SAMPLE CHAPTER

Spring Integration IN ACTION

Mark Fisher
Jonas Partner
Marius Bogoevici
Iwein Fuld

FOREWORD BY Rod Johnson

Spring Integration in Action

by Mark Fisher, Jonas Partner,
Marius Bogoevici, Iwein Fuld

Chapter 3

brief contents

PART 1 BACKGROUND 1

1 ■ Introduction to Spring Integration 3
2 ■ Enterprise integration fundamentals 24

PART 2 MESSAGING 43

3 ■ Messages and channels 45
4 ■ Message Endpoints 63
5 ■ Getting down to business 80
6 ■ Go beyond sequential processing: routing and filtering 104
7 ■ Splitting and aggregating messages 122

PART 3 INTEGRATING SYSTEMS 139

8 ■ Handling messages with XML payloads 141
9 ■ Spring Integration and the Java Message Service 155
10 ■ Email-based integration 180
11 ■ Filesystem integration 191
12 ■ Spring Integration and web services 208
13 ■ Chatting and tweeting 219

PART 4 ADVANCED TOPICS.....237

14 ■ Monitoring and management 239
15 ■ Managing scheduling and concurrency 258
16 ■ Batch applications and enterprise integration 276
17 ■ Scaling messaging applications with OSGi 292
18 ■ Testing 304

3 Messages and channels

This chapter covers

- Introducing messages and channels
- How different types of channels work
- Customizing channel functionality with dispatchers and interceptors

Now that you have an idea of the high-level concepts of messaging and Spring Integration, it's time to dive in deeper. As mentioned in chapter 1, the three fundamental concepts in Spring Integration are messages, channels, and endpoints. Endpoints are discussed in a chapter of their own (chapter 4), but first you get to read about messages and channels.

In our introduction to messages, we show how the information exchanged by Spring Integration is organized. When describing channels, we discuss the conduits through which messages travel and how the different components are connected. Spring Integration provides a variety of options when it comes to channels, so it's important to know how to choose the one that's right for the job. Finally, there are a couple of more advanced components you can use to enhance the functionality of channels: dispatchers and interceptors, which we discuss at the end of the chapter.

Throughout the remaining chapters of this book, many include a section called "Under the hood," where we discuss the reasoning behind the API and provide details about the internal workings of the concepts discussed. But in this chapter we focus on the API itself, because messages and channels are fundamental concepts. Understanding how Spring Integration represents them is an essential foundation for building the rest of your knowledge about the framework. The best way to get to trust something is to understand how it works.

Now let's go inside and look around!

3.1 Introducing Spring Integration messages

As you saw in chapter 2, message-based integration is one of the major enterprise integration styles. Spring Integration is based on it because it's the most flexible and allows the loosest coupling between components.

We start our journey through Spring Integration with one of its building blocks: the *message*. In this section, we discuss the message as a fundamental enterprise integration pattern and provide an in-depth look at how it's implemented in Spring Integration.

3.1.1 What's in a message?

A message is a discrete piece of information sent from one component of the software system to another. By *piece of information,* we mean data that can be represented in the system (objects, byte arrays, strings, and so forth) and passed along as part of a message. *Discrete* means that messages are typically independent units. Each discrete message contains all the information that's needed by the receiving component for performing a particular operation.

Besides the application data, which is destined to be consumed by the recipient, a message usually packs meta-information. The meta-information is of no interest to the application per se, but it's critical for allowing the infrastructure to handle the information correctly. We call this meta-information *message headers*.

The obvious analogy here is a letter. A letter contains an individual message destined to the recipient but is also wrapped in an envelope containing information that's useful to the mailing system but not to the recipient: a delivery address, the stamp indicating that mailing fees are paid, and various stamps applied by the mailing office for routing the message more efficiently.

Based on the role they fulfill, we distinguish three types of messages:

- Document messages that contain only information
- Command messages that instruct the recipient to perform various operations
- Event messages that indicate notable occurrences in the system and to which the recipient may react

Now that we have an overview of the message concept, we can talk about Spring Integration's approach to it. Next we look at it from an API and implementation standpoint and then see how to create a message. Usually, you won't need to create the

message yourself, but to understand the Spring Integration design, it's important to see how message creation works.

3.1.2 How it's done in Spring Integration

The nature of a message in Spring Integration is best expressed by the `Message` interface:

```
package org.springframework.integration;

public interface Message<T> {

  MessageHeaders getHeaders();

  T getPayload();

}
```

Let's examine this code in detail. A message is a generic wrapper around data that's transported between the different components of the system. The wrapper allows the framework to ignore the type of the content and to treat all messages in the same way when dealing with dispatching and routing.

Having an interface as the top-level abstraction enables the framework to be extensible by allowing extensions to create their own message implementations in addition to those included with the framework. It's important to remember that, as a user, creating your own message implementations is hardly necessary because you usually won't need to deal with the message objects directly.

The content wrapped by the message can be any Java `Object`, and because the interface is parameterized, you can retrieve the payload in a type-safe fashion.

Besides the payload, a message contains a number of *headers*, which are metadata associated with that message. For example, every message has an identifier header that uniquely identifies the message instance. In other cases, message headers may contain correlation information indicating that various individual messages belong to the same group. A message's headers may even contain information about the origin of the message. For example, if the message was created from data coming from an external source (like a JMS [Java Message Service] queue or the filesystem), the message may contain information specific to that source (like the JMS properties or the path to the file in the filesystem).

The `MessageHeaders` are a `Map` with some additional behavior. Their role is to store the header values, which can be any Java `Object`, and each value is referenced by a header name:

```
package org.springframework.integration;

public final class MessageHeaders
    implements Map<String, Object>, Serializable {
   /* implementation omitted */
}
```

WHERE ARE THE SETTERS?

As you may have noticed from the definition of the `Message` interface, there is no way to *set* something on a `Message`. Even the `MessageHeaders` are an immutable `Map`: trying to set a header on a given `Message` instance will yield an `UnsupportedOperationException`. This happens for a good reason: messages aren't modifiable. You might worry about how these messages are created and why they can't be modified. The short answer is that you don't need to create them yourself and that they're immutable to avoid issues with concurrency.

In publish-subscribe scenarios, the same instance of a message may be shared among different consumers. Immutability is a simple and effective way to ensure thread safety. Imagine what would happen if multiple concurrent consumers were to modify the same `Message` instance simultaneously by changing its payload or one of its header values.

As mentioned, Spring Integration provides a few implementations of its own for the `Message` interface, mostly for internal use. Instead of using those implementations directly or creating implementations of your own, you can use Spring Integration's `MessageBuilder` when you need to create a new `Message` instance.

The `MessageBuilder` creates messages from either a given payload or another message (a facility necessary for creating message copies). There are three steps in this process, shown in figure 3.1.

The figure shows the builder creating a message from the payload, but the same steps apply for creating a message from another message. Steps 1 and 3 are mandatory in that they have to take place whenever a new message is created using the `MessageBuilder`, whereas step 2 is optional and needs to be performed only when custom headers are required. It should be noted that the `MessageBuilder` initializes a number of headers that are needed by default by the framework, and this is one of the significant advantages of using the `MessageBuilder` over instantiating messages directly: you don't have to worry about setting up those headers yourself.

As an example, creating a new message with a `String` payload can take place as follows:

```
Message<String> helloMessage =
    MessageBuilder.withPayload("Hello, world!").build();
```

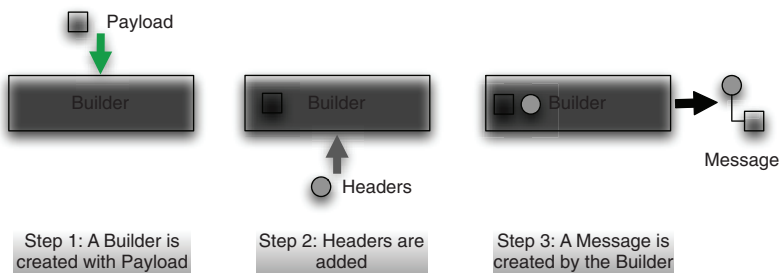

Figure 3.1 Creating a message with the `MessageBuilder`

The `MessageBuilder` also provides a variety of methods for setting the headers, including options for setting a header value only if that header hasn't already been set. A more elaborate variant of creating the `helloMessage` could be something like this:

```
Message<String> helloMessage =
   MessageBuilder.withPayload("Hello, world!")
                 .setHeader("custom.header", "Value")
                 .setHeaderIfAbsent("custom.header2", "Value2")
                 .build();
```

You can also create messages by copying the payload and headers of other messages and can change the headers of the resulting instance as necessary:

```
Message<String> anotherHelloMessage =
   MessageBuilder.fromMessage(helloMessage)
                 .setHeader("custom.header", "ChangedValue")
                 .setHeaderIfAbsent("custom.header2", "IgnoredValue")
                 .build();
```

Now you know what messages are in Spring Integration and how you can create them. Next, we discuss how messages are sent and received through *channels*.

3.2 Introducing Spring Integration channels

Messages don't achieve anything by sitting there all by themselves. To do something useful with the information they're packaging, they need to travel from one component to another, and for this they need *channels*, which are well-defined conduits for transporting messages across the system.

Let's go back to the letter analogy. The sender creates the letter and hands it off to the mailing system by depositing it in a well-known location: the mailbox. From there on, the letter is completely under the control of the mailing system, which delivers it to various waypoints until it reaches the recipient. The most that the sender can expect is a reply. The sender is unaware of who routes the message or, sometimes, even who may be the physical reader of the letter (think about writing to a government agency). From a logical standpoint, the channel is much like a mailbox: a place where components (producers) deposit messages that are later processed by other components (consumers). This way, producers and consumers are decoupled from each other and are only concerned about what kinds of messages they can send and receive, respectively.

One distinctive trait of Spring Integration, which differentiates it from other enterprise integration frameworks, is its emphasis on the role of channels in defining the enterprise integration strategy. Channels aren't just information transfer components; they play an active role in defining the overall application behavior. The business processing takes place in the endpoints, but you can alter the channel configuration to completely change the runtime characteristics of the application.

We explain channels from a logical perspective and offer overviews of the various channel implementations provided by the framework: what's characteristic to each of

them, and how you can get the most from your application by using the right kind of channel for the job.

3.2.1 Using channels to move messages

To connect the producers and consumers configured in an application, you use a channel. All channels in Spring Integration implement the following `MessageChannel` interface, which defines standard methods for sending messages. Note that it provides no methods for receiving messages:

```
package org.springframework.integration;

public interface MessageChannel {

    boolean send(Message<?> message);

    boolean send(Message<?> message, long timeout);

}
```

The reason no methods are provided for receiving messages is because Spring Integration differentiates clearly between two mechanisms through which messages are handed over to the next endpoint—polling and subscription—and provides two distinct types of channels accordingly.

3.2.2 I'll let you know when I've got something!

Channels that implement the `SubscribableChannel` interface, shown below, take responsibility for notifying subscribers when a message is available:

```
package org.springframework.integration.core;

public interface SubscribableChannel extends MessageChannel {

    boolean subscribe(MessageHandler handler);

    boolean unsubscribe(MessageHandler handler);

}
```

3.2.3 Do you have any messages for me?

The alternative is the `PollableChannel`, whose definition follows. This type of channel requires the receiver or the framework acting on behalf of the receiver to periodically check whether messages are available on the channel. This approach has the advantage that the consumer can choose when to process messages. The approach can also have its downsides, requiring a trade-off between longer poll periods, which may introduce latency in receiving a message, and computation overhead from more frequent polls that find no messages:

```
package org.springframework.integration.core;

public interface PollableChannel extends MessageChannel {

    Message<?> receive();
```

```
    Message<?> receive(long timeout);
}
```

It's important to understand the characteristics of each message delivery strategy because the decision to use one over the other affects the timeliness and scalability of the system. From a logical point of view, the responsibility of connecting a consumer to a channel belongs to the framework, thus alleviating the complications of defining the appropriate consumer types. To put it plainly, your job is to configure the appropriate channel type, and the framework will select the appropriate consumer type (polling or event-driven).

Also, subscription versus polling is the most important criterion for classifying channels, but it's not the only one. In choosing the right channels for your application, you must consider a number of other criteria, which we discuss next.

3.2.4 The right channel for the job

Spring Integration offers a number of channel implementations, and because `MessageChannel` is an interface, you're also free to provide your own implementations. The type of channel you select has significant implications for your application, including transactional boundaries, latency, and overall throughput. This section walks you through the factors to consider and through a practical scenario for selecting appropriate channels. In the configuration, we use the namespace, and we also discuss which concrete channel implementation will be instantiated by the framework.

In our flight-booking internet application, a booking confirmation results in a number of actions. Foremost for many businesses is the need to get paid, so making sure you can charge the provided credit card is a high priority. You also want to ensure that, as seats are booked, an update occurs to indicate one less seat is available on the flight so you don't overbook the flight. The system must also send a confirmation email with details of the booking and additional information on the check-in process. In addition to a website, the internet booking application exposes a REST interface to allow third-party integration for flight comparison sites and resellers. Because most of the airline's premium customers come through the airline's website, any design should allow you to prioritize bookings originating from its website over third-party integration requests to ensure that the airline's direct customers experience a responsive website even during high load.

The selection of channels is based on both functional and nonfunctional requirements, and several factors can help you make the right choice. Table 3.1 provides a brief overview of the technical criteria and the best practices you should consider when selecting the most appropriate channels.

Let's see how these criteria apply to our flight-booking sample.

Table 3.1 How do you decide what channel to use?

Decision factor	What factors must you consider?
Sharing context	– Do you need to propagate context information between the successive steps of a process? – Thread-local variables are used to propagate context when needed in several places where passing via the stack would needlessly increase coupling, such as in the transaction context. – Relying on the thread context is a subtle form of coupling and has an impact when considering the adoption of a highly asynchronous staged event-driven architecture (SEDA) model. It may prevent splitting the processing into concurrently executing steps, prevent partial failures, or introduce security risks such as leaking permissions to the processing of different messages.
Atomic boundaries	– Do you have all-or-nothing scenarios? – Classic example: bank transaction where credit and debit should either both succeed or both fail. – Typically used to decide transaction boundaries, which makes it a specific case of context sharing. Influences the threading model and therefore limits the available options when choosing a channel type.
Buffering messages	– Do you need to consider variable load? What is immediate and what can wait? – The ability of systems to withstand high loads is an important performance factor, but load is typically fluctuating, so adopting a thread-per-message-processing scenario requires more hardware resources for accommodating peak load situations. Those resources are unused when the load decreases, so this approach could be expensive and inefficient. Moreover, some of the steps may be slow, so resources may be blocked for long durations. – Consider what requires an immediate response and what can be delayed; then use a buffer to store incoming requests at peak rate, and allow the system to process them at its own pace. Consider mixing the types of processing—for example, an online purchase system that immediately acknowledges receipt of the request, performs some mandatory steps (credit card processing, order number generation), and responds to the client but does the actual handling of the request (assembling the items, shipping, and so on) asynchronously in the background.
Blocking and nonblocking operations	– How many messages can you buffer? What should you do when you can't cope with demand? – If your application can't cope with the number of messages being received and no limits are in place, you may exhaust your capacity for storing the message backlog or breach quality-of-service guarantees in terms of response turnaround. – Recognizing that the system can't cope with demands is usually a better option than continuing to build up a backlog. A common approach is to apply a degree of self-limiting behavior to the system by blocking the acceptance of new messages where the system is approaching its maximum capacity. This limit commonly is a maximum number of messages awaiting processing or a measure of requests received per second. – Where the requester has a finite number of threads for issuing requests, blocking those threads for long periods of time may result in timeouts or quality-of-service breaches. It may be preferable to accept the message and then discard it later if system capacity is being exceeded or to set a timeout on the blocking operation to avoid indefinite blocking of the requester.

Table 3.1 How do you decide what channel to use? *(continued)*

Decision factor	What factors must you consider?
Consumption model	– How many components are interested in receiving a particular message? – There are two major messaging paradigms: point-to-point and publish-subscribe. In the former, a message is consumed by exactly one recipient connected to the channel (even if there are more of them), and in the latter, the message is received by all recipients. – If the processing requirements are that the same message should be handled by multiple consumers, the consumers can work concurrently and a publish-subscribe channel can take care of that. An example is a mashup application that aggregates results from searching flight bookings. Requests are broadcast simultaneously to all potential providers, which will respond by indicating whether they can offer a booking. – Conversely, if the request should always be handled by a single component (for example, for processing a payment), you need a point-to-point strategy.

3.2.5 A channel selection example

Using the default channel throughout, we have three channels—one accepting requests and the other two connecting the services:

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

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
                   ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                   output-channel="emailConfirmationRequests"
                   ref="seatAvailabilityService" />

<channel id="emailConfirmationRequests" />

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

In Spring Integration, the default channels are `SubscribableChannels`, and the message transmission is synchronous. The effect is simple: one thread is responsible for invoking the three services sequentially, as shown in figure 3.2.

Because all operations are executing in a single thread, a single transaction encompasses those invocations. That assumes that the transaction configuration doesn't require new transactions to be created for any of the services.

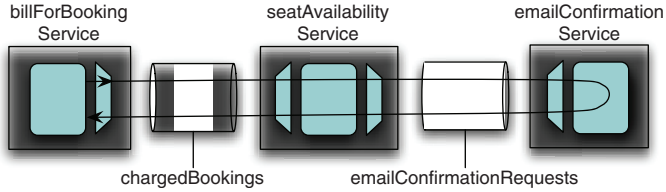

Figure 3.2 Diagram of threading model of service invocation in the airline application

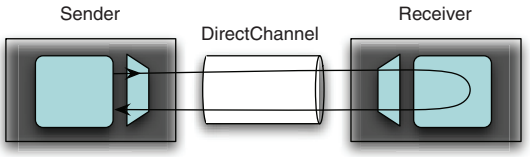

Figure 3.3 Diagram of threading model of service invocation when using a default channel

Figure 3.3 shows what you get when you configure an application using the default channels, which are subscribable and synchronous. But having all service invocations happening in one thread and encompassed by a single transaction is a mixed blessing: it could be a good thing in applications where all three operations must be executed atomically, but it takes a toll on the scalability and robustness of the application.

BUT EMAIL IS SLOW AND OUR SERVERS ARE UNRELIABLE

The basic configuration is good in the sunny-day case where the email server is always up and responsive, and the network is 100% reliable. Reality is different. Your application needs to work in a world where the email server is sometimes overloaded and the network sometimes fails. Analyzing your actions in terms of what you need to do now and what you can afford to do later is a good way of deciding what service calls you should block on. Billing the credit card and updating the seat availability are clearly things you need to do now so you can respond with confidence that the booking has been made. Sending the confirmation email isn't time critical, and you don't want to refuse bookings simply because the mail server is down. Therefore, introducing a queue between the mainline business logic execution and the confirmation email service will allow you to do just that: charge the card, update availability, and send the email confirmation when you can.

Introducing a queue on the `emailConfirmationRequests` channel allows the thread passing in the initial message to return as soon as the credit card has been charged and the seat availability has been updated. Changing the Spring Integration configuration to do this is as trivial as adding a child `<queue/>` element to the `<channel/>`:[1]

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

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
                   ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                   output-channel="emailConfirmationRequests"
                   ref="seatAvailabilityService" />

<channel id="emailConfirmationRequests">
   <queue />
</channel>

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

[1] This will also require either an explicit or default poller configuration for the consuming endpoint connected to the queue channel. We'll discuss that in detail within chapter 15.

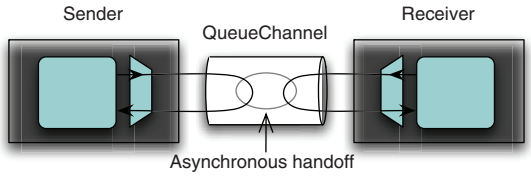

Figure 3.4 Diagram of threading model of service invocation when using a `QueueChannel`

Let's recap how the threading model changes by introducing the `QueueChannel`, shown in figure 3.4.

Because a single thread context no longer encompasses all invocations, the transaction boundaries change as well. Essentially, every operation that's executing on a separate thread executes in a separate transaction, as shown in figure 3.5.

By replacing one of the default channels with a buffered `QueueChannel` and setting up an asynchronous communication model, you gain some confidence that long-running operations won't block the application because some component is down or takes a long time to respond. But now you have another challenge: what if you need to connect one producer with not just one, but two (or more) consumers?

TELLING EVERYONE WHO NEEDS TO KNOW THAT A BOOKING OCCURRED

We've looked at scenarios where a number of services are invoked in sequence with the output of one service becoming the input of the next service in the sequence. This works well when the result of a service invocation needs to be consumed only once, but it's common that more than one consumer may be interested in receiving certain messages. In our current version of the channel configuration, successfully billed bookings that have been recorded by the seat availability service pass directly into a queue for email confirmation. In reality, this information would be of interest to a number of services within the application and systems within the enterprise, such as customer relationship management systems tracking customer purchases to better target promotions and finance systems monitoring the financial health of the enterprise as a whole.

To allow delivery of the same message to more than one consumer, you introduce a publish-subscribe channel after the availability check. The publish-subscribe channel provides one-to-many semantics rather than the one-to-one semantics provided by most channel implementations. One-to-many semantics are particularly useful when you want the flexibility to add additional consumers to the configuration; if the name of the publish-subscribe channel is known, that's all that's required for the configuration of additional consumers with no changes to the core application configuration.

The publish-subscribe channel doesn't support queueing, but it does support asynchronous operation if you provide a task executor that delivers messages to each of

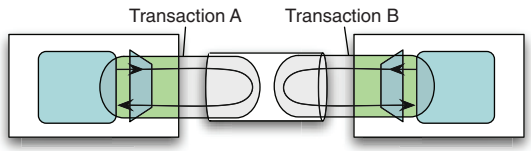

Figure 3.5 Diagram of transactional boundaries when a `QueueChannel` is used

the subscribers in separate threads. But this approach may still block the main thread sending the message on the channel where the task executor is configured to use the caller thread or block the caller thread when the underlying thread pool is exhausted.

To ensure that a backlog in sending email confirmations doesn't block either the sender thread or the entire thread pool for the task executor, you can connect the new publish-subscribe channel to the existing email confirmation queue by means of a *bridge*. The bridge is an enterprise integration pattern that supports the connection of two channels, which allows the publish-subscribe channel to deliver to the queue and then have the thread return immediately:

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

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
                   ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                   output-channel="completedBookings"
                   ref="seatAvailabilityService" />

<publish-subscribe-channel id="completedBookings" />

<bridge input-channel="completedBookings"
        output-channel="emailConfirmationRequests" />

<channel id="emailConfirmationRequests">
    <queue />
</channel>

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

Now it's possible to connect one producer with multiple consumers by means of a publish-subscribe channel. Let's get to the last challenge and emerge victoriously with our dramatically improved application: what if "first come, first served" isn't always right?

SOME CUSTOMERS ARE MORE EQUAL THAN OTHERS

Let's say you want to ensure that the airline's direct customers have the best possible user experience. To do that, you must prioritize the processing of their requests so you can render the response as quickly as possible. Using a comparator that prioritizes direct customers over indirect, you can modify the first channel to be a priority queue. This causes the framework to instantiate an instance of `PriorityChannel`, which results in a queue that can prioritize the order in which the messages are received. In this case, you provide an instance of a class implementing `Comparator<Message<?>>`:

```
<channel id="bookingConfirmationRequests">
    <priority-queue comparator="customerPriorityComparator" />
</channel>

<service-activator input-channel="bookingConfirmationRequests"
                   output-channel="chargedBookings"
```

```
                    ref="billForBookingService" />

<channel id="chargedBookings" />

<service-activator input-channel="chargedBookings"
                    output-channel="completedBookings"
                    ref="seatAvailabilityService" />

<publish-subscribe-channel id="completedBookings" />

<bridge input-channel="completedBookings"
        output-channel="emailConfirmationRequests" />

<channel id="emailConfirmationRequests">
    <queue />
</channel>

<outbound-channel-adapter channel="emailConfirmationRequests"
                          ref="emailConfirmationService" />
```

The configuration changes made in this section are an example of applying different types of channels for solving the different requirements. Starting with the defaults and working through the example, we replaced several channel definitions with the ones most suitable for each particular situation encountered. What's most important is that every type of channel has its own justification, and what may be advisable in one use case may not be advisable in another. We illustrated the decision process with the criteria we find most relevant in each case.

This wraps up the overview of the Spring Integration channels and makes way for some more particular components that can be used for fine tuning the functionality of a channel.

3.3 Channel collaborators

From time to time, creating more advanced applications requires going beyond sending and receiving messages. For this purpose, Spring Integration provides additional components that act as channel collaborators.

We next look at the `MessageDispatcher`, which controls how messages sent to a channel are passed to any registered handlers, and then we look at the `ChannelInterceptor`, which allows interception at key points, like the channel send and receive operations.

3.3.1 MessageDispatcher

In most of the examples given so far, a channel has a single message handler connected to it. Though having a single component that takes care of processing the messages that arrive on a channel is a pretty common occurrence, multiple handlers processing messages arriving on the same channel is also a typical configuration. Depending on how messages are distributed to the different message handlers, we can have either a competing consumers scenario or a broadcasting scenario.

In the broadcasting scenario, multiple (or all) handlers will receive the message. In the competing consumers scenario, from the multiple handlers that are connected

to a given channel, only one will receive and handle the message. Having multiple receivers that are capable of handling the message, even if only one of them will actually be selected to do the processing, is useful for load balancing or failover.

The strategy for determining how the channel implementation dispatches the message to the handlers is defined by the following `MessageDispatcher` interface:

```
package org.springframework.integration.dispatcher;

public interface MessageDispatcher {

    boolean addHandler(MessageHandler handler);

    boolean removeHandler(MessageHandler handler);

    boolean dispatch(Message<?> message);

}
```

Spring Integration provides two dispatcher implementations out of the box: `UnicastingDispatcher`, which, as the name suggests, delivers the message to at most one `MessageHandler`, and `BroadcastingDispatcher`, which delivers to zero or more. The `UnicastingDispatcher` provides an additional strategy interface, `LoadBalancingStrategy`, shown in the next code snippet, which determines which single `MessageHandler` of potentially many should receive any given message. The only provided implementation of this is the `RoundRobinLoadBalancingStrategy`, which works through the list of handlers, passing one message to each in turn.

```
package org.springframework.integration.dispatcher;

public interface LoadBalancingStrategy {

    public Iterator<MessageHandler>
               getHandlerIterator(Message<?> message,
                                  List<MessageHandler> handlers);

}
```

Providing your own implementations of `MessageDispatcher` is uncommon and should be resorted to only where other options don't provide the level of control desired over the process of delivering messages to handlers. An example where a custom `MessageDispatcher` could be appropriate is where different handlers are being used to deal with varying service levels. In this scenario, a custom dispatcher could be used to prioritize messages according to the defined service-level agreements. The dispatcher implementation could inspect the message and attempt to dispatch to the full set of handlers only where a message is determined to be high priority:

```
public class ServiceLevelAgreementAwareMessageDispatcher
                                  implements MessageDispatcher {

    private List<MessageHandler> highPriorityHandlers;

    private List<MessageHandler> lowPriorityHandlers;


    public boolean dispatch(Message<?> message){
       boolean highPriority = isHighPriority(message);
```

```
        boolean delivered = false;
        if(highPriority){
           delivered = attemptDelivery(highPriorityHandlers);
        }

        if(!delivered){
            delivered = attemptDelivery(lowPriorityHandlers);
        }

        return delivered;
    }

    ...
}
```

So far, we've seen how to control the way messages from a channel are distributed to the message handlers that are listening on that particular channel. Now let's see how to intervene in the process of sending and receiving messages.

3.3.2 ChannelInterceptor

Another important requirement for an integration system is that it can be notified as messages are traveling through the system. This functionality can be used for several purposes, ranging from monitoring of messages as they pass through the system to vetoing send and receive operations for security reasons. For supporting this, Spring Integration provides a special type of component called a *channel interceptor*.

The channel implementations provided by Spring Integration all allow the registration of one or more `ChannelInterceptor` instances. Channel interceptors can be registered for individual channels or globally. The `ChannelInterceptor` interface allows implementing classes to hook into the sending and receiving of messages by the channel:

```
package org.springframework.integration.channel;

public interface ChannelInterceptor {

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

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

    boolean preReceive(MessageChannel channel);

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

Just by looking at the names of the methods, it's easy to get an idea when these methods are invoked, but there's more to this component than simple notification. Let's review them one by one:

- `preSend` is invoked before a message is sent and returns the message that will be sent to the channel when the method returns. If the method returns null, nothing is sent. This allows the implementation to control what gets sent to the channel, effectively filtering the messages.

- `postSend` is invoked after an attempt to send the message has been made. It indicates whether the attempt was successful through the `boolean` flag it passes as an argument. This allows the implementation to monitor the message flow and learn which messages are sent and which ones fail.
- `preReceive` applies only if the channel is pollable. It's invoked when a component calls `receive()` on the channel, but before a Message is actually read from that channel. It allows implementers to decide whether the channel can return a message to the caller.
- `postReceive`, like `preReceive`, applies only to pollable channels. It's invoked after a message is read from a channel but before it's returned to the component that called `receive()`. If it returns null, then no message is received. This allows the implementer to control what, if anything, is actually received by the poller.

Creating a new type of interceptor is typically done by implementing the `ChannelInterceptor` interface:[2]

```
package siia.channels;

import org.springframework.integration.Message;
import org.springframework.integration.MessageChannel;
import org.springframework.integration.channel.interceptor
        ➥.ChannelInterceptorAdapter;

public class ChannelAuditor extends ChannelInterceptorAdapter {

  private AuditService auditService;

  public void setAuditService(AuditService auditService) {
      this.auditService = auditService;
  }

  public Message<?> preSend(Message<?> message,
                            MessageChannel channel) {
      this.auditService.audit(message.getPayload());
      return message;
  }
}
```

In this case, the interceptor intercepts the messages and sends their payloads to an audit service (injected in the interceptor itself). Before an email request is sent out, the audit service logs the charged bookings that were created by the application.

Setting up the interceptor on a channel is also straightforward. An interceptor is defined as a regular bean, and a special namespace element takes care of adding it to the channel, as follows:

```
<beans:bean id="auditInterceptor"
            class="siia.channels.ChannelAuditor">
   <beans:property name="auditService" ref="auditService"/>
</beans:bean>
<beans:bean id="auditService"  class="siia.channels.AuditService"/>
```

[2] Spring Integration provides a no-op implementation, `ChannelInterceptorAdapter`, that framework users can extend. It allows users to implement only the needed functionalities.

```
<channel id="chargedBookings">
    ....
    <interceptors>
       <beans:ref bean="auditInterceptor"/>
    </interceptors>
</channel>
```

Some of the best examples for understanding the use of the `ChannelInterceptor` come from Spring Integration, which supports two different types of interception: monitoring and filtering messages before they are sent on a channel.

EVEN SPRING INTEGRATION HAS ITS OWN INTERCEPTORS!

For the monitoring scenario, Spring Integration provides `WireTap`, an implementation of the more general *Wire Tap* enterprise integration pattern. As you saw earlier, it's easy to audit messages that arrive on a channel using a custom interceptor, but `WireTap` enables you to do this in an even less invasive fashion by defining an interceptor that sends copies of messages on a distinct channel. This means that monitoring is completely separated from the actual business flow, from a logical standpoint, but also that it can take place asynchronously. Here's an example:

```
<channel id="monitoringChannel/>

<channel id="chargedBookings">
    ....
    <interceptors>
       <wire-tap channel="monitoringChannel"/>
    </interceptors>
</channel>
```

For each booking you charge, you send a copy of the message on the `monitoringChannel`, where it can be analyzed by a monitoring handler independently of the main application flow.

The filtering scenario is based on the idea that only certain types of messages can be sent to a given channel. For this purpose, Spring Integration provides a `MessageSelectingInterceptor` that uses a `MessageSelector` to decide whether a message is allowed to be sent on a channel.

The `MessageSelector` is used in several other places in the framework, and its role is to encapsulate the decision whether a message is acceptable, taking into consideration a given set of criteria, as shown here:

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

One of the implementations of a `MessageSelector` provided by the framework is the `PayloadTypeSelector`, which accepts only the payload types it's configured with, so combining it with a `MessageSelectingInterceptor` allows you to implement another enterprise integration pattern, the *Datatype Channel*, which allows only certain types of messages to be sent on a given channel:[3]

[3] The Datatype Channel pattern is also supported by simply adding a `datatype` attribute to any channel element.

```
<beans:bean id="typeSelector"
   class="org.springframework.integration.selector.PayloadTypeSelector">
  <beans:constructor-arg value="siia.channels.ChargedBooking" />
</beans:bean>

<beans:bean" id="typeSelectingInterceptor"
   class="org.springframework.integration.channel
     ➥.interceptor.MessageSelectingInterceptor">
  <beans:constructor-arg ref="typeSelector"/>
</beans:bean>

<channel id="chargedBookings">
   ...
   <interceptors>
      <ref bean="typeSelectingInterceptor"/>
   </interceptors>
</channel>
```

With this setup, only messages with a `ChargedBooking` payload are allowed to be sent on the `chargedBookings` channel.

> **MESSAGESELECTOR IS MULTIVALENT** As you progress further, you'll encounter the `MessageSelector` being used by another component, the `MessageFilter`, which has similar functionality but is a message handler, not a channel interceptor. The difference is subtle, as the role of filters is mostly to prevent messages from reaching other endpoints (especially in a chain setup), while selectors prevent messages from being sent on channels; they do pretty much the same thing but in different scenarios.

3.4 Summary

The concepts of messages and channels are essential to the flexibility inherent in applications built on Spring Integration. The ease of swapping channel implementations provides a high degree of flexibility in controlling threading models, thread utilization, and latency. In choosing the correct channel, it's vital to understand the behavior of the provided implementations because choosing incorrectly can have serious performance implications or can invalidate the correctness of the application by altering the transactional boundaries.

In this chapter, you learned exactly what a message is to Spring Integration and how you can create messages of your own. You also learned what channels are, and we gave you examples to help you choose the right channel for the job.

In the next chapter, we dive into the components that are connected by channels. The endpoints in Spring Integration contain your business logic but can also be infrastructural components like routers or splitters. Read on for details.

JAVA/SPRING

Spring Integration IN ACTION

Fisher • Partner • Bogoevici • Fuld

Free eBook
SEE INSERT

Spring Integration extends the Spring Framework to support the patterns described in Gregor Hohpe and Bobby Woolf's *Enterprise Integration Patterns*. Like the Spring Framework itself, it focuses on developer productivity, making it easier to build, test, and maintain enterprise integration solutions.

Spring Integration in Action is an introduction and guide to enterprise integration and messaging using the Spring Integration framework. The book starts off by reviewing core messaging patterns, such as those used in transformation and routing. It then drills down into real-world enterprise integration scenarios using JMS, web services, filesystems, email, and more. You'll find an emphasis on testing, along with practical coverage of topics like concurrency, scheduling, system management, and monitoring.

What's Inside

- Realistic examples
- Expert advice from Spring Integration creators
- Detailed coverage of Spring Integration 2 features

This book is accessible to developers who know Java. Experience with Spring and EIP is helpful but not assumed.

Mark Fisher is the Spring Integration founder and project lead. **Jonas Partner**, **Marius Bogoevici**, and **Iwein Fuld** have all been project committers and are recognized experts on Spring and Spring Integration.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/SpringIntegrationinAction

"A wealth of good advice based on experience."
—From the Foreword by Rod Johnson
Founder of the Spring Framework

"Informative and well-written ... makes Spring Integration fun!"
—John Guthrie, SAP

"Bridges the gap between Spring and Enterprise Integration workspaces."
—Rick Wagner, Red Hat

"Comprehensive coverage of features and capabilities."
—Doug Warren, Java Web Services

"Spring Integration from its creators."
—Arnaud Cogoluègnes, coauthor of *Spring Batch in Action* and *Spring Dynamic Modules in Action*

ISBN 13: 978-1-935182-43-6
ISBN 10: 1-935182-43-9
54999
9 781935 182436

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]