Spring Integration Reference Manual

Mark Fisher

1.0.0.m1 (Milestone 1)

Table of Contents

1. Spring Integration Overview

1.1 Background

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

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

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

As an extension of the Spring programming model, Spring Integration provides a wide variety of configuration options including annotations, XML with namespace support, XML with generic "bean" elements, and of course direct usage of the underlying API. That API is based upon well-defined strategy interfaces and non-invasive, delegating adapters. Spring Integration's design is inspired by the recognition of a strong affinity between common patterns within Spring and the well-known Enterprise Integration Patterns [http://www.eaipatterns.com] as described in the book of the same name by Gregor Hohpe and Bobby Woolf (Addison Wesley, 2003). 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 header and has a unique identifier. The payload can be of any type and the header holds commonly required information such as timestamp, expiration, and return address. Developers can also store any arbitrary key-value properties or attributes in the header.

Message Channel

A Message Channel represents the "pipe" of a pipes-and-filters architecture. Producers send Messages to a MessageChannel, and consumers receive Messages from a MessageChannel. The send and receive methods both come in two forms: one that blocks indefinitely and one that accepts a timeout (for an immediate return, specify a timeout value of 0). There are two main types of channels: *Point-to-Point*

channels where typically a single consumer will receive the Message and *Publish-Subscribe* channels where all subscribers should receive the Message.

Message Endpoint

A Message Endpoint represents the "filter" of a pipes-and-filters architecture. 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 messaging framework. This is similar to the role of a Controller in the MVC paradigm. Just as a Controller handles HTTP requests, the endpoint handles Messages. Just as Controllers are mapped to URL patterns, endpoints are mapped to Message Channels. The goal is the same in both cases: isolate application code from the infrastructure. In Spring Integration, the Message Endpoint invokes a `MessageHandler` callback interface as described in Section 2.3, “MessageHandler”.

Message Router

A Message Router is a particular type of endpoint that is capable of receiving a Message and then deciding what channel or channels should receive the Message next. Typically the decision is based upon the Message's content and/or metadata. A Message Router is often used as a dynamic alternative to configuring the input and output channels for an endpoint.

Channel Adapter

A Channel Adapter is used to connect components to a Message Channel when those components are not themselves Message Endpoints. These adapters provide a mechanism for connecting to external systems, such as JMS queues or a File system. Channel Adapters may be configured for input and/or output. An input (source) adapter will receive (or poll for) data, convert that data to a Message, and then send that Message to its Message Channel. An output (target) adapter is simply another type of Message Endpoint, but when it receives a Message, it will convert it to the target's expected type and then "send" it (publish to a JMS queue, write to a File, etc.).

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.

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 a header containing user-extensible properties as key-value pairs. Here is the definition of the `Message` interface:

```
public interface Message<T> {
    Object getId();
    MessageHeader getHeader();
    T getPayload();
}
```

And the header provides the following properties:

Table 2.1. Properties of the MessageHeader

Property Name	Property Type
timestamp	java.util.Date
expiration	java.util.Date
correlationId	java.lang.Object
replyChannelName	java.lang.String
sequenceNumber	int
sequenceSize	int
properties	java.util.Properties
attributes	Map<String,Object>

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

```
new GenericMessage<T>(Object id, T payload);
new GenericMessage<T>(T payload);
```

When no id is provided, a random unique id will be generated. There are also two convenient subclasses available currently: `StringMessage` and `ErrorMessage`. The latter accepts any `Throwable` object as its payload.

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 header (the 'properties' and 'attributes').

2.2 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 `MessageChannel` interface is defined as follows.

```
public interface MessageChannel {
    String getName();
    DispatcherPolicy getDispatcherPolicy();
    boolean send(Message message);
    boolean send(Message message, long timeout);
    Message receive();
    Message receive(long timeout);
}
```

The `SimpleChannel` implementation wraps a queue. It provides a no-argument constructor as well as a constructor that accepts the queue capacity:

```
public SimpleChannel(int capacity)
```

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*. Likewise when receiving a message, the return value will be *null* in the case of a timeout or interrupt.

2.3 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 business components that consume the message payload. That said, developers will rarely need to implement this callback 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.

2.4 MessageBus

There is a rather obvious gap in what we have reviewed thus far. The `MessageChannel` provides a

`receive()` method that returns a `Message`, and the `MessageHandler` provides a `handle()` method that accepts a `Message`, but how do the messages get passed from the channel to the handler? As mentioned earlier, the `MessageBus` provides a runtime form of inversion of control, and so the short answer is: you don't need to worry about it. Nevertheless since this is a reference guide, we will explore this in a bit of detail.

The `MessageBus` is an example of a mediator. It performs a number of roles - mostly by delegating to other strategies. One of its fundamental responsibilities is to manage registration of the `MessageChannels` and `MessageHandlers`. It provides the following methods:

```
public void registerChannel(String name, MessageChannel channel)
public void registerHandler(String name, MessageHandler handler, Subscription subscription)
public void registerHandler(String name, MessageHandler handler, Subscription subscription, ConcurrencyPolicy c
```

As those method signatures reveal, the message bus is handling several of the concerns here so that the channel and handler objects can be as simple as possible. These responsibilities include the creation and lifecycle management of message dispatchers, the activation of handler subscriptions, and the configuration of thread pools. The bus coordinates all of that behavior based upon the metadata provided via these registration methods. We will briefly take a look at each of those metadata objects.

The bus creates and manages dispatchers that pull messages from a channel in order to push those messages to handlers subscribed to that channel. Each channel has a `DispatcherPolicy` that contains metadata for configuring those dispatchers:

Table 2.2. Properties of the DispatcherPolicy

Property Name	Default Value	Description
maxMessagesPerTask	1	maximum number of messages to retrieve per poll
receiveTimeout	1000 (milliseconds)	how long to block on the receive call (0 for no blocking, -1 for indefinite block)
rejectionLimit	5	maximum number of attempts to invoke handlers (e.g. no threads available)
retryInterval	1000 (milliseconds)	amount of time to wait between successive attempts to invoke handlers
shouldFailOnRejectionLimit	true	whether to throw a `MessageDeliveryException` if the 'rejectionLimit' is reached - if this is set to 'false', then such undeliverable messages would be dropped silently

The bus registers handlers with a channel's dispatcher based upon the `Subscription` metadata provided to the `registerHandler()` method.

Table 2.3. Properties of the Subscription

Property Name	Description
channel	the channel instance to subscribe to (an object reference)
channelName	the name of the channel to subscribe to - only used as a fallback if 'channel' is null
schedule	the scheduling metadata (see below)

The scheduling metadata is provided with an instance of the `Schedule` interface. This is an abstraction designed to allow extensibility of schedulers for messaging tasks. Currently, there is a single implementation called `PollingSchedule` that provides the following properties:

Table 2.4. Properties of the PollingSchedule

Property Name	Default Value	Description
period	N/A	the delay interval between each poll
initialDelay	0	the delay prior to the first poll
timeUnit	TimeUnit.MILLISECONDS	time unit for 'period' and 'initialDelay'
fixedRate	false	'false' indicates fixed-delay (no backlog)

The `PollingSchedule` constructor requires the 'period' value.

The `ConcurrencyPolicy` is an optional parameter to provide when registering a handler. When the `MessageBus` registers a handler, it will use these properties to configure that handler's thread pool. These parameters are configurable on a per-handler basis since handlers may have different performance characteristics and may have different expectations with regard to the volume of throughput. The following table lists the available properties and their default values:

Table 2.5. Properties of the ConcurrencyPolicy

Property Name	Default Value	Description
coreSize	1	the core size of the thread pool
maxSize	10	the maximum size the thread pool can reach when under

Property Name	Default Value	Description
		demand
queueCapacity	0	capacity of the queue which defers an increase of the pool size
keepAliveSeconds	60	how long added threads (beyond core size) should remain idle before being removed from the pool

2.5 MessageEndpoint

When `MessageHandlers` are registered with the `MessageBus`, the bus assigns the handler to a dispatcher based on the provided schedule as described above. Internally, the bus is creating and registering an instance that implements the `MessageEndpoint` interface. This is where other handler metadata enters the picture (e.g. the concurrency settings). Basically, you can consider the endpoint to be a composite handler built from a simple implementation of the `MessageHandler` along with its metadata. In fact, the `MessageEndpoint` does extend the `MessageHandler` interface.

```
public interface MessageEndpoint extends MessageHandler {
    String getName();
    Subscription getSubscription();
    ConcurrencyPolicy getConcurrencyPolicy();
}
```

When using the API, it's simpler to register handlers with metadata and leave the message endpoint as an internal responsibility of the bus. However, it is possible to create endpoints directly. Spring Integration provides a single implementation: `DefaultMessageEndpoint`.

3. Channel Adapters

3.1 Introduction

Channel Adapters are the components responsible for interacting with external systems or other components that are external to the messaging system. As the name implies, the interaction consists of adapting the external system or component to send-to and/or receive-from a `MessageChannel`. Within Spring Integration, there is a distinction between *source adapters* and *target adapters*. In the 1.0 Milestone 1 release, Spring Integration includes adapters for JMS, Files, Streams, and Spring ApplicationEvents.

3.2 JMS Adapters

Spring Integration provides two adapters for accepting JMS messages: `JmsPollingSourceAdapter` and `JmsMessageDrivenSourceAdapter`. 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 `JmsPollingSourceAdapter` 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 `JmsPollingSourceAdapter` also requires a 'channel' property that should be a reference to a `MessageChannel` instance. The adapter accepts additional properties such as: period, initialDelay, maxMessagesPerTask, and sendTimeout. The following example defines a JMS source adapter that polls every 5 seconds and then sends to the "exampleChannel":

```
<bean class="org.springframework.integration.adapter.jms.JmsPollingSourceAdapter">
    <constructor-arg ref="jmsTemplate"/>
    <property name="channel" ref="exampleChannel"/>
    <property name="period" value="5000"/>
</bean>
```

In most cases, Spring Integration's message-driven JMS adapter is more appropriate since it delegates to a `MessageListener` container and supports dynamically adjusting concurrent consumers. The `JmsMessageDrivenSourceAdapter` requires references to a `MessageChannel`, a `ConnectionFactory`, and a `Destination` (or 'destinationName'). The following example defines a JMS message-driven source adapter that receives from the JMS queue called "exampleQueue" and then sends to the Spring Integration channel named "exampleChannel":

```
<bean class="org.springframework.integration.adapter.jms.JmsMessageDrivenSourceAdapter">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destinationName" value="exampleQueue"/>
    <property name="channel" ref="exampleChannel"/>
</bean>
```

For both source adapter types, Spring's `MessageConverter` strategy is used to convert the JMS message into a plain Java object, and then Spring Integration's `MessageMapper` strategy is used to convert from the plain object to a `Message`.

The `JmsTargetAdapter` is a `MessageHandler` implementation that 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. In the section called “Configuring Channel Adapters”, you will see how to configure a JMS target adapter with Spring Integration's namespace support.

3.3 File Adapters

The `FileSourceAdapter` extends the generic `PollingSourceAdapter` (just as the polling JMS adapter does). It requires the following constructor arguments:

```
public FileSourceAdapter(File directory, MessageChannel channel, int period)
```

Optional properties include 'initialDelay' and 'maxMessagesPerTask'.

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

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

As with the JMS adapters, the most convenient way to configure File adapters is with the namespace support. For examples, see the section called “Configuring Channel Adapters”.

3.4 Stream Adapters

Spring Integration also provides adapters for streams. Both `ByteStreamSourceAdapter` and `CharacterStreamSourceAdapter` extend the `PolllingSourceAdapter` so that the polling period can be configured, and the Message Bus can automatically detect and schedule them. Both require an `InputStream` as the single constructor argument. The `ByteStreamSourceAdapter` 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: `ByteStreamTargetAdapter` and `CharacterStreamTargetAdapter`. Each defines a constructor that requires an `OutputStream`, and each provides a second constructor that adds the optional 'bufferSize' property. Since both of these ultimately implement the `MessageHandler` interface, they can be referenced from an endpoint configuration as will be described in more detail in the section called “Configuring Message Endpoints”.

3.5 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 `ApplicationEventSourceAdapter` (as with all source adapters, if a `MessageBus` is defined, it will automatically detect the event source adapter). The `ApplicationEventSourceAdapter` 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 `ApplicationEventTargetAdapter` class as the handler of an endpoint (such configuration will be described in detail in the section called “Configuring Message Endpoints”). This adapter implements Spring's `ApplicationEventPublisherAware` 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. Of course, it is also possible to always stick with a single approach. The main point is that these are *options* for configuration motivated by the need to support a user community with a wide range of preferences. That said, there has also been a concerted effort to provide consistent naming so that, for example, the 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 [http://www.eaipatterns.com].

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, use the 'channel' element:

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

You can also specify the channel's capacity:

```
<channel id="exampleChannel" capacity="100"/>
```

The default channel type is *Point to Point*. To create a *Publish Subscribe* channel, provide a value of *true* for the 'publish-subscribe' attribute of the channel element:

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

When the `MessageBus` detects and registers channels, it will establish a dispatcher for each channel. The default dispatcher settings were previously displayed in Table 2.2, “Properties of the DispatcherPolicy”. To customize these settings for a particular channel, add the 'dispatcher-policy' sub-element and provide one or more of the attributes shown below:

```
<channel id="exampleChannel" publish-subscribe="true">
    <dispatcher-policy max-messages-per-task="25"
                       receive-timeout="10"
                       rejection-limit="3"
                       retry-interval="500"
                       should-fail-on-rejection-limit="false"/>
</channel>
```

Configuring Message Endpoints

To create a Message Endpoint instance, use the 'endpoint' element with the 'input-channel' and 'handler-ref' attributes:

```
<endpoint input-channel="exampleChannel" handler-ref="exampleHandler"/>
```

The configuration above assumes that "exampleHandler" is an actual implementation of the

`MessageHandler` interface as described in Section 2.3, “MessageHandler”. To delegate to an arbitrary method of any object, simply add the "handler-method" attribute.

```
<endpoint input-channel="exampleChannel" handler-ref="somePojo" handler-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 for a value in the message header's 'replyChannelName' property. If that value is available, it will attempt to resolve the channel by performing a lookup in the `ChannelRegistry`. If the message header does not contain a 'replyChannelName' property, then it will fallback to its own 'defaultOutputChannel' property. If neither is available, then a `MessageHandlingException` will be thrown. To configure the default output channel when using the XML namespace, provide the 'default-output-channel' attribute:

```
<endpoint input-channel="exampleChannel"
          handler-ref="somePojo"
          handler-method="someMethod"
          default-output-channel="replyChannel"/>
```

When the `MessageBus` registers the endpoint, it will activate the subscription by assigning the endpoint to the input channel's dispatcher. The dispatcher is capable of handling multiple endpoint subscriptions for its channel and delegates to a scheduler for managing the tasks that pull messages from the channel and push them to the endpoints. To configure the polling period for an individual endpoint's schedule, provide a 'schedule' sub-element with the 'period' in milliseconds:

```
<endpoint input-channel="exampleChannel" handler-ref="exampleHandler"/>
    <schedule period="3000"/>
</endpoint>
```

Note

Individual endpoint schedules only apply for "Point-to-Point" channels, since in that case only a single subscriber needs to receive the message. On the other hand, when a Spring Integration channel is configured as a "Publish-Subscribe" channel, then the dispatcher will drive all endpoint notifications according to its own default schedule, and any 'schedule' element configured for those endpoints will be ignored.

One of the most important configuration options for endpoints is the concurrency policy. Each endpoint is capable of managing a thread pool for its handler, and the values you provide for that pool's core and max size can make a substantial difference in how the handler performs under load. These settings are available per-endpoint since the performance characteristics of an endpoint's handler 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 'concurrency' sub-element and one or more of the properties shown below:

```
<endpoint input-channel="exampleChannel" handler-ref="exampleHandler"/>
    <concurrency core="5" max="25" queue-capacity="20" keep-alive="120"/>
</endpoint>
```

Recall the default concurrency policy values as listed in Table 2.5, “Properties of the ConcurrencyPolicy”.

Tip

The default queue capacity of 0 triggers the creation of a `SynchronousQueue`. In many cases, this is preferable since the direct handoff eliminates the chance of a message handling task being "stuck" in the queue (thread pool executors will favor adding to the queue rather than increasing the pool size). Specifically, whenever a dispatcher for a Point-to-Point channel has more than one subscribed endpoint, a task that is rejected due to an exhausted thread pool can be handled immediately by another endpoint whose pool has one or more threads available. On the other hand, when a particular channel/endpoint may be expecting bursts of activity, setting a queue capacity value might be the best way to accommodate the volume.

Configuring the Message Bus

As described in Section 2.4, “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 the 'message-bus' element accepts a reference with its 'error-channel' attribute:

```
<message-bus error-channel="errorChannel"/>

<channel id="errorChannel" publish-subscribe="true" capacity="500"/>
```

When exceptions occur in an endpoint's execution of its `MessageHandler` callback, those exceptions will be wrapped in `ErrorMessages` and sent to the Message Bus' 'errorChannel' by default. To enable global error handling, simply register a handler on that channel. For example, you can configure Spring Integration's `PayloadTypeRouter` 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.

The 'message-bus' element accepts two more optional attributes. First is the size of the dispatcher thread pool. The dispatcher threads are responsible for polling channels and then passing the messages to handlers. 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. Finally, the Message Bus is capable of automatically creating channel instances (with default settings) if an endpoint registers a subscription by providing the name of a channel that the bus does not recognize.

```
<message-bus dispatcher-pool-size="25" auto-create-channels="true"/>
```

Configuring Channel Adapters

The most convenient way to configure Channel Adapters is by using the namespace support. The following examples demonstrate the namespace-based configuration of source and target adapters (Spring Integration 1.0 M1 includes namespace support for JMS and Files):

```
<jms-source connection-factory="connectionFactory" destination="inputQueue" channel="inputChannel1"/

<jms-target connection-factory="connectionFactory" destination="outputQueue" channel="outputChannel1"/>

<file-source directory="/tmp/input" channel="inputChannel2" poll-period="10000"/>

<file-target directory="/tmp/output" channel="outputChannel2"/>
```

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) {
        ...
    }
}
```

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.

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

    @Handler
```

```
    public void processFoo(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 message header's 'replyChannelName' property will be used if available, and the endpoint's default output is the fallback. To configure the default output for an annotation-driven endpoint, provide the 'defaultOutput' attribute on the `@MessageEndpoint`.

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

Finally, just as the 'schedule' sub-element and its 'period' attribute can be provided for a namespace-based endpoint, the 'pollPeriod' attribute can be provided on the `@MessageEndpoint`.

```
@MessageEndpoint(input="exampleChannel", pollPeriod=3000)
```

Two additional annotations are supported, and both act as a special form of handler method: `@Router` and `@Splitter`. As with the `@Handler` annotation, methods annotated with either of these two annotations can either accept the `Message` itself or the message payload type as the parameter. 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 map to either a property or attribute name.

```
@Router(property="customerType")
public String route(String customerType)

@Router(attribute="orderStatus")
public List<String> route(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. The `@Splitter` annotation expects a 'channel' attribute that specifies the channel name to which those messages should be sent.

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

The `@Publisher` annotation is a convenience for sending messages with AOP *after-returning advice*. 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";
}
```

Similarly, the `@Subscriber` annotation triggers the retrieval of messages from a channel, and the payload of each message will then be sent as input to an arbitrary method. This is one of the simplest ways to configure asynchronous, event-driven behavior:

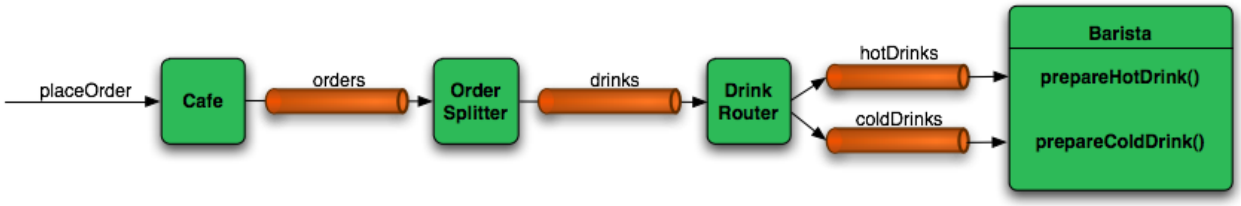
```
@Subscriber(channel="fooChannel")
public void log(String foo) {
    System.out.println(foo);
}
```

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 Milestone 1 release. This sample is inspired by one of the samples featured in Gregor Hohpe's Ramblings [http://www.eaipatterns.com/ramblings.html].

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



The `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"/>

    <endpoint input-channel="coldDrinks" handler-ref="barista" handler-method="prepareColdDrink"/>
    <endpoint input-channel="hotDrinks" handler-ref="barista" handler-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")
public class OrderSplitter {

    @Splitter(channel="drinks")
    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 <endpoint> 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 = 1000;
    private long coldDrinkDelay = 700;

    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);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("prepared hot drink #" + hotDrinkCounter.incrementAndGet() + ": " + drink);
    }

    public void prepareColdDrink(Drink drink) {
        try {
            Thread.sleep(this.coldDrinkDelay);
```

```
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("prepared cold drink #" + coldDrinkCounter.incrementAndGet() + ": " + drink);
    }
}
```

As you can see from the code excerpt above, the barista methods have different delays. 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 70 of the hot drinks. This is to be expected based on their respective delays of 700 and 1000 milliseconds. However, by configuring the endpoint concurrency, you can dramatically change the results. For example, on my machine, the following single modification causes all 100 hot drinks to be prepared before the 4th cold drink is ready:

```
<endpoint input-channel="coldDrinks" handler-ref="barista" handler-method="prepareColdDrink"/>

<endpoint input-channel="hotDrinks" handler-ref="barista" handler-method="prepareHotDrink">
    <concurrency core="25" max="50"/>
</endpoint>
```

In addition to experimenting with the 'concurrency' settings, you can also try adding the 'schedule' sub-element as described in the section called “Configuring Message Endpoints”. Additionally, you can experiment with the channel's configuration, such as adding a 'dispatcher-policy' as described in the section called “Configuring Message Channels”. If you want to explore the sample in more detail, the source JAR is available in the "dist" directory: 'spring-integration-samples-sources-1.0.0.m1.jar'.

6. Additional Resources

6.1 Spring Integration Home

The definitive source of information about Spring Integration is the Spring Integration Home [http://www.springframework.org/spring-integration] 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.