Spring Integration Reference Manual

Mark Fisher
Marius Bogoevici

1.0.0.M4 (Milestone 4)

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:

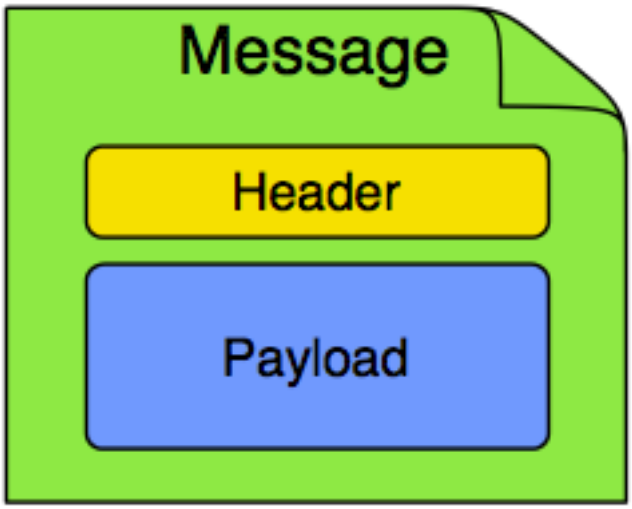
- Components should be *loosely coupled* for modularity and testability.
- The framework should enforce *separation of concerns* between business logic and integration logic.
- Extension points should be abstract in nature but within well-defined boundaries to promote *reuse* and *portability*.

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.

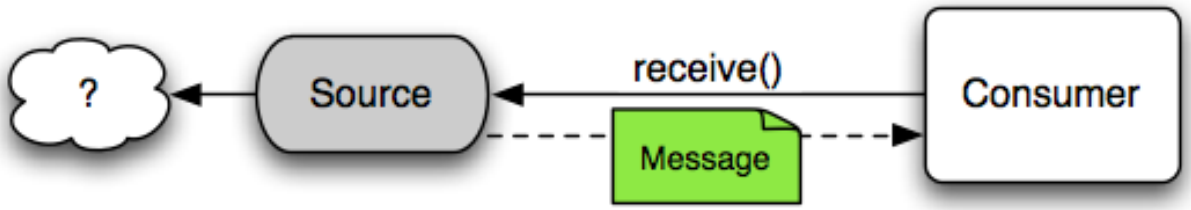
Message

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 Source

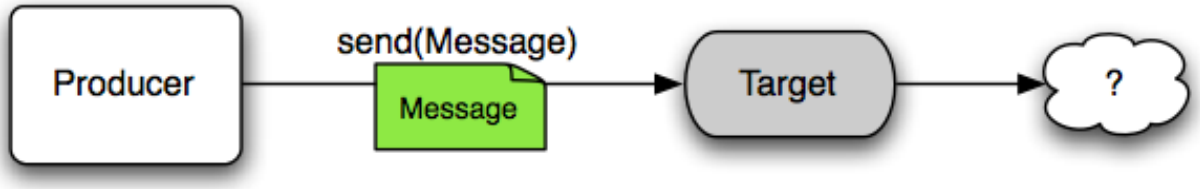
Since a Spring Integration Message is a generic wrapper for any Object, there is no limit to the number of potential sources for such messages. In fact, a Source implementation can act as an adapter that converts Objects from any other system into Spring Integration Messages.



To facilitate the conversion of Objects to Messages, Spring Integration also defines a strategy interface for creating Messages called `MessageCreator`. While it is relatively easy to implement Source directly, an adapter is also available for invoking arbitrary methods on plain Objects. Also, several Source implementations are already available within the Spring Integration Adapters module. For a detailed discussion of the various adapters, see Chapter 3, *Adapters*.

Message Target

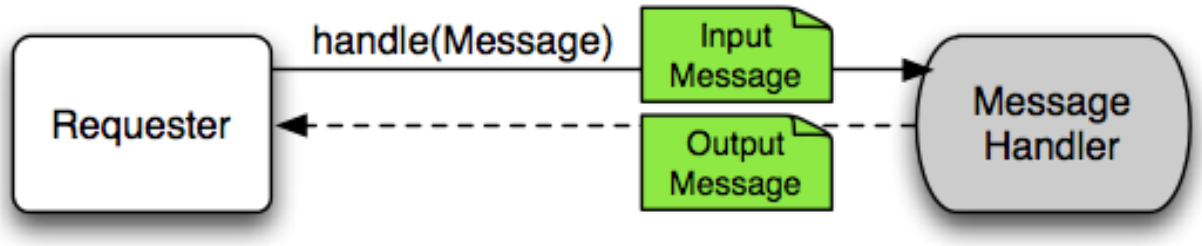
Just as a Source enables Message reception, a Target handles the responsibility of sending Messages. As with a Source, a Target can act as an adapter that converts Messages into the Objects expected by some other system.



Spring Integration provides a strategy interface for mapping Messages to Objects called `MessaegMapper`. The Target interface may be implemented directly, but an adapter is also available for invoking arbitrary methods on plain Objects (delegating to the Message-mapping strategy in the process). As with Sources, several Target implementations are already available within the Spring Integration Adapters module as discussed in Chapter 3, *Adapters*.

Message Handler

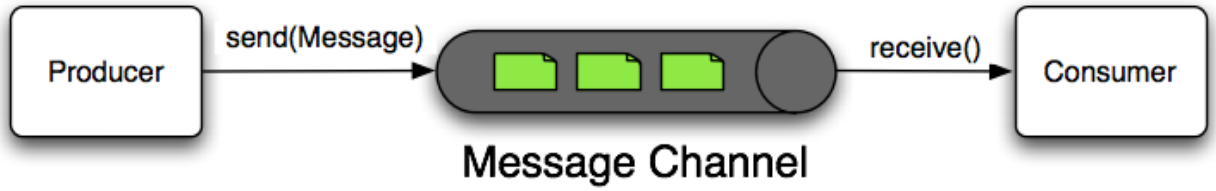
As described above, the Source and Target components support conversion between Objects and Messages so that application code and/or external systems can be connected to a Spring Integration application rather easily. However, both Source and Target are unidirectional while the application code or external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



As with the Source and Target, Spring Integration also provides an adapter that itself implements the Message Handler interface while supporting the invocation of arbitrary methods on plain Objects. The adapter relies upon the message-creating and message-mapping strategies to handle the bidirectional Object/Message conversion. 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 Source and Target.



Spring Integration provides a number of different channel implementations: QueueChannel, PriorityChannel, RendezvousChannel, DirectChannel, and ThreadLocalChannel. These are described in detail in Section 2.4, “MessageChannel”.

Message Endpoint

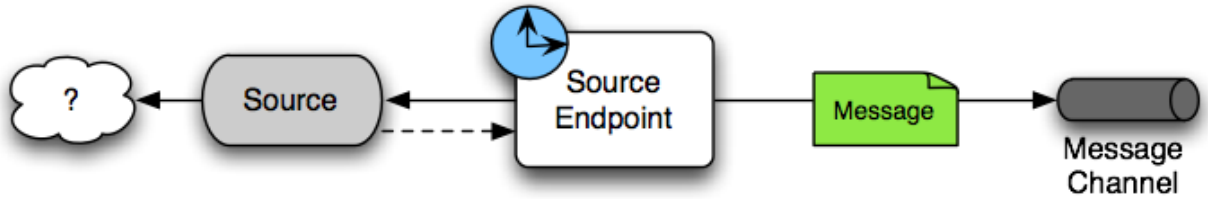
Thus far, the component diagrams show Consumers, Producers, and Requesters invoking the Source,

Target, and Message Handlers 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 Producers, Consumers, 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 three types of endpoints - one for each of the component types described above: Source Endpoint, Target Endpoint, and Handler Endpoint.

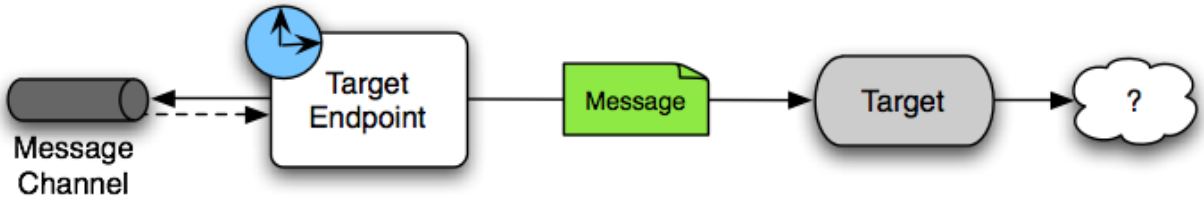
Source Endpoint

A Source Endpoint connects any Source implementation to a Message Channel. The invocation of the Source's receive operation is controlled by scheduling information provided within the Source Endpoint's configuration. Any time the receive operation returns a non-null Message, it is sent to the channel.



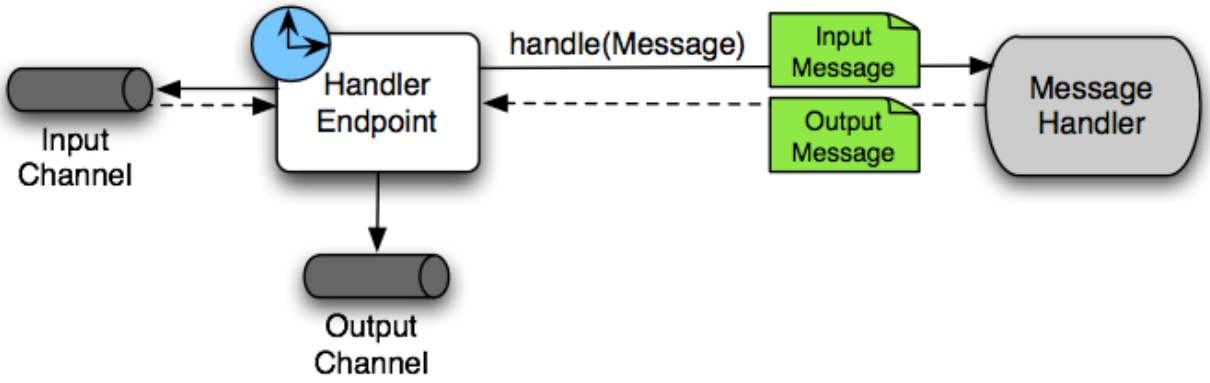
Target Endpoint

A Target Endpoint connects a Message Channel to any Target implementation. The invocation of the Message Channel's receive operation is controlled by scheduling information provided within the Target Endpoint's configuration. Any time a non-null Message is received from the channel, it is sent to the Target.



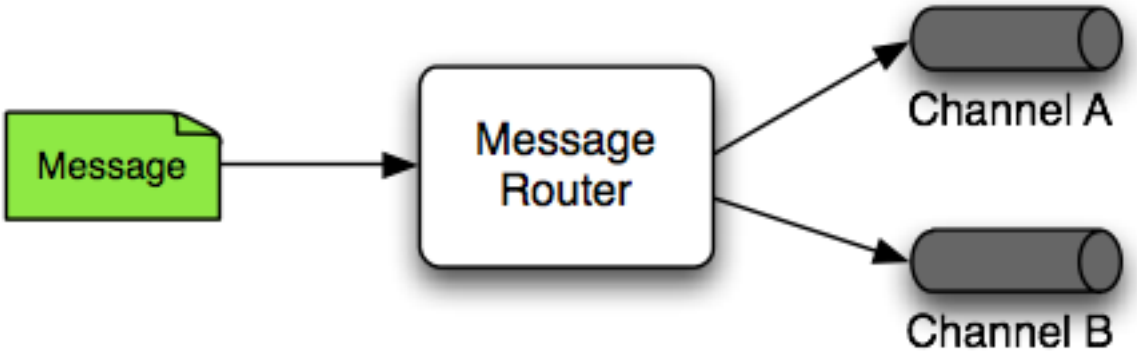
Handler Endpoint

Since Message Handler's are capable of returning reply Messages, the Handler Endpoint has some additional responsibilities. The general behavior is the same as the Target Endpoint, but the Handler Endpoint must make a distinction between "input-channel" and "output-channel". Whenever the Message Handler does return 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 as the Message header's "return address" if available.



Message Router

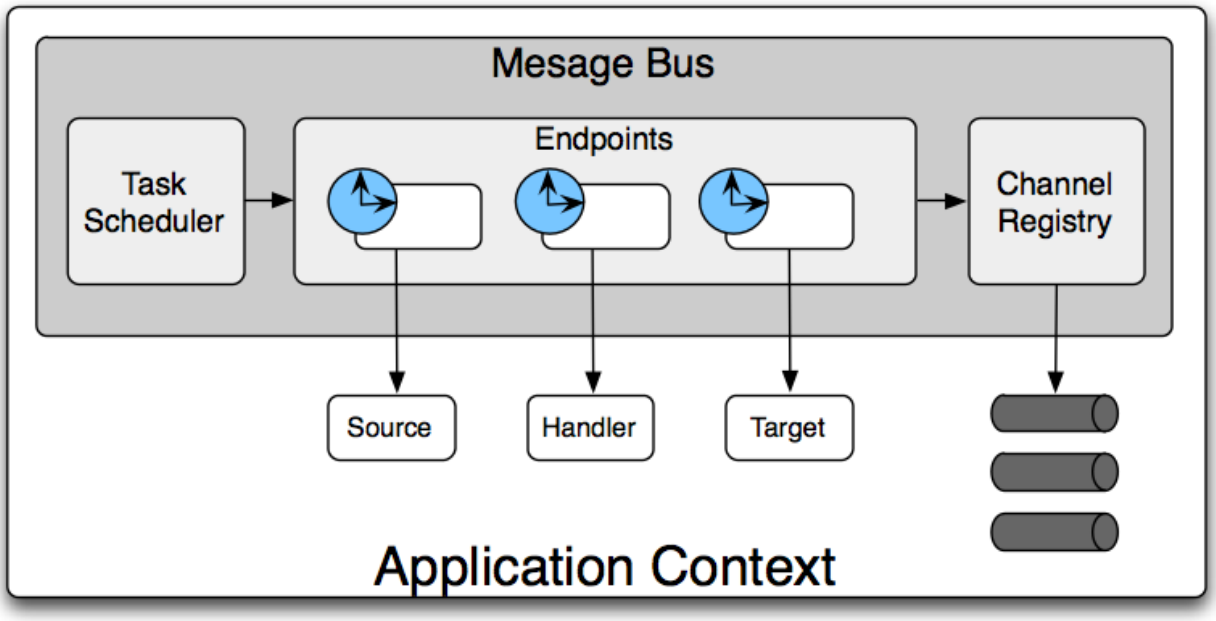
A Message Router is a particular type of `MessageHandler` that is capable of receiving a Message and then deciding what channel or channels should receive the Message next. Typically the decision is based upon the Message's content and/or metadata. A Message Router is often used as a dynamic alternative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



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();
    boolean isExpired();
}
```

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
returnAddress	java.lang.Object (can be a String or MessageChannel)
sequenceNumber	int
sequenceSize	int
priority	MessagePriority (an *enum*)
properties	java.util.Properties
attributes	Map<String,Object>

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

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

When no id is provided, a random unique id will be generated. The constructor that accepts a `MessageHeader` will copy properties, attributes, and any 'returnAddress' from the provided header. There are also two convenient subclasses available currently: `StringMessage` and `ErrorMessage`.

The latter accepts any `Throwable` object as its payload.

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

2.2 Source

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

```
public interface Source<T> {
    Message<T> receive();
}
```

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.setMethod("sourceMethod");
Message<?> result = source.receive();
```

It is generally more common to configure a `MethodInvokingSource` in XML by providing a bean reference.

```
<source-adapter id="source" ref="sourceObject" method="sourceMethod"/>
```

2.3 Target

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

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

As with the `Source`, 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"));
```

Likewise, the corresponding XML configuration is very similar to that of `MethodInvokingSource`.

```
<target-adapter id="target" ref="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 `MessageChannel` interface is defined as follows.

```
public interface MessageChannel {
    String getName();
    void setName(String name);
    DispatcherPolicy getDispatcherPolicy();
    boolean send(Message message);
    boolean send(Message message, long timeout);
    Message receive();
    Message receive(long timeout);
    List<Message<?>> clear();
    List<Message<?>> purge(MessageSelector selector);
}
```

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.

Spring Integration provides several different implementations of the `MessageChannel` interface. Each is briefly described in the sections below.

QueueChannel

The `QueueChannel` implementation wraps a queue. 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`' property within each message's header. 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` is significantly different than the channel implementations described thus far. It's primary purpose is to enable a single thread to perform the operations on "both sides" of the channel. For example, if a `HandlerEndpoint` is subscribed to a `DirectChannel`, then sending a Message to that channel will trigger invocation of the handler *directly in the sender's thread*. The key motivation for providing a channel implementation with this behavior is to support transactions. If the send call is invoked within the scope of a transaction, then the outcome of the handler 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 could collect its replies.

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 {
    boolean preSend(Message<?> message, MessageChannel channel);
    void postSend(Message<?> message, MessageChannel channel, boolean sent);
    boolean preReceive(MessageChannel channel);
    void 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 `boolean` value can return '`false`' to prevent the send or receive operation from proceeding (send would return 'false' and receive would return 'null').

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` methods are empty, and the `boolean` methods return `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 boolean preSend(Message<?> message, MessageChannel channel) {
        sendCount.incrementAndGet();
        return true;
    }
}
```

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 `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 one of the primary responsibilities that it assumes is connecting the channels to the handlers. It also connects Sources and Targets to channels, and it manages the scheduling of pollers and dispatchers.

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 `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 concurrencyPolicy)
```

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, and typically developers will not even use this API directly since the metadata can be provided in XML and/or annotations. 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
publishSubscribe	false	whether the dispatcher should attempt to publish to all of its handlers (rather than just one)
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)

Property Name	Default Value	Description
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 as an implementation of the `Schedule` interface. This is an abstraction designed to allow extensibility of schedulers for messaging tasks. Currently, there is a single implementation named `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

Property Name	Default Value	Description
		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 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

2.8 MessageEndpoint

As described in Chapter 1, *Spring Integration Overview*, there are three implementations of the `MessageEndpoint` interface: `SourceEndpoint`, `TargetEndpoint`, and `HandlerEndpoint`. These endpoints provide the metadata necessary for the `MessageBus` to manage `Sources`, `Targets`, and `MessageHandlers` respectively.

For a `SourceEndpoint`, the `MessageBus` schedules a task for polling the `Source` based on the provided schedule.

When a `Target` or `MessageHandler` is registered with the `MessageBus`, the bus assigns it to a dispatcher that polls a `MessageChannel` based on the provided schedule. Targets and handlers may also provide concurrency settings in which case a thread pool will be created for asynchronous processing of messages.

Rather than programming to the API directly, it is simpler and more common to register sources, targets, and handlers with either XML or annotation-based metadata. Then, the message endpoint is an internal responsibility of the bus. The configuration options are discussed in detail in the section called “Configuring Message Endpoints”.

2.9 MessageSelector

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

```
public interface MessageSelector {
    boolean accept(Message<?> message);
}
```

A `MessageEndpoint` can be configured with zero or more selectors, 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, the `MessageChannel`'s 'purge' method accepts a selector:

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

There is even 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 RequestReplyTemplate

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 `RequestReplyTemplate` that supports a variety of request-reply scenarios. For example, it is possible to send a request and wait for a reply.

```
RequestReplyTemplate template = new RequestReplyTemplate(requestChannel);
Message reply = template.request(new StringMessage("test"));
```

In that example, a temporary anonymous channel would be used internally by the template. However, the 'replyChannel' may be configured explicitly in which case the template will manage the reply correlation.

```
RequestReplyTemplate template = new RequestReplyTemplate(requestChannel);
template.setReplyChannel(replyChannel);
Message reply = template.request(new StringMessage("test"));
```

2.11 MessagingGateway

Even though the `RequestReplyTemplate` 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) { ... }
```

It enables configuration of a request and/or reply channel and delegates to 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.

3. Adapters

3.1 Introduction

Spring Integration provides a number of implementations of the `Source` and `Target` interfaces that serve as adapters for interacting with external systems or components that are not part of the messaging system. Configuring these source and target implementations within `SourceEndpoints` and `TargetEndpoints` provides an implementation of the *Channel Adapter* pattern. Essentially, the external system or component sends-to and/or receives-from a `MessageChannel`. In the 1.0 Milestone 4 release, Spring Integration includes source and target implementations for JMS, RMI, Files, Streams, Spring's HttpInvoker and Spring ApplicationEvents. A source adapter for FTP is also available as well as target adapters for sending e-mail and invoking Web Services.

Adapters that allow the 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". For example, Spring Integration provides a `JmsSource` that is *polled by* the bus-managed scheduler, but it 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.

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 `SourceEndpoint` 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 `Target` 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 `RmiSourceAdapter` 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:

```
RmiSourceAdapter rmiSourceAdapter = new RmiSourceAdapter(channel);
```

The `RmiTargetAdapter` encapsulates the creation of a proxy that is capable of communicating with an `RmiSourceAdapter` 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 `RmiSourceAdapter` (the prefix is available as a constant).

```
String url = "http://somehost:1099/" + RmiSourceAdapter.SERVICE_NAME_PREFIX + "someChannel";
RmiTargetAdapter rmiTargetAdapter = new RmiTargetAdapter(url);
```

3.4 HttpInvoker Adapters

The source and target 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 `HttpInvokerSourceAdapter` 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.HttpInvokerSourceAdapter">
    <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 source adapter. As with the `RmiTargetAdapter`, the `HttpInvokerTargetAdapter` only requires the URL that matches an instance of `HttpInvokerSourceAdapter` 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 `SourceEndpoint`.

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 `SourceEndpoint`. 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.FtpSourceAdapter">
    <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 attributes in the `MessageHeader` with the following constants defining the keys:

```
MailAttributeKeys.SUBJECT
MailAttributeKeys.TO
MailAttributeKeys.CC
MailAttributeKeys.BCC
MailAttributeKeys.FROM
MailAttributeKeys.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="mailTargetAdapter"
    class="org.springframework.integration.adapter.mail.MailTargetAdapter">
    <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: `SimpleWebServiceTargetAdapter` and `MarshallingWebServiceTargetAdapter`. 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.

```
simpleAdapter = new SimpleWebServiceTargetAdapter(uri);

marshallingAdapter = new MarshallingWebServiceTargetAdapter(uri, marshaller);
```

Either adapter can then be referenced from a `HandlerEndpoint` that is subscribed to a `MessageChannel`. The endpoint is then responsible for passing the response to the proper reply channel. It will first check for an "output-channel" on the endpoint itself and will fallback to a *returnAddress* on the original message's header.

For more detail on the inner workings, see the Spring Web Services reference guide's chapter covering client access [http://static.springframework.org/spring-ws/site/reference/html/client.html] as well as the chapter covering Object/XML mapping [http://static.springframework.org/spring-ws/site/reference/html/oxm.html].

3.9 Stream Adapters

Spring Integration also provides adapters for streams. Both `ByteStreamSource` and `CharacterStreamSource` implement the `Source` interface. By configuring one of these within a `SourceEndpoint`, 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 `Target` interface, they can be referenced from a `TargetEndpoint` 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 `SourceEndpoint` 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 `TargetEndpoint` (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 [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, you can use the generic 'channel' element:

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

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>
```

To create a Datatype Channel [http://www.eaipatterns.com/DatatypeChannel.html] that only accepts messages containing a certain payload type, provide the fully-qualified class name in the channel element's `datatype` attribute:

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

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

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

When using the "channel" element, the creation of the channel instances will be deferred to the `ChannelFactory` defined on the `MessageBus` (see below).

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 <priority-channel/> element

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

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

By default, the channel will consult the `MessagePriority` value in the message's header. 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 three endpoint types (source, target, and handler) has its own element in the namespace.

The <source-endpoint/> element

A `SourceEndpoint` connects an implementation of the `Source` interface to a `MessageChannel`. The <source-endpoint/> therefore requires these two references as well as the scheduling information so that the `MessageBus` can manage the message-receiving tasks.

```
<source-endpoint source="exampleSource" channel="exampleChannel">
    <schedule period="5000"/>
</source-endpoint>
```

The <target-endpoint/> element

A `TargetEndpoint` connects a `MessageChannel` to an implementation of the `Target` interface. The <target-endpoint/> requires these two references.

```
<target-endpoint input-channel="exampleChannel" target="exampleTarget"/>
```

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:

```
<target-endpoint input-channel="exampleChannel" target="exampleTarget">
    <schedule period="3000"/>
</target-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.

The <target-endpoint/> accepts additional attributes and child elements, but since these configuration options are also available for the <handler-endpoint/> element, they will be discussed below.

The <handler-endpoint/> element

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

```
<handler-endpoint input-channel="exampleChannel" handler="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.

```
<handler-endpoint input-channel="exampleChannel" handler="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 an "output-channel" was provided in the endpoint configuration:

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

If no "output-channel" is available, it will next check the message header's '`returnAddress`' property. If that value is available, it will then check its type. If it is a `MessageChannel`, the reply message will be sent to that channel. If it is a `String`, then the endpoint will attempt to resolve the channel by performing a lookup in the `ChannelRegistry`.

To reverse the order so that the 'returnAddress' is given priority over the endpoint's "output-channel", then provide the "return-address-overrides" attribute with a value of 'true':

```
<handler-endpoint input-channel="exampleChannel" output-channel="replyChannel"
                  handler="somePojo" method="someMethod" return-address-overrides="true"/>
```

If neither is available, then a `MessageHandlingException` will be thrown.

Handler and Target 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.

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

Another important configuration option for handler and target endpoints is the concurrency policy. Each endpoint is capable of managing a thread pool for its handler or target, and the values you provide for that pool's core and max size can make a substantial difference in how the handler or target performs under load. These settings are available per-endpoint since the performance characteristics of an endpoint's handler or target 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:

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

```
</handler-endpoint>
```

Recall the default concurrency policy values as listed in Table 2.5, “Properties of the ConcurrencyPolicy”. If no concurrency settings are provided (i.e. a *null* `ConcurrencyPolicy`), the endpoint's handler or target will be invoked in the caller's thread. Note that the "caller" is usually the dispatcher except in the case of a `DirectChannel` (see the section called “DirectChannel” for more detail).

Tip

For the concurrency settings, 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 2.7, “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 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 a concurrent 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. 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 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 concurrency policy is provided to an endpoint, then it will be invoked in the dispatcher's thread (with the exception of `DirectChannels`).

Also, the Message Bus is capable of automatically creating channel instances if an endpoint registers a subscription by providing the name of a channel that the bus does not recognize.

```
<message-bus auto-create-channels="true"/>
```

Finally, the type of channel that gets created automatically by the bus can be customized by using the "channel-factory" attribute on the "message-bus" definition as in the following example:

```
<message-bus channel-factory="channelFactoryBean"/>

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

With this definition, all the channels created automatically will be `PriorityChannel` instances. Without the "channel-factory" element, 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 sources and targets:

```
<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-source id="rmiSource" request-channel="rmiSourceInput"/>

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

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

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

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

<ws-target 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 `Source` and `Target` interfaces do not accept any 'channel' references. To connect such sources and targets to a channel, register them within an endpoint. For example, here is a File source with an endpoint whose polling will be scheduled to execute every 30 seconds by the `MessageBus`.

```
<source-endpoint source="fileSource" channel="exampleChannel">
    <schedule period="30000"/>
</source-endpoint>

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

Likewise, here is an example of a JMS target that is registered with a target-endpoint whose Messages will be received from the "exampleChannel" that is polled every 500 milliseconds.

```
<target-endpoint input-channel="exampleChannel" target="jmsTarget">
    <schedule period="500"/>
</target-endpoint>

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

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 bar(Foo foo) {
        ...
    }
}
```

When the method parameter should be mapped from a value in the `MessageHeader`, another option is to use the `@HeaderAttribute` and/or `@HeaderProperty` parameter annotations.

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

    @Handler
    public void bar(@HeaderAttribute("fooAttrib") Foo foo) {
        ...
    }
}
```

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

    @Handler
    public void bar(@HeaderProperty("foo") String input) {
        ...
    }
}
```

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 'returnAddress' 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 'schedule' sub-element and its 'period' attribute can be provided for a namespace-based endpoint, the `@Polled` annotation can be provided with the `@MessageEndpoint` annotation.

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

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

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

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

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 use the same parameter annotations that were introduced above.

```
@Router
public String route(@HeaderProperty("customerType") String customerType)

@Router
public List<String> route(@HeaderAttribute("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 `@Publisher` annotation is convenient 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:

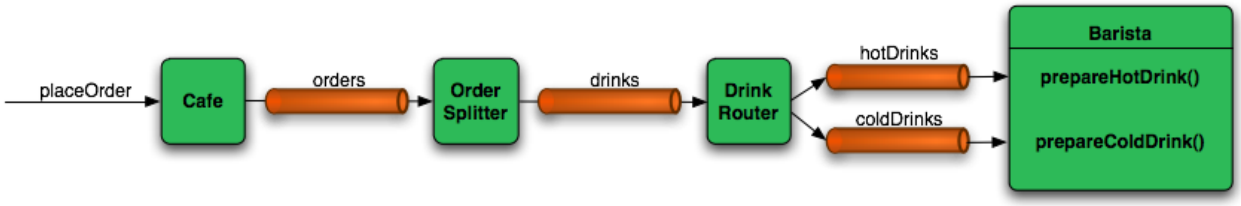
```
@Subscriber(channel="fooChannel")
public void log(String foo) {
    System.out.println(foo);
}
```

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 distribution. 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"/>

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

    <handler-endpoint input-channel="hotDrinks" handler="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 <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 = 2000;
    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);
        } 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 50 of the hot drinks. This is to be expected based on their respective delays of 1000 and 2000 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 5th cold drink is ready:

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

<handler-endpoint input-channel="hotDrinks" handler="barista" method="prepareHotDrink">
    <concurrency core="25" max="50"/>
</handler-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 "src" directory: 'org.springframework.integration.samples-sources-1.0.0.M4.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.