-meta-data="false"
                                               order=""                          ❷
                                               sql-parameter-source-factory=""
                                               use-payload-as-parameter-source="">
    <int:poller fixed-rate=""/>
    <int-jdbc:sql-parameter-definition name=""/>
    <int-jdbc:parameter name=""/>

</int-jdbc:stored-proc-outbound-channel-adapter>
```

❶ The receiving Message Channel of this endpoint. *Required*.

❷ Specifies the order for invocation when this endpoint is connected as a subscriber to a channel. This is particularly relevant when that channel is using a *failover* dispatching strategy. It has no effect when this endpoint itself is a Polling Consumer for a channel with a queue. *Optional*.

Stored Procedure Outbound Gateway

```
<int-jdbc:stored-proc-outbound-gateway request-channel=""                        ❶
                                       stored-procedure-name=""
                                       data-source=""
                                  auto-startup="true"
                                  id=""
                                  ignore-column-meta-data="false"
                                  is-function="false"
                                  order=""
                                  reply-channel=""                               ❷
                                  reply-timeout=""                               ❸
                                  return-value-required="false"                  ❹
                                  skip-undeclared-results=""                     ❺
                                  sql-parameter-source-factory=""
                                  use-payload-as-parameter-source="">
<int-jdbc:sql-parameter-definition name="" direction="IN"
                                  type=""
                                  scale="10"/>
<int-jdbc:sql-parameter-definition name=""/>
<int-jdbc:parameter name="" type="" value=""/>
<int-jdbc:parameter name="" expression=""/>
<int-jdbc:returning-resultset name="" row-mapper="" />
```

❶ The receiving Message Channel of this endpoint. *Required*.

❷ Message Channel to which replies should be sent, after receiving the database response. *Optional*.

❸ Allows you to specify how long this gateway will wait for the reply message to be sent successfully before throwing an exception. Keep in mind that when sending to a `DirectChannel`, the invocation will occur in the sender's thread so the failing of the send operation may be caused by other components further downstream. By default the Gateway will wait indefinitely. The value is specified in milliseconds. *Optional*.

❹ Indicates whether this procedure's return value should be included. *Optional*.

❺ If the `skip-undeclared-results` attribute is set to `true`, then all results from a stored procedure call that don't have a corresponding `SqlOutParameter` declaration will be bypassed.

E.g. Stored Procedures may return an update count value, even though your Stored Procedure only declared a single result parameter. The exact behavior depends on the used database. The value is set on the underlying `JdbcTemplate`.

Few developers will probably ever want to process update counts, thus the value defaults to `true`. *Optional*.

Examples

In the following two examples we call Apache Derby Stored Procedures. The first procedure will call a Stored Procedure that returns a `ResultSet`, and using a `RowMapper` the data is converted into a domain object, which then becomes the Spring Integration message payload.

In the second sample we call a Stored Procedure that uses Output Parameters instead, in order to return data.

Note

Please have a look at the *Spring Integration Samples* project, located at https://github.com/SpringSource/spring-integration-samples

The project contains the Apache Derby example referenced here, as well as instruction on how to run it. The *Spring Integration Samples* project also provides an example using Oracle Stored Procedures.

In the first example, we call a Stored Procedure named *FIND_ALL_COFFEE_BEVERAGES* that does not define any input parameters but which returns a `ResultSet`.

In Apache Derby, Stored Procedures are implemented using Java. Here is the method signature followed by the corresponding Sql:

```
public static void findAllCoffeeBeverages(ResultSet[] coffeeBeverages)
            throws SQLException {
    ...
}
```

```
CREATE PROCEDURE FIND_ALL_COFFEE_BEVERAGES() \
PARAMETER STYLE JAVA LANGUAGE JAVA MODIFIES SQL DATA DYNAMIC RESULT SETS 1 \
EXTERNAL NAME
 'org.springframework.integration.jdbc.storedproc.derby.DerbyStoredProcedures.findAllCoffeeBeverages';
```

In Spring Integration, you can now call this Stored Procedure using e.g. a `stored-proc-outbound-gateway`

```
<int-jdbc:stored-proc-outbound-gateway id="outbound-gateway-storedproc-find-all"
                                       data-source="dataSource"
                                       request-channel="findAllProcedureRequestChannel"
                                       expect-single-result="true"
                                       stored-procedure-name="FIND_ALL_COFFEE_BEVERAGES">
<int-jdbc:returning-resultset name="coffeeBeverages"
    row-mapper="org.springframework.integration.support.CoffeBeverageMapper"/>
</int-jdbc:stored-proc-outbound-gateway>
```

In the second example, we call a Stored Procedure named *FIND_COFFEE* that has one input parameter. Instead of returning a ResultSet, an output parameter is used:

```
public static void findCoffee(int coffeeId, String[] coffeeDescription)
            throws SQLException {
    ...
}
```

```
CREATE PROCEDURE FIND_COFFEE(IN ID INTEGER, OUT COFFEE_DESCRIPTION VARCHAR(200)) \
PARAMETER STYLE JAVA LANGUAGE JAVA EXTERNAL NAME \
'org.springframework.integration.jdbc.storedproc.derby.DerbyStoredProcedures.findCoffee';
```

In Spring Integration, you can now call this Stored Procedure using e.g. a `stored-proc-outbound-gateway`

```
<int-jdbc:stored-proc-outbound-gateway id="outbound-gateway-storedproc-find-coffee"
                                       data-source="dataSource"
                                       request-channel="findCoffeeProcedureRequestChannel"
                                       skip-undeclared-results="true"
                                       stored-procedure-name="FIND_COFFEE"
                                       expect-single-result="true">
    <int-jdbc:parameter name="ID" expression="payload" />
</int-jdbc:stored-proc-outbound-gateway>
```

18. JPA Support

Spring Integration's JPA (Java Persistence API) module provides components for performing various database operations using JPA. The following components are provided:

- *Inbound Channel Adapter*
- *Outbound Channel Adapter*
- *Updating Outbound Gateway*
- *Retrieving Outbound Gateway*

These components can be used to perform *select*, *create*, *update* and *delete* operations on the targeted databases by sending/receiving messages to them.

The JPA *Inbound Channel Adapter* lets you poll and retrieve (select) data from the database using JPA whereas the JPA *Outbound Channel Adapter* lets you create, update and delete entities.

Outbound Gateways for JPA can be used to persist entities to the database, yet allowing you to continue with the flow and execute further components downstream. Similarly, you can use an Outbound Gateway to retrieve entities from the database.

For example, you may use the Outbound Gateway, which receives a Message with a user Id as payload on its request channel, to query the database and retrieve the User entity and pass it downstream for further processing.

Recognizing these semantic differences, Spring Integration provides 2 separate JPA Outbound Gateways:

- *Retrieving Outbound Gateway*
- *Updating Outbound Gateway*

Functionality

All JPA components perform their respective JPA operations by using either one of the following:

- *Entity classes*
- *Java Persistence Query Language (JPQL) for update, select and delete (inserts are not supported by JPQL)*
- *Native Query*
- *Named Query*

In the following sections we will describe each of these components in more detail.

18.1 Supported Persistence Providers

The Spring Integration JPA support has been tested using the following persistence providers:

- *Hibernate*
- *OpenJPA*
- *EclipseLink*

When using a persistence provider, please ensure that the provider is compatible with JPA 2.0.

18.2 Java Implementation

Each of the provided components will use the `o.s.i.jpa.core.JpaExecutor` class which in turn will use an implementation of the `o.s.i.jpa.core.JpaOperations` interface. `JpaOperations` operates like a typical Data Access Object (DAO) and provides methods such as *find*, *persist*, *executeUpdate* etc. For most use cases the provided default implementation `o.s.i.jpa.core.DefaultJpaOperations` should be sufficient. Nevertheless, you have the option to optionally specify your own implementation in case you require custom behavior.

For initializing a `JpaExecutor` you have to use one of 3 available constructors that accept one of:

- *EntityManagerFactory*
- *EntityManager* or
- *JpaOperations*

Note

The XML Namespace Support described further below is also very flexible and provides configuration attributes for each JPA component to pass in an *EntityManagerFactory*, *EntityManager* or *JpaOperations* reference.

Java Configuration Example

The following example of a JPA *Retrieving Outbound Gateway* is configured purely through Java. In typical usage scenarios you will most likely prefer the XML Namespace Support described further below. However, the example illustrates how the classes are wired up. Understanding the inner workings can also be very helpful for debugging or customizing the individual JPA components.

First, we instantiate a `JpaExecutor` using an `EntityManager` as constructor argument. The `JpaExecutor` is then in return used as constructor argument for the `o.s.i.jpa.outbound.JpaOutboundGateway` and the `JpaOutboundGateway` will be passed as constructor argument into the `EventDrivenConsumer`.

```
<bean id="jpaExecutor" class="o.s.i.jpa.core.JpaExecutor">
    <constructor-arg name="entityManager" ref="entityManager"/>
    <property name="entityClass"        value="o.s.i.jpa.test.entity.StudentDomain"/>
    <property name="jpaQuery"           value="select s from Student s where s.id = :id"/>
    <property name="expectSingleResult" value="true"/>
    <property name="jpaParameters" >
        <util:list>
            <bean class="org.springframework.integration.jpa.support.JpaParameter">
                <property name="name"       value="id"/>
                <property name="expression" value="payload"/>
            </bean>
        </util:list>
    </property>
</bean>

<bean id="jpaOutboundGateway" class="o.s.i.jpa.outbound.JpaOutboundGateway">
    <constructor-arg ref="jpaExecutor"/>
    <property        name="gatewayType"   value="RETRIEVING"/>
    <property        name="outputChannel" ref="studentReplyChannel"/>
</bean>

<bean id="getStudentEndpoint"
      class="org.springframework.integration.endpoint.EventDrivenConsumer">
    <constructor-arg name="inputChannel" ref="getStudentChannel"/>
    <constructor-arg name="handler"      ref="jpaOutboundGateway"/>
</bean>
```

Note

For more examples of constructing JPA components purely through Java, see the JUnit test-cases for the JPA Adapters.

18.3 Namespace Support

When using XML namespace support, the underlying parser classes will instantiate the relevant Java classes for you. Thus, you typically don't have to deal with the inner workings of the JPA adapter. This section will document the XML Namespace Support provided by the Spring Integration and will show you how to use the XML Namespace Support to configure the Jpa components.

Common XML Namespace Configuration Attributes

Certain configuration parameters are shared amongst all JPA components and are described below:

auto-startup

Lifecycle attribute signaling if this component should be started during Application Context startup. Defaults to `true`. *Optional*.

id

Identifies the underlying Spring bean definition, which is an instance of either `EventDrivenConsumer` or `PollingConsumer`. *Optional*.

entity-manager-factory

The reference to the JPA Entity Manager Factory that will be used by the adapter to create the `EntityManager`. Either this attribute or the *entity-manager* attribute or the *jpa-operations* attribute must be provided.

entity-manager

The reference to the JPA Entity Manager that will be used by the component. Either this attribute or the *enity-manager-factory* attribute or the *jpa-operations* attribute must be provided.

Note

Usually your Spring Application Context only defines a JPA Entity Manager Factory and the EntityManager is injected using the @PersistenceContext annotation. This, however, is not applicable for the Spring Integration JPA components. Usually, injecting the JPA Entity Manager Factory will be best but in case you want to inject an EntityManager explicitly, you have to define a `SharedEntityManagerBean`. For more information, please see the relevant JavaDoc.

```
<bean id="entityManager"
      class="org.springframework.orm.jpa.support.SharedEntityManagerBean">
    <property name="entityManagerFactory" ref="entityManagerFactoryBean" />
</bean>
```

jpa-operations

Reference to a bean implementing the `JpaOperations` interface. In rare cases it might be advisable to provide your own implementation of the `JpaOperations` interface, instead of relying on the default implementation `org.springframework.integration.jpa.core.DefaultJpaOperations`. As `JpaOperations` wraps the necessary datasource; the JPA Entity Manager or JPA Entity Manager Factory must not be provided, if the *jpa-operations* attribute is used.

entity-class

The fully qualified name of the entity class. The exact semantics of this attribute vary, depending on whether we are performing a persist/update operation or whether we are retrieving objects from the database.

When retrieving data, you can specify the *entity-class* attribute to indicate that you would like to retrieve objects of this type from the database. In that case you must not define any of the query attributes (*jpa-query*, *native-query* or *named-query*)

When persisting data, the *entity-class* attribute will indicate the type of object to persist. If not specified (for persist operations) the entity class will be automatically retrieved from the Message's payload.

jpa-query

Defines the JPA query (Java Persistence Query Language) to be used.

native-query

Defines the native SQL query to be used.

named-query

This attribute refers to a named query. A named query can either be defined in Native SQL or JPAQL but the underlying JPA persistence provider handles that distinction internally.

Providing JPA Query Parameters

For providing parameters, the *parameter* XML sub-element can be used. It provides a mechanism to provide parameters for the queries that are either based on the Java Persistence Query Language (JPQL) or native SQL queries. Parameters can also be provided for Named Queries.

Expression based Parameters

```
<int-jpa:parameter expression="payload.name" name="firstName"/>
```

Value based Parameters

```
<int-jpa:parameter name="name" type="java.lang.String" value="myName"/>
```

Positional Parameters

```
<int-jpa:parameter expression="payload.name"/>
<int-jpa:parameter type="java.lang.Integer" value="21"/>
```

Transaction Handling

All JPA operations like Insert, Update and Delete require a transaction to be active whenever they are performed. For Inbound Channel Adapters there is nothing special to be done, it is similar to the way we configure transaction managers with pollers used with other inbound channel adapters.The xml snippet below shows a sample where a transaction manager is configured with the poller used with an *Inbound Channel Adapter*.

```
<int-jpa:inbound-channel-adapter
    channel="inboundChannelAdapterOne"
    entity-manager="em"
    auto-startup="true"
    jpa-query="select s from Student s"
    expect-single-result="true"
    delete-after-poll="true">
    <int:poller fixed-rate="2000" >
        <int:transactional propagation="REQUIRED"
            transaction-manager="transactionManager"/>
    </int:poller>
</int-jpa:inbound-channel-adapter>
```

However, it may be necessary to specifically start a transaction when using an *Outbound Channel Adapter*/*Gateway*. If a *DirectChannel* is an input channel for the outbound adapter/gateway, and if transaction is active in the current thread of execution, the JPA operation will be performed in the same transaction context. We can also configure to execute this JPA operation in a new transaction as below.

```
<int-jpa:outbound-gateway
    request-channel="namedQueryRequestChannel"
    reply-channel="namedQueryResponseChannel"
    named-query="updateStudentByRollNumber"
    entity-manager="em"
    gateway-type="UPDATING">
    <int-jpa:parameter name="lastName" expression="payload"/>
    <int-jpa:parameter name="rollNumber" expression="headers['rollNumber']"/>

      <int-jpa:transactional propagation="REQUIRES_NEW"
        transaction-manager="transactionManager"/>

</int-jpa:outbound-gateway>
```

As we can see above, the *transactional* sub element of the outbound gateway/adapter will be used to specify the transaction attributes. It is optional to define this child element if you have *DirectChannel* as an input channel to the adapter and you want the adapter to execute the operations in the same

transaction context as the caller. If, however, you are using an *ExecutorChannel*, it is required to have the *transactional* sub element as the invoking client's transaction context is not propagated.

Note

Unlike the *transactional* sub element of the poller which is defined in the spring integration's namespace, the *transactional* sub element for the outbound gateway/adapter is defined in the jpa namespace.

18.4 Inbound Channel Adapter

An *Inbound Channel Adapter* is used to execute a *select* query over the database using JPA QL and return the result. The message payload will be either a single entity or a `List` of entities. Below is a sample xml snippet that shows a sample usage of *inbound-channel-adapter*.

```
<int-jpa:inbound-channel-adapter channel="inboundChannelAdapterOne" ❶
                    entity-manager="em" ❷
                    auto-startup="true" ❸
                    query="select s from Student s" ❹
                    expect-single-result="true" ❺
                    delete-after-poll="true"> ❻
    <int:poller fixed-rate="2000" >
      <int:transactional propagation="REQUIRED" transaction-manager="transactionManager"/>
    </int:poller>
</int-jpa:inbound-channel-adapter>
```

❶ The channel over which the *inbound-channel-adapter* will put the messages with the payload received after executing the provided JPA QL in the *query* attribute.

❷ The `EntityManager` instance that will be used to perform the required JPA operations.

❸ Attribute signalling if the component should be automatically started on startup of the Application Context. The value defaults to `true`

❹ The JPA QL that needs to be executed and whose result needs to be sent out as the payload of the message

❺ The attribute that tells if the executed JPQL query gives a single entity in the result or a `List` of entities. If the value is set to `true`, the single entity retrieved is sent as the payload of the message. If, however, multiple results are returned after setting this to `true`, a `MessagingException` is thrown. The value defaults to `false`.

❻ Set this value to `true` if you want to delete the rows received after execution of the query. Please ensure that the component is operating as part of a transaction. Otherwise, you may encounter an Exception such as: *java.lang.IllegalArgumentException: Removing a detached instance ...*

Configuration Parameter Reference

```
<int-jpa:inbound-channel-adapter
  auto-startup="true" ❶
  channel="" ❷
  delete-after-poll="false"  ❸
  delete-per-row="false"  ❹
  entity-class=""  ❺
  entity-manager="" ❻
  entity-manager-factory="" ❼
  expect-single-result="false" ❽
  id=""
  jpa-operations="" ❾
  jpa-query="" ❿
  named-query="" 11
  native-query="" 12
  parameter-source="" 13
  send-timeout="" 14>
  <int:poller ref="myPoller"/>
 </int-jpa:inbound-channel-adapter>
```

❶ This *Lifecycle* attribute signaled if this component should be started during startup of the Application Context. This attribute defaults to true.*Optional*.

❷ The channel to which the adapter will send a message with the payload that was received after performing the desired JPA operation.

❸ A boolean flag that indicates whether the records selected are to be deleted after they are being polled by the adapter. By default the value is `false`, that is, the records will not be deleted. Please ensure that the component is operating as part of a transaction. Otherwise, you may encounter an Exception such as: *java.lang.IllegalArgumentException: Removing a detached instance ....Optional*.

❹ A boolean flag that indicates whether the records can be deleted in bulk or are deleted one record at a time. By default the value is `false`, that is, the records are bulk deleted.*Optional*.

❺ The fully qualified name of the entity class that would be queried from the database. The adapter will automatically build a JPA Query to be executed based on the entity class name provided.*Optional*.

❻ An instance of `javax.persistence.EntityManager` that will be used to perform the JPA operations. *Optional*.

❼ An instance of `javax.persistence.EntityManagerFactory` that will be used to obtain an instance of `javax.persistence.EntityManager` that will perform the JPA operations. *Optional*.

❽ A boolean flag indicating whether the select operation is expected to return a single result or a `List` of results. If this flag is set to `true`, the single entity selected is sent as the payload of the message. If multiple entities are returned, an exception is thrown. If `false`, the `List` of entities is being sent as the payload of the message. By default the value is `false`.*Optional*.

❾ An implementation of `org.springframework.integration.jpa.core.JpaOperations` that would be used to perform the JPA operations. It is recommended not to provide an implementation of your own but use the default `org.springframework.integration.jpa.core.DefaultJpaOperations` implementation. Either of the *entity-manager*, *entity-manager-factory* or *jpa-operations* attributes is to be used. *Optional*.

❿ The JPA QL that needs to be executed by this adapter.*Optional*.

11 The named query that needs to be executed by this adapter.*Optional*.

12 The native query that will be executed by this adapter. Either of the *jpa-query*, *named-query*,*entity-class* or *native-query* attributes are to be used. *Optional*.

13 An implementation of `o.s.i.jpa.support.parametersource.ParameterSource` which will be used to resolve the values of the parameters provided in the query. Ignored if *entity-class* attribute is provided.*Optional*.

14 Maximum amount of time in milliseconds to wait when sending a message to the channel.*Optional*.

18.5 Outbound Channel Adapter

The JPA Outbound channel adapter allows you to accept messages over a request channel. The payload can either be used as the entity to be persisted, or used along with the headers in parameter expressions for a defined JPQL query to be executed. In the following sub sections we shall see what those possible ways of performing these operations are.

Using an Entity Class

The XML snippet below shows how we can use the Outbound Channel Adapter to persist an entity to the database.

```
<int-jpa:outbound-channel-adapter channel="entityTypeChannel" ❶
    entity-class="org.springframework.integration.jpa.test.entity.Student" ❷
    persist-mode="PERSIST" ❸
    entity-manager="em"/ >❹
```

❶ The channel over which a valid JPA entity will be sent to the JPA Outbound Channel Adapter.

❷ The fully qualified name of the entity class that would be accepted by the adapter to be persisted in the database. You can actually leave off this attribute in most cases as the adapter can determine the entity class automatically from the Spring Integration Message payload.

❸ The operation that needs to be done by the adapter, valid values are *PERSIST*, *MERGE* and *DELETE*. The default value is *MERGE*.

❹ The JPA entity manager to be used.

As we can see above these 4 attributes of the *outbound-channel-adapter* are all we need to configure it to accept entities over the input channel and process them to *PERSIST*,*MERGE* or *DELETE* it from the underlying data source.

Using JPA Query Language (JPA QL)

We have seen in the above sub section how to perform a *PERSIST* action using an entity We will now see how to use the outbound channel adapter which uses JPA QL (Java Persistence API Query Language)

```
<int-jpa:outbound-channel-adapter channel="jpaQlChannel" ❶
  jpa-query="update Student s set s.firstName = :firstName where s.rollNumber
 = :rollNumber" ❷
  entity-manager="em"> ❸
    <int-jpa:parameter name="firstName"  expression="payload['firstName']"/> ❹
    <int-jpa:parameter name="rollNumber" expression="payload['rollNumber']"/>
</int-jpa:outbound-channel-adapter>
```

❶ The input channel over which the message is being sent to the outbound channel adapter

❷ The JPA QL that needs to be executed.This query may contain parameters that will be evaluated using the *parameter* child tag.

❸ The entity manager used by the adapter to perform the JPA operations

❹ This sub element, one for each parameter will be used to evaluate the value of the parameter names specified in the JPA QL specified in the *query* attribute

The *parameter* sub element accepts an attribute *name* which corresponds to the named parameter specified in the provided JPA QL (point 2 in the above mentioned sample). The value of the parameter can either be static or can be derived using an expression. The static value and the expression to derive the value is specified using the *value* and the *expression* attributes respectively. These attributes are mutually exclusive.

If the *value* attribute is specified we can provide an optional *type* attribute. The value of this attribute is the fully qualified name of the class whose value is represented by the *value* attribute. By default the type is assumed to be a `java.lang.String`.

```
<int-jpa:outbound-channel-adapter ... >
    <int-jpa:parameter name="level" value="2" type="java.lang.Integer"/>
    <int-jpa:parameter name="name" expression="payload['name']"/>
</int-jpa:outbound-channel-adapter>
```

As seen in the above snippet, it is perfectly valid to use multiple *parameter* sub elements within an outbound channel adapter tag and derive some parameters using expressions and some with static value. However, care should be taken not to specify the same parameter name multiple times, and, provide one *parameter* sub element for each named parameter specified in the JPA query. For example, we are specifying two parameters *level* and *name* where *level* attribute is a static value of type `java.lang.Integer`, where as the *name* attribute is derived from the payload of the message

Note

Though specifying *select* is valid for JPA QL, it makes no sense as outbound channel adapters will not be returning any result. If you want to select some values, consider using the outbound gateway instead.

Using Native Queries

In this section we will see how to use native queries to perform the operations using JPA outbound channel adapter. Using native queries is similar to using JPA QL, except that the query specified here is a native database query. By choosing native queries we lose the database vendor independence which we get using JPA QL.

One of the things we can achieve using native queries is to perform database inserts, which is not possible using JPA QL (To perform inserts we send JPA entities to the channel adapter as we have seen earlier). Below is a small xml fragment that demonstrates the use of native query to insert values in a table. Please note that we have only mentioned the important attributes below. All other attributes like *channel*, *entity-manager* and the *parameter* sub element has the same semantics as when we use JPA QL.

Important

Please be aware that named parameters may not be supported by your JPA provider in conjunction with native SQL queries. While they work fine using Hibernate, OpenJPA and EclipseLink do NOT support them: https://issues.apache.org/jira/browse/OPENJPA-111 Section 3.8.12 of the JPA 2.0 spec states: "Only positional parameter binding and positional access to result items may be portably used for native queries."

```
<int-jpa:outbound-channel-adapter channel="nativeQlChannel"
  native-query="insert into STUDENT_TABLE(FIRST_NAME,LAST_UPDATED) values
 (:lastName,:lastUpdated)" ❶
  entity-manager="em">
    <int-jpa:parameter name="lastName" expression="payload['updatedLastName']"/>
    <int-jpa:parameter name="lastUpdated" expression="new java.util.Date()"/>
</int-jpa:outbound-channel-adapter>
```

❶ The native query that will be executed by this outbound channel adapter

Using Named Queries

We will now see how to use named queries after seeing using entity, JPA QL and native query in previous sub sections. Using named query is also very similar to using JPA QL or a native query, except that we specify a named query instead of a query. Before we go further and see the xml fragment for the declaration of the *outbound-channel-adapter*, we will see how named JPA named queries are defined.

In our case, if we have an entity called `Student`, then we have the following in the class to define two named queries *selectStudent* and *updateStudent*. Below is a way to define named queries using annotations

```
@Entity
@Table(name="Student")
@NamedQueries({
    @NamedQuery(name="selectStudent",
        query="select s from Student s where s.lastName = 'Last One'"),
    @NamedQuery(name="updateStudent",
        query="update Student s set s.lastName = :lastName,
               lastUpdated = :lastUpdated where s.id in (select max(a.id) from Student
 a)")
})
public class Student {

...
```

You can alternatively use the *orm.xml* to define named queries as seen below

```
<entity-mappings ...>
    ...
    <named-query name="selectStudent">
        <query>select s from Student s where s.lastName = 'Last One'</query>
    </named-query>
</entity-mappings>
```

Now that we have seen how we can define named queries using annotations or using *orm.xml*, we will now see a small xml fragment for defining an *outbound-channel-adapter* using named query

```
<int-jpa:outbound-channel-adapter channel="namedQueryChannel"
            named-query="updateStudent" ❶
            entity-manager="em">
        <int-jpa:parameter name="lastName" expression="payload['updatedLastName']"/>
        <int-jpa:parameter name="lastUpdated" expression="new java.util.Date()"/>
</int-jpa:outbound-channel-adapter>
```

❶ The named query that we want the adapter to execute when it receives a message over the channel

Configuration Parameter Reference

```
<int-jpa:outbound-channel-adapter
  auto-startup="true" ❶
  channel="" ❷
  entity-class="" ❸
  entity-manager="" ❹
  entity-manager-factory="" ❺
  id=""
  jpa-operations="" ❻
  jpa-query="" ❼
  named-query="" ❽
  native-query="" ❾
  order="" ❿
  parameter-source-factory="" 11
  persist-mode="MERGE" 12
  use-payload-as-parameter-source="true" 13
 <int:poller/>
 <int-jpa:transactional/> 14
 <int-jpa:parameter/> 15
</int-jpa:outbound-channel-adapter>
```

❶ Lifecycle attribute signaling if this component should be started during Application Context startup. Defaults to `true`. *Optional*.

❷ The channel from which the outbound adapter will receive messages for performing the desired operation.

❸ The fully qualified name of the entity class for the JPA Operation. The attributes *entity-class*, *query* and *named-query* are mutually exclusive. *Optional*.

❹ An instance of `javax.persistence.EntityManager` that will be used to perform the JPA operations. *Optional*.

❺ An instance of `javax.persistence.EntityManagerFactory` that will be used to obtain an instance of `javax.persistence.EntityManager` that will perform the JPA operations. *Optional*.

❻ An implementation of `org.springframework.integration.jpa.core.JpaOperations` that would be used to perform the JPA operations. It is recommended not to provide an implementation of your own but use the default `org.springframework.integration.jpa.core.DefaultJpaOperations` implementation. Either of the *entity-manager*, *entity-manager-factory* or *jpa-operations* attributes is to be used. *Optional*.

❼ The JPA QL that needs to be executed by this adapter.*Optional*.

❽ The named query that needs to be executed by this adapter.*Optional*.

❾ The native query that will be executed by this adapter. Either of the *jpa-query*, *named-query* or *native-query* attributes are to be used. *Optional*.

❿ The order for this consumer when multiple consumers are registered thereby managing load-balancing and/or failover. Optional (Defaults to *Ordered.LOWEST_PRECEDENCE*).

11 An instance of `o.s.i.jpa.support.parametersource.ParameterSourceFactory` that will be used to get an instance of `o.s.i.jpa.support.parametersource.ParameterSource` which will be used to resolve the values of the parameters provided in the query. Ignored if operations are performed using a JPA entity. If a parameter sub element is used, the factory must be of type `ExpressionEvaluatingParameterSourceFactory` located in package `o.s.i.jpa.support.parametersource`. *Optional*.

12 Accepts one of the *PERSIST*,*MERGE* or *DELETE*. Indicates the operation that the adapter needs to perform. Relevant only if an entity is being used for JPA operations. Ignored if JPA QL, named query or native query is provided. Defaults to *MERGE*. *Optional*.

13 If set to true, the payload of the Message will be used as a source for providing parameters. If false, however, the entire Message will be available as a source for parameters.*Optional*.

14 Defines the transaction management attributes and the reference to transaction manager to be used by the JPA adapter.*Optional*.

15 One or more *parameter* attributes, one for each parameter used in the query. The value or expression provided will be evaluated to compute the value of the parameter.*Optional*.

18.6 Outbound Gateways

The JPA *Inbound Channel Adapter* allows you to poll a database in order to retrieve one or more JPA entities and the retrieved data is consequently used to start a Spring Integration flow using the retrieved data as message payload.

Additionally, you may use JPA *Outbound Channel Adapters* at the end of your flow in order to persist data, essentially terminating the flow at the end of the persistence operation.

However, how can you execute JPA persistence operation in the middle of a flow? For example, you may have business data that you are processing in your Spring Integration message flow, that you would like to persist, yet you still need to execute other components further downstream. Or instead of polling the database using a poller, you rather have the need to execute JPQL queries and retrieve data actively which then is used to being processed in subsequent components within your flow.

This is where JPA Outbound Gateways come into play. They give you the ability to persist data as well as retrieving data. To facilitate these uses, Spring Integration provides two types of JPA Outbound Gateways:

- *Updating Outbound Gateway*
- *Retrieving Outbound Gateway*

Whenever the Outbound Gateway is used to perform an action that saves, updates or soley deletes some records in the database, you need to use an *Updating Outbound Gateway* gateway. If for example an *entity* is used to persist it, then a merged/persisted entity is returned as a result. In other cases the number of records affected (updated or deleted) is returned instead.

When retrieving (selecting) data from the database, we use a *Retrieving Outbound Gateway*. With a *Retrieving Outbound Gateway* gateway, we can use either JPQL, Named Queries (native or JPQL-based) or Native Queries (SQL) for selecting the data and retrieving the results.

An *Updating Outbound Gateway* is functionally very similar to an *Outbound Channel Adapter*, except that an *Updating Outbound Gateway* is used to send a result to the Gateway's *reply channel* after performing the given JPA operation.

A *Retrieving Outbound Gateway* is quite similar to an *Inbound Channel Adapter*.

Note

We recommend you to first refer to the JPA Outbound Channel Adapter section and the JPA Inbound Channel Adapter sections above, as most of the common concepts are being explained there.

This similarity was the main factor to use the central `JpaExecutor` class to unify common functionality as much as possible.

Common for all JPA Outbound Gateways and simlar to the *outbound-channel-adapter*, we can use

- *Entity classes*
- *JPA Query Language (JPQL)*
- *Native query*
- *Named query*

for performing various JPA operations. For configuration examples please see the section called “JPA Outbound Gateway Samples”.

Common Configuration Parameters

JPA Outbound Gateways always have access to the Spring Integration Message as input. As such the following parameters are available:

parameter-source-factory

An instance of `o.s.i.jpa.support.parametersource.ParameterSourceFactory` that will be used to get an instance of `o.s.i.jpa.support.parametersource.ParameterSource`. The *ParameterSource* is used to resolve the values of the parameters provided in the query. The *parameter-source-factory* attribute is ignored, if operations are performed using a JPA entity. If a *parameter* sub-element is used, the factory must be of type `ExpressionEvaluatingParameterSourceFactory`, located in package *o.s.i.jpa.support.parametersource*. *Optional*.

use-payload-as-parameter-source

If set to *true*, the payload of the Message will be used as a source for providing parameters. If set to *false*, the entire Message will be available as a source for parameters. If no JPA Parameters are passed in, this property will default to *true*. This means that using a default `BeanPropertyParameterSourceFactory`, the bean properties of the payload will be used as a source for parameter values for the to-be-executed JPA query. However, if JPA Parameters are passed in, then this property will by default evaluate to *false*. The reason is that JPA Parameters allow for SpEL Expressions to be provided and therefore it is highly beneficial to have access to the entire Message, including the Headers.

Updating Outbound Gateway

```
<int-jpa:updating-outbound-gateway request-channel="" ❶
    auto-startup="true"
    entity-class=""
    entity-manager=""
    entity-manager-factory=""
    id=""
    jpa-operations=""
    jpa-query=""
    named-query=""
    native-query=""
    order=""
    parameter-source-factory=""
    persist-mode="MERGE"
    reply-channel="" ❷
    reply-timeout="" ❸
    use-payload-as-parameter-source="true">

    <int:poller/>
    <int-jpa:transactional/>

    <int-jpa:parameter name="" type="" value=""/>
    <int-jpa:parameter name="" expression=""/>
</int-jpa:updating-outbound-gateway>
```

❶ The channel from which the outbound gateway will receive messages for performing the desired operation. This attribute is similar to *channel* attribute of the outbound-channel-adapter.*Optional*.

❷ The channel to which the gateway will send the response after performing the required JPA operation. If this attribute is not defined, the request message must have a replyChannel header. *Optional*.

❸ Specifies the time the gateway will wait to send the result to the reply channel. Only applies when the reply channel itself might block the send (for example a bounded QueueChannel that is currently full). By default the Gateway will wait indefinitely. The value is specified in milliseconds. *Optional*.

Retrieving Outbound Gateway

```
<int-jpa:retrieving-outbound-gateway request-channel=""
    auto-startup="true"
    delete-after-poll="false"
    delete-in-batch="false"
    entity-class=""
    entity-manager=""
    entity-manager-factory=""
    expect-single-result="false" ❶
    id=""
    jpa-operations=""
    jpa-query=""
    max-number-of-results="" ❷
    named-query=""
    native-query=""
    order=""
    parameter-source-factory=""
    reply-channel=""
    reply-timeout=""
    use-payload-as-parameter-source="true">
    <int:poller></int:poller>
    <int-jpa:transactional/>

    <int-jpa:parameter name="" type="" value=""/>
    <int-jpa:parameter name="" expression=""/>
</int-jpa:retrieving-outbound-gateway>
```

❶ A boolean flag indicating whether the select operation is expected to return a single result or a `List` of results. If this flag is set to `true`, the single entity selected is sent as the payload of the message. If multiple entities are returned, an exception is thrown. If `false`, the `List` of entities is being sent as the payload of the message. By default the value is `false`.*Optional*.

❷ This non zero, non negative integer value tells the adapter not to select more than given number of rows on execution of the select operation. By default, if this attribute is not set, all the possible records are selected by given query.*Optional*.

Important

When choosing to delete entities upon retrieval and you have retrieved a collection of entities, please be aware that by default entities are deleted on a per entity basis. This may cause performance issues.

Alternatively, you can set attribute *deleteInBatch* to *true*, which will perform a batch delete. However, please be aware of the limitation that in that case cascading deletes are not supported.

JSR 317: Java™ Persistence 2.0 states in chapter Chapter 4.10, Bulk Update and Delete Operations that:

"A delete operation only applies to entities of the specified class and its subclasses. It does not cascade to related entities."

For more information please see JSR 317: Java™ Persistence 2.0

JPA Outbound Gateway Samples

This section contains various examples of the *Updating Outbound Gateway* and *Retrieving Outbound Gateway*

Update using an Entity Class

In this example an *Updating Outbound Gateway* is persisted using solely the entity class `org.springframework.integration.jpa.test.entity.Student` as JPA defining parameter.

```
<int-jpa:updating-outbound-gateway request-channel="entityRequestChannel" ❶
    reply-channel="entityResponseChannel" ❷
    entity-class="org.springframework.integration.jpa.test.entity.Student"
    entity-manager="em"/>
```

❶ This is the request channel for the outbound gateway, this is similar to the *channel* attribute of the *outbound-channel-adapter*

❷ This is where a gateway differs from an outbound adapter, this is the channel over which the reply of the performed JPA operation is received. If,however, you are not interested in the reply received and just want to perform the operation, then using a JPA *outbound-channel-adapter* is the appropriate choice. In above case, where we are using entity class, the reply will be the entity object that was created/merged as a result of the JPA operation.

Update using JPQL

In this example, we will see how we can update an entity using the Java Persistence Query Language (JPQL). For this we use an *Updating Outbound Gateway*.

```
<int-jpa:updating-outbound-gateway request-channel="jpaqlRequestChannel"
  reply-channel="jpaqlResponseChannel"
  jpa-query="update Student s set s.lastName = :lastName where s.rollNumber
 = :rollNumber" ❶
  entity-manager="em">
    <int-jpa:parameter name="lastName" expression="payload"/>
    <int-jpa:parameter name="rollNumber" expression="headers['rollNumber']"/>
</int-jpa:updating-outbound-gateway>
```

❶ The JPQL query that will be executed by the gateway. Since an *Updating Outbound Gateway* is used, only *update* and *delete* JPQL queries would be sensible choices.

When sending a message with a String payload and containing a header *rollNumber* with a *long* value, the last name of the student with the provided roll number is updated to the value provided in the message payload. When using an *UPDATING* gateway, the return value is *always* an integer value which denotes the number of records affected by execution of the JPA QL.

Retrieving an Entity using JPQL

The following examples uses a *Retrieving Outbound Gateway* together with JPQL to retrieve (select) one or more entities from the database.

```
<int-jpa:retrieving-outbound-gateway request-channel="retrievingGatewayReqChannel"
    reply-channel="retrievingGatewayReplyChannel"
    jpa-query="select s from Student s where s.firstName = :firstName and s.lastName
 = :lastName"
    entity-manager="em">
    <int-jpa:parameter name="firstName" expression="payload"/>
    <int-jpa:parameter name="lastName" expression="headers['lastName']"/>
</int-jpa:outbound-gateway>
```

Update using a Named Query

Using a Named Query is basically the same as using a JPQL query directly. The difference is that the *named-query* attribute is used instead, as seen in the xml snippet below.

```
<int-jpa:updating-outbound-gateway request-channel="namedQueryRequestChannel"
    reply-channel="namedQueryResponseChannel"
    named-query="updateStudentByRollNumber"
    entity-manager="em">
    <int-jpa:parameter name="lastName" expression="payload"/>
    <int-jpa:parameter name="rollNumber" expression="headers['rollNumber']"/>
</int-jpa:outbound-gateway>
```

Note

You can find a complete Sample application for using Spring Integration's JPA adapter at:

https://github.com/SpringSource/spring-integration-samples/tree/master/basic/jpa

19. 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.

As you can see from above by using `JmsTemplate` and `MessageListener` container Spring Integration relies on Spring's JMS support. This is important to understand since most of the attributes exposed on these adapters will configure the underlying Spring's `JmsTemplate` and/or `MessageListener` container. For more details about `JmsTemplate` and `MessageListener` container please refer to Spring JMS documentation.

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` (or `request-destination-name` or `request-destination-expression`) and then receives a reply Message. The `reply-destination` reference (or `reply-destination-name` or `reply-destination-expression`) can be configured explicitly or else the outbound gateway will use a JMS TemporaryQueue.

Prior to *Spring Integration 2.2*, if necessary, a `TemporaryQueue` was created (and removed) for each request/reply. Beginning with *Spring Integration 2.2*, the outbound gateway can be configured to use a `MessageListener` container to receive replies instead of directly using a new (or cached) `Consumer` to receive the reply for each request. When so configured, and no explicit reply destination is provided, a single `TemporaryQueue` is used for each gateway instead of one for each request.

19.1 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.

```
<int-jms:inbound-channel-
adapter id="jmsIn" destination="inQueue" channel="exampleChannel">
    <int:poller fixed-rate="30000"/>
</int-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.

```
<int-jms:inbound-channel-adapter id="jmsIn"
    destination="inQueue"
    channel="exampleChannel"
    extract-payload="false"/>
    <int:poller fixed-rate="30000"/>
</int-jms:inbound-channel-adapter>
```

19.2 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.

```
<int-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.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 “Enter the GatewayProxyFactoryBean”.

```
<int-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.

19.3 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".

```
<int-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).

19.4 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".

```
<int-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"/>

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

You might notice that this example looks very similar to that included within the section called “Enter the 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.

19.5 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`.

Caution

Using a reply-destination (or reply-destination-name), together with a `CachingConnectionFactory` with *cacheConsumers* set to *true*, can cause Out of Memory conditions. This is because each request gets a new consumer with a new selector (selecting on the correlation-key value, or on the sent JMSMessageID when there is no correlation-key). Given that these selectors are unique, they will remain in the cache unused after the current request completes.

If you specify a reply destination, you are advised to NOT use cached consumers. Alternatively, consider using a <reply-listener/> as described below.

```
<int-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.

<reply-listener/>

Spring Integration 2.2 introduced an alternative technique for handling replies. If you add a `<reply-listener/>` child element to the gateway, instead of creating a consumer for each reply, a `MessageListener` container is used to receive the replies and hand them over to the requesting thread. This provides a number of performance benefits as well as alleviating the cached consumer memory utilization problem described in the caution above.

When using a <reply-listener/>, instead of creating a `TemporaryQueue` for each request, a single `TemporaryQueue` is used (the gateway will create an additional `TemporaryQueue`, as necessary, if the connection to the broker is lost and recovered).

When using a *correlation-key*, multiple gateways can share the same reply destination because the listener container uses a selector that is unique to each gateway.

Caution

If you specify a reply listener, and specify a reply destination (or reply destination name), but provide NO correlation key, the gateway will log a warning and fall back to pre-2.2 behavior. This is because there is no way to configure a selector in this case, thus there is no way to avoid a reply going to a different gateway that might be configured with the same reply destination.

Note that, in this situation, a new consumer is used for each request, and consumers can build up in memory as described in the caution above; therefore cached consumers should not be used in this case.

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

In the above example, a reply listener with default attributes is used. The listener is very lightweight and it is anticipated that, in most cases, only a single consumer will be needed. However, attributes such as *concurrent-consumers*, *max-concurrent-consumers* etc., can be added. Refer to the schema for a complete list of supported attributes, together with the Spring JMS documentation for their meanings.

Attribute Reference

```
<int-jms:outbound-gateway
    connection-factory="connectionFactory"❶
    correlation-key=""❷
    delivery-persistent=""❸
    destination-resolver=""❹
    explicit-qos-enabled=""❺
    extract-reply-payload="true"❻
    extract-request-payload="true"❼
    header-mapper=""❽
    message-converter=""❾
    priority=""❿
    receive-timeout=""11
    reply-channel=""12
    reply-destination=""13
    reply-destination-expression=""14
    reply-destination-name=""15
    reply-pub-sub-domain=""16
    reply-timeout=""17
    request-channel=""18
    request-destination=""19
    request-destination-expression=""20
    request-destination-name=""21
    request-pub-sub-domain=""22
    time-to-live="">23
    <int-jms:reply-listener />24
</int-jms:outbound-gateway>
```

❶ Reference to a `javax.jms.ConnectionFactory`; default `connectionFactory`.

❷ The name of a property that will contain correlation data to correlate responses with replies. If omitted, the gateway will expect the responding system to return the value of the outbound JMSMessageID header in the JMSCorrelationID header. If specified, the gateway will generate a correlation id and populate the specified property with it; the responding system must echo back that value in the same property. Can be set to `JMSCorrelationID`, in which case the standard header is used instead of a simple String property to hold the correlation data. When a `<reply-container/>` is used, the correlation-key MUST be specified if an explicit `reply-destination` is provided.

❸ A boolean value indicating whether the delivery mode should be DeliveryMode.PERSISTENT (true) or DeliveryMode.NON_PERSISTENT (false). This setting will only take effect if `explicit-qos-enabled` is `true`.

❹ A `DestinationResolver`; default is a `DynamicDestinationResolver` which simply maps the destination name to a queue or topic of that name.

❺ When set to `true`, enables the use of quality of service attributes - `priority`, `delivery-mode`, `time-to-live`.

❻ When set to `true` (default), the payload of the Spring Integration reply Message will be created from the JMS Reply Message's body (using the `MessageConverter`). When set to `false`, the entire JMS Message will become the payload of the Spring Integration Message.

❼ When set to `true` (default), the payload of the Spring Integration Message will be converted to a JMSMessage (using the `MessageConverter`). When set to `false`, the entire Spring Integration Message will be converted to the the JMSMessage. In both cases, the Spring Integration Message Headers are mapped to JMS headers and properties using the HeaderMapper.

❽ A `HeaderMapper` used to map Spring Integration Message Headers to/from JMS Message Headers/Properties.

❾ A reference to a `MessageConverter` for converting between JMS Messages and the Spring Integration Message payloads (or messages if `extract-request-payload` is `false`). Default is a `SimpleMessageConverter`.

❿ The default priority of request messages. Overridden by the message priority header, if present; range 0-9. This setting will only take effect if `explicit-qos-enabled` is `true`.

11 The time (in millseconds) to wait for a reply. Default 5 seconds.

12 The channel to which the reply message will be sent.

13 A reference to a `Destination` which will be set as the JMSReplyTo header. At most, only one of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is allowed. If none is provided, a `TemporaryQueue` is used for replies to this gateway.

14 A SpEL expression evaluating to a `Destination` which will be set as the JMSReplyTo header. The expression can result in a `Destination` object, or a `String`, which will be used by the `DestinationResolver` to resolve the actual `Destination`. At most, only one of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is allowed. If none is provided, a `TemporaryQueue` is used for replies to this gateway.

15 The name of the destination which will be set as the JMSReplyTo header; used by the `DestinationResolver` to resolve the actual `Destination`. At most, only one of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is allowed. If none is provided, a `TemporaryQueue` is used for replies to this gateway.

16 When set to `true`, indicates that any reply `Destination` resolved by the `DestinationResolver` should be a `Topic` rather then a `Queue`.

17 The time the gateway will wait when sending the reply message to the `reply-channel`. This only has an effect if the `reply-channel` can block - such as a `QueueChannel` with a capacity limit that is currently full. Default: infinity.

18 The channel on which this gateway receives request messages.

19 A reference to a `Destination` to which request messages will be sent. One, and only one, of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is required.

20 A SpEL expression evaluating to a `Destination` to which request messages will be sent. The expression can result in a `Destination` object, or a `String`, which will be used by the `DestinationResolver` to resolve the actual `Destination`. One, and only one, of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is required.

21 The name of the destination to which request messages will be sent; used by the `DestinationResolver` to resolve the actual `Destination`. One, and only one, of `reply-destination`, `reply-destination-expression`, or `reply-destination-name` is required.

22 When set to `true`, indicates that any request `Destination` resolved by the `DestinationResolver` should be a `Topic` rather then a `Queue`.

23 Specify the message time to live. This setting will only take effect if `explicit-qos-enabled` is `true`.

24 When this element is included, replies are received by a `MessageListenerContainer` rather than creating a consumer for each reply. This can be more efficient in many cases.

19.6 Mapping Message Headers to/from JMS Message

JMS Message can contain meta-information such as JMS API headers as well as simple properties. You can map those to/from Spring Integration Message Headers using `JmsHeaderMapper`. The JMS API headers are passed to the appropriate setter methods (e.g. setJMSReplyTo) whereas other headers will be copied to the general properties of the JMS Message. JMS Outbound Gateway is bootstrapped

with the default implementation of `JmsHeaderMapper` which will map standard JMS API Headers as well as primitive/String Message Headers. Custom header mapper could also be provided via `header-mapper` attribute of inbound and outbound gateways.

19.7 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 affect 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.

19.8 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.

```
<int-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.

```
<int-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.

```
<int-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.

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

19.9 Using JMS Message Selectors

With JMS message selectors you can filter JMS Messages based on JMS headers as well as JMS properties. For example, if you want to listen to messages whose custom JMS header property *fooHeaderProperty* equals *bar*, you can specify the following expression:

```
fooHeaderProperty = 'bar'
```

Message selector expressions are a subset of the SQL-92 conditional expression syntax, and are defined as part of the *Java Message Service* specification (Version 1.1 April 12, 2002). Specifically, please see chapter "3.8 Message Selection". It contains a detailed explanation of the expressions syntax.

You can specify the JMS message *selector* attribute using XML Namespace configuration for the following Spring Integration JMS components:

- JMS Channel
- JMS Publish Subscribe Channel
- JMS Inbound Channel Adapter
- JMS Inbound Gateway
- JMS Message-driven Channel Adapter

Important

It is important to remember that you cannot reference message body values using JMS Message selectors.

19.10 JMS Samples

To experiment with these JMS adapters, check out the JMS samples available in the *Spring Integration Samples* Git repository:

- https://github.com/SpringSource/spring-integration-samples/tree/master/basic/jms

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 samples' *common.xml Spring Application Context* 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 *common.xml* file to use *tcp://localhost:61616* (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.

20. Mail Support

20.1 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`

20.2 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.

20.3 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:int-mail="http://www.springframework.org/schema/integration/mail"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration/mail
    http://www.springframework.org/schema/integration/mail/spring-integration-mail.xsd">
```

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

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

Alternatively, provide the host, username, and password:

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

Note

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

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

```
<int-mail:header-enricher input-channel="expressionsInput" default-overwrite="false">
 <int-mail:to expression="payload.to"/>
 <int-mail:cc expression="payload.cc"/>
 <int-mail:bcc expression="payload.bcc"/>
 <int-mail:from expression="payload.from"/>
 <int-mail:reply-to expression="payload.replyTo"/>
 <int-mail:subject expression="payload.subject" overwrite="true"/>
</int-mail:header-enricher>
```

This example assumes the payload is a JavaBean with appropriate getters for the specified properties, but any SpEL expression can be used. Alternatively, use the `value` attribute to specify a literal. Notice also that you can specify `default-overwrite` and individual `overwrite` attributes to control the behavior with existing headers.

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="receiveChannel"
      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="receiveChannel"
      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.

Important

If your username contains the '@' character use '%40' instead of '@' to avoid parsing errors from the underlying JavaMail API.

```
<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>
```

By default, the `ImapMailReceiver` will search for Messages based on the default `SearchTerm` which is *All mails that are RECENT (if supported), that are NOT ANSWERED, that are NOT DELETED, that are NOT SEEN and have not been processed by this mail receiver (enabled by the use of the custom USER flag or simply NOT FLAGGED if not supported)*. Since version 2.2, the `SearchTerm` used by the `ImapMailReceiver` is fully configurable via the `SearchTermStrategy` which you can inject via the `search-term-strategy` attribute. `SearchTermStrategy` is a simple strategy interface with a single method that allows you to create an instance of the `SearchTerm` that will be used by the `ImapMailReceiver`.

```
public interface SearchTermStrategy {

 SearchTerm generateSearchTerm(Flags supportedFlags, Folder folder);

}
```

For example:

```
<mail:imap-idle-channel-adapter id="customAdapter"
   store-uri="imap:foo"
   …
   search-term-strategy="searchTermStrategy"/>

<bean id="searchTermStrategy"
  class="o.s.i.mail.config.ImapIdleChannelAdapterParserTests.TestSearchTermStrategy"/>
```

In the above example instead of relying on the default `SearchTermStrategy` the `TestSearchTermStrategy` will be used instead

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.

Important

It is important to understand that that these actions (marking messages read, and deleting messages) are performed after the messages are received, but before they are processed. This can cause messages to be lost.

You may wish to consider using transaction synchronization instead - see Section 20.5, “Transaction Synchronization”

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.

Note

Beginning with the 3.0 release, the IMAP idle adapter emits application events (specifically `ImapIdleExceptionEvents`) when exceptions occur. This allows applications to detect and act on those exceptions. The events can be obtained using an `<int-event:inbound-channel-adapter>` or any `ApplicationListener` configured to receive an `ImapIdleExceptionEvent` or one of its super classes.

20.4 Email Message Filtering

Very often you may encounter a requirement to filter incoming messages (e.g., You want to only read emails that have 'Spring Integration' in the *Subject* line). This could be easily accomplished by connecting Inbound Mail adapter with an expression-based *Filter*. Although it would work, there is a downside to this approach. Since messages would be filtered after going through inbound mail adapter all such messages would be marked as read (SEEN) or Un-read (depending on the value of `should-mark-messages-as-read` attribute). However in reality what would be more useful is to mark messages as SEEN only if they passed the filtering criteria. This is very similar to looking at your email client while scrolling through all the messages in the preview pane, but only flagging messages as SEEN that were actually opened and read.

In Spring Integration 2.0.4 we've introduced `mail-filter-expression` attribute on `inbound-channel-adapter` and `imap-idle-channel-adapter`. This attribute allows you to provide an

expression which is a combination of SpEL and Regular Expression. For example if you would like to read only emails that contain 'Spring Integration' in the Subject line, you would configure `mail-filter-expression` attribute like this this: `mail-filter-expression="subject matches '(?i).*Spring Integration.*"`

Since `javax.mail.internet.MimeMessage` is the root context of SpEL Evaluation Context, you can filter on any value available through MimeMessage including the actual body of the message. This one is particularly important since reading the body of the message would typically result in such message to be marked as SEEN by default, but since we now setting PEAK flag of every incomming message to 'true', only messages that were explicitly marked as SEEN will be seen as read.

So in the below example only messages that match the filter expression will be output by this adapter and only those messages will be marked as SEEN. In this case based on the `mail-filter-expression` only messages that contain 'Spring Integration' in the subject line will be produced by this adapter.

```
<int-mail:imap-idle-channel-adapter id="customAdapter"
 store-uri="imaps://some_google_address:${password}@imap.gmail.com/INBOX"
 channel="receiveChannel"
 should-mark-messages-as-read="true"
 java-mail-properties="javaMailProperties"
 mail-filter-expression="subject matches '(?i).*Spring Integration.*'"/>
```

Another reasonable question is what happens on the next poll, or idle event, or what happens when such adapter is restarted. Will there be a potential duplication of massages to be filtered? In other words if on the last retrieval where you had 5 new messages and only 1 passed the filter what would happen with the other 4. Would they go through the filtering logic again on the next poll or idle? After all they were not marked as SEEN. The actual answer is no. They would not be subject of duplicate processing due to another flag (RECENT) that is set by the Email server and is used by Spring Integration mail search filter. Folder implementations set this flag to indicate that this message is new to this folder, that is, it has arrived since the last time this folder was opened. In other while our adapter may peek at the email it also lets the email server know that such email was touched and therefore will be marked as RECENT by the email server.

20.5 Transaction Synchronization

Transaction synchronization for inbound adapters allows you to take different actions after a transaction commits, or rolls back. Transaction synchronization is enabled by adding a <transactional/> element to the poller for the polled <inbound-adapter/>, or to the <imap-idle-inbound-adapter/>. Even if there is no 'real' transaction involved, you can still enable this feature by using a `PseudoTransactionManager` with the <transactional/> element. For more information, see Section C.3, “Transaction Synchronization”.

Because of the many different mail servers, and specifically the limitations that some have, at this time we only provide a strategy for these transaction synchronizations. You can send the messages to some other Spring Integration components, or invoke a custom bean to perform some action. For example, to move an IMAP message to a different folder after the transaction commits, you might use something similar to the following:

```
<int-mail:imap-idle-channel-adapter id="customAdapter"
    store-uri="imaps://foo.com:password@imap.foo.com/INBOX"
    channel="receiveChannel"
    auto-startup="true"
    should-delete-messages="false"
    java-mail-properties="javaMailProperties">
    <int:transactional synchronization-factory="syncFactory"/>
</int-mail:imap-idle-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="@syncProcessor.process(payload)"/>
</int:transaction-synchronization-factory>

<bean id="syncProcessor" class="foo.bar.Mover"/>
```

```
public class Mover {

    public void process(MimeMessage message) throws Exception{
        Folder folder = message.getFolder();
        folder.open(Folder.READ_WRITE);
        String messageId = message.getMessageID();
        Message[] messages = folder.getMessages();
        FetchProfile contentsProfile = new FetchProfile();
        contentsProfile.add(FetchProfile.Item.ENVELOPE);
        contentsProfile.add(FetchProfile.Item.CONTENT_INFO);
        contentsProfile.add(FetchProfile.Item.FLAGS);
        folder.fetch(messages, contentsProfile);
        // find this message and mark for deletion
        for (int i = 0; i < messages.length; i++) {
            if (((MimeMessage) messages[i]).getMessageID().equals(messageId)) {
                messages[i].setFlag(Flags.Flag.DELETED, true);
                break;
            }
        }

        Folder fooFolder = store.getFolder("FOO"));
        fooFolder.appendMessages(new MimeMessage[]{message});
        folder.expunge();
        folder.close(true);
        fooFolder.close(false);
    }
}
```

Important

For the message to be still available for manipulation after the transaction, *should-delete-messages* must be set to 'false'.

21. MongoDb Support

As of version 2.1 Spring Integration introduces support for MongoDB: a *"high-performance, open source, document-oriented database"*. This support comes in the form of a MongoDB-based MessageStore.

21.1 Introduction

To download, install, and run MongoDB please refer to the MongoDB documentation.

21.2 Connecting to MongoDb

To begin interacting with MongoDB you first need to connect to it. Spring Integration builds on the support provided by another Spring project, Spring Data MongoDB, which provides a factory class called `MongoDbFactory` that simplifies integration with the MongoDB Client API.

MongoDbFactory

To connect to MongoDB you can use an implementation of the `MongoDbFactory` interface:

```
public interface MongoDbFactory {

 /**
  * Creates a default {@link DB} instance.
  *
  * @return the DB instance
  * @throws DataAccessException
  */
 DB getDb() throws DataAccessException;

 /**
  * Creates a {@link DB} instance to access the database with the given name.
  *
  * @param dbName must not be {@literal null} or empty.
  *
  * @return the DB instance
  * @throws DataAccessException
  */
 DB getDb(String dbName) throws DataAccessException;
}
```

The example below shows `SimpleMongoDbFactory`, the out-of-the-box implementation:

In Java:

```
MongoDbFactory mongoDbFactory = new SimpleMongoDbFactory(new Mongo(), "test");
```

Or in Spring's XML configuration:

```
<bean id="mongoDbFactory" class="o.s.data.mongodb.core.SimpleMongoDbFactory">
    <constructor-arg>
        <bean class="com.mongodb.Mongo"/>
    </constructor-arg>
    <constructor-arg value="test"/>
</bean>
```

As you can see `SimpleMongoDbFactory` takes two arguments: 1) a `Mongo` instance and 2) a String specifying the name of the database. If you need to configure properties such as `host`, `port`, etc,

you can pass those using one of the constructors provided by the underlying `Mongo` class. For more information on how to configure MongoDB, please refer to the Spring-Data-Document reference.

21.3 MongoDB Message Store

As described in EIP, a Message Store allows you to persist Messages. This can be very useful when dealing with components that have a capability to buffer messages (*QueueChannel, Aggregator, Resequencer*, etc.) if reliability is a concern. In Spring Integration, the MessageStore strategy also provides the foundation for the ClaimCheck pattern, which is described in EIP as well.

Spring Integration's MongoDB module provides the `MongoDbMessageStore` which is an implementation of both the `MessageStore` strategy (mainly used by the *QueueChannel* and *ClaimCheck* patterns) and the `MessageGroupStore` strategy (mainly used by the *Aggregator* and *Resequencer* patterns).

```
<bean id="mongoDbMessageStore" class="o.s.i.mongodb.store.MongoDbMessageStore">
    <constructor-arg ref="mongoDbFactory"/>
</bean>

<int:channel id="somePersistentQueueChannel">
    <int:queue message-store="mongoDbMessageStore"/>
<int:channel>

<int:aggregator input-channel="inputChannel" output-channel="outputChannel"
         message-store="mongoDbMessageStore"/>
```

Above is a sample `MongoDbMessageStore` configuration that shows its usage by a *QueueChannel* and an *Aggregator*. As you can see it is a simple bean configuration, and it expects a `MongoDbFactory` as a constructor argument.

21.4 MongoDB Inbound Channel Adapter

The *MongoDb Inbound Channel Adapter* is a polling consumer that reads data from MongoDb and sends it as a Message payload.

```
<int-mongodb:inbound-channel-adapter id="mongoInboundAdapter"
       channel="replyChannel"
       query="{'name' : 'Bob'}"
       entity-class="java.lang.Object"
       auto-startup="false">
  <int:poller fixed-rate="100"/>
</int-mongodb:inbound-channel-adapter>
```

As you can see from the configuration above, you configure a *MongoDb Inbound Channel Adapter* using the `inbound-channel-adapter` element, providing values for various attributes such as:

- `query` or `query-expression` - a JSON query (see MongoDb Querying)
- `entity-class` - the type of the payload object; if not supplied, a `com.mongodb.DBObject` will be returned.
- `collection-name` or `collection-name-expression` - Identifies the name of the MongoDb collection to use.
- `mongodb-factory` - reference to an instance of `o.s.data.mongodb.MongoDbFactory`

• `mongo-template` - reference to an instance of `o.s.data.mongodb.core.MongoTemplate` and other attributes that are common across all other inbound adapters (e.g., 'channel').

Note

You cannot set both `mongo-template` and `mongodb-factory`.

The example above is relatively simple and static since it has a literal value for the `query` and uses the default name for a `collection`. Sometimes you may need to change those values at runtime, based on some condition. To do that, simply use their `-expression` equivalents (`query-expression` and `collection-name-expression`) where the provided expression can be any valid SpEL expression.

Also, you may wish to do some post-processing to the successfully processed data that was read from the MongoDb. For example; you may want to move or remove a document after its been processed. You can do this using Transaction Synchronization feature that was added with Spring Integration 2.2.

```
<int-mongodb:inbound-channel-adapter id="mongoInboundAdapter"
    channel="replyChannel"
    query="{'name' : 'Bob'}"
    entity-class="java.lang.Object"
    auto-startup="false">
    <int:poller fixed-rate="200" max-messages-per-poll="1">
        <int:transactional synchronization-factory="syncFactory"/>
    </int:poller>
</int-mongodb:inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
 <int:after-commit expression="@documentCleaner.remove(#mongoTemplate, payload,
 headers.mongo_collectionName)" channe="someChannel"/>
</int:transaction-synchronization-factory>

<bean id="documentCleaner" class="foo.bar.DocumentCleaner"/>

<bean id="transactionManager" class="o.s.i.transaction.PseudoTransactionManager"/>
```

```
public class DocumentCleaner {
  public void remove(MongoOperations mongoOperations, Object target, String
 collectionName) {
   if (target instanceof List<?>){
    List<?> documents = (List<?>) target;
    for (Object document : documents) {
     mongoOperations.remove(new BasicQuery(JSON.serialize(document)), collectionName);
    }
   }
  }
}
```

As you can see from the above, all you need to do is declare your poller to be transactional with a `transactional` element. This element can reference a real transaction manager (for example if some other part of your flow invokes JDBC). If you don't have a 'real' transaction, you can use a `org.springframework.integration.transaction.PseudoTransactionManager` which is an implementation of Spring's `PlatformTransactionManager` and enables the use of the transaction synchronization features of the mongo adapter when there is no actual transaction.

Important

This does NOT make MongoDB itself transactional, it simply allows the synchronization of actions to be taken before/after success (commit) or after failure (rollback).

Once your poller is transactional all you need to do is set an instance of the `org.springframework.integration.transaction.TransactionSynchronizationFactory` on the `transactional` element. `TransactionSynchronizationFactory` will create an instance of the `TransactioinSynchronization`. For your convenience, we've exposed a default SpEL-based `TransactionSynchronizationFactory` which allows you to configure SpEL expressions, with their execution being coordinated (synchronized) with a transaction. Expressions for before-commit, after-commit, and after-rollback are supported, together with a channel for each where the evaluation result (if any) will be sent. For each sub-element you can specify `expression` and/or `channel` attributes. If only the `channel` attribute is present the received Message will be sent there as part of the particular synchronization scenario. If only the `expression` attribute is present and the result of an expression is a non-Null value, a Message with the result as the payload will be generated and sent to a default channel (NullChannel) and will appear in the logs (DEBUG). If you want the evaluation result to go to a specific channel add a `channel` attribute. If the result of an expression is null or void, no Message will be generated.

For more information about transaction synchronization, see Section C.3, “Transaction Synchronization”.

21.5 MongoDB Outbound Channel Adapter

The *MongoDb Outbound Channel Adapter* allows you to write the Message payload to a MongoDb document store

```
<int-mongodb:outbound-channel-adapter id="fullConfigWithCollectionExpression"
 collection-name="myCollection"
 mongo-converter="mongoConverter"
 mongodb-factory="mongoDbFactory" />
```

As you can see from the configuration above, you configure a *MongoDb Outbound Channel Adapter* using the `outbound-channel-adapter` element, providing values for various attributes such as:

- `collection-name` or `collection-name-expression` - Identifies the name of the MongoDb collection to use.
- `mongo-converter` - reference to an instance of `o.s.data.mongodb.core.convert.MongoConverter` to assist with converting a raw java object to a JSON document representation
- `mongodb-factory` - reference to an instance of `o.s.data.mongodb.MongoDbFactory`
- `mongo-template` - reference to an instance of `o.s.data.mongodb.core.MongoTemplate` (NOTE: you can not have both mongo-template and mongodb-factory set)

and other attributes that are common across all other inbound adapters (e.g., 'channel').

The example above is relatively simple and static since it has a literal value for the `collection-name`. Sometimes you may need to change this value at runtime based on some condition. To do that, simply use `collection-name-expression` where the provided expression can be any valid SpEL expression.

22. Redis Support

Since version 2.1 Spring Integration introduces support for Redis: *"an open source advanced key-value store".* This support comes in the form of a Redis-based MessageStore as well as Publish-Subscribe Messaging adapters that are supported by Redis via its PUBLISH, SUBSCRIBE and UNSUBSCRIBE commands.

22.1 Introduction

To download, install and run Redis please refer to the Redis documentation.

22.2 Connecting to Redis

To begin interacting with Redis you first need to connect to it. Spring Integration uses support provided by another Spring project, Spring Data Redis, which provides typical Spring constructs: `ConnectionFactory` and `Template`. Those abstractions simplify integration with several Redis-client Java APIs. Currently Spring-Data-Redis supports jedis, jredis and rjc

RedisConnectionFactory

To connect to Redis you would use one of the implementations of the `RedisConnectionFactory` interface:

```
public interface RedisConnectionFactory extends PersistenceExceptionTranslator {

 /**
  * Provides a suitable connection for interacting with Redis.
  *
  * @return connection for interacting with Redis.
  */
 RedisConnection getConnection();
}
```

The example below shows how to create a `JedisConnectionFactory`.

In Java:

```
JedisConnectionFactory jcf = new JedisConnectionFactory();
jcf.afterPropertiesSet();
```

Or in Spring's XML configuration:

```
<bean id="redisConnectionFactory"
    class="o.s.data.redis.connection.jedis.JedisConnectionFactory">
    <property name="port" value="7379" />
</bean>
```

The implementations of RedisConnectionFactory provide a set of properties such as port and host that can be set if needed. Once an instance of RedisConnectionFactory is created, you can create an instance of RedisTemplate and inject it with the RedisConnectionFactory.

RedisTemplate

As with other template classes in Spring (e.g., `JdbcTemplate`, `JmsTemplate`) `RedisTemplate` is a helper class that simplifies Redis data access code. For more information about `RedisTemplate` and its variations (e.g., `StringRedisTemplate`) please refer to the Spring-Data-Redis documentation

The code below shows how to create an instance of `RedisTemplate`:

In Java:

```
RedisTemplate rt = new RedisTemplate<String, Object>();
rt.setConnectionFactory(redisConnectionFactory);
```

Or in Spring's XML configuration::

```
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
 <property name="connectionFactory" ref="redisConnectionFactory"/>
</bean>
```

22.3 Messaging with Redis

As mentioned in the introduction Redis provides support for Publish-Subscribe messaging via its PUBLISH, SUBSCRIBE and UNSUBSCRIBE commands. As with JMS and AMQP, Spring Integration provides Message Channels and adapters for sending and receiving messages via Redis.

Redis Publish/Subscribe channel

Similar to the JMS there are cases where both the producer and consumer 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, however just like with Spring Integration's JMS support, there is a simpler approach to address this use case.

```
<int-redis:publish-subscribe-channel id="redisChannel" topic-name="si.test.topic"/>
```

The publish-subscribe-channel (above) will behave much like a normal `<publish-subscribe-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 Redis topic name - a String value specified by the `topic-name` attribute. However unlike JMS this topic doesn't have to be created in advance or even auto-created by Redis. In Redis topics are simple String values that play the role of an address, and all the producer and consumer need to do to communicate is use the same String value as their topic name. A simple subscription to this channel means that asynchronous pub-sub messaging is possible between the producing and consuming endpoints, but unlike the asynchronous Message Channels created by adding a `<queue/>` sub-element within a simple Spring Integration `<channel/>` element, the Messages are not just stored in an in-memory queue. Instead those Messages are passed through Redis allowing you to rely on its support for persistence and clustering as well as its interoperability with other non-java platforms.

Redis Inbound Channel Adapter

The Redis-based Inbound Channel Adapter adapts incoming Redis messages into Spring Integration Messages in the same way as other inbound adapters. It receives platform-specific messages (Redis in this case) and converts them to Spring Integration Messages using a `MessageConverter` strategy.

```
<int-redis:inbound-channel-adapter id="redisAdapter"
       topics="foo, bar"
       channel="receiveChannel"
       error-channel="testErrorChannel"
       message-converter="testConverter" />

<bean id="redisConnectionFactory"
    class="o.s.data.redis.connection.jedis.JedisConnectionFactory">
    <property name="port" value="7379" />
</bean>

<bean id="testConverter" class="foo.bar.SampleMessageConverter" />
```

Above is a simple but complete configuration of a Redis Inbound Channel Adapter. Note that the above configuration relies on the familiar Spring paradigm of auto-discovering certain beans. In this case the `redisConnectionFactory` is implicitly injected into the adapter. You can of course specify it explicitly using the `connection-factory` attribute instead.

Also, note that the above configuration injects the adapter with a custom `MessageConverter`. The approach is similar to JMS where `MessageConverters` are used to convert between Redis Messages and the Spring Integration Message payloads. The default is a `SimpleMessageConverter`.

Inbound adapters can subscribe to multiple topic names hence the comma-delimited set of values in the `topics` attribute.

Redis Outbound Channel Adapter

The Redis-based Outbound Channel Adapter adapts outgoing Spring Integration messages into Redis messages in the same way as other outbound adapters. It receives Spring Integration messages and converts them to platform-specific messages (Redis in this case) using a `MessageConverter` strategy.

```
<int-redis:outbound-channel-adapter id="outboundAdapter"
    channel="sendChannel"
    topic="foo"
    message-converter="testConverter"/>

<bean id="redisConnectionFactory"
    class="o.s.data.redis.connection.jedis.JedisConnectionFactory">
    <property name="port" value="7379"/>
</bean>

<bean id="testConverter" class="foo.bar.SampleMessageConverter" />
```

As you can see the configuration is similar to the Redis Inbound Channel Adapter. The adapter is implicitly injected with a `RedisConnectionFactory` which was defined with '`redisConnectionFactory`' as its bean name. This example also includes the optional, custom `MessageConverter` (the '`testConverter`' bean).

22.4 Redis Message Store

As described in EIP, a Message Store allows you to persist Messages. This can be very useful when dealing with components that have a capability to buffer messages (*QueueChannel, Aggregator, Resequencer*, etc.) if reliability is a concern. In Spring Integration, the MessageStore strategy also provides the foundation for the ClaimCheck pattern, which is described in EIP as well.

Spring Integration's Redis module provides the `RedisMessageStore` which is an implementation of both the the `MessageStore` strategy (mainly used by the *QueueChannel* and *ClaimCheck* patterns) and the `MessageGroupStore` strategy (mainly used by the *Aggregator* and *Resequencer* patterns).

```
<bean id="redisMessageStore" class="o.s.i.redis.store.RedisMessageStore">
    <constructor-arg ref="redisConnectionFactory"/>
</bean>

<int:channel id="somePersistentQueueChannel">
    <int:queue message-store="redisMessageStore"/>
<int:channel>

<int:aggregator input-channel="inputChannel" output-channel="outputChannel"
         message-store="redisMessageStore"/>
```

Above is a sample `RedisMessageStore` configuration that shows its usage by a *QueueChannel* and an *Aggregator*. As you can see it is a simple bean configuration, and it expects a `RedisConnectionFactory` as a constructor argument.

By default the `RedisMessageStore` will use Java serialization to serialize the Message. However if you want to use a different serialization technique (e.g., JSON), you can provide your own serializer via the `valueSerializer` property of the `RedisMessageStore`.

22.5 RedisStore Inbound Channel Adapter

The *RedisStore Inbound Channel Adapter* is a polling consumer that reads data from a Redis collection and sends it as a Message payload.

```
<int-redis:store-inbound-channel-adapter id="listAdapter"
    connection-factory="redisConnectionFactory"
    key="myCollection"
    channel="redisChannel"
    collection-type="LIST" >
    <int:poller fixed-rate="2000" max-messages-per-poll="10"/>
</int-redis:store-inbound-channel-adapter>
```

As you can see from the configuration above you configure a *Redis Store Inbound Channel Adapter* using the `store-inbound-channel-adapter` element, providing values for various attributes such as:

- `key` or `key-expression` - The name of the key for the collection being used.
- `collection-type` - enumeration of the Collection types supported by this adapter. Supported Collections are: LIST, SET, ZSET, PROPERTIES, MAP
- `connection-factory` - reference to an instance of `o.s.data.redis.connection.RedisConnectionFactory`
- `redis-template` - reference to an instance of `o.s.data.redis.core.RedisTemplate`

and other attributes that are common across all other inbound adapters (e.g., 'channel').

Note

You cannot set both `redis-template` and `connection-factory`.

Important

By default, the adapter uses a `StringRedisTemplate`; this uses `StringRedisSerializers` for keys, values, hash keys and hash values. If your Redis store contains objects that are serialized with other techniques, you must supply a `RedisTemplate` configured with appropriate serializers. For example, if the store is written to using a RedisStore Outbound Adapter that has its `extract-payload-elements` set to false, you must provide a `RedisTemplate` configured thus:

```
<bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">
    <property name="connectionFactory" ref="redisConnectionFactory"/>
    <property name="keySerializer">

  <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
    </property>
    <property name="hashKeySerializer">

  <bean class="org.springframework.data.redis.serializer.StringRedisSerializer"/>
    </property>
</bean>
```

This uses String serializers for keys and hash keys and the default JDK Serialization serializers for values and hash values.

The example above is relatively simple and static since it has a literal value for the `key`. Sometimes, you may need to change the value of the key at runtime based on some condition. To do that, simply use `key-expression` instead, where the provided expression can be any valid SpEL expression.

Also, you may wish to perform some post-processing to the successfully processed data that was read from the Redis collection. For example; you may want to move or remove the value after its been processed. You can do this using the Transaction Synchronization feature that was added with Spring Integration 2.2.

```
<int-redis:store-inbound-channel-adapter id="zsetAdapterWithSingleScoreAndSynchronization"
        connection-factory="redisConnectionFactory"
        key-expression="'presidents'"
        channel="otherRedisChannel"
        auto-startup="false"
        collection-type="ZSET">
            <int:poller fixed-rate="1000" max-messages-per-poll="2">
                <int:transactional synchronization-factory="syncFactory"/>
            </int:poller>
</int-redis:store-inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
 <int:after-commit expression="payload.removeByScore(18, 18)"/>
</int:transaction-synchronization-factory>

<bean id="transactionManager" class="o.s.i.transaction.PseudoTransactionManager"/>
```

As you can see from the above all, you need to do is declare your poller to be transactional with a `transactional` element. This element can reference a real transaction manager (for example if some other part of your flow invokes JDBC). If you don't have a 'real' transaction, you can use a `o.s.i.transaction.PseudoTransactionManager` which is an implementation of Spring's `PlatformTransactionManager` and enables the use of the transaction synchronization features of the redis adapter when there is no actual transaction.

Important

This does NOT make the Redis activities themselves transactional, it simply allows the synchronization of actions to be taken before/after success (commit) or after failure (rollback).

Once your poller is transactional all you need to do is set an instance of the `org.springframework.integration.transaction.TransactionSynchronizationFactory` on the `transactional` element. `TransactionSynchronizationFactory` will create an instance of the `TransactionSynchronization`. For your convenience we've exposed a default SpEL-based `TransactionSynchronizationFactory` which allows you to configure SpEL expressions, with their execution being coordinated (synchronized) with a transaction. Expressions for before-commit, after-commit, and after-rollback are supported, together with a channel for each where the evaluation result (if any) will be sent. For each sub-element you can specify `expression` and/or `channel` attributes. If only the `channel` attribute is present the received Message will be sent there as part of the particular synchronization scenario. If only the `expression` attribute is present and the result of an expression is a non-Null value, a Message with the result as the payload will be generated and sent to a default channel (NullChannel) and will appear in the logs (DEBUG). If you want the evaluation result to go to a specific channel add a `channel` attribute. If the result of an expression is null or void, no Message will be generated.

For more information about transaction synchronization, see Section C.3, “Transaction Synchronization”.

22.6 RedisStore Outbound Channel Adapter

The *RedisStore Outbound Channel Adapter* allows you to write a Message payload to a Redis collection

```
<int-redis:store-outbound-channel-adapter id="redisListAdapter"
          collection-type="LIST"
          channel="requestChannel"
          key="myCollection" />
```

As you can see from the configuration above, you configure a *Redis Store Outbound Channel Adapter* using the `store-inbound-channel-adapter` element, providing values for various attributes such as:

- `key` or `key-expression` - The name of the key for the collection being used.
- `extract-payload-elements` - If set to `true` (Default) and the payload is an instance of a "multi-value" object (i.e., Collection or Map) it will be stored using addAll/ putAll semantics. Otherwise, if set to `false` the payload will be stored as a single entry regardless of its type. If the payload is not an instance of a "multi-value" object, the value of this attribute is ignored and the payload will always be stored as a single entry.
- `collection-type` - enumeration of the Collection types supported by this adapter. Supported Collections are: LIST, SET, ZSET, PROPERTIES, MAP
- `map-key-expression` - SpEL expression that returns the name of the key for entry being stored. Only applies if the `collection-type` is MAP or PROPERTIES and 'extract-payload-elements' is false.
- `connection-factory` - reference to an instance of `o.s.data.redis.connection.RedisConnectionFactory`

- `redis-template` - reference to an instance of `o.s.data.redis.core.RedisTemplate`

and other attributes that are common across all other inbound adapters (e.g., 'channel').

Note

You cannot set both `redis-template` and `connection-factory`.

Important

By default, the adapter uses a `StringRedisTemplate`; this uses `StringRedisSerializers` for keys, values, hash keys and hash values. However, if `extract-payload-elements` is set to false, a `RedisTemplate` using `StringRedisSerializers` for keys and hash keys, and `JdkSerializationRedisSerializers` for values and hash values will be used. With the JDK serializer, it is important to understand that java serialization is used for all values, regardless of whether the value is actually a collection or not. If you need more control over the serialization of values, you may want to consider providing your own `RedisTemplate` rather than relying upon these defaults.

The example above is relatively simple and static since it has a literal values for the `key` and other attributes. Sometimes you may need to change the values dynamically at runtime based on some condition. To do that simply use their `-expression` equivalents (`key-expression`, `map-key-expression` etc.) where the provided expression can be any valid SpEL expression.

23. Resource Support

23.1 Introduction

The *Resource Inbound Channel Adapter* builds upon Spring's `Resource` abstraction to support greater flexibility across a variety of actual types of underlying resources, such as a file, a URL, or a class path resource. Therefore, it's similar to but more generic than the *File Inbound Channel Adapter*.

23.2 Resource Inbound Channel Adapter

The *Resource Inbound Channel Adapter* is a polling adapter that creates a `Message` whose payload is a collection of `Resource` objects.

`Resource` objects are resolved based on the pattern specified using the `pattern` attribute. The collection of resolved `Resource` objects is then sent as a payload within a `Message` to the adapter's channel. That is one major difference between *Resource Inbound Channel Adapter* and *File Inbound Channel Adapter*; the latter buffers File objects and sends a single `File` object per `Message`.

Below is an example of a very simple configuration which will find all files ending with the 'properties' extension in the `foo.bar` package available on the classpath and will send them as the payload of a Message to the channel named '`resultChannel`':

```
<int:resource-inbound-channel-adapter id="resourceAdapter"
                channel="resultChannel"
                pattern="classpath:foo/bar/*.properties">
    <int:poller fixed-rate="1000"/>
</int:resource-inbound-channel-adapter>
```

The *Resource Inbound Channel Adapter* relies on the `org.springframework.core.io.support.ResourcePatternResolver` strategy interface to resolve the provided pattern. It defaults to an instance of the current `ApplicationContext`. However you may provide a reference to an instance of your own implementation of `ResourcePatternResolver` using the `pattern-resolver` attribute:

```
<int:resource-inbound-channel-adapter id="resourceAdapter"
                channel="resultChannel"
                pattern="classpath:foo/bar/*.properties"
                pattern-resolver="myPatternResolver">
    <int:poller fixed-rate="1000"/>
</int:resource-inbound-channel-adapter>

<bean id="myPatternResolver" class="org.example.MyPatternResolver"/>
```

You may have a use case where you need to further filter the collection of resources resolved by the `ResourcePatternResolver`. For example, you may want to prevent resources that were resolved already from appearing in a collection of resolved resources ever again. On the other hand your resources might be updated rather often and you *do* want them to be picked up again. In other words there is a valid use case for defining an additional filter as well as disabling filtering altogether. You can provide your own implementation of the `org.springframework.integration.util.CollectionFilter` strategy interface:

```
public interface CollectionFilter<T> {

    Collection<T> filter(Collection<T> unfilteredElements);

}
```

As you can see the `CollectionFilter` receives a collection of un-filtered elements (which would be `Resource` objects in this case), and it returns a collection of filtered elements of that same type.

If you are defining the adapter via XML but you do not specify a filter reference, a default implementation of `CollectionFilter` will be used by the *Resource Inbound Channel Adapter*. The implementation class of that default filter is `org.springframework.integration.util.AcceptOnceCollectionFilter`. It remembers the elements passed in the previous invocation in order to avoid returning those elements more than once.

To inject your own implementation of `CollectionFilter` instead, use the `filter` attribute.

```
<int:resource-inbound-channel-adapter id="resourceAdapter"
                channel="resultChannel"
                pattern="classpath:foo/bar/*.properties"
                filter="myFilter">
    <int:poller fixed-rate="1000"/>
</int:resource-inbound-channel-adapter>

<bean id="myFilter" class="org.example.MyFilter"/>
```

If you don't need any filtering and want to disable even the default `CollectionFilter` strategy, simply provide an empty value for the filter attribute (e.g., `filter=""`)

24. RMI Support

24.1 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.

24.2 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>
```

24.3 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>
```

Important

If you use an `errorChannel` on an inbound gateway, it would be normal for the error flow to return a result (or throw an exception). This is because it is likely that there is a corresponding outbound gateway waiting for a response of some kind. Consuming a message on the error flow, and not replying, will result in no reply at the inbound gateway. Exceptions (on the main flow when there is no errorChannel, or on the error flow) will be propagated to the corresponding inbound gateway.

24.4 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.

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

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

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

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

<int-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.

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

25. SFTP Adapters

Spring Integration provides support for file transfer operations via SFTP.

25.1 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 three *client* side endpoints: *Inbound Channel Adapter*, *Outbound Channel Adapter*, and *Outbound Gateway* It also provides convenient namespace configuration to define these *client* components.

```
xmlns:int-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.xsd"
```

25.2 SFTP Session Factory

Important

Starting with version 3.0, sessions are no longer cached by default. See Section 25.3, “SFTP Session Caching”.

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

```
<beans:bean id="sftpSessionFactory"
    class="org.springframework.integration.sftp.session.DefaultSftpSessionFactory">
    <beans:property name="host" value="localhost"/>
    <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`, a new SFTP session is being created. Under the covers, the SFTP Session Factory relies on the JSch library to provide the SFTP capabilities.

However, Spring Integration also supports the caching of SFTP sessions, please see Section 25.3, “SFTP Session Caching” for more information.

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). Please also see Section 25.7, “SFTP/JSCH Logging”.

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*.

Configuration Properties

Below you will find all properties that are exposed by the `DefaultSftpSessionFactory`.

clientVersion

Allows you to set the client version property. It's default depends on the underlying JSch version but it will look like: *SSH-2.0-JSCH-0.1.45*

enableDaemonThread

If `true`, all threads will be daemon threads. If set to `false`, normal non-daemon threads will be used instead. This property will be set on the underlying JSch `Session`. There, this property will default to `false`, if not explicitly set.

host

The url of the host you want connect to. *Mandatory*.

hostKeyAlias

Sets the host key alias, used when comparing the host key to the known hosts list.

knownHosts

Specifies the filename that will be used to create a host key repository. The resulting file has the same format as OpenSSH's *known_hosts* file.

password

The password to authenticate against the remote host. If a *password* is not provided, then the *privateKey* property is mandatory.

port

The port over which the SFTP connection shall be established. If not specified, this value defaults to 22. If specified, this properties must be a positive number.

privateKey

Allows you to set a `Resource`, which represents the location of the private key used for authenticating against the remote host. If the *privateKey* is not provided, then the *password* property is mandatory.

privateKeyPassphrase

The password for the private key. Optional.

proxy

Allows for specifying a JSch-based `Proxy`. If set, then the proxy object is used to create the connection to the remote host.

serverAliveCountMax

Specifies the number of server-alive messages, which will be sent without any reply from the server before disconnecting. If not set, this property defaults to `1`.

serverAliveInterval

Sets the timeout interval (milliseconds) before a server alive message is sent, in case no message is received from the server.

sessionConfig

Using `Properties`, you can set additional configuration setting on the underlying JSch Session.

socketFactory

Allows you to pass in a `SocketFactory`. The socket factory is used to create a socket to the target host. When a proxy is used, the socket factory is passed to the proxy. By default plain TCP sockets are used.

timeout

The timeout property is used as the socket timeout parameter, as well as the default connection timeout. Defaults to `0`, which means, that no timeout will occur.

user

The remote user to use. *Mandatory*.

25.3 SFTP Session Caching

Important

Starting with version 3.0, sessions are no longer cached by default; the `cache-sessions` attribute is no longer supported on endpoints. You must now use a `CachingSessionFactory` (see below) if you wish to cache sessions.

In versions prior to 3.0, the sessions were cached automatically by default. A `cache-sessions` attribute was available for disabling the auto caching, but that solution did not provide a way to configure other session caching attributes. For example, you could not limit on the number of sessions created. To support that requirement and other configuration options, a `CachingSessionFactory` was provided. It provides `sessionCacheSize` and `sessionWaitTimeout` properties. As its name suggests, the `sessionCacheSize` property controls how many active sessions the factory will maintain in its cache (the DEFAULT is unbounded). If the `sessionCacheSize` threshold has been reached, any attempt to acquire another session will block until either one of the cached sessions becomes available or until the wait time for a Session expires (the DEFAULT wait time is Integer.MAX_VALUE). The `sessionWaitTimeout` property enables configuration of that value.

If you want your Sessions to be cached, simply configure your default Session Factory as described above and then wrap it in an instance of `CachingSessionFactory` where you may provide those additional properties.

```
<bean id="sftpSessionFactory"
    class="org.springframework.integration.sftp.session.DefaultSftpSessionFactory">
    <property name="host" value="localhost"/>
</bean>

<bean id="cachingSessionFactory"
    class="org.springframework.integration.file.remote.session.CachingSessionFactory">
    <constructor-arg ref="sftpSessionFactory"/>
    <property name="sessionCacheSize" value="10"/>
    <property name="sessionWaitTimeout" value="1000"/>
</bean>
```

In the above example you see a `CachingSessionFactory` created with the `sessionCacheSize` set to 10 and the `sessionWaitTimeout` set to 1 second (its value is in millliseconds).

25.4 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.

```
<int-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"
   local-filename-generator-expression="#this.toUpperCase() + '.a'"
   local-filter="myFilter"
   delete-remote-files="false">
  <int:poller fixed-rate="1000"/>
</int-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.

By default the transferred file will carry the same name as the original file. If you want to override this behavior you can set the `local-filename-generator-expression` attribute which allows you to provide a SpEL Expression to generate the name of the local file. Unlike outbound gateways and adapters where the root object of the SpEL Evaluation Context is a `Message`, this inbound adapter does not yet have the Message at the time of evaluation since that's what it ultimately generates with the transferred file as its payload. So, the root object of the SpEL Evaluation Context is the original name of the remote file (String).

Sometimes 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. This filter determines which remote files are retrieved.

Note

Beginning with 3.0, you can also specify a filter used to filter the files locally, once they have been retrieved. The default filter is an `AcceptOnceFileListFilter` which prevents processing files with the same name multiple times in the same JVM execution; this can now be overridden (for example with an `AcceptAllFileListFilter`), using the `local-filter` attribute. Previously, the default `AcceptOnceFileListFilter` could not be overridden.

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

Sometimes 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"/>
```

25.5 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'.

Avoiding Partially Written Files

One of the common problems, when dealing with file transfers, is the possibility of processing a *partial file* - a file might appear in the file system before its transfer is actually complete.

To deal with this issue, Spring Integration SFTP adapters use a very common algorithm where files are transferred under a temporary name and than renamed once they are fully transferred.

By default, every file that is in the process of being transferred will appear in the file system with an additional suffix which, by default, is `.writing`; this can be changed using the `temporary-file-suffix` attribute.

However, there may be situations where you don't want to use this technique (for example, if the server does not permit renaming files). For situations like this, you can disable this feature by setting `use-temporary-file-name` to `false` (default is `true`). When this attribute is `false`, the file is written with its final name and the consuming application will need some other mechanism to detect that the file is completely uploaded before accessing it.

25.6 SFTP Outbound Gateway

The *SFTP Outbound Gateway* provides a limited set of commands to interact with a remote SFTP server.

Commands supported are:

- ls (list files)
- get (retrieve file)
- mget (retrieve file(s))
- rm (remove file(s))
- mv (move/rename file)

ls

ls lists remote file(s) and supports the following options:

- -1 - just retrieve a list of filenames, default is to retrieve a list of `FileInfo` objects.
- -a - include all files (including those starting with '.')
- -f - do not sort the list

- -dirs - include directories (excluded by default)
- -links - include symbolic links (excluded by default)

In addition, filename filtering is provided, in the same manner as the `inbound-channel-adapter`.

The message payload resulting from an *ls* operation is a list of file names, or a list of `FileInfo` objects. These objects provide information such as modified time, permissions etc.

The remote directory that the *ls* command acted on is provided in the `file_remoteDirectory` header.

get

get retrieves a remote file and supports the following option:

- -P - preserve the timestamp of the remote file

The message payload resulting from a *get* operation is a `File` object representing the retrieved file.

The remote directory is provided in the `file_remoteDirectory` header, and the filename is provided in the `file_remoteFile` header.

mget

mget retrieves multiple remote files based on a pattern and supports the following option:

- -x - Throw an exception if no files match the pattern (otherwise an empty list is returned)

The message payload resulting from an *mget* operation is a `List<File>` object - a List of File objects, each representing a retrieved file.

The remote directory is provided in the `file_remoteDirectory` header, and the pattern for the filenames is provided in the `file_remoteFile` header.

rm

The *rm* command has no options.

The message payload resulting from an *rm* operation is Boolean.TRUE if the remove was successful, Boolean.FALSE otherwise. The remote directory is provided in the `file_remoteDirectory` header, and the filename is provided in the `file_remoteFile` header.

mv

The *mv* command has no options.

The *expression* attribute defines the "from" path and the *rename-expression* attribute defines the "to" path. By default, the *rename-expression* is `headers['file_renameTo']`. This expression must not evaluate to null, or an empty `String`. If necessary, any remote directories needed will be created. The payload of the result message is `Boolean.TRUE`. The original remote directory is provided in the `file_remoteDirectory` header, and the filename is provided in the `file_remoteFile` header. The new path is in the `file_renameTo` header.

For all commands, the PATH that the command acts on is provided by the 'expression' property of the gateway. For the mget command, the expression might evaluate to '*', meaning retrieve all files, or 'somedirectory/*' etc.

Here is an example of a gateway configured for an ls command...

```
<int-ftp:outbound-gateway id="gateway1"
  session-factory="ftpSessionFactory"
  request-channel="inbound1"
  command="ls"
  command-options="-1"
  expression="payload"
  reply-channel="toSplitter"/>
```

The payload of the message sent to the toSplitter channel is a list of String objects containing the filename of each file. If the `command-options` was omitted, it would be a list of `FileInfo` objects. Options are provided space-delimited, e.g. `command-options="-1 -dirs -links"`.

25.7 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
```

26. Stream Support

26.1 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.

26.2 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>
```

26.3 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 3.3, “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>
```

26.4 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:int-stream="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.xsd
      http://www.springframework.org/schema/integration/stream
      http://www.springframework.org/schema/integration/stream/spring-integration-
stream.xsd">
```

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

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

<int-stream: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.

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

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

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

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

27. Syslog Support

27.1 Introduction

Spring Integration 2.2 introduced the Syslog transformer `SyslogToMapTransformer`. This transformer, together with a `UDP` or `TCP` inbound adapter could be used to receive and analyze syslog records from other hosts. The transformer creates a message payload containing a map of the elements from the syslog message.

Spring Integration 3.0 introduced convenient namespace support for configuring a Syslog inbound adapter in a single element.

27.2 Syslog <inbound-channel-adapter>

This element encompases a `UDP` or `TCP` inbound channel adapter and a `MessageConverter` to convert the Syslog message to a Spring Integration message. The `DefaultMessageConverter` delegates to the `SyslogToMapTransformer`, creating a message with its payload being the `Map` of Syslog fields. In addition, all fields except the message are also made available as headers in the message, prefixed with `syslog_`.

Example Configuration

```
<int-syslog:inbound-channel-adapter id="syslogIn" port="1514" />
```

A `UDP` adapter that sends messages to channel `syslogIn` (the adapter bean name is `syslogIn.adapter`). The adapter listens on port `1514`.

```
<int-syslog:inbound-channel-adapter id="syslogIn"
 channel="fromSyslog" port="1514" />
```

A `UDP` adapter that sends message to channel `fromSyslog` (the adapter bean name is `syslogIn`). The adapter listens on port `1514`.

```
<int-syslog:inbound-channel-adapter id="bar" protocol="tcp" port="1514" />
```

A `TCP` adapter that sends messages to channel `syslogIn` (the adapter bean name is `syslogIn.adapter`). The adapter listens on port `1514`.

Note the addition of the `protocol` attribute. This attribute can contain `udp` or `tcp`; it defaults to `udp`.

```
<int-syslog:inbound-channel-adapter id="udpSyslog"
 channel="fromSyslog"
 auto-startup="false"
 phase="10000"
 converter="converter"
 send-timeout="1000"
 error-channel="errors">
  <int-syslog:udp-attributes port="1514" lookup-host="false" />
</int-syslog:inbound-channel-adapter>
```

A `UDP` adapter that sends messages to channel `fromSyslog`. It also shows the `SmartLifecyle` attributes `auto-startup` and `phase`. It has a reference to a custom `org.springframework.integration.syslog.MessageConverter` with id `converter` and an

`error-channel`. Also notice the `udp-attributes` child element. You can set various UDP attributes here, as defined in Table 16.2, “UDP Inbound Channel Adapter Attributes”.

Note

When using the `udp-attributes` element, the `port` attribute must be provided there rather than on the `inbound-channel-adapter` element itself.

```
<int-syslog:inbound-channel-adapter id="TcpSyslog"
 protocol="tcp"
 channel="fromSyslog"
 connection-factory="cf" />

<int-ip:tcp-connection-factory id="cf" type="server" port="1514" />
```

A `TCP` adapter that sends messages to channel `fromSyslog`. It also shows how to reference an externally defined connection factory, which can be used for advanced configuration (socket keep alive etc). For more information, see Section 16.3, “TCP Connection Factories”.

Note

The externally configured `connection-factory` must be of type `server` and, the port is defined there rather than on the `inbound-channel-adapter` element itself.

28. 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.

28.1 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

Previous versions of Spring Integration were dependent upon the Twitter4J API, but with the release of Spring Social 1.0 GA, Spring Integration, as of version 2.1, now builds directly upon Spring Social's Twitter support, instead of Twitter4J.

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

```
xmlns:int-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.xsd"
```

28.2 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.

28.3 Twitter Template

As mentioned above, Spring Integration relies upon Spring Social, and that library provides an implementation of the template pattern, `o.s.social.twitter.api.impl.TwitterTemplate` to

interact with Twitter. For anonymous operations (e.g., search), you don't have to define an instance of `TwitterTemplate` explicitly, since a default instance will be created and injected into the endpoint. However, for authenticated operations (update status, send direct message, etc.), you must configure a `TwitterTemplate` as a bean and inject it explicitly into the endpoint, because the authentication configuration is required. Below is a sample configuration of TwitterTemplate:

```
<bean id="twitterTemplate" class="o.s.social.twitter.api.impl.TwitterTemplate">
 <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="o.s.social.twitter.api.impl.TwitterTemplate">
    <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>
```

28.4 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 a bean named 'metadataStore' within the Application Context.

```
<bean id="metadataStore" class="o.s.i.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.

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

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

Mentions Inbound Message Channel Adapter

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

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

```
<int-twitter:search-inbound-channel-adapter
    query="#springintegration"
  channel="inboundMentionsChannel">
     <int:poller fixed-rate="5000" max-messages-per-poll="3"/>
</int-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.social.twitter.api.Tweet` simply invoke the `getText()` method.

28.5 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.

```
<int-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.

```
<int-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>.

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

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

29. Web Services Support

29.1 Outbound Web Service Gateways

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

```
simpleGateway = new SimpleWebServiceOutboundGateway(destinationProvider);

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

Note

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

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

29.2 Inbound Web Service Gateways

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

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

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

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

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

29.3 Web Service Namespace Support

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

```
<int-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":

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

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

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

And for inbound:

```
<int-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.

29.4 Outbound URI Configuration

For all URI-schemes supported by Spring Web Services (URIs and Transports) `<uri-variable/>` substitution is provided:

```
<ws:outbound-gateway id="gateway" request-channel="input"
        uri="http://springsource.org/{foo}-{bar}">
    <ws:uri-variable name="foo" expression="payload.substring(1,7)"/>
    <ws:uri-variable name="bar" expression="headers.x"/>
</ws:outbound-gateway>

<ws:outbound-gateway request-channel="inputJms"
        uri="jms:{destination}?deliveryMode={deliveryMode}&amp;priority={priority}"
        message-sender="jmsMessageSender">
    <ws:uri-variable name="destination" expression="headers.jmsQueue"/>
    <ws:uri-variable name="deliveryMode" expression="headers.deliveryMode"/>
    <ws:uri-variable name="priority" expression="headers.jms_priority"/>
</ws:outbound-gateway>
```

If a `DestinationProvider` is supplied, variable substitution is not supported and a configuration error will result if variables are provided.

30. XML Support - Dealing with XML Payloads

30.1 Introduction

Spring Integration's XML support extends the core of Spring Integration with the following components:

- *Marshalling Transformer*
- *Unmarshalling Transformer*
- *Xslt Transformer*
- *XPath Transformer*
- *XPath Splitter*
- *XPath Router*
- *XPath Header Enricher*
- *XPath Filter*
- *Validating Filter*

These components are 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. In cases where you require greater control over document creation, you can provide an appropriately configured instance of `DocumentBuilder`.

30.2 Namespace Support

All components within the Spring Integration XML module provide namespace support. In order to enable namespace support, you need to import the respective schema for the Spring Integration XML Module. A typical setup is shown below:

```
<?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:int="http://www.springframework.org/schema/integration"
  xmlns:int-xml="http://www.springframework.org/schema/integration/xml"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/xml
    http://www.springframework.org/schema/integration/xml/spring-integration-xml.xsd">
</beans>
```

XPath Expressions

Many of the components within the Spring Integration XML module work with XPath Expressions. Each of those components will either reference an XPath Expression that has been defined as top-level element or via a nested `<xpath-expression/>` element.

All forms of XPath expressions result in the creation of an `XPathExpression` using the Spring `org.springframework.xml.xpath.XPathExpressionFactory`. When creating XPath expressions, the best XPath implementation that is available on the classpath is being used, either JAXP 1.3+ or Jaxen, whereby JAXP is preferred.

Note

Spring Integration under the covers uses the XPath functionality as provided by the *Spring Web Services* project (http://www.springsource.org/spring-web-services). Specifically, Spring Web Services' XML module (spring-xml-x.x.x.jar) is being used. Therefore, for a deeper understanding, please refer to the respective documentation as well at:

- http://static.springsource.org/spring-ws/sites/2.0/reference/html/common.html#xpath

Here is an overview of all available configuration parameters of the `xpath-expression` element:

```
<int-xml:xpath-expression expression=""❶
          id=""             ❷
          namespace-map=""  ❸
          ns-prefix=""      ❹
          ns-uri="">        ❺
    <map></map>             ❻
</int-xml:xpath-expression>
```

❶ Defines an XPath xpression. *Required*.

❷ The Identifier of the underlying bean definition. Will be an instance of `org.springframework.xml.xpath.XPathExpression` *Optional*.

❸ Reference to a map containing namespaces. The key of the map defines the namespace prefix and the value of the map sets the namespace URI. It is not valid to specify both this attribute and the `map` sub element, or setting the `ns-prefix` and `ns-uri` attribute. *Optional*.

❹ Allows you to set the namspace prefix directly as and attribute on the XPath expression element. If you set `ns-prefix`, you must also set the `ns-uri` attribute. *Optional*.

❺ Allows you to set the namspace URI directly as an attribute on the XPath expression element. If you set `ns-uri`, you must also set the `ns-prefix` attribute. *Optional*.

❻ Defines a map containing namespaces. Only one map child element is allowed. The key of the map defines the namespace prefix and the value of the map sets the namespace URI.

It is not valid to specify both this sub-element and the `map` attribute, or setting the `ns-prefix` and `ns-uri` attributes. *Optional*.

Providing Namespaces (Optional) to XPath Expressions

For the XPath Expression Element, namespace information can be optionally provided as configuration parameters. As such, namespaces can be defined using one of the following 3 choices:

- Reference a map using the `namespace-map` attribute
- Provide a map of namespaces using the `map` sub-element

- Specifying the `ns-prefix` and the `ns-uri` attribute

All three options are mutially exlusive. Only one option can be set.

Below, please find several different usage examples on how to use XPath expressions using the XML namespace support including the various option for setting the XML namespaces as discussed above.

```
<int-xml:xpath-filter id="filterReferencingXPathExpression"
                      xpath-expression-ref="refToXpathExpression"/>

<int-xml:xpath-expression id="refToXpathExpression" expression="/name"/>

<int-xml:xpath-filter id="filterWithoutNamespace">
   <int-xml:xpath-expression expression="/name"/>
</int-xml:xpath-filter>

<int-xml:xpath-filter id="filterWithOneNamespace">
    <int-xml:xpath-expression expression="/ns1:name"
                              ns-prefix="ns1" ns-uri="www.example.org"/>
</int-xml:xpath-filter>

<int-xml:xpath-filter id="filterWithTwoNamespaces">
    <int-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>
    </int-xml:xpath-expression>
</int-xml:xpath-filter>

<int-xml:xpath-filter id="filterWithNamespaceMapReference">
    <int-xml:xpath-expression expression="/ns1:name/ns2:type"
                              namespace-map="defaultNamespaces"/>
</int-xml:xpath-filter>

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

Using XPath Expressions with Default Namespaces

When working with default nanmespaces, you may run into situations that behave differently than originally expected. Let's assume we have the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<order>
 <orderItem>
  <isbn>0321200683</isbn>
  <quantity>2</quantity>
 </orderItem>
 <orderItem>
  <isbn>1590596439</isbn>
  <quantity>1</quantity>
 </orderItem>
</order>
```

This document is not declaring any namespace. Therefore, applying the following XPath Expression will work as expected:

```
<int-xml:xpath-expression expression="/order/orderItem" />
```

You might expect that the same expression will also work for the following XML file. It looks exactly the same as the previous example but in addition it also declares a default namespace:

http://www.example.org/orders

```
<?xml version="1.0" encoding="UTF-8"?>
<order xmlns="http://www.example.org/orders">
 <orderItem>
  <isbn>0321200683</isbn>
  <quantity>2</quantity>
 </orderItem>
 <orderItem>
  <isbn>1590596439</isbn>
  <quantity>1</quantity>
 </orderItem>
</order>
```

However, the XPath Expression used previously will fail in this case.

In order to solve this issue, you must provide a namespace prefix and a namespace URI using either the *ns-prefix* and *ns-uri* attibute or by providing a *namespace-map* attribute instead. The namespace URI must match the namespace declared in your XML document, which in this example is *http://www.example.org/orders*.

The namespace prefix, however, can be arbitrarily chosen. In fact, just providing an empty String will actually work (Null is not allowed). In the case of a namespace prefix consisting of an empty String, your Xpath Expression will use a colon (":") to indicate the default namespace. If you leave the colon off, the XPath expression will not match. The following XPath Expression will match agains the XML document above:

```
<si-xml:xpath-expression expression="/:order/:orderItem"
    ns-prefix="" ns-uri="http://www.example.org/prodcuts"/>
```

Of course you can also provide any other arbitrarily chosen namespace prefix. The following XPath expression using the *myorder* namespace prefix will match also:

```
<si-xml:xpath-expression expression="/myorder:order/myorder:orderItem"
    ns-prefix="myorder" ns-uri="http://www.example.org/prodcuts"/>
```

It is important to remember that the namespace URI is the really important piece of information to declare, not the prefix itself. The Jaxen FAQ summarizes the point very well:

“ In XPath 1.0, all unprefixed names are unqualified. There is no requirement that the prefixes used in the XPath expression are the same as the prefixes used in the document being queried. Only the namespace URIs need to match, not the prefixes. ”

30.3 Transforming XML Payloads

Configuring Transformers as Beans

This section will explain the workings of the following transformers and how to configure them as *beans*:

- `UnmarshallingTransformer`

- `MarshallingTransformer`
- `XsltPayloadTransformer`

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

An `UnmarshallingTransformer` allows an XML `Source` to be unmarshalled using implementations of the Spring OXM `Unmarshaller`. Spring's Object/XML Mapping support provides several implementations supporting marshalling and unmarshalling using JAXB, Castor and JiBX amongst others. The unmarshaller requires an instance of `Source`. If the message payload is not an instance of `Source`, conversion will be attempted. Currently `String`, `File` and `org.w3c.dom.Document` payloads are supported. Custom conversion to a `Source` is also supported by injecting an implementation of a `SourceFactory`.

Note

If a `SourceFactory` is not set explicitly, the property on the `UnmarshallingTransformer` will by default be set to a `DomSourceFactory`.

```
<bean id="unmarshallingTransformer" class="o.s.i.xml.transformer.UnmarshallingTransformer">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="contextPath" value="org.example" />
        </bean>
    </constructor-arg>
</bean>
```

MarshallingTransformer

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="o.s.i.xml.transformer.MarshallingTransformer">
    <constructor-arg>
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="contextPath" value="org.example"/>
        </bean>
    </constructor-arg>
    <constructor-arg>
        <bean class="o.s.i.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

`XsltPayloadTransformer` transforms XML payloads using Extensible Stylesheet Language Transformations (XSLT). The transformer's constructor requires an instance of either `Resource` or `Templates` to be passed in. Passing in a `Templates` instance allows for greater configuration of the `TransformerFactory` used to create the template instance.

As with the `UnmarshallingTransformer`, the `XsltPayloadTransformer` will do the actual XSLT transformation using instances of `Source`. Therefore, if the message payload is not an instance of `Source`, conversion will be attempted. `String` and `Document` payloads are supported directly.

Custom conversion to a `Source` is also supported by injecting an implementation of a `SourceFactory`.

Note

If a `SourceFactory` is not set explicitly, the property on the `XsltPayloadTransformer` will by default be set to a `DomSourceFactory`.

By default, the `XsltPayloadTransformer` will create a message with a `Result` payload, similar to the `XmlPayloadMarshallingTransformer`. This can be customised by providing a `ResultFactory` and/or a `ResultTransformer`.

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

ResultTransformers

Both the `MarshallingTransformer` and the `XsltPayloadTransformer` allow you to specify a `ResultTransformer`. Thus, if the Marshalling or XSLT transformation returns a `Result`, than you have the option to also use a `ResultTransformer` to transform the `Result` into another format. Spring Integration provides 2 concrete `ResultTransformer` implementations:

- ResultToDocumentTransformer
- ResultToStringTransformer

Using ResultTransformers with the MarshallingTransformer

By default, the *MarshallingTransformer* will always return a `Result`. By specifying a `ResultTransformer`, you can customize the type of payload returned.

Using ResultTransformers with the XsltPayloadTransformer

The behavior is slighly more complex for the *XsltPayloadTransformer*. By default, if the input payload is an instance of `String` or `Document` the *resultTransformer* property is ignored.

However, if the input payload is a `Source` or any other type, then the *resultTransformer* property is applied. Additionally, you can set the property *alwaysUseResultFactory* to `true`, which will also cause the specified *resultTransformer* to being used.

For more information and examples, please see the section called “Namespace Configuration and ResultTransformers”

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.

UnmarshallingTransformer

The namespace support for the `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.

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

<int-xml:unmarshalling-transformer id="unmarshallerWithPoller"
    input-channel="input" output-channel="output"
    unmarshaller="unmarshaller">
    <int:poller fixed-rate="2000"/>
<int-xml:unmarshalling-transformer/>
```

MarshallingTransformer

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).

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

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

<bean id="resultTransformer" class="o.s.i.xml.transformer.ResultToStringTransformer"/>
```

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. The attributes *result-type* and *result-factory* are mutually exclusive.

Note

Internally, the result types `StringResult` and `DomResult` are represented by the `ResultFactorys` `StringResultFactory` and `DomResultFactory` respectively.

XsltPayloadTransformer

Namespace support for the `XsltPayloadTransformer` allows you to either pass in a `Resource`, in order to create the `Templates` instance, or alternatively, you can pass in a precreated `Templates` instance 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.

Important

If you specify the `result-factory` or the `result-type` attribute, then the `alwaysUseResultFactory` property on the underlying `XsltPayloadTransformer` will be set to `true` by the `XsltPayloadTransformerParser`.

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

<int-xml:xslt-transformer id="xsltTransformerWithTemplatesAndResultTransformer"
    input-channel="withTemplatesAndResultTransformerIn" output-channel="output"
    xsl-templates="templates"
    result-transformer="resultTransformer"/>
```

Often you may need to have access to Message data, such as the Message Headers, in order to assist with transformation. 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, as illustrated in following example:

```
<int-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 the *<xslt-param/>* sub element. There you can use either the `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 values. 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.

Namespace Configuration and ResultTransformers

The usage of `ResultTransformers` was previously introduced in the section called "ResultTransformers". The following example illustrates several special use-cases using XML namespace configuration. First, we define the `ResultTransformer`:

```
<beans:bean id="resultToDoc" class="o.s.i.xml.transformer.ResultToDocumentTransformer"/>
```

This `ResultTransformer` will accept either a `StringResult` or a `DOMResult` as input and converts the input into a `Document`.

Now, let's declare the transformer:

```
<int-xml:xslt-transformer input-channel="in" output-channel="fahrenheitChannel"
    xsl-resource="classpath:noop.xslt" result-transformer="resultToDoc"/>
```

If the incoming message's payload is of type `Source`, then as first step the `Result` is determined using the `ResultFactory`. As we did not specify a `ResultFactory`, the default `DomResultFactory` is used, meaning that the transformation will yield a `DomResult`.

However, as we specified a *ResultTransformer*, it will be used and the resulting Message payload will be of type `Document`.

Important

If the incoming message's payload is of type `String`, the payload after the Xslt transformation will be a String. Similarly, if the incoming message's payload is of type `Document`, the payload after the Xslt transformation will be a `Document`. The specified *ResultTransformer* will be ignored with `String` or `Document` payloads.

If the message payload is neither a `Source`, `String` or `Document`, as a fallback option, it is attempted to create a `Source` using the default `SourceFactory`. As we did not specify a `SourceFactory` explicitly using the *source-factory* attribute, the default `DomSourceFactory` is used. If successful, the XSLT transformation is executed as if the payload was of type `Source`, which we described in the previous paragraphs.

Note

The `DomSourceFactory` supports the creation of a `DOMSource` from a either `Document`, `File` or `String` payloads.

The next transformer declaration adds a *result-type* attribute using `StringResult` as its value. First, the *result-type* is internally represented by the `StringResultFactory`. Thus, you could have also added a reference to a `StringResultFactory`, using the *result-factory* attribute, which would haven been the same.

```
<int-xml:xslt-transformer input-channel="in" output-channel="fahrenheitChannel"
  xsl-resource="classpath:noop.xslt" result-transformer="resultToDoc"
  result-type="StringResult"/>
```

Because we are using a `ResultFactory`, the *alwaysUseResultFactory* property of the `XsltPayloadTransformer` class will be implicitly set to `true`. Consequently, the referenced `ResultToDocumentTransformer` will be used.

Therefore, if you transform a payload of type `String`, the resulting payload will be of type `Document`.

XsltPayloadTransformer and <xsl:output method="text"/>

`<xsl:output method="text"/>` tells the XSLT template to only produce text content from the input source. In this particuliar case there is no reason to have a `DomResult`. Therefore, the `XsltPayloadTransformer` defaults to `StringResult` if the output property called `method` of the underlying `javax.xml.transform.Transformer` returns `"text"`. This coercion is performed independent from the inbound payload type. Keep in mind that this “smart” behavior is only available, if the `result-type` or `result-factory` attributes aren't provided for the respective `<int-xml:xslt-transformer>` component.

30.4 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:

```
<int-xml: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 the capability to perform simple conversion of extracted elements to a desired type. Valid return types are defined in `javax.xml.xpath.XPathConstants` and follows the conversion rules specified by the `javax.xml.xpath.XPath` interface.

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

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

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

<int-xml: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:

```
<int-xml: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 an implementation of the `org.springframework.integration.xml.XmlPayloadConverter` to provide more granular transformation:

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

. . . 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();
  }
}
```

The DefaultXmlPayloadConverter is used if this reference is not provided, and it should be sufficient in most cases since it can convert from Node, Document, Source, File, and String typed payloads. If you need to extend beyond the capabilities of that default implementation, then an upstream Transformer is probably a better option than providing a reference to a custom implementation of this strategy here.

30.5 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>
```

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 -->
<int-xml:xpath-splitter id="orderItemSplitter"
                        input-channel="orderChannel"
                        output-channel="orderItemsChannel">
    <int-xml:xpath-expression expression="/order/items"/>
</int-xml:xpath-splitter>

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

30.6 Routing XML Messages Using XPath

Similar to SpEL-based routers, Spring Integration provides support for routing messages based on XPath expressions, allowing you to create a Message Endpoint with an input channel but no output channel. Instead, one or more output channels are determined dynamically.

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

Note

For an overview of attributes that are common among Routers, please see chapter: *Common Router Parameters*

Internally XPath expressions will be evaluated as *NODESET* type and converted to a `List<String>` representing channel names. Typically such a list will contain a single channel name. However, based on the results of an XPath Expression, the XPath router can also take on the characteristics of a *Recipient List Router* if the XPath Expression returns more then one value. In that case, the `List<String>` will contain more then one channel name and consequently Messages will be sent to all channels in the list.

Thus, 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-->
<int-xml:xpath-router id="responderRouter" input-channel="orderChannel">
    <int-xml:xpath-expression expression="/request/responders"/>
</int-xml:xpath-router>
```

If the returned values do not represent the channel names directly, additional mapping parameters can be specified, in order to map those returned values to actual channel names. 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 configuration such as the following:

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

As already mentioned, the default evaluation type for XPath expressions is *NODESET*, which is converted to a `List<String>` of channel names, therefore handling single channel scenarios as well as multiple ones.

Nonetheless, certain XPath expressions may evaluate as String type from the very beginning. Take for example the following XPath Expression:

```
name(./node())
```

This expression will return the name of the root node. It will resulting in an exception, if the default evaluation type *NODESET* is being used.

For these scenarious, you may use the `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.

Note

To provide some background information: XPath 1.0 specifies 4 data types:

- Node-sets
- Strings
- Number
- Boolean

When the XPath Router evaluates expressions using the optional `evaluate-as-string` attribute, the return value is determined per the `string()` function as defined in the XPath specification. This means that if the expression selects multiple nodes, it will return the string value of the first node.

For further information, please see:

- Specification: XML Path Language (XPath) Version 1.0: http://www.w3.org/TR/xpath/
- XPath specification - string() function: http://www.w3.org/TR/xpath/#function-string

For example if we want to route based on the name of the root node, we can 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>
```

XML Payload Converter

For XPath Routers, you can also specify the Converter to use when converting payloads prior to XPath evaluation. As such, the XPath Router supports custom implementations of the `XmlPayloadConverter` strategy, and when configuring an `xpath-router` element in XML, a reference to such an implementation may be provided via the `converter` attribute.

If this reference is not explicitly provided, the `DefaultXmlPayloadConverter` is used. It should be sufficient in most cases, since it can convert from Node, Document, Source, File, and String typed payloads. If you need to extend beyond the capabilities of that default implementation, then an upstream Transformer is generally a better option in most cases, rather than providing a reference to a custom implementation of this strategy here.

30.7 XPath Header Enricher

The XPath Header Enricher defines a Header Enricher Message Transformer that evaluates XPath expressions against the message payload and inserts the result of the evaluation into a messsage header.

Please see below for an overview of all available configuration parameters:

```
<int-xml:xpath-header-enricher default-overwrite="true"   ❶
                           id=""                          ❷
                           input-channel=""               ❸
                           output-channel=""              ❹
                           should-skip-nulls="true">      ❺
    <int:poller></int:poller>                             ❻
    <int-xml:header name=""                               ❼
                    evaluation-type="STRING_RESULT"       ❽
                    overwrite="true"                      ❾
                    xpath-expression=""                   ❿
                    xpath-expression-ref=""/>             11
</int-xml:xpath-header-enricher>
```

❶ Specify the default boolean value for whether to overwrite existing header values. This will only take effect for sub-elements that do not provide their own 'overwrite' attribute. If the 'default- overwrite' attribute is not provided, then the specified header values will NOT overwrite any existing ones with the same header names. *Optional*.

❷ Id for the underlying bean definition. *Optional*.

❸ The receiving Message channel of this endpoint. *Optional*.

❹ Channel to which enriched messages shall be send to. *Optional*.

❺ Specify whether null values, such as might be returned from an expression evaluation, should be skipped. The default value is true. Set this to false if a null value should trigger removal of the corresponding header instead. *Optional*.

❻ *Optional*.

❼ The name of the header to be enriched. *Mandatory*.

❽ The result type expected from the XPath evaluation. This will be the type of the header value. The following values are allowed: BOOLEAN_RESULT, STRING_RESULT, NUMBER_RESULT, NODE_RESULT and NODE_LIST_RESULT. Defaults internally to `XPathEvaluationType.STRING_RESULT` if not set. *Optional*.

❾ Boolean value to indicate whether this header value should overwrite an existing header value for the same name if already present on the input Message.

❿ The XPath Expression as a String. Either this attribute or `xpath-expression-ref` must be provided, but not both.

11 The XPath Expression reference. Either this attribute or `xpath-expression` must be provided, but not both.

30.8 Using the XPath Filter

This component defines an XPath-based Message Filter. Under the covers this components uses a `MessageFilter` that wraps an instance of `AbstractXPathMessageSelector`.

Note

Please also refer to the chapter on Message Filters for further details.

In order to use the XPath Filter you must as a minimum provide an XPath Expression either by declaring the `xpath-expression` sub-element or by referencing an XPath Expression using the `xpath-expression-ref` attribute.

If the provided XPath expression will evaluate to a `boolean` value, no further configuration parameters are necessary. However, if the XPath expression will evaluate to a String, the `match-value` attribute should be specified against which the evaluation result will be matched.

There are three options for the `match-type`:

- **exact** - correspond to `equals` on `java.lang.String`. The underlying implementation uses a `StringValueTestXPathMessageSelector`
- **case-insensitive** - correspond to `equals-ignore-case` on `java.lang.String`. The underlying implementation uses a `StringValueTestXPathMessageSelector`
- **regex** - matches operations one `java.lang.String`. The underlying implementation uses a `RegexTestXPathMessageSelector`

When providing a 'match-type' value of 'regex', the value provided with thos `match-value` attribute must be a valid Regular Expression.

```
<int-xml:xpath-filter discard-channel=""                          ❶
                      id=""                                       ❷
                      input-channel=""                            ❸
                      match-type="exact"                          ❹
                      match-value=""                              ❺
                      output-channel=""                           ❻
                      throw-exception-on-rejection="false"        ❼
                      xpath-expression-ref="">                    ❽
    <int-xml:xpath-expression ... />                              ❾
    <int:poller ... />                                            ❿
</int-xml:xpath-filter>
```

❶ Message Channel where you want rejected messages to be sent. *Optional*.

❷ Id for the underlying bean definition. *Optional*.

❸ The receiving Message channel of this endpoint. *Optional*.

❹ Type of match to apply between the XPath evaluation result and the *match-value*. Default is *exact*. *Optional*.

❺ String value to be matched against the XPath evaluation result. If this attribute is not provided, then the XPath evaluation MUST produce a boolean result directly. *Optional*.

❻ The channel to which Messages that matched the filter criterias shall be dispatched to. *Optional*.

❼ By default, this property is set to *false* and rejected Messages (Messages that did not match the filter criteria) will be silently dropped. However, if set to *true* message rejection will result in an error condition and the exception will be propagated upstream to the caller. *Optional*.

❽ Reference to an XPath expression instance to evaluate.

❾ This sub-element sets the XPath expression to be evaluated. If this is not defined you MUST define the `xpath-expression-ref` attribute. Also, only one `xpath-expression` element can be set.

❿ *Optional*.

30.9 XML Validating Filter

The XML Validating Filter allows you to validate incoming messages against provided schema instances. The following schema types are supported:

- xml-schema (http://www.w3.org/2001/XMLSchema)
- relax-ng (http://relaxng.org/ns/structure/1.0)

Messages that fail validation can either be silently dropped or they can be forwarded to a definable `discard-channel`. Furthermore you can configure this filter to throw an `Exception` in case validation fails.

Please see below for an overview of all available configuration parameters:

```
<int-xml:validating-filter discard-channel=""                     ❶
                           id=""                                  ❷
                           input-channel=""                       ❸
                           output-channel=""                      ❹
                           schema-location=""                     ❺
                           schema-type="xml-schema"               ❻
                           throw-exception-on-rejection="false"   ❼
                           xml-validator="">                      ❽
    <int:poller .../>                                             ❾
</int-xml:validating-filter>
```

❶ Message Channel where you want rejected messages to be sent. *Optional*.
❷ Id for the underlying bean definition. *Optional*.
❸ The receiving Message channel of this endpoint. *Optional*.
❹ Message Channel where you want accepted messages to be sent. *Optional*.
❺ Sets the location of the schema to validate the Message's payload against. Internally uses the `org.springframework.core.io.Resource` interface. You can set this attribute or the `xml-validator` attribute but not both. *Optional*.
❻ Sets the schema type. Can be either `xml-schema` or `relax-ng`. *Optional*. If not set it defaults to `xml-schema` which internally translates to `org.springframework.xml.validation.XmlValidatorFactory#SCHEMA_W3C_XML`
❼ If `true` a `MessageRejectedException` is thrown in case validation fails for the provided Message's payload. *Optional*. Defaults to `false` if not set.
❽ Reference to a custom `sorg.springframework.xml.validation.XmlValidator` strategy. You can set this attribute or the `schema-location` attribute but not both. *Optional*.
❾ *Optional*.

31. XMPP Support

Spring Integration provides Channel Adapters for XMPP.

31.1 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* and *ejabberd*

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:int-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.xsd"
```

31.2 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.

```
<int-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.

31.3 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.

```
<int-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>
```

31.4 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.

31.5 Advanced Configuration

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="o.s.i.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

Appendix A. Spring Expression Language (SpEL)

A.1 Introduction

Many Spring Integration components can be configured using expressions. These expressions are written in the Spring Expression Language.

In most cases, the *#root* object is the `Message` which, of course, has two properties - `headers` and `payload` - allowing such expressions as `payload`, `payload.foo`, `headers['my.header']` etc.

In some cases, additional variables are provided, for example the `<int-http:inbound-gateway/>` provides `#requestParams` (parameters from the HTTP request) and `#pathVariables` (values from path placeholders in the URI).

For all SpEL expressions, a `BeanResolver` is available, enabling references to any bean in the application context. For example `@myBean.foo(payload)`. In addition, two `PropertyAccessors` are available; a `MapAccessor` enables accessing values in a `Map` using a key, and a `ReflectivePropertyAccessor` which allows access to fields and or JavaBean compliant properties (using getters and setters). This is how the `Message` headers and payload properties are accessible.

A.2 SpEL Evaluation Context Customization

Starting with Spring Integration 3.0, it is possible to add additional `PropertyAccessors` to the SpEL evaluation context. In fact, the framework provides one such accessor, the `JsonPropertyAccessor` which can be used (read-only) to access fields from a `JsonNode`, or JSON in a `String`. Or you can create your own `PropertyAccessor` if you have specific needs.

In addition, custom functions can be added. Custom functions are `static` methods declared on a class. Functions and property accessors are available in any SpEL expression used throughout the framework.

To configure your custom accessors and functions, add an `IntegrationEvaluationContextFactoryBean` with `id="integrationEvaluationContext"` to your application context, with the appropriate configuration; for example:

```
<beans:bean id="integrationEvaluationContext"
   class="org.springframework.integration.config.IntegrationEvaluationContextFactoryBean">
 <property name="propertyAccessors">
  <list>
   <bean class="foo.MyCustomPropertyAccessor"/>
  </list>
 </property>
 <property name="functions">
  <map>
   <entry key="barcalc" value="#{T(foo.MyFunctions).getMethod('calc', T(foo.MyBar))}"/>
  </map>
 </property>
</bean>
```

This factory bean definition will override the default `integrationEvaluationContext` bean definition, adding the custom accessor to the list (which also includes the standard accessors mentioned above), and one custom function.

Note that custom functions are static methods. In the above example, the custom function is a static method `calc` on class `MyFunctions` and takes a single parameter of type `MyBar`.

Say you have a `Message` with a payload that has a type `MyFoo` on which you need to perform some action to create a `MyBar` object from it, and you then want to invoke a custom function `calc` on that object.

The standard property accessors wouldn't know how to get a `MyBar` from a `MyFoo` so you could write and configure a custom property accessor to do so. So, your final expression might be `"#barcalc(payload.myBar)"`.

A.3 SpEL Functions

Namespace support is provided for easy addition of SpEL custom functions. You can specify `<spel-function/>` components to provide custom SpEL functions to the `EvaluationContext` used throughout the framework. Instead of configuring the factory bean above, simply add one or more of these components and the framework will automatically add them to the default *integrationEvaluationContext* factory bean.

For example, assuming we have a useful static method to evaluate XPath:

```
<int:spel-function id="xpath"
 class="com.foo.test.XPathUtils" method="evaluate(java.lang.String, java.lang.Object)"/>

<int:transformer input-channel="in" output-channel="out"
   expression="#xpath('//foo/@bar', payload)" />
```

With this sample:

- The default `IntegrationEvaluationContextFactoryBean` bean with id *integrationEvaluationContext* is registered with the application context.
- The `<spel-function/>` is parsed and added to the `functions` Map of *integrationEvaluationContext* as map entry with `id` as the key and the static `Method` as the value.
- The *integrationEvaluationContext* factory bean creates a new `StandardEvaluationContext` instance, and it is configured with the default `PropertyAccessors`, `BeanResolver` and the custom function.
- That `EvaluationContext` instance is injected into the `ExpressionEvaluatingTransformer` bean.

Note

SpEL functions declared in a parent context are also made available in any child context(s). Each context has its own instance of the *integrationEvaluationContext* factory bean because each needs a different `BeanResolver`, but the function declarations are inherited (and can be overridden if needed by declaring a SpEL function with the same name. The functions themselves are processed by the framework - they do not appear as beans in the application context.

Note

At this time, `PropertyAccessors` are not inherited and must be declared as described above in each context.

Appendix B. 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.

B.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:

```
<int: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:

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

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

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

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

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

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

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

Appendix C. Transaction Support

C.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:

```
<int: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.

```
<int: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 you 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.

C.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.

C.3 Transaction Synchronization

In some environments, it is advantageous to synchronize operations with a transaction that encompasses the entire flow. For example, consider a <file:inbound-channel-adapter/> at the start of a flow, that performs a number of database updates. If the transaction commits, we might want to move the file to a *success* directory, while we might want to move it to a *failures* directory if the transaction rolls back.

Spring Integration 2.2 introduces the capability of synchronizing these operations with a transaction. In addition, you can configure a `PseudoTransactionManager` if you don't have a 'real' transaction, but still want to perform different actions on success, or failure. For more information, see Section C.4, “Pseudo Transactions”.

Key strategy interfaces for this feature are

```
public interface TransactionSynchronizationFactory {

    TransactionSynchronization create(Object key);
}

public interface TransactionSynchronizationProcessor {

    void processBeforeCommit(IntegrationResourceHolder holder);

    void processAfterCommit(IntegrationResourceHolder holder);

    void processAfterRollback(IntegrationResourceHolder holder);

}
```

The factory is responsible for creating a TransactionSynchronization object. You can implement your own, or use the one provided by the framework: `DefaultTransactionSynchronizationFactory`. This implementation returns a `TransactionSynchronization` that delegates to a default implementation of `TransactionSynchronizationProcessor`, the `ExpressionEvaluatingTransactionSynchronizationProcessor`. This processor supports three SpEL expressions, *beforeCommitExpression*, *afterCommitExpression*, and *afterRollbackExpression*.

These actions should be self-explanatory to those familiar with transactions. In each case, the *#root* variable is the original `Message`; in some cases, other SpEL variables are made available, depending on the `MessageSource` being polled by the poller. For example, the `MongoDbMessageSource` provides the *#mongoTemplate* variable which references the message source's `MongoTemplate`; the `RedisStoreMessageSource` provides the *#store* variable which references the `RedisStore` created by the poll.

To enable the feature for a particular poller, you provide a reference to the `TransactionSynchronizationFactory` on the poller's <transactional/> element using the *synchronization-factory* attribute.

To simplify configuration of these components, namespace support for the default factory has been provided. Configuration is best described using an example:

```
<int-file:inbound-channel-adapter id="inputDirPoller"
    channel="someChannel"
    directory="/foo/bar"
    filter="filter"
    comparator="testComparator">
    <int:poller fixed-rate="5000">
        <int:transactional transaction-manager="transactionManager" synchronization-
factory="syncFactory" />
    </int:poller>
</int-file:inbound-channel-adapter>

<int:transaction-synchronization-factory id="syncFactory">
    <int:after-commit expression="payload.renameTo('/success/' +
 payload.name)" channel="committedChannel" />
    <int:after-rollback expression="payload.renameTo('/failed/' +
 payload.name)" channel="rolledBackChannel" />
</int:transaction-synchronization-factory>
```

The result of the SpEL evaluation is sent as the payload to either the *committedChannel* or *rolledBackChannel* (in this case, this would be `Boolean.TRUE` or `Boolean.FALSE` - the result of the `java.io.File.renameTo()` method call).

If you wish to send the entire payload for further Spring Integration processing, simply use the expression 'payload'.

Important

It is important to understand that this is simply synchronizing the actions with a transaction, it does not make a resource that is not inherently transactional actually transactional. Instead, the transaction (be it JDBC or otherwise) is started before the poll, and committed/rolled back when the flow completes, followed by the synchronized action.

It is also important to understand that if you provide a custom `TransactionSynchronizationFactory`, it is responsible for creating a resource synchronization that will cause the bound resource to be unbound automatically, when the transaction completes. The default `TransactionSynchronizationFactory` does this by returning a subclass of `ResourceHolderSynchronization`, with the default *shouldUnbindAtCompletion()* returning `true`.

In addition to the *after-commit* and *after-rollback* expressions, *before-commit* is also supported. In that case, if the evaluation (or downstream processing) throws an exception, the transaction will be rolled back instead of being committed.

C.4 Pseudo Transactions

Referring to the above section, you may be thinking it would be useful to take these 'success' or 'failure' actions when a flow completes, even if there is no 'real' transactional resources (such as JDBC) downstream of the poller. For example, consider a <file:inbound-channel-adapter/> followed by an <ftp:outbout-channel-adapter/>. Neither of these components is transactional but we might want to move the input file to different directories, based on the success or failure of the ftp transfer.

To provide this functionality, the framework provides a `PseudoTransactionManager`, enabling the above configuration even when there is no real transactional resource involved. If the flow completes normally, the *beforeCommit* and *afterCommit* synchronizations will be called, on failure the *afterRollback*

will be called. Of course, because it is not a real transaction there will be no actual commit or rollback. The pseudo transaction is simply a vehicle used to enable the synchronization features.

To use a `PseudoTransactionManager`, simply define it as a <bean/>, in the same way you would configure a real transaction manager:

```
<bean id="transactionManager" class="o.s.i.transaction.PseudoTransactionManager" />
```

Appendix D. Security in Spring Integration

D.1 Introduction

Spring Integration builds upon the Spring Security project to enable role based security checks to be applied to channel send and receive invocations.

D.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:int="http://www.springframework.org/schema/integration"
   xmlns:int-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.xsd
      http://www.springframework.org/schema/security
      http://www.springframework.org/schema/security/spring-security.xsd
      http://www.springframework.org/schema/integration
      http://www.springframework.org/schema/integration/spring-integration.xsd
      http://www.springframework.org/schema/integration/security
      http://www.springframework.org/schema/integration/security/spring-integration-
security.xsd">

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

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

Appendix E. Spring Integration Samples

E.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.

E.2 Where to get Samples

The Spring Integration Samples project is hosted on GitHub. You can find the repository at:

https://github.com/SpringSource/spring-integration-samples

In order to check out or *clone* (Git parlance) the samples, please make sure you have a Git client installed on your system. There are several GUI-based products available for many platforms, e.g. EGit for the Eclipse IDE. A simple Google search will help you find them. Of course you can also just use the command line interface for Git.

Note

If you need more information on how to install and/or use Git, please visit: http://git-scm.com/.

In order to checkout (clone in Git terms) the Spring Integration samples repository using the Git command line tool, issue the following commands:

```
$ git clone https://github.com/SpringSource/spring-integration-samples.git
```

That is all you need to do in order to clone the entire samples repository into a directory named *spring-integration-samples* within the working directory where you issued that *git* command. Since the samples repository is a live repository, you might want to perform periodic *pulls* (updates) to get new samples, as well as updates to the existing samples. In order to do so issue the following git *PULL* command:

```
$ git pull
```

E.3 Submitting Samples or Sample Requests

How can I contribute my own Samples?

Github is for social coding: if you want to submit your own code examples to the Spring Integration Samples project, we encourage contributions through *pull requests* from *forks* of this repository. If you want to contribute code this way, please reference, if possible, a *JIRA Ticket* that provides some details regarding the provided sample.

Sign the contributor license agreement

Very important: before we can accept your Spring Integration sample, we will need you to sign the SpringSource contributor license agreement (CLA). Signing the contributor's agreement does not grant anyone commit rights to the main repository, but it does mean that we can accept your contributions, and you will get an author credit if we do. In order to read and sign the CLA, please go to:

https://support.springsource.com/spring_committer_signup

As Project, please select *Spring Integration*. The Project Lead is *Mark Fisher*.

Code Contribution Process

For the actual code contribution process, please read the the *Contributor Guidelines* for Spring Integration, they apply for this project as well:

https://github.com/SpringSource/spring-integration/wiki/Contributor-Guidelines

This process ensures that every commit gets peer-reviewed. As a matter of fact, the core committers follow the exact same rules. We are gratefully looking forward to your Spring Integration Samples!

Sample Requests

As mentioned earlier, the *Spring Integration Samples* project has a dedicated JIRA Issue tracking system. To submit new sample requests, please visit our JIRA Issue Tracking system:

https://jira.springframework.org/browse/INTSAMPLES.

E.4 Samples Structure

Starting with Spring Integration 2.0, 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!

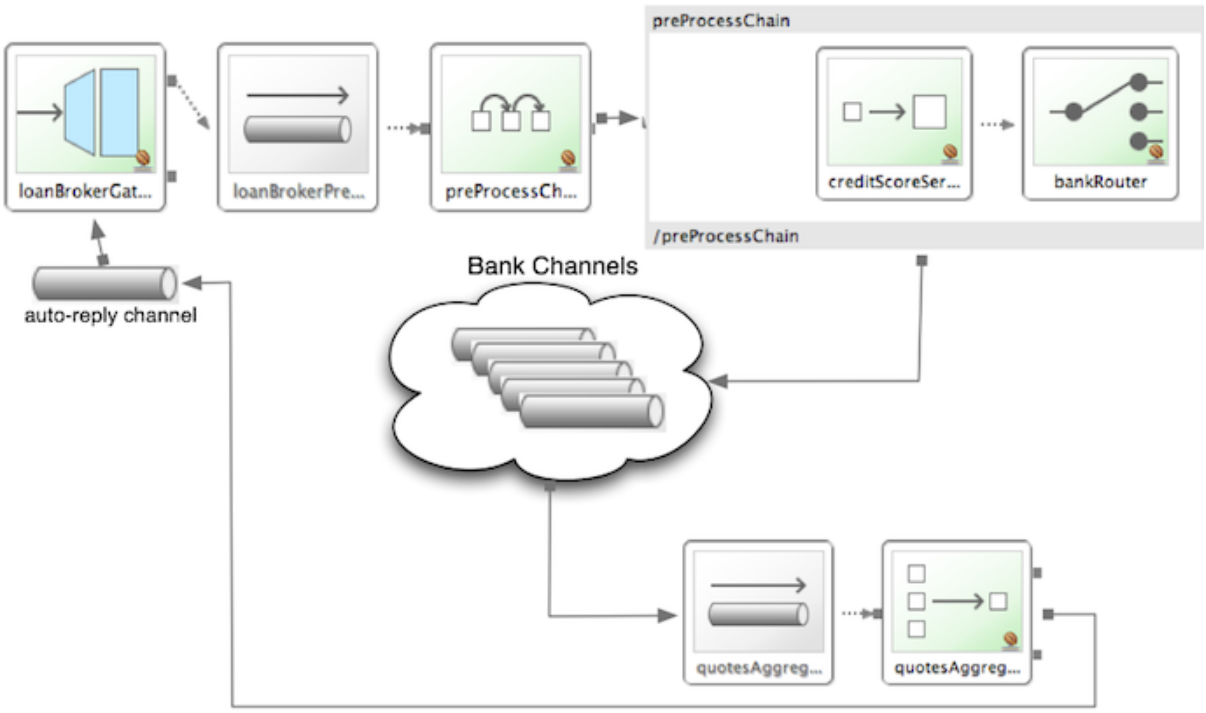
E.5 Samples

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.

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.

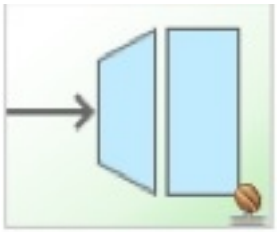
Obviously the real process of obtaining a loan quote is a bit more complex, but since our goal here is to demonstrate how Enterprise Integration Patterns are realized and implemented within SI, the use case has been simplified to concentrate only on the Integration aspects of the process. It is not an 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.

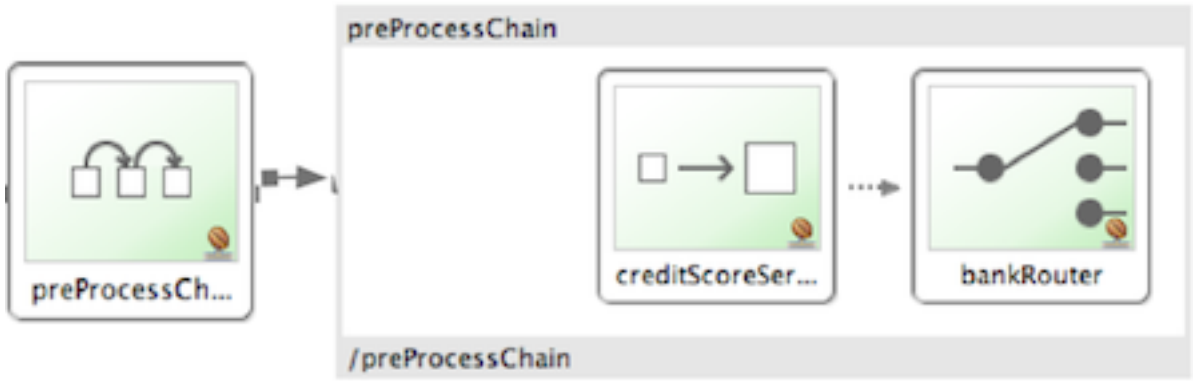
```
<int:gateway id="loanBrokerGateway"
  default-request-channel="loanBrokerPreProcessingChannel"
  service-
interface="org.springframework.integration.samples.loanbroker.LoanBrokerGateway">
  <int:method name="getBestLoanQuote">
    <int:header name="RESPONSE_TYPE" value="BEST"/>
  </int:method>
</int:gateway>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

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.

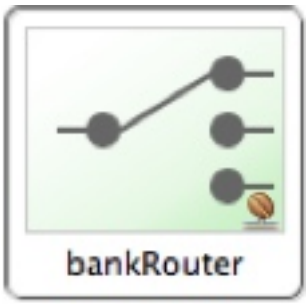
Composed Message Processor

The *Composed Message Processor* pattern describes rules around building endpoints that maintain control over message flow which consists of multiple message processors. In Spring 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.

```
<int:aggregator id="quotesAggregator"
      input-channel="quotesAggregationChannel"
      method="aggregateQuotes">
  <beans:bean class="org.springframework.integration.samples.loanbroker.LoanQuoteAggregator"/
>
</int: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

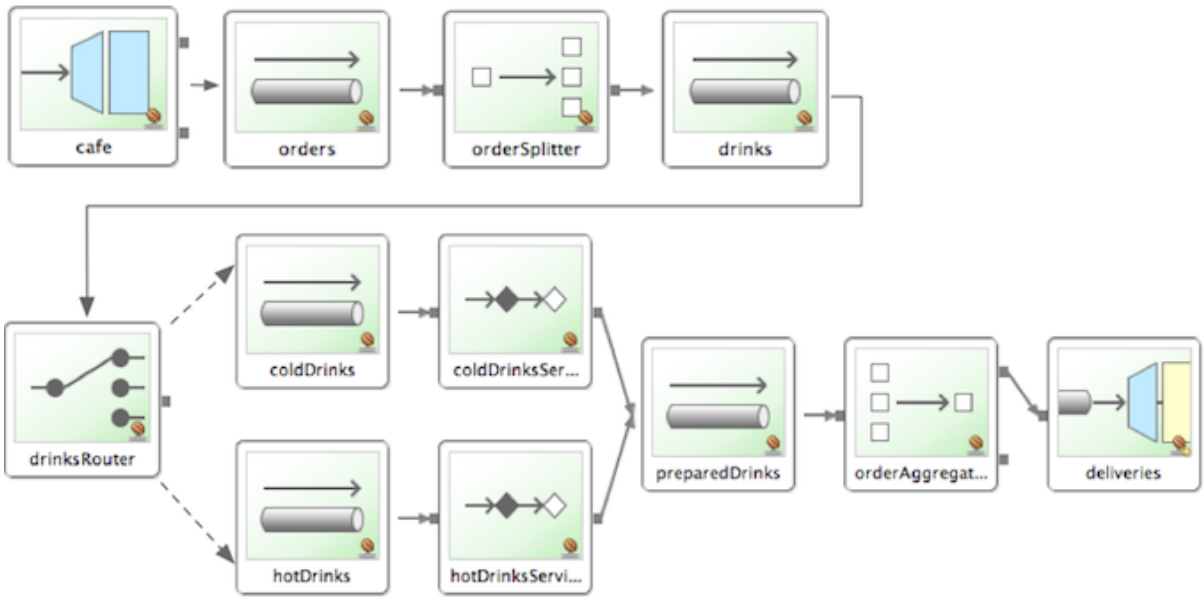
As you can see a rather complex process was assembled based on POJO (read existing, legacy), light weight, embeddable messaging framework (Spring Integration) with a loosely coupled programming model intended to simplify integration of heterogeneous systems without requiring a heavy-weight ESB-like engine 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 built on top of Enterprise Integration Patterns and can be considered "building blocks" for YOUR solution; they are not intended to be complete solutions. Integration concerns exist in all types of application (whether server based or not). It should not require change in design, testing and deployment strategy if such applications need to be integrated.

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.

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:int="http://www.springframework.org/schema/integration"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xmlns:int-stream="http://www.springframework.org/schema/integration/stream"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/integration
  http://www.springframework.org/schema/integration/spring-integration.xsd
  http://www.springframework.org/schema/integration/stream
  http://www.springframework.org/schema/integration/stream/spring-integration-stream.xsd">

    <int:gateway id="cafe" service-interface="o.s.i.samples.cafe.Cafe"/>

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

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

    <int:channel id="coldDrinks"><int:queue capacity="10"/></int:channel>
    <int:service-activator input-channel="coldDrinks" ref="barista"
                           method="prepareColdDrink" output-channel="preparedDrinks"/>

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

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

    <int-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="o.s.i.samples.cafe.xml.Barista"/>
    <beans:bean id="waiter"  class="o.s.i.samples.cafe.xml.Waiter"/>

    <int: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:

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

  <int:service-activator input-channel="hotDrinks"
                       ref="barista"
                       method="prepareHotDrink"
                       output-channel="preparedDrinks">
      <int:poller task-executor="pool" fixed-rate="1000"/>

  </int: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.

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

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

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

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

Appendix F. Configuration

F.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.

F.2 Namespace Support

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

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

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

You can choose any name after "xmlns:"; *int* 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.xsd
           http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration.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:int="http://www.springframework.org/schema/integration"
  xmlns:int-file="http://www.springframework.org/schema/integration/file"
  xmlns:int-jms="http://www.springframework.org/schema/integration/jms"
  xmlns:int-mail="http://www.springframework.org/schema/integration/mail"
  xmlns:int-rmi="http://www.springframework.org/schema/integration/rmi"
  xmlns:int-ws="http://www.springframework.org/schema/integration/ws"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/integration/spring-integration.xsd
    http://www.springframework.org/schema/integration/file
    http://www.springframework.org/schema/integration/file/spring-integration-file.xsd
    http://www.springframework.org/schema/integration/jms
    http://www.springframework.org/schema/integration/jms/spring-integration-jms.xsd
    http://www.springframework.org/schema/integration/mail
    http://www.springframework.org/schema/integration/mail/spring-integration-mail.xsd
    http://www.springframework.org/schema/integration/rmi
    http://www.springframework.org/schema/integration/rmi/spring-integration-rmi.xsd
    http://www.springframework.org/schema/integration/ws
    http://www.springframework.org/schema/integration/ws/spring-integration-ws.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.

F.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 consider. 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 relies on an instance of ThreadPoolTaskScheduler as described in the Task Execution and Scheduling section of the Spring Framework reference manual. That default TaskScheduler will startup automatically with a pool of 10 threads. If you provide your own TaskScheduler instance instead, you can set the 'autoStartup' property to *false*, and/or you can provide your own pool size value.

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.

Important

When running in a JEE container, you may need to use Spring's `TimerManagerTaskScheduler` as described here, instead of the default *taskScheduler*. To do that, simply define a bean with the appropriate JNDI name for your environment, for example:

```
<bean id="taskScheduler" class="o.s.scheduling.commonj.TimerManagerTaskScheduler">
    <property name="timerManagerName" value="tm/MyTimerManager" />
    <property name="resourceRef" value="true" />
</bean>
```

The next section will describe what happens if Exceptions occur within the asynchronous invocations.

F.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.

```
<int:channel id="errorChannel">
    <int:queue capacity="500"/>
</int: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.

F.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 it is itself annotated with Spring's `@Component` annotation and is therefore recognized automatically as a bean definition when using Spring component-scanning.

Even more important 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. Annotations available in Spring Integration include:

- @Aggregator
- @Filter
- @Router
- @ServiceActivator
- @Splitter
- @Transformer

The behavior of each is described in its own chapter or section within this reference.

Note

If you are using XML configuration in combination with annotations, the `@MessageEndpoint` annotation is not required. If you want to configure a POJO reference from the "ref" attribute of a <service-activator/> element, it is sufficient to provide the method-level annotations. In 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.

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 3.4, “Messaging Bridge”.

Also see the section called “Advising Endpoints Using Annotations”.

F.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:

```
<int:service-activator input-channel="input" output-channel="output" method="foo">
  <bean class="org.bar.Foo"/>
</int: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);
}
```

```
<int:service-activator input-channel="input" output-channel="output" method="bar">
  <bean class="org.bar.Foo"/>
</int:service-activator>
```

Now there is no ambiguity since the configuration explicitly maps to the 'bar' method which has no name conflicts.

Appendix G. Additional Resources

G.1 Spring Integration Home

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

Appendix H. Change History

H.1 Changes between 1.0 and 2.0

For a detailed migration guide in regards to upgrading an existing application that uses Spring Integration older than version 2.0, please see:

http://www.springsource.org/node/2976

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. 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 3.3, “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 15.3, “Http Outbound Gateway” for more details.

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 book.

Message History

We now provide support for the Message History 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 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 8.3, “Message Store”, 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 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 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 8.4, “Control Bus” and the section called “Control Bus” for more details.

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

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 28, *Twitter Adapter* for more details.

XMPP Adapters

The new XMPP adapters support both Chat Messages and Presence events. See Chapter 31, *XMPP Support* for more details.

FTP/FTPS Adapters

Inbound and outbound File transfer support over FTP/FTPS is now available. See Chapter 13, *FTP/FTPS Adapters* for more details.

SFTP Adapters

Inbound and outbound File transfer support over SFTP is now available. See Chapter 25, *SFTP Adapters* for more details.

Feed Adapters

We have also added Channel Adapters for receiving news feeds (ATOM/RSS). See Chapter 11, *Feed Adapter* for more details.

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 7.6, “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.

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, “Messaging Gateways”, Section 15.3, “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").

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.

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 We have also created many new samples, including samples for every new Adapter.

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, please download it here:

http://www.springsource.com/landing/best-development-tool-enterprise-java

H.2 Changes between 2.0 and 2.1

New Components

JSR-223 Scripting Support

In Spring Integration 2.0, support for Groovy was added. With Spring Integration 2.1 we expanded support for additional languages substantially by implementing support for JSR-223 (Scripting for the Java™ Platform). Now you have the ability to use any scripting language that supports JSR-223 including:

- Javascript
- Ruby/JRuby
- Python/Jython
- Groovy

For further details please see Section 7.5, “Scripting support”.

GemFire Support

Spring Integration provides support for GemFire by providing inbound adapters for entry and continuous query events, an outbound adapter to write entries to the cache, and `MessageStore` and `MessageGroupStore` implementations. Spring integration leverages the *Spring Gemfire* project, providing a thin wrapper over its components.

For further details please see Chapter 14, *GemFire Support*.

AMQP Support

Spring Integration 2.1 adds several Channel Adapters for receiving and sending messages using the *Advanced Message Queuing Protocol* (AMQP). Furthermore, Spring Integration also provides a point-to-point Message Channel, as well as a publish/subscribe Message Channel that are backed by AMQP Exchanges and Queues.

For further details please see Chapter 9, *AMQP Support*.

MongoDB Support

As of version 2.1 Spring Integration provides support for MongoDB by providing a MongoDB-based MessageStore.

For further details please see Chapter 21, *MongoDb Support*.

Redis Support

As of version 2.1 Spring Integration supports Redis, an advanced key-value store, by providing a Redis-based MessageStore as well as Publish-Subscribe Messaging adapters.

For further details please see Chapter 22, *Redis Support*.

Support for Spring's Resource abstraction

As of version 2.1, we've introduced a new *Resource Inbound Channel Adapter* that builds upon Spring's Resource abstraction to support greater flexibility across a variety of actual types of underlying resources, such as a file, a URL, or a class path resource. Therefore, it's similar to but more generic than the *File Inbound Channel Adapter*.

For further details please see Section 23.2, “Resource Inbound Channel Adapter”.

Stored Procedure Components

With Spring Integration 2.1, the `JDBC` Module also provides Stored Procedure support by adding several new components, including inbound/outbound channel adapters and an Outbound Gateway. The Stored Procedure support leverages Spring's `SimpleJdbcCall` class and consequently supports stored procedures for:

- Apache Derby
- DB2
- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL
- Sybase

The Stored Procedure components also support Sql Functions for the following databases:

- MySQL
- Microsoft SQL Server
- Oracle
- PostgreSQL

For further details please see Section 17.5, “Stored Procedures”.

XPath and XML Validating Filter

Spring Integration 2.1 provides a new XPath-based Message Filter, that is part of the `XML` module. The XPath Filter allows you to filter messages using provided XPath Expressions. Furthermore, documentation was added for the XML Validating Filter.

For more details please see Section 30.8, “Using the XPath Filter” and Section 30.9, “XML Validating Filter”.

Payload Enricher

Since Spring Integration 2.1, the Payload Enricher is provided. A Payload Enricher defines an endpoint that typically passes a `Message` to the exposed request channel and then expects a reply message. The reply message then becomes the root object for evaluation of expressions to enrich the target payload.

For further details please see the section called “Payload Enricher”.

FTP and SFTP Outbound Gateways

Spring Integration 2.1 provides two new Outbound Gateways in order to interact with remote File Transfer Protocol (FTP) or Secure File Transfer Protocol (SFT) servers. These two gateways allow you to directly execute a limited set of remote commands.

For instance, you can use these Outbound Gateways to list, retrieve and delete remote files and have the Spring Integration message flow continue with the remote server's response.

For further details please see Section 13.5, “FTP Outbound Gateway” and Section 25.6, “SFTP Outbound Gateway”.

FTP Session Caching

As of version 2.1, we have exposed more flexibility with regards to session management for remote file adapters (e.g., FTP, SFTP etc).

Specifically, the `cache-sessions` attribute, which is available via the XML namespace support, is now *deprecated*. Alternatively, we added the `sessionCacheSize` and `sessionWaitTimeout` attributes on the `CachingSessionFactory`.

For further details please see Section 13.6, “FTP Session Caching” and Section 25.3, “SFTP Session Caching”.

Framework Refactoring

Standardizing Router Configuration

Router parameters have been standardized across all router implementations with Spring Integration 2.1 providing a more consistent user experience.

With Spring Integration 2.1 the `ignore-channel-name-resolution-failures` attribute has been removed in favor of consolidating its behavior with the `resolution-required` attribute. Also, the `resolution-required` attribute now defaults to `true`.

Starting with Spring Integration 2.1, routers will no longer silently drop any messages, if no default output channel was defined. This means, that by default routers now require at least one resolved channel (if no `default-output-channel` was set) and by default will throw a `MessageDeliveryException` if no channel was determined (or an attempt to send was not successful).

If, however, you do desire to drop messages silently, simply set `default-output-channel="nullChannel"`.

Important

With the standardization of Router parameters and the consolidation of the parameters described above, there is the possibility of breaking older Spring Integration based applications.

For further details please see Section 5.1, “Routers”

XML Schemas updated to 2.1

Spring Integration 2.1 ships with an updated XML Schema (version 2.1), providing many improvements, e.g. the Router standardizations discussed above.

From now on, users *must* always declare the latest XML schema (currently version 2.1). Alternatively, they can use the version-less schema. Generally, the best option is to use version-less namespaces, as these will automatically use the latest available version of Spring Integration.

Declaring a version-less Spring Integration namespace:

```
<?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:int="http://www.springframework.org/schema/integration"
   xsi:schemaLocation="http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
...
</beans>
```

Declaring a Spring Integration namespace using an explicit version:

```
<?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:int="http://www.springframework.org/schema/integration"
   xsi:schemaLocation="http://www.springframework.org/schema/integration
           http://www.springframework.org/schema/integration/spring-integration-2.2.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">
...
</beans>
```

The old 1.0 and 2.0 schemas are still there, but if an Application Context still references one of those deprecated schemas, the validator will fail on initialization.

Source Control Management and Build Infrastructure

Source Code now hosted on Github

Since version 2.0, the Spring Integration project uses Git for version control. In order to increase community visibility even further, the project was moved from SpringSource hosted Git repositories to Github. The Spring Integration Git repository is located at: https://github.com/SpringSource/spring-integration/

For the project we also improved the process of providing code contributions and we ensure that every commit is peer-reviewed. In fact, core committers now follow the same process as contributors. For more details please see:

https://github.com/SpringSource/spring-integration/wiki/Contributor-Guidelines

Improved Source Code Visibility with Sonar

In an effort to provide better source code visibility and consequently to monitor the quality of Spring Integration's source code, an instance of Sonar was setup and metrics are gathered nightly and made avaiblable at:

https://sonar.springsource.org/

New Samples

For the 2.1 release of Spring Integration we also expanded the Spring Integration Samples project and added many new samples, e.g. samples covering AMQP support, the new payload enricher, a sample

illustrating techniques for testing Spring Integration flow fragments, as well as an example for executing Stored Procedures against Oracle. For details please visit:

https://github.com/SpringSource/spring-integration-samples

H.3 Changes between 2.1 and 2.2

New Components

RedisStore Inbound and Outbound Channel Adapters

Spring Integration now has RedisStore Inbound and Outbound Channel Adapters allowing you to write and read Message payloads to/from Redis collection(s). For more information please see Section 22.6, “RedisStore Outbound Channel Adapter” and Section 22.5, “RedisStore Inbound Channel Adapter”.

MongoDB Inbound and Outbound Channel Adapters

Spring Integration now has MongoDB Inbound and Outbound Channel Adapters allowing you to write and read Message payloads to/from a MongoDB document store. For more information please see Section 21.5, “MongoDB Outbound Channel Adapter” and Section 21.4, “MongoDB Inbound Channel Adapter”.

JPA Endpoints

Spring Integration now includes components for the Java Persistence API (JPA) for retrieving and persisting JPA entity objects. The JPA Adapter includes the following components:

- *Inbound Channel Adapter*
- *Outbound Channel Adapter*
- *Updating Outbound Gateway*
- *Retrieving Outbound Gateway*

For more information please see Chapter 18, *JPA Support*

General Changes

Spring 3.1 Used by Default

Spring Integration now uses Spring 3.1.

Adding Behavior to Endpoints

The ability to add an <advice-chain/> to a poller has been available for some time. However, the behavior added by this affects the entire integration flow. It did not address the ability to add, say, retry, to an individual endpoint. The 2.2. release introduces the <request-handler-advice-chain/> to many endpoints.

In addition, 3 standard Advice classes have been provided for this purpose:

- MessageHandlerRetryAdvice
- MessageHandlerCircuitBreakerAdvice
- ExpressionEvaluatingMessageHandlerAdvice

For more information, see Section 7.7, “Adding Behavior to Endpoints”.

Transaction Synchronization and Pseudo Transactions

Pollers can now participate in Spring's *Transaction Synchronization* feature. This allows for synchronizing such operations as renaming files by an inbound channel adapter depending on whether the transaction commits, or rolls back.

In addition, these features can be enabled when there is not a 'real' transaction present, by means of a `PseudoTransactionManager`.

For more information see Section C.3, “Transaction Synchronization”.

File Adapter - Improved File Overwrite/Append Handling

When using the *File Oubound Channel Adapter* or the *File Outbound Gateway*, a new *mode* property was added. Prior to *Spring Integration 2.2*, target files were replaced when they existed. Now you can specify the following options:

- REPLACE (Default)
- APPEND
- FAIL
- IGNORE

For more information please see the section called “Dealing with Existing Destination Files”.

Reply-Timeout added to more Outbound Gateways

The XML Namespace support adds the *reply-timeout* attribute to the following *Outbound Gateways*:

- Amqp Outbound Gateway
- File Outbound Gateway
- Ftp Outbound Gateway
- Sftp Outbound Gateway
- Ws Outbound Gateway

Spring-AMQP 1.1

Spring Integration now uses Spring AMQP 1.1. This enables several features to be used within a Spring Integration application, including...

- A fixed reply queue for the outbound gateway
- HA (mirrored) queues
- Publisher Confirms
- Returned Messages
- Support for Dead Letter Exchanges/Dead Letter Queues

JDBC Support - Stored Procedures Components

SpEL Support

When using the Stored Procedure components of the Spring Integration JDBC Adapter, you can now provide Stored Procedure Names or Stored Function Names using Spring Expression Language (SpEL).

This allows you to specify the Stored Procedures to be invoked at runtime. For example, you can provide Stored Procedure names that you would like to execute via Message Headers. For more information please see Section 17.5, “Stored Procedures”.

JMX Support

The Stored Procedure components now provide basic JMX support, exposing some of their properties as MBeans:

- Stored Procedure Name
- Stored Procedure Name Expression
- JdbcCallOperations Cache Statistics

JDBC Support - Outbound Gateway

When using the JDBC Outbound Gateway, the update query is no longer mandatory. You can now provide solely a select query using the request message as a source of parameters.

JDBC Support - Channel-specific Message Store Implementation

A new *Message Channel*-specific Message Store Implementation has been added, providing a more scalable solution using database-specific SQL queries. For more information please see: the section called “Backing Message Channels”.

Orderly Shutdown

A method `stopActiveComponents()` has been added to the IntegrationMBeanExporter. This allows a Spring Integration application to be shut down in an orderly manner, disallowing new inbound messages to certain adapters and waiting for some time to allow in-flight messages to complete.

JMS Oubound Gateway Improvements

The JMS Outbound Gateway can now be configured to use a `MessageListener` container to receive replies. This can improve performance of the gateway.

object-to-json-transformer

The `ObjectToJsonTransformer` now sets the *content-type* header to *application/json* by default. For more information see Section 6.1, “Transformer”.

HTTP Support

Java serialization over HTTP is no longer enabled by default. Previously, when setting a `expected-response-type` to a `Serializable` object, the `Accept` header was not properly set up. The `SerializingHttpMessageConverter` has now been updated to set the Accept header to `application/x-java-serialized-object`. However, because this could cause incompatibility

with existing applications, it was decided to no longer automatically add this converter to the HTTP endpoints.

If you wish to use Java serialization, you will need to add the `SerializingHttpMessageConverter` to the appropriate endpoints, using the `message-converters` attribute, when using XML configuration, or using the `setMessageConverters()` method.

Alternatively, you may wish to consider using JSON instead which is enabled by simply having `Jackson` on the classpath.