Spring Integration Reference Manual

Mark Fisher
Marius Bogoevici

1.0.0.M6 (Milestone 6)

Table of Contents

1. Spring Integration Overview

1.1 Background

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

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

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

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

1.2 Goals and Principles

Spring Integration is motivated by the following goals:

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

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

Spring Integration is guided by the following principles:

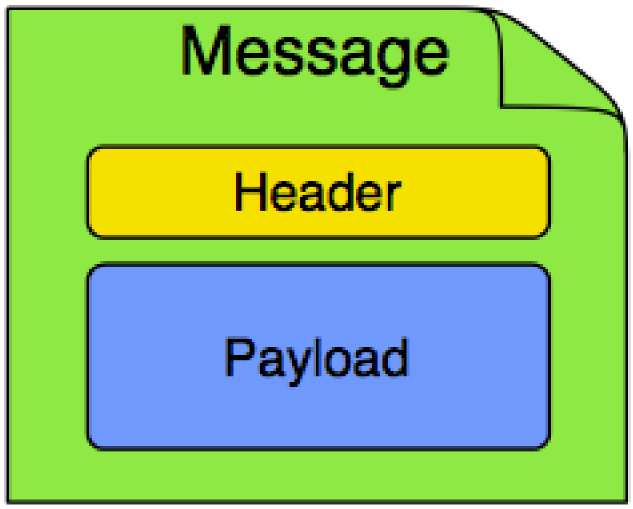
- Components should be *loosely coupled* for modularity and testability.
- The framework should enforce *separation of concerns* between business logic and integration logic.
- Extension points should be abstract in nature but within well-defined boundaries to promote *reuse* and *portability*.

1.3 Main Components

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

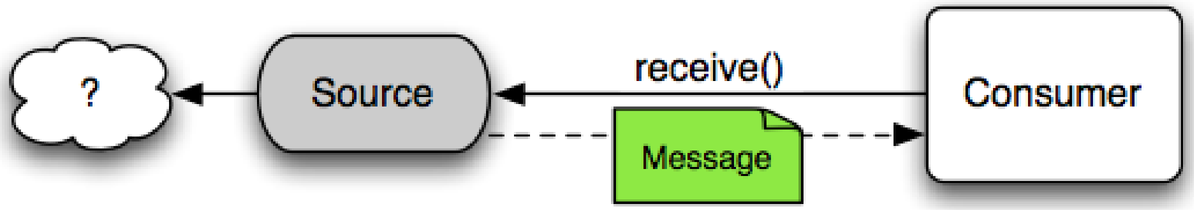
Message

In Spring Integration, a Message is a generic wrapper for any Java object combined with metadata used by the framework while handling that object. It consists of a payload and headers. The payload can be of any type and the headers hold commonly required information such as id, timestamp, expiration, and return address. Developers can also store any arbitrary key-value pair in the headers.



Message Source

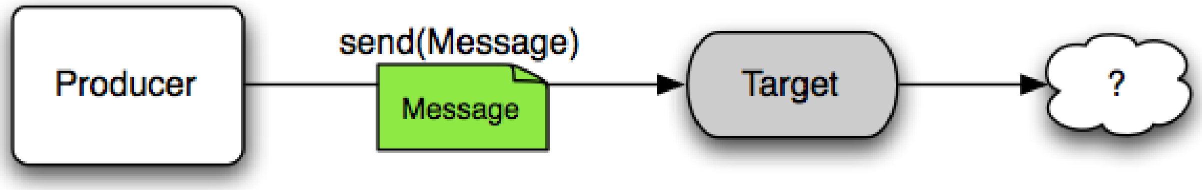
Since a Spring Integration Message is a generic wrapper for any Object, there is no limit to the number of potential sources for such messages. In fact, a source implementation can act as an adapter that converts Objects from any other system into Spring Integration Messages.



There are two types of source: those which require polling and those which send Messages directly. Therefore, Spring Integration provides two main interfaces that extend the `MessageSource` interface: `PollableSource` and `SubscribableSource`. While it is relatively easy to implement these interfaces directly, an adapter is also available for invoking arbitrary methods on plain Objects. Also, several `MessageSource` implementations are already available within the Spring Integration Adapters module. For a detailed discussion of the various adapters, see Chapter 3, *Adapters*.

Message Target

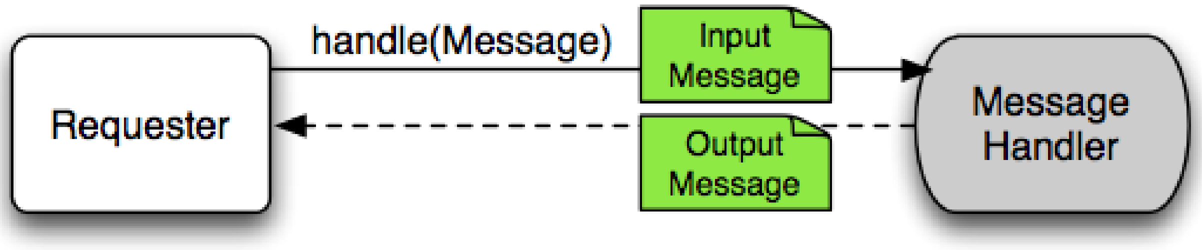
Just as the `MessageSource` implementations enable Message reception, a `MessageTarget` handles the responsibility of sending Messages. As with the `MessageSource`, a `MessageTarget` can act as an adapter that converts Messages into the Objects expected by some other system.



The MessageTarget interface may be implemented directly, but an adapter is also available for invoking arbitrary methods on plain Objects (delegating to a `MessageMapper` strategy in the process). As with MessageSources, several MessageTarget implementations are already available within the Spring Integration Adapters module as discussed in Chapter 3, *Adapters*.

Message Handler

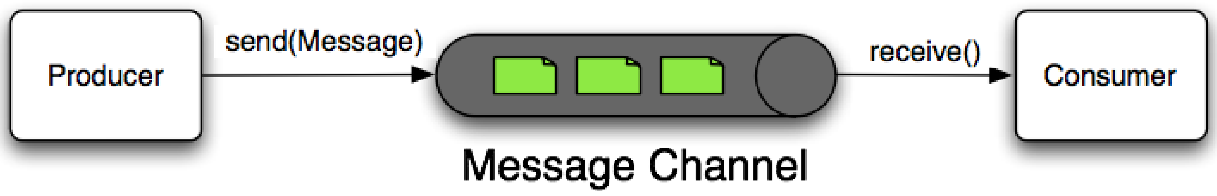
As described above, the MessageSource and MessageTarget components support conversion between Objects and Messages so that application code and/or external systems can be connected to a Spring Integration application rather easily. However, both MessageSource and MessageTarget are unidirectional while the application code or external system to be invoked may provide a return value. The `MessageHandler` interface supports these request-reply scenarios.



As with the MessageSource and MessageTarget, Spring Integration also provides an adapter that itself implements the `MessageHandler` interface while supporting the invocation of arbitrary methods on plain Objects. For more information about the Message Handler, see Section 2.6, “MessageHandler”.

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. By providing both send and receive operations, a Message Channel basically combines the roles of MessageSource and MessageTarget.



Every channel is also a `MessageTarget`, so Messages can be sent to a channel. Likewise, every channel is a `MessageSource`, but as discussed above, Spring Integration defines two types of source: pollable and subscribable. Subscribable channels include PublishSubscribeChannel and DirectChannel. Pollable channels include QueueChannel, PriorityChannel, RendezvousChannel, and ThreadLocalChannel. These are described in detail in Section 2.4, “MessageChannel”.

Message Endpoint

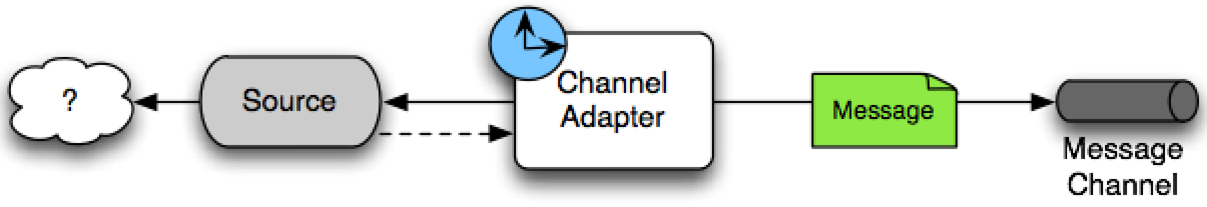
Thus far, the component diagrams show consumers, producers, and requesters invoking the MessageSource, MessageTarget, and MessageHandlers respectively. However, one of the primary goals of Spring Integration is to simplify the development of enterprise integration solutions through *inversion of control*. This means that you should not have to implement such consumers, producers, and requesters directly. Instead, you should be able to focus on your domain logic with an implementation based on plain Objects. Then, by providing declarative configuration, you can "connect" your application code to the messaging infrastructure provided by Spring Integration. The components responsible for these connections are 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 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. Spring Integration provides Message Endpoints for connecting each of the component types described above. The description of each of the main types of endpoint follows.

Channel Adapter

A Channel Adapter is an endpoint that connects either a MessageSource or a MessageTarget to a MessageChannel. If a MessageSource is being adapted, then the adapter is responsible for receiving Messages from the MessageSource and sending them to the MessageChannel. If a Message Target is being adapted, then the adapter is responsible for receiving Messages from the MessageChannel and sending them to the MessageTarget.

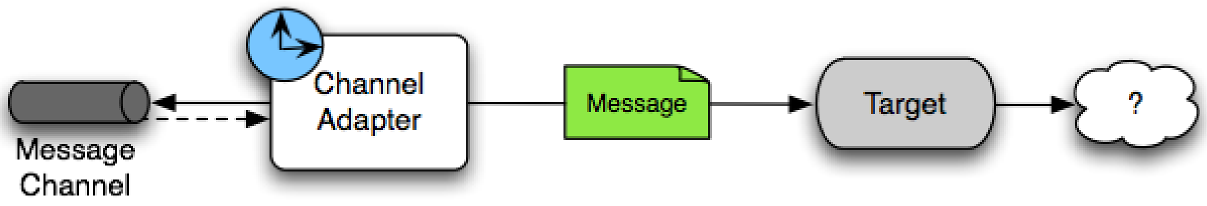
When a Channel Adapter is used to connect a PollableSource implementation to a Message Channel, the invocation of the MessageSource's receive operation may be controlled by scheduling information provided within the Channel Adapter's configuration. Any time the receive operation returns a non-null Message, it is sent to the MessageChannel.



An inbound "Channel Adapter" endpoint connects a MessageSource to a MessageChannel

If the source being connected by the Channel Adapter is a SubscribableSource, then no scheduling information should be provided. Instead the source will send the Message directly, and the Channel Adapter will pass it along to its Message Channel.

When a Channel Adapter is used to connect a MessageTarget implementation to a Pollable Message Channel, the invocation of the MessageChannel's receive operation may be controlled by scheduling information provided within the Channel Adapter's configuration. Any time a non-null Message is received from the MessageChannel, it is sent to the MessageTarget.

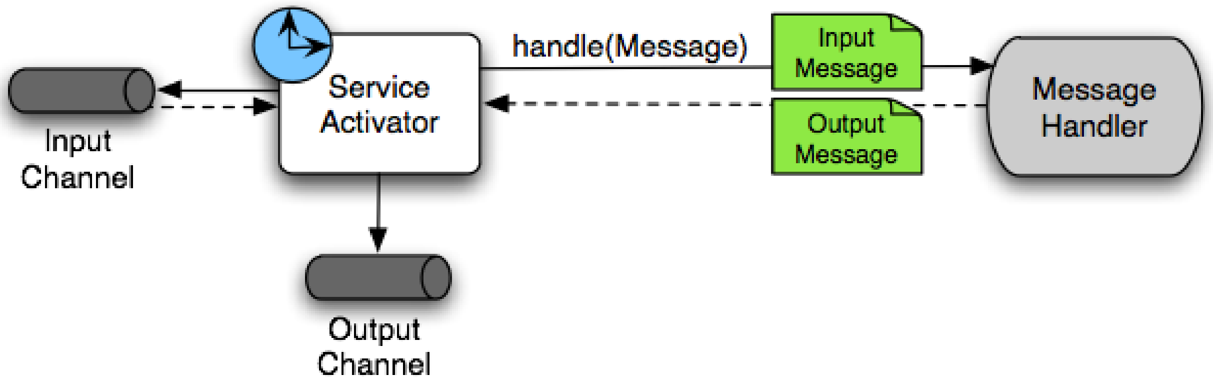


An outbound "Channel Adapter" endpoint connects a MessageChannel to a MessageTarget

If the MessageChannel being connected is not Pollable but Subscribable (e.g. Direct Channel or Publish Subscribe Channel), then no scheduling information should be provided. Instead the channel will send Messages directly, and the Channel Adapter will pass them along to the MessageTarget.

Service Activator

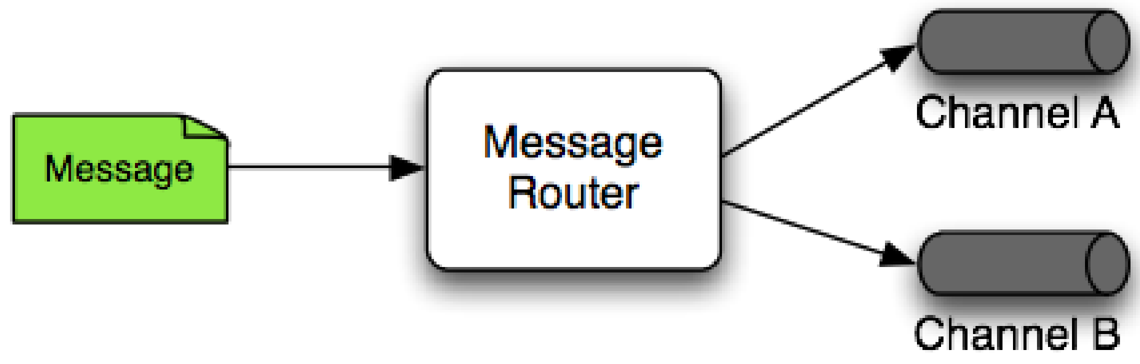
When the Object to be invoked is capable of returning a value, another type of endpoint is needed to accommodate the additional responsibilities of the *request/reply* interaction. The general behavior is similar to a Channel Adapter, but this type of endpoint - the Service Activator - must make a distinction between the "input-channel" and the "output-channel". Also, the Service Activator invokes an operation on some Message Handler to process the request Message. Whenever the Message-handling Object returns a reply Message, that 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 MessageHeader's "return address" if available.



A request-reply "Service Activator" endpoint connects a MessageHandler to input and output MessageChannels.

Message Router

A Message Router is a particular type of Message Endpoint that is capable of receiving a Message from a MessageChannel and then 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 MessageHeader. A Message Router is often used as a dynamic alternative to a statically configured output channel on a Service Activator or other Message-handling endpoint.



Splitter

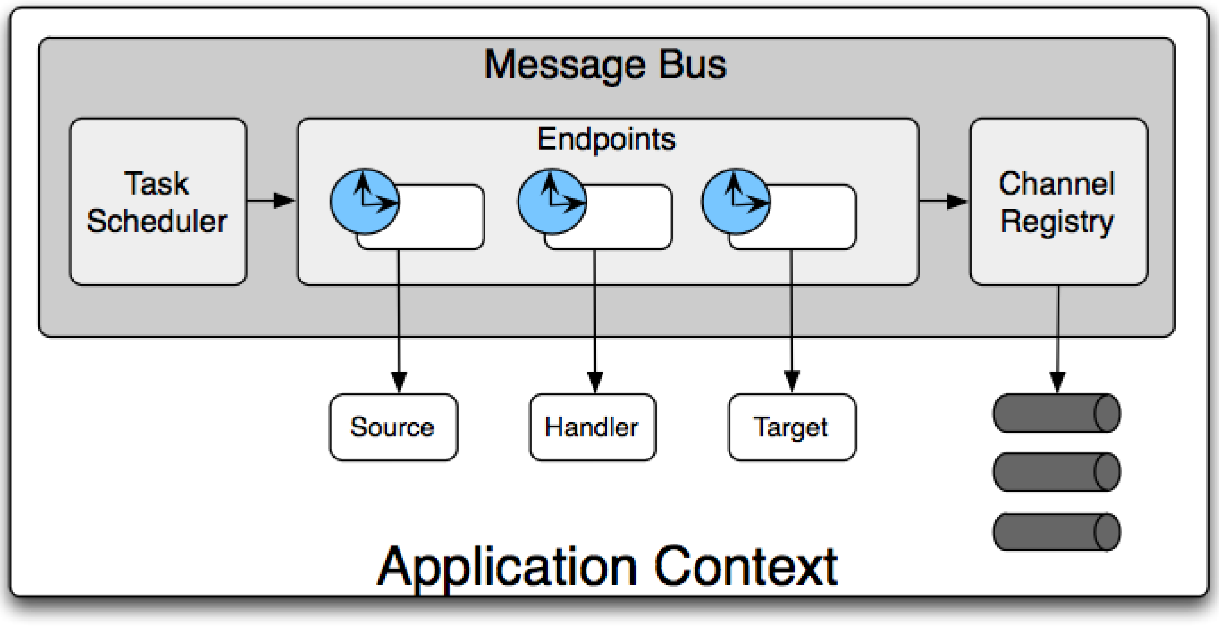
A Splitter is another type of Message Endpoint whose responsibility is to accept a Message from its input channel, split that Message into multiple Messages, and then send each of those to its output channel. This is typically used for dividing a "composite" payload object into a group of Messages containing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

Message Bus

The Message Bus acts as a registry for Message Channels and Message Endpoints. It also encapsulates the complexity of message retrieval and dispatching. Essentially, the Message Bus forms a logical extension of the Spring application context into the messaging domain. For example, it will automatically detect Message Channel and Message Endpoint components from within the application context. It handles the scheduling of pollers, the creation of thread pools, and the lifecycle management of all messaging components that can be initialized, started, and stopped. The Message Bus is the primary example of inversion of control within Spring Integration.


Message Bus
Task Scheduler
Endpoints
Channel Registry
Source
Handler
Target
Application Context

2. The Core API

2.1 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. Here is the definition of the `Message` interface:

```
public interface Message<T> {

    T getPayload();

    MessageHeaders getHeaders();

}
```

And the following headers are pre-defined:

Table 2.1. Pre-defined Message Headers

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

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

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

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

When a Message is created, a random unique id will be generated. The constructor that accepts a Map of headers will copy the provided headers to the newly created Message. There are also two convenient subclasses available currently: `StringMessage` and `ErrorMessage`. The latter accepts any `Throwable` object as its payload.

You may notice that the Message interface defines retrieval methods for its payload and headers but no setters. This is fully intentional so that each Message is unmodifiable after 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 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 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.fromPayload("test")
        .setHeader("foo", "bar")
        .build();

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

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

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

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

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

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

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

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

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

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

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

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

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

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 the system evolves to support new types, or when the types themselves are modified and/or extended, the messaging system will not be affected by such changes. On the other hand, when some component in the messaging system *does* require access to information about the `Message`, such metadata can typically be stored to and retrieved from the metadata in the Message Headers.

2.2 MessageSource

As alluded to in the overview, the `MessageSource` interface is itself a marker interface for any "source" of Messages. The two sub-interfaces - `PollableSource` and `SubscribableSource` - accommodate two types of source: those that must be polled and those that send Messages on their own. An example of the first type would be a source that represents a directory in the File-system, and an example of the second type would be an inbound RMI invocation.

The PollableSource interface defines a single method for receiving `Message` objects.

```
public interface PollableSource<T> extends MessageSource<T> {
    Message<T> receive();
}
```

The `BlockingSource` interface extends `PollableSource` and adds a single method with a timeout:

```
Message<T> receive(long timeout);
```

Spring Integration also provides a `MethodInvokingSource` implementation that serves as an adapter for invoking any arbitrary method on a plain Object (i.e. there is no need to implement an interface). To use the `MethodInvokingSource`, provide the Object reference and the method name.

```
MethodInvokingSource source = new MethodInvokingSource();
```

```
source.setObject(new SourceObject());
source.setMethodName("sourceMethod");
Message<?> result = source.receive();
```

It is also possible to configure a `MethodInvokingSource` in XML by providing a bean reference in the "source" attribute of a <channel-adapter> element along with a "method" attribute.

```
<channel-adapter source="sourceObject" method="sourceMethod" channel="someChannel"/>
```

2.3 MessageTarget

The `MessageTarget` interface defines a single method for sending `Message` objects.

```
public interface MessageTarget {
    boolean send(Message<?> message);
}
```

As with the `MessageSource`, Spring Integration also provides a `MethodInvokingTarget` adapter class.

```
MethodInvokingTarget target = new MethodInvokingTarget();
target.setObject(new TargetObject());
target.setMethodName("targetMethod");
target.afterPropertiesSet();
target.send(new StringMessage("test"));
```

When creating a Channel Adapter for this target, the corresponding XML configuration is very similar to that of `MethodInvokingSource`.

```
<channel-adapter channel="someChannel" target="targetObject" method="targetMethod"/>
```

2.4 MessageChannel

While the `Message` plays the crucial role of encapsulating data, it is the `MessageChannel` that decouples message producers from message consumers. Spring Integration's top-level `MessageChannel` interface is defined as follows.

```
public interface MessageChannel extends MessageSource, BlockingTarget {
    String getName();
}
```

Because it extends `BlockingTarget`, it inherits the following methods:

```
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*.

Since Message Channels are also Message Sources, there are two sub-interfaces corresponding to the two source types. Here is the definition of `PollableChannel`.

```
public interface PollableChannel extends MessageChannel, BlockingSource {

    List<Message<?>> clear();

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

}
```

Since the PollableChannel interface extends BlockingSource, it also inherits the following methods:

```
Message<T> receive();

Message<T> 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.

The subscribable channels implement the `SubscribableSource` interface. Instead of providing receive methods for polling, these channels will send messages directly to their subscribers. The `SubscribableSource` interface defines the following two methods:

```
boolean subscribe(MessageTarget target);

boolean unsubscribe(MessageTarget target);
```

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

PublishSubscribeChannel

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

QueueChannel

The `QueueChannel` implementation wraps a queue. Unlike, the `PublishSubscribeChannel`, the `QueueChannel` has point-to-point semantics. In other words, even if the channel has multiple consumers, only one of them should receive any Message sent to that channel. It provides a no-argument constructor (that uses a default capacity of 100) as well as a constructor that accepts the queue capacity:

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

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

PriorityChannel

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

RendezvousChannel

The `RendezvousChannel` enables a "direct-handoff" scenario where a sender will block until another party invokes the channel's `receive()` method or vice-versa. Internally, this implementation is quite similar to the `QueueChannel` except that it uses a `SynchronousQueue` (a zero-capacity implementation of `BlockingQueue`). This works well in situations where the sender and receiver are operating in different threads but simply dropping the message in a queue asynchronously is too dangerous. For example, the sender's thread could roll back a transaction if the send operation times out, whereas with a `QueueChannel`, the message would have been stored to the internal queue and potentially never received.

The `RendezvousChannel` is also useful for implementing request-reply operations. The sender can create a temporary, anonymous instance of `RendezvousChannel` which it then sets as the 'returnAddress' on a Message. After sending that Message, the sender can immediately call receive (optionally providing a timeout value) in order to block while waiting for a reply Message.

DirectChannel

The `DirectChannel` has point-to-point semantics, but otherwise is more similar to the `PublishSubscribeChannel` than any of the queue-based channel implementations described above. In other words, it does not implement the `PollableChannel` interface, but rather dispatches Messages directly to a subscriber. As a point-to-point channel, however, it will only send each Message to a *single* subscribed `MessageTarget`. Its primary purpose is to enable a single thread to perform the operations on "both sides" of the channel. For example, if a receiving target is subscribed to a `DirectChannel`, then sending a Message to that channel will trigger invocation of that target's `send(Message)` method *directly in the sender's thread*. The key motivation for providing a channel implementation with this behavior is to support transactions that must span across the channel while still benefiting from the abstraction and loose coupling that the channel provides. If the send call is invoked within the scope of a transaction, then the outcome of the target's invocation can play a role in determining the ultimate result of that transaction (commit or rollback).

ThreadLocalChannel

The final channel implementation type is `ThreadLocalChannel`. This channel also delegates to a queue internally, but the queue is bound to the current thread. That way the thread that sends to the channel will later be able to receive those same Messages, but no other thread would be able to access them. While probably the least common type of channel, this is useful for situations where `DirectChannels` are being used to enforce a single thread of operation but any reply Messages should be sent to a "terminal" channel. If that terminal channel is a `ThreadLocalChannel`, the original sending thread can collect its replies from it.

2.5 ChannelInterceptor

One of the advantages of a messaging architecture is the ability to provide common behavior and capture meaningful information about the messages passing through the system in a non-invasive way. Since the `Messages` are being sent to and received from `MessageChannels`, those channels provide an opportunity for intercepting the send and receive operations. The `ChannelInterceptor` strategy interface provides methods for each of those operations:

```
public interface ChannelInterceptor {

    Message<?> preSend(Message<?> message, MessageChannel channel);

    void postSend(Message<?> message, MessageChannel channel, boolean sent);

    boolean preReceive(MessageChannel channel);

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

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

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

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

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

```
public class CountingChannelInterceptor extends ChannelInterceptorAdapter {

    private final AtomicInteger sendCount = new AtomicInteger();

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

2.6 MessageHandler

So far we have seen that generic message objects are sent-to and received-from simple channel objects. Here is Spring Integration's callback interface for handling the `Messages`:

```
public interface MessageHandler {
    Message<?> handle(Message<?> message);
}
```

The handler plays an important role, since it is typically responsible for translating between the generic `Message` objects and the domain objects or primitive values expected by business components that consume the message payload. That said, developers will rarely need to implement this interface directly. While that option will always be available, we will soon discuss the higher-level configuration options including both annotation-driven techniques and XML-based configuration with convenient namespace support.

2.7 MessageBus

So far, you have seen that the `PollableChannel` provides a `receive()` method that returns a `Message`, the subscribable MessageChannels invoke one or more subscribers directly, and the `MessageHandler` provides a `handle()` method that accepts a `Message`. However, we have not yet discussed how messages get passed from a channel to a handler. As mentioned earlier, the `MessageBus` provides a runtime form of inversion of control, and one of the primary responsibilities that it assumes is connecting the channels to the handlers. It also connects MessageSources and MessageTargets to channels (thereby creating Channel Adapters), and it manages the scheduling of polling dispatchers. Ultimately, every MessageHandler should be invoked as if it is an event-driven consumer, and this works fine when the handler's input source is a `SubscribableSource`. However, the bus creates and manages these polling dispatchers so that even when handlers receive input from a `PollableSource`, they will still behave as event-driven consumers.

The `MessageBus` is an example of a mediator. It performs a number of roles - mostly by delegating to other strategies. One of its main responsibilities is to manage registration of the `MessageChannels` and endpoints, such as *Channel Adapters* and *Service Activators*. It recognizes any of these instances that have been defined within its `ApplicationContext`.

The message bus handles several of the concerns so that the channels, sources, targets, and Message-handling objects can be as simple as possible. These responsibilities include the lifecycle management of message endpoints, the activation of subscriptions, and the scheduling of dispatchers (including the configuration of thread pools). The bus coordinates all of that behavior based upon the metadata provided in bean definitions. Furthermore, those bean definitions may be provided via XML and/or annotations (we will look at examples of both configuration options shortly).

The bus is responsible for activating all of its registered endpoints by connecting them to channels within its registry, and if necessary scheduling a poller so that the endpoint can be event-driven even when connected to a channel that requires polling. For example, the poller for an outbound *Channel Adapter* will poll the referenced "channel", and the poller for a *Service Activator* will poll the referenced "input-channel". If that "channel" or "input-channel" is subscribable rather than pollable, the bus will simply activate the subscription. The important point is that the endpoint itself does not need to know whether its source is pollable or subscribable.

2.8 MessageEndpoint

As described in Chapter 1, *Spring Integration Overview*, there are different types of Message Endpoint, such as the *Channel Adapter* (inbound or outbound) and the *Service Activator*. Spring Integration provides many other components that are also endpoints, such as Routers, Splitters, and Aggregators. Each endpoint may provide its own specific metadata so that the `MessageBus` can manage its connection to channels and its poller (if necessary).

The scheduling metadata is provided as an implementation of the `Schedule` interface. This is an abstraction designed to allow extensibility of schedulers for messaging tasks. Currently, there are two implementations: `PollingSchedule` and `CronSchedule`. The former has a *period* property, and the latter has a *cronExpression*. The polling schedule may be configured based on throughput expectations and/or the type of MessageSource (e.g. file-system vs. JMS).

While the MessageBus manages the scheduling of the pollers, it is often beneficial to have multiple task executors with different concurrency settings for an endpoint or group of endpoints. This provides more control over the number of threads available for each receive-and-handle unit of work and depending on the type of task executor, may also enable dynamic adjustments. When the `MessageBus` activates an endpoint, it will create and schedule the poller for that endpoint based on the endpoint's configuration. This will be described in more detail in the section called “Configuring Message Endpoints”.

2.9 MessageSelector

As described above, each endpoint is registered with the message bus and is thereby subscribed to a channel. Often it is necessary to provide additional *dynamic* logic to determine what messages the endpoint should receive. The `MessageSelector` strategy interface fulfills that role.

```
public interface MessageSelector {

    boolean accept(Message<?> message);

}
```

A `MessageEndpoint` can be configured with a selector (or selector-chain) and will only receive messages that are accepted by each selector. Even though the interface is simple to implement, a couple common selector implementations are provided. For example, the `PayloadTypeSelector` provides similar functionality to Datatype Channels (as described in the section called “Configuring Message Channels”) except that in this case the type-matching can be done by the endpoint rather than the channel.

```
PayloadTypeSelector selector = new PayloadTypeSelector(String.class, Integer.class);
assertTrue(selector.accept(new StringMessage("example")));
assertTrue(selector.accept(new GenericMessage<Integer>(123)));
assertFalse(selector.accept(new GenericMessage<SomeObject>(someObject)));
```

Another simple but useful `MessageSelector` provided out-of-the-box is the `UnexpiredMessageSelector`. As the name suggests, it only accepts messages that have not yet expired.

Essentially, using a selector provides *reactive* routing whereas the Datatype Channel and Message Router provide *proactive* routing. However, selectors accommodate additional uses. For example, a `PollableChannel`'s 'purge' method accepts a selector:

```
channel.purge(someSelector);
```

There is a `ChannelPurger` utility class whose purge operation is a good candidate for Spring's JMX support:

```
ChannelPurger purger = new ChannelPurger(new ExampleMessageSelector(), channel);
purger.purge();
```

Implementations of `MessageSelector` might provide opportunities for reuse on channels in addition to endpoints. For that reason, Spring Integration provides a simple selector-wrapping `ChannelInterceptor` that accepts one or more selectors in its constructor.

```
MessageSelectingInterceptor interceptor =
        new MessageSelectingInterceptor(selector1, selector2);
channel.addInterceptor(interceptor);
```

2.10 MessageExchangeTemplate

Whereas the `MessageHandler` interface provides the foundation for many of the components that enable non-invasive invocation of your application code *from the messaging system*, sometimes it is necessary to invoke the messaging system *from your application code*. Spring Integration provides a `MessageExchangeTemplate` that supports a variety of message-exchanges, including request/reply scenarios. For example, it is possible to send a request and wait for a reply.

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

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

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

public Message<?> sendAndReceive(final Message<?> request, final MessageTarget target) { .. }

public Message<?> receive(final PollableSource<?> source) { ... }

public boolean receiveAndForward(final PollableSource<?> source, final MessageTarget target) { ... }
```

Additionally, a 'transactionManager' can be configured on a MessageExchangeTemplate as well as the various transaction attributes:

```
template.setTransactionManager(transactionManager);
template.setPropagationBehaviorName(propagationBehavior);
template.setIsolationLevelName(isolationLevel);
template.setTransactionTimeout(transactionTimeout);
template.setTransactionReadOnly(readOnly);
template.setReceiveTimeout(receiveTimeout);
template.setSendTimeout(sendTimeout);
```

Finally, there is a also an asynchronous version called `AsyncMessageExchangeTemplate` whose constructor accepts a `TaskExecutor`, and whose Message-returning methods return an `AsyncMessage`. That is essentially a wrapper for any Message that also implements `Future<Message<T>>`:

```
AsyncMessageExchangeTemplate template = new AsyncMessageExchangeTemplate(taskExecutor);
Message reply = template.sendAndReceive(new StringMessage("test"), someChannel);
// do some work in the meantime
reply.getPayload(); // blocks if still waiting for actual reply
```

2.11 MessagingGateway

Even though the `MessageExchangeTemplate` is fairly straightforward, it does not hide the details of messaging from your application code. To support working with plain Objects instead of messages, Spring Integration provides `SimpleMessagingGateway` with the following methods:

```
public void send(Object object) { ... }

public Object receive() { ... }

public Object sendAndReceive(Object object) { ... }

public void receiveAndForward() { ... }
```

It enables configuration of a request and/or reply channel and delegates to an instance of the `MessageMapper` and `MessageCreator` strategy interfaces.

```
SimpleMessagingGateway gateway = new SimpleMessagingGateway();
gateway.setRequestChannel(requestChannel);
gateway.setReplyChannel(replyChannel);
gateway.setMessageCreator(messageCreator);
gateway.setMessageMapper(messageMapper);
Object result = gateway.sendAndReceive("test");
```

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

```
<gateway id="fooService"
         service-interface="org.example.FooService"
         request-channel="requestChannel"
```

```
          reply-channel="replyChannel"
          message-creator="messageCreator"
          message-mapper="messageMapper"/>
```

Then, the "fooService" can be injected into other beans, and the code that invokes the methods on that proxied instance of the FooService interface has no awareness of the Spring Integration API. The general approach is similar to that of Spring Remoting (RMI, HttpInvoker, etc.).

3. Adapters

3.1 Introduction

Spring Integration provides a number of implementations of the `MessageSource` and `MessageTarget` interfaces that serve as adapters for interacting with external systems or components that are not part of the messaging system. These source and target implementations can be configured within the same *channel-adapter* element that we have already discussed. Essentially, the external system or component sends-to and/or receives-from a `MessageChannel`. In the 1.0 Milestone 6 release, Spring Integration includes source and target implementations for JMS, Files, FTP, Streams, and Spring ApplicationEvents.

Adapters that allow an external system to perform request-reply operations across Spring Integration `MessageChannels` are actually examples of the *Messaging Gateway* pattern. Therefore, those implementations are typically called "gateways" (whereas "source" and "target" are in-only and out-only interactions respectively). For example, Spring Integration provides a `JmsSource` that is *polled by* the bus-managed scheduler, but also provides a `JmsGateway`. The gateway differs from the source in that it is an *event-driven consumer* rather than a *polling consumer*, and it is capable of waiting for reply messages. Spring Integration also provides gateways for RMI and Spring's HttpInvoker.

Finally, adapters that enable interaction with external systems by *invoking them* for request/reply interactions (the response is sent back on a Message Channel) are typically called *handlers* in Spring Integration, since they implement the `MessageHandler` interface. Basically, these types of adapters can be configured exactly like any POJO with the <service-activator> element. Spring Integration provides RMI, HttpInvoker, and Web Service handler implementations.

All of these adapters are discussed in this section. However, namespace support is provided for many of them and is typically the most convenient option for configuration. For examples, see the section called “Configuring Adapters”.

3.2 JMS Adapters

Spring Integration provides two adapters for accepting JMS messages (as mentioned above): `JmsSource` and `JmsGateway`. The former uses Spring's `JmsTemplate` to receive based on a polling period. The latter configures and delegates to an instance of Spring's `DefaultMessageListenerContainer`.

The `JmsSource` 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 `JmsSource` can then be referenced from a "channel-adapter" element that connects the source to a `MessageChannel` instance. The following example defines a JMS source with a `JmsTemplate` as a constructor-argument.

```
<bean id="jmsSource" class="org.springframework.integration.adapter.jms.JmsSource">
    <constructor-arg ref="jmsTemplate"/>
</bean>
```

In most cases, Spring Integration's message-driven `JmsGateway` is more appropriate since it delegates to a `MessageListener` container, supports dynamically adjusting concurrent consumers, and can also handle replies. The `JmsGateway` requires references to a `ConnectionFactory`, and a `Destination` (or 'destinationName'). The following example defines a `JmsGateway` that receives from the JMS queue called "exampleQueue". Note that the 'expectReply' property has been set to 'true' (it is 'false' by default):

```
<bean class="org.springframework.integration.adapter.jms.JmsGateway">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destinationName" value="exampleQueue"/>
    <property name="expectReply" value="true"/>
</bean>
```

The `JmsTarget` implements the `MessageTarget` interface and is capable of mapping 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). In the section called “Configuring Adapters”, you will see how to configure a JMS target adapter with Spring Integration's namespace support.

3.3 RMI Adapters

The `RmiGateway` is built upon Spring's `RmiServiceExporter`. However, since it is adapting a `MessageChannel`, there is no need to specify the *serviceInterface*. Likewise, the *serviceName* is automatically generated based on the channel name. Therefore, creating the adapter is as simple as providing a reference to its channel:

```
RmiGateway rmiGateway = new RmiGateway(channel);
```

The `RmiHandler` encapsulates the creation of a proxy that is capable of communicating with an `RmiGateway` running in another process. Since the interface is already known, the only required information is the URL. The URL should include the host, port (default is '1099'), and 'serviceName'. The 'serviceName' must match that created by the `RmiGateway` (the prefix is available as a constant).

```
String url = "http://somehost:1099/" + RmiGateway.SERVICE_NAME_PREFIX + "someChannel";
RmiHandler rmiHandler = new RmiHandler(url);
```

3.4 HttpInvoker Adapters

The adapters for HttpInvoker are very similar to the RMI adapters. For a source, only the channel needs to be provided, and for a target, only the URL. If running in a Spring MVC environment, then the

`HttpInvokerGateway` simply needs to be defined and provided in a `HandlerMapping`. For example, the following would be exposed at the path "http://somehost/path-mapped-to-dispatcher-servlet/httpInvokerAdapter" when a simple `BeanNameUrlHandlerMapping` strategy is enabled:

```
<bean name="/httpInvokerAdapter"
    class="org.springframework.integration.adapter.httpinvoker.HttpInvokerGateway">
    <constructor-arg ref="someChannel"/>
</bean>
```

When not running in a Spring MVC application, simply define a servlet in 'web.xml' whose type is `HttpRequestHandlerServlet` and whose name matches the bean name of the gateway adapter. As with the `RmiHandler`, the `HttpInvokerHandler` only requires the URL that matches an instance of `HttpInvokerGateway` running in a web application.

3.5 File Adapters

The `FileSource` requires the directory as a constructor argument:

```
public FileSource(File directory)
```

It can then be connected to a `MessageChannel` when referenced from a "channel-adapter" element.

The `FileTarget` constructor also requires the 'directory' argument. The target adapter also accepts an implementation of the `FileNameGenerator` strategy that defines the following method:

```
String generateFileName(Message message)
```

3.6 FTP Adapters

To poll a directory with FTP, configure an instance of `FtpSource` and then connect it to a channel by configuring a `channel-adapter`. The `FtpSource` expects a number of properties for connecting to the FTP server as shown below.

```
<bean id="ftpSource"
    class="org.springframework.integration.adapter.ftp.FtpSource">
    <property name="host" value="example.org"/>
    <property name="username" value="someuser"/>
    <property name="password" value="somepassword"/>
    <property name="localWorkingDirectory" value="/some/path"/>
    <property name="remoteWorkingDirectory" value="/some/path"/>
</bean>
```

3.7 Mail Adapters

Spring Integration currently provides support for *outbound* email only with the `MailTarget`. This adapter delegates to a configured instance of Spring's `JavaMailSender`, and its various mapping strategies use Spring's `MailMessage` abstraction. By default text-based mails are created when the handled message has a String-based payload. If the message payload is a byte array, then that will be mapped to an attachment.

The adapter also delegates to a `MailHeaderGenerator` for providing the mail's properties, such as the recipients (TO, CC, and BCC), the from/reply-to, and the subject.

```
public interface MailHeaderGenerator {
    void populateMailMessageHeader(MailMessage mailMessage, Message<?> message);
}
```

The default implementation will look for values in the `MessageHeaders` with the following constants defining the header names:

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

A static implementation is also available out-of-the-box and may be useful for testing. However, when customizing, the properties would typically be generated dynamically based on the message itself. The following is an example of a configured mail adapter.

```
<bean id="mailTarget"
    class="org.springframework.integration.adapter.mail.MailTarget">
    <property name="mailSender" ref="javaMailSender"/>
    <property name="headerGenerator" ref="dynamicMailMessageHeaderGenerator"/>
</bean>
```

3.8 Web Service Adapters

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

```
simpleHandler = new SimpleWebServiceHandler(uri);

marshallingHandler = new MarshallingWebServiceHandler(uri, marshaller);
```

Either adapter can then be referenced from a `service-activator` element that is subscribed to an input-channel. The endpoint is then responsible for passing the response to the proper reply channel. It will first check for an "output-channel" on the service-activator and will fallback to a *RETURN_ADDRESS* in the original message's headers.

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.

3.9 Stream Adapters

Spring Integration also provides adapters for streams. Both `ByteStreamSource` and `CharacterStreamSource` implement the `PollableSource` interface. 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 `ByteStreamSource` also accepts the 'bytesPerMessage' property to determine how many bytes it will attempt to read into each `Message`.

For target streams, there are also two implementations: `ByteStreamTarget` and `CharacterStreamTarget`. 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' property. Since both of these ultimately implement the `MessageTarget` interface, they can be referenced from a *channel-adapter* configuration as will be described in more detail in the section called “Configuring Message Endpoints”.

3.10 ApplicationEvent Adapters

Spring `ApplicationEvents` can also be integrated as either a source or target for Spring Integration message channels. To receive the events and send to a channel, simply define an instance of Spring Integration's `ApplicationEventSource` (as with all source implementations, this can then be configured within a "channel-adapter" element and automatically detected by the message bus). The `ApplicationEventSource` also implements Spring's `ApplicationListener` interface. By default it will pass all received events as Spring Integration Messages. To limit based on the type of event, configure the list of event types that you want to receive with the 'eventTypes' property.

To send Spring `ApplicationEvents`, register an instance of the `ApplicationEventTarget` class as the 'target' of a Channel Adapter (such configuration will be described in detail in the section called “Configuring Message Endpoints”). This target also implements Spring's `ApplicationEventPublisherAware` interface and thus acts as a bridge between Spring Integration `Messages` and `ApplicationEvents`.

4. Configuration

4.1 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 yet another option and is described in detail in Chapter 2, *The Core API*. We expect that most users will choose one of the higher-level options, such as the namespace-based or annotation-driven configuration.

4.2 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 namespace support within your Spring configuration files, add the following namespace reference and schema mapping in your top-level 'beans' element:

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

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

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

When using this alternative, no prefix is necessary for the Spring Integration elements. On the other hand, if you want to define a generic Spring "bean" within the same configuration file, then a prefix would be required for the bean element (<beans:bean ... />). Since it is generally a good idea to modularize the configuration files themselves based on responsibility and/or architectural layer, you may find it appropriate to use the latter approach in the integration-focused configuration files, since generic beans are seldom necessary within those same files. For purposes of this documentation, we will assume the "integration" namespace is primary.

Configuring Message Channels

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

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

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

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

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

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

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

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

When using the "channel" element, the creation of the channel instances will be deferred to the `ChannelFactory` bean whose name is "channelFactory" if defined within the ApplicationContext. If no such bean is defined, the default factory will be used. The default implementation is `QueueChannelFactory`.

It is also possible to use more specific elements for the various channel types (as described in Section 2.4, “MessageChannel”). Depending on the channel, these may provide additional configuration options. Examples of each are shown below.

The <queue-channel/> element

To create a `QueueChannel`, use the "queue-channel" element. By using this element, you can also specify the channel's capacity:

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

The <publish-subscribe-channel/> element

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

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

The <priority-channel/> element

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

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

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

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

The <rendezvous-channel/> element

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

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

The <direct-channel/> element

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

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

The <thread-local-channel/> element

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

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

Message channels may also have interceptors as described in Section 2.5, “ChannelInterceptor”. One or more <interceptor> elements can be added as sub-elements of <channel> (or the more specific element types). Provide the "ref" attribute to reference any Spring-managed object that implements the `ChannelInterceptor` interface:

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

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

Configuring Message Endpoints

Each of the endpoint types (channel-adapter, service-activator, etc) has its own element in the namespace.

The inbound <channel-adapter/> element with a MessageSource

A "channel-adapter" element can connect any implementation of the `MessageSource` interface to a `MessageChannel`. When the `MessageBus` registers the endpoint, it will activate the subscription and if necessary create a poller for the endpoint. The Message Bus delegates to a `TaskScheduler` for scheduling the poller based on its schedule. To configure the polling 'period' or 'cronExpression' for an individual channel-adapter's schedule, provide a 'poller' sub-element with the 'period' (in milliseconds) or 'cron' attribute:

```
<channel-adapter source="source1" channel="channel1">
    <poller period="5000"/>
</channel-adapter>

<channel-adapter source="source2" channel="channel2">
    <poller cron="30 * * * * ?"/>
</channel-adapter>
```

Note

Cron support does require the Quartz JAR and its transitive dependencies. Also, keep in mind that pollers only apply for `PollableChannel` implementations. On the other hand, subscribable channels (PublishSubscribeChannel and DirectChannel) will send Messages to their subscribed targets directly.

The outbound <channel-adapter/> with a MessageTarget

A "channel-adapter" element can also connect a `MessageChannel` to any implementation of the `MessageTarget` interface.

```
<channel-adapter channel="exampleChannel" target="exampleTarget"/>
```

Again, it is possible to provide a poller:

```
<channel-adapter channel="exampleChannel" target="exampleTarget">
    <poller period="3000"/>
</channel-adapter>
```

The <service-activator/> element

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

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

The configuration above assumes that "exampleHandler" is an actual implementation of the `MessageHandler` interface as described in Section 2.6, “MessageHandler”. To delegate to an arbitrary method of any object, simply add the "method" attribute.

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

In either case (`MessageHandler` or arbitrary object/method), when the handling 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 the NEXT_TARGET header contains a non-null value, next it will check if an "output-channel" was provided in the endpoint configuration:

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

If no "output-channel" is available, it will finally check the message header's RETURN_ADDRESS property. If that value is available, it will then check its type. If it is a `MessageTarget`, the reply message will be sent to that target. If it is a `String`, then the endpoint will attempt to resolve the channel by performing a lookup in the `ChannelRegistry`. If the target cannot be resolved, then a `MessageHandlingException` will be thrown.

Message Endpoints also support `MessageSelectors` as described in Section 2.9, “MessageSelector”. To configure a selector with namespace support, simply add the "selector" attribute to the endpoint definition and reference an implementation of the `MessageSelector` interface.

```
<service-activator id="endpoint" input-channel="channel" ref="handler"
                   selector="exampleSelector"/>
```

Another important configuration option for message endpoints is the inclusion of `EndpointInterceptors`. The interface is defined as follows:

```
public interface EndpointInterceptor {

    Message<?> preHandle(Message<?> requestMessage);

    Message<?> aroundHandle(Message<?> requestMessage, MessageHandler handler);

    Message<?> postHandle(Message<?> replyMessage);

}
```

There is also an EndpointInterceptorAdapter that provides no-op methods for convenience when subclassing. Within an endpoint configuration, interceptors can be added within the <interceptors> sub-element. It accepts either "ref" elements or inner "beans":

```
<service-activator id="exampleEndpoint"
                   input-channel="requestChannel"
                   ref="someObject"
                   method="someMethod"
                   output-channel="replyChannel">
    <poller period="1000"/>
    <interceptors>
        <ref bean="someInterceptor"/>
        <beans:bean class="example.AnotherInterceptor"/>
    </interceptors>
</service-activator>
```

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:

```
<service-activator id="exampleEndpoint"
                   input-channel="requestChannel"
                   ref="someObject"
                   method="someMethod"
                   output-channel="replyChannel">
    <poller period="1000">
        <transactional transaction-manager="txManager"
                       propagation="REQUIRES_NEW"
                       isolation="REPEATABLE_READ"
                       timeout="10000"
                       read-only="false"/>
    </poller>
</service-activator>
```

Spring Integration also provides support for executing the pollers with a `TaskExceutor`. This enables concurrency for an endpoint or group of endpoints. As a convenience, there is also namespace support for creating a simple thread pool executor. The <pool-executor/> element defines attributes for common concurrency settings such as core-size, max-size, and queue-capacity. Configuring a thread-pooling executor can make a substantial difference in how the endpoint performs under load. These settings are available per-endpoint since the performance characteristics of an endpoint's handler or 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 an 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:

```
<service-activator input-channel="exampleChannel" ref="exampleHandler">
    <poller period="5000" task-executor="pool"/>
</service-activator>

<pool-executor id="pool" core-size="5" max-size="25" queue-capacity="20" keep-alive-seconds="120"/>
```

If no 'task-executor' is provided, the endpoint's handler or target will be invoked in the caller's thread. Note that the "caller" is usually the MessageBus' task scheduler except in the case of a subscribable channel. Also, keep in mind that you the 'task-executor' attribute can provide a reference to any implementation of Spring's `TaskExecutor` interface.

Configuring the Message Bus

As described in Section 2.7, “MessageBus”, the `MessageBus` plays a central role. Nevertheless, its configuration is quite simple since it is primarily concerned with managing internal details based on the configuration of channels and endpoints. The bus is aware of its host application context, and therefore is also capable of auto-detecting the channels and endpoints. Typically, the `MessageBus` can be configured with a single empty element:

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

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

```
<message-bus/>

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

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

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

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

Another configurable property is the size of the default dispatcher thread pool. The dispatcher threads are responsible for polling channels and then passing the messages to handlers.

```
<message-bus dispatcher-pool-size="25"/>
```

When the endpoints are concurrency-enabled as described in the previous section, the invocation of the handling methods will happen within the handler thread pool and not the dispatcher pool. However, when no task-executor is provided to an endpoint's poller, then it will be invoked in the dispatcher's thread (with the exception of subscribable channels).

Finally, the type of channel that gets created automatically by the bus can be customized by defining a bean that implements the ChannelFactory interface and whose name is "channelFactory".

```
<message-bus/>

<beans:bean id="channelFactory"
    class="org.springframework.integration.channel.factory.PriorityChannelFactory"/>
```

With this definition, all the channels created automatically will be `PriorityChannel` instances. Without a "channelFactory" bean, the Message Bus will assume a default `QueueChannelFactory`.

Configuring Adapters

The most convenient way to configure Source and Target adapters is by using the namespace support. The following examples demonstrate the namespace-based configuration of several source, target, gateway, and handler adapters:

```
<jms-source id="jmsSource" connection-factory="connFactory" destination="inQueue"/>

<!-- using the default "connectionFactory" reference -->
<jms-target id="jmsTarget" destination="outQueue"/>

<file-source id="fileSource" directory="/tmp/in"/>

<file-target id="fileTarget" directory="/tmp/out"/>

<rmi-gateway id="rmiSource" request-channel="rmiSourceInput"/>

<rmi-handler id="rmiTarget"
             local-channel="rmiTargetOutput"
             remote-channel="someRemoteChannel"
             host="somehost"/>

<httpinvoker-gateway id="httpSource" name="/some/path" request-channel="httpInvokerInput"/>

<httpinvoker-handler id="httpTarget" channel="httpInvokerOutput" url="http://somehost/test"/>

<mail-target id="mailTarget" host="somehost" username="someuser" password="somepassword"/>

<ws-handler id="wsTarget" uri="http://example.org" channel="wsOutput"/>

<ftp-source id="ftpSource"
            host="example.org"
            username="someuser"
            password="somepassword"
            local-working-directory="/some/path"
            remote-working-directory="/some/path"/>
```

In the examples above, notice that simple implementations of the `MessageSource` and `MessageTarget` interfaces do not accept any 'channel' references. To connect such sources and targets to a channel, register them within a 'channel-adapter'. For example, here is a File source with an endpoint whose polling will be scheduled to execute every 30 seconds by the `MessageBus`.

```
<channel-adapter source="fileSource" channel="exampleChannel">
    <poller period="30000"/>
</channel-adapter>

<file-source id="fileSource" directory="/tmp/in"/>
```

Likewise, here is an example of a JMS target that is registered within a 'channel-adapter' and whose Messages will be received from the "exampleChannel" that is polled every 500 milliseconds.

```
<channel-adapter channel="exampleChannel" target="jmsTarget">
    <poller period="500"/>
</channel-adapter>

<jms-target id="jmsTarget" destination="targetDestination"/>
```

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 <channel-adapter/> element. Therefore, if the "channel" is not provided, the "id" is required.

Enabling Annotation-Driven Configuration

The next section will describe Spring Integration's support for annotation-driven configuration. To enable those features, add this single element to the XML-based configuration:

```
<annotation-driven/>
```

4.3 Annotations

In addition to the XML namespace support for configuring Message Endpoints, it is also possible to use annotations. The class-level `@MessageEndpoint` annotation indicates that the annotated class is capable of being registered as an endpoint, and the method-level `@Handler` annotation indicates that the annotated method is capable of handling a message.

```
@MessageEndpoint(input="fooChannel")
public class FooService {

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

The @MessageEndpoint is not required. If you want to configure a POJO reference from the "ref" attribute of a <service-activator/> element, it is sufficient to provide the @Handler method annotation. As long as the "annotation-driven" support is enabled, a Spring-managed object with that method annotation (or the others which are described below) will be post-processed such that it can be used as a reference from an XML-configured endpoint.

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

```
public class FooService {

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

}
```

When the method parameter should be mapped from a value in the `MessageHeader`, another option is to use the parameter-level `@Header` annotation.

```
@MessageEndpoint(input="fooChannel")
public class FooService {

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

}
```

As described in the previous section, when the handler method returns a non-null value, the endpoint will attempt to send a reply. This is consistent across both configuration options (namespace and annotations) in that the the endpoint's output channel will be used if available, and the message header's RETURN_ADDRESS value will be the fallback. To configure the output channel for an annotation-driven endpoint, provide the 'output' attribute on the `@MessageEndpoint`.

```
@MessageEndpoint(input="exampleChannel", output="replyChannel")
```

Just as the 'poller' sub-element and its 'period' attribute can be provided for a namespace-based endpoint, the `@Poller` annotation can be provided with the `@MessageEndpoint` annotation.

```
@MessageEndpoint(input="exampleChannel")
@Poller(period=3000)
public class FooService {
    ...
}
```

Likewise, `@Concurrency` provides an annotation-based equivalent of the <pool-executor/> element:

```
@MessageEndpoint(input="fooChannel")
@Concurrency(coreSize=5, maxSize=20)
public class FooService {

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

}
```

Several additional annotations are supported, and three of these act as a special form of handler method: `@Router`, `@Splitter` and `@Aggregator`. As with the `@Handler` annotation, methods annotated with these 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:

```
someMethod(String payload, @Header("x") int valueX, @Header("y") int valueY);
```

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

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

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

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

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

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

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

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

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

The `@Aggregator` annotation may be used on a method that accepts a collection of Messages or Message payload types and whose return value is a single Message or single Object that will be used as the payload of a Message.

```
@Aggregator
public Message<?> aggregateMessages(List<Message<?>> messages) { ... }

@Aggregator
public Order aggregateOrder(List<LineItem> items) { ... }
```

Finally, the `@Publisher` is an annotation that triggers the creation of a Spring AOP Proxy such that the return value, exception, or method invcation arguments can be sent to a Message Channel. For example, each time the following method is invoked, its return value will be sent to the "fooChannel":

```
@Publisher(channel="fooChannel")
public String foo() {
    return "bar";
}
```

The return value is published by default, but you can also configure the payload type:

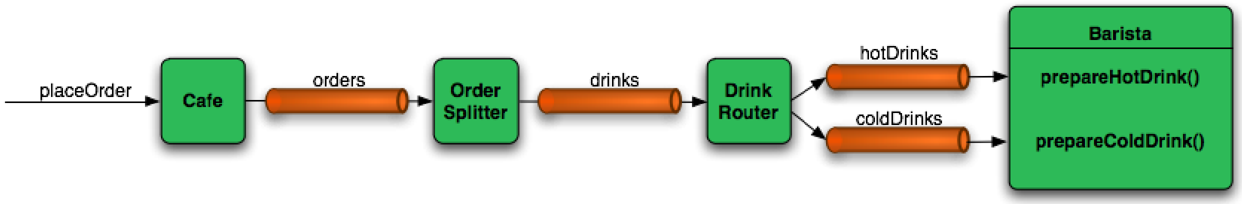
```
@Publisher(channel="testChannel", payloadType=MessagePublishingInterceptor.PayloadType.ARGUMENTS)
public void publishArguments(String s, Integer n) {
    ...
}

@Publisher(channel="testChannel", payloadType=MessagePublishingInterceptor.PayloadType.EXCEPTION)
public void publishException() {
    throw new RuntimeException("oops!");
}
```

5. Spring Integration Samples

5.1 The Cafe Sample

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

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



The `DrinkOrder` object may contain multiple `Drinks`. 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 `Drink` object's 'isIced' property). Finally the `Barista` prepares each drink, but hot and cold drink preparation are handled by two distinct methods: 'prepareHotDrink' and 'prepareColdDrink'.

Here is the XML configuration:

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

    <message-bus/>
    <annotation-driven/>

    <context:component-scan base-package="org.springframework.integration.samples.cafe"/>

    <channel id="orders"/>
    <channel id="drinks"/>
    <channel id="coldDrinks"/>
    <channel id="hotDrinks"/>

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

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

    <beans:bean id="cafe" class="org.springframework.integration.samples.cafe.Cafe">
        <beans:property name="orderChannel" ref="orders"/>
    </beans:bean>

</beans:beans>
```

Notice that the Message Bus is defined. It will automatically detect and register all channels and endpoints. The 'annotation-driven' element will enable the detection of the splitter and router - both of which carry the `@MessageEndpoint` annotation. That annotation extends Spring's "stereotype" annotations (by relying on the @Component meta-annotation), and so all classes carrying the endpoint annotation are capable of being detected by the component-scanner.

```
@MessageEndpoint(input="orders", output="drinks")
public class OrderSplitter {

    @Splitter
    public List<Drink> split(DrinkOrder order) {
        return order.getDrinks();
    }
}
```

```
@MessageEndpoint(input="drinks")
public class DrinkRouter {

    @Router
    public String resolveDrinkChannel(Drink drink) {
        return (drink.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. The 'barista' could have been defined in the XML, but instead the `@Component` annotation is applied:

```
@Component
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 void prepareHotDrink(Drink drink) {
        try {
            Thread.sleep(this.hotDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                + " prepared hot drink #" + hotDrinkCounter.incrementAndGet() + ": " + drink);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }

    public void prepareColdDrink(Drink drink) {
        try {
            Thread.sleep(this.coldDrinkDelay);
            System.out.println(Thread.currentThread().getName()
                + " prepared cold drink #" + coldDrinkCounter.incrementAndGet() + ": " + drink);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

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.

```
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);
    }
    context.start();
    Cafe cafe = (Cafe) context.getBean("cafe");
    DrinkOrder order = new DrinkOrder();
    Drink hotDoubleLatte = new Drink(DrinkType.LATTE, 2, false);
    Drink icedTripleMocha = new Drink(DrinkType.MOCHA, 3, true);
    order.addDrink(hotDoubleLatte);
    order.addDrink(icedTripleMocha);
    for (int i = 0; i < 100; i++) {
        cafe.placeOrder(order);
    }
}
```

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

When you run cafeDemo, you will see that all 100 cold drinks are prepared in roughly the same amount of time as only 20 of the hot drinks. 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:

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

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

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

Also, notice that the worker thread name is displayed with each invocation. You should see that most of the hot drinks are prepared by the task-executor threads, but that occasionally it throttles the input by forcing the message-bus (the caller) to invoke the operation. In addition to experimenting with the 'concurrency' settings, you can also add the 'transactional' sub-element as described in the section called “Configuring Message Endpoints”. If you want to explore the sample in more detail, the source JAR is available in the "src" directory: 'org.springframework.integration.samples-sources-1.0.0.M6.jar'.

6. Additional Resources

6.1 Spring Integration Home

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