

Spring Integration IN ACTION

Mark Fisher
Jonas Partner
Marius Bogoevici
Iwein Fuld





**MEAP Edition
Manning Early Access Program
Spring Integration in Action Production Version**

Copyright 2012 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part 1: Background

- Chapter 1) Introduction to Spring integration
- Chapter 2) Enterprise integration fundamentals

Part 2: Massaging

- Chapter 3) Messages and channels
- Chapter 4) Message endpoints
- Chapter 5) Getting down to business
- Chapter 6) Go beyond sequential processing: routing and filtering

Part 3: Integrating Systems

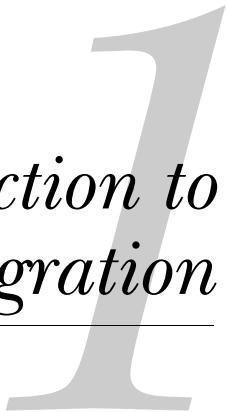
- Chapter 7) Splitting and aggregating messages
- Chapter 8) Handling messages with XML payloads
- Chapter 9) Spring integration and the Java Message Service
- Chapter 10) Going e-postal
- Chapter 11) Filesystem integration
- Chapter 12) Spring Integration and web services

Part 4: Advanced Topics

- Chapter 13) Chatting and tweeting
- Chapter 14) Monitoring and management
- Chapter 15) Managing scheduling and concurrency
- Chapter 16) Batch applications and enterprise integration
- Chapter 17) Scaling messaging applications with OSGi™ and AMQP
- Chapter 18) Testing

Part 1

Introduction to Spring Integration



This chapter covers

- Spring Integration architecture
- Support for enterprise integration patterns
- Inversion of Control

We live in an event-driven world. Throughout each day, we're continuously bombarded by phone calls, emails, and instant messages. As if we're not distracted enough by all of this, we also subscribe to RSS feeds and sign up for Twitter accounts and other social media sites that add to the overall event noise. In fact, technological progress seems to drive a steady increase in the number and types of events we're expected to handle. In today's world of hyperconnectivity, it's a wonder we can ever focus and get any real work done. What saves us from such event-driven paralysis is that we can respond to most events and messages at our convenience. Like a return address on an envelope, events usually carry enough information for us to know how and where to respond.

Now, let's turn our attention to software. When we design and build a software application, we strive to provide a foundation that accurately models the

application’s domain. The domain is like a slice of reality that has particular relevance from the perspective of a given business. Therefore, successful software projects are accurate reflections of the real world, and as such, the event-driven nature plays an increasingly important role. Whereas many software applications are based on a conversational model between a client and a server, that paradigm doesn’t always provide an adequate reflection. Sometimes the act of making a request and then waiting for a reply is not only inefficient but artificial when compared with the actual business actions being represented.

For example, consider an online travel booking application. When you plan a trip by booking a flight, hotel, and rental car, you don’t typically sit and wait at the computer for all of the trip details. You may receive a confirmation number for the trip itself and a high-level summary of the various reservations, but the full details will arrive later. For example, you may receive an email telling you to log in and review the details. In other words, even though the application may be described as a service, it’s likely implemented as an event-driven system. Too many different services are involved to wrap the whole thing in a single synchronous action as the traditional client/server conversational model may suggest. Instead, your request is likely processed by a series of events, or *messages*, being passed across a range of participating systems. When designing software, it’s often useful to consider the events and messages that occur within the domain at hand. Focusing on events and messages rather than being excessively service-oriented may lead to a more natural way to think about the problem.

As a result of all this, it’s increasingly common that enterprise developers must build solutions that respond to a wide variety of events. In many cases, these solutions are replacements for outdated client/server versions, and in other cases, they’re replacements for scheduled back-office processes. Sure, those nightly batch-processing systems that grab a file and process it shortly after midnight still exist, but it’s increasingly common to encounter requirements to refactor those systems to be more timely. Perhaps a new service-level agreement (SLA) establishes that files must be processed within an hour of their arrival, or maybe the nightly batch option is now insufficient due to 24/7 availability and globalized clientèle. These are, after all, the motivating factors behind such hyped, if oxymoronic, phrases as “near real time.” That phrase usually suggests that legacy file-drop systems need to be replaced or augmented with message-driven solutions. Waiting for the result until the next day is no longer a valid option. Perhaps the entry point is now a web service invocation, or an email, or even a Twitter message. And by the way, those legacy systems won’t be completely phased out for approximately another 10 years, so you need to support all of the above. That means you also need to be sure that the refactoring process can be done in an incremental fashion.

Spring Integration addresses these challenges. It aims to increase productivity, simplify development, and provide a solid platform from which you can tackle the complexities. It offers a lightweight, noninvasive, and declarative model for constructing

message-driven applications. On top of this, it includes a toolbox of commonly required integration components and adapters. With these tools in hand, developers can build the types of applications that literally change the way their companies do business.

Spring Integration stands on the shoulders of two giants. First is the Spring Framework, a nearly ubiquitous and highly influential foundation for enterprise Java applications that has popularized a programming model which is powerful because of its simplicity. Second is the book *Enterprise Integration Patterns* (Hohpe and Woolf, Addison-Wesley, 2003), which has standardized the vocabulary and catalogued the patterns of common integration challenges. The original prototype that eventually gave birth to the Spring Integration project began with the recognition that these two giants could produce ground-breaking offspring.

By the end of this chapter, you should have a good understanding of how the Spring Integration framework extends the Spring programming model into the realm of enterprise integration patterns. You'll see that a natural synergy exists between that model and the patterns. If the patterns are *what* Spring Integration supports, the Spring programming model is *how* it supports them. Ultimately, software patterns exist to describe solutions to common problems, and frameworks are designed to support those solutions. Let's begin by zooming out to see what solutions Spring Integration supports at a very high level.

1.1 Spring Integration's architecture

From the 10,000-foot view, Spring Integration consists of two parts. At its core, it's a messaging framework that supports lightweight, event-driven interactions within an application. On top of that core, it provides an adapter-based platform that supports flexible integration of applications across the enterprise. These two roles are depicted in figure 1.1.

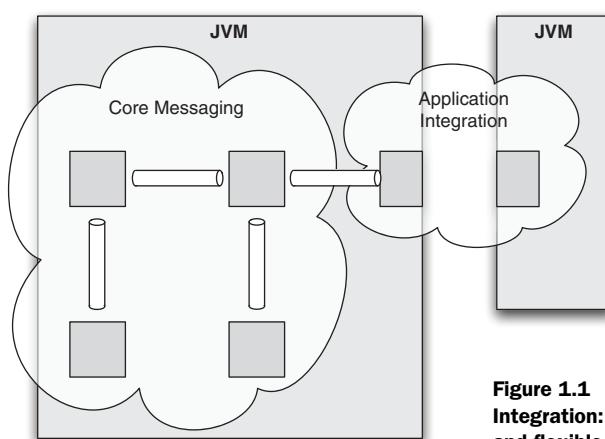


Figure 1.1 Two areas of focus for Spring Integration: Lightweight intra-application messaging and flexible interapplication integration

Everything depicted in the Core Messaging area of the figure would exist within the scope of a single Spring application context. Those components would exchange messages in a lightweight manner because they’re running in the same instance of a Java Virtual Machine (JVM). There’s no need to worry about serialization, and unless necessary for a particular component, the message content doesn’t need to be represented in XML. Instead, most messages will contain plain old Java object (POJO) instances as their payloads.

The Application Integration area is different. There, adapters are used to map the content from outbound messages into the format that some external system expects to receive and to map inbound content from those external systems into messages. The way mapping is implemented depends on the particular adapter, but Spring Integration provides a consistent model that’s easy to extend. The Spring Integration 2.0 distribution includes support for the following adapters:

- File system, FTP, or Secured File Transfer Protocol (SFTP)
- User Datagram Protocol (UDP)
- TCP
- HTTP (Representational State Transfer [REST])
- Web services (SOAP)
- Mail (POP3 or IMAP for receiving, SMTP for sending)
- Java Message Service (JMS)
- JDBC
- Java Management Extensions (JMX)
- Remote Method Invocation (RMI)
- Really Simple Syndication (RSS) feeds
- Twitter
- Extensible Messaging and Presence Protocol (XMPP)

Most of the protocols and transports listed here can act as either an inbound source or an outbound target for Spring Integration messages. In Spring Integration, the pattern name *Channel Adapter* applies to any unidirectional inbound or outbound adapter. In other words, an inbound Channel Adapter supports an in-only message exchange, and an outbound Channel Adapter supports an out-only exchange. Any bidirectional, or request-reply, adapter is known as a *Gateway* in Spring Integration. In part 2 of this book, you’ll learn about Channel Adapters and Gateways in detail.

Figure 1.1 obviously lacks detail, but it captures the core architecture of Spring Integration surprisingly well. The figure contains several boxes, and those boxes are connected via pipes. Now substitute “filters” for boxes, and you have the classic pipes-and-filters architectural style.¹

¹ In this context, it’s probably better to think of *filter* as meaning *processor*.

Anyone familiar with a UNIX-based operating system can appreciate the pipes-and-filters style: it provides the foundation of such operating systems. Consider a basic example:

```
$> echo foo | sed s/foo/bar/  
bar
```

You can see that it's literally the pipe symbol being used to connect two commands (the filters). It's easy to swap out different processing steps or to extend the chain to accomplish more complex tasks while still using these same building blocks:

```
$> cat /usr/share/dict/words | grep ^foo | head -9 | sed s/foo/bar/  
bar bard barder bardful bardless bardlessness bardstuff bardy barfaraw
```

To avoid digressing into a foofaraw,² we should turn back to the relevance of this architectural style for Spring Integration. Those of us using the UNIX pipes-and-filters model on a day-to-day basis may take it for granted, but it provides a great example of two of the most universally applicable characteristics of good software design: low coupling and high cohesion.

Thanks to the pipe, the processing components aren't connected directly to each other but may be used in various loosely coupled combinations. Likewise, to provide useful functionality in a wide variety of such combinations, each processing component should be focused on one task with clearly defined input and output requirements so that the implementation itself is as cohesive, and hence reusable, as possible.

These same characteristics also describe the foundation of a well-designed messaging architecture. *Enterprise Integration Patterns* introduces Pipes-and-Filters as a general style that promotes modularity and flexibility when designing messaging applications. Many of the other patterns discussed in that book can be viewed as more specialized versions of the pipes-and-filters style.

The same holds true for Spring Integration. At the lowest level, it has simple building blocks based on the pipes-and-filters style. As you move up the stack to more specialized components, they exhibit the characteristics and perform the roles of other patterns described in *Enterprise Integration Patterns*. In other words, if it were representing an actual Spring Integration application, the boxes in figure 1.1 could be labeled with the names of those patterns to depict the actual roles being performed. All of this makes sense when you recall our description of Spring Integration as essentially the Spring programming model applied to those patterns. Let's take a quick tour of the main patterns now. Then we'll see how the Spring programming model enters the picture.

² “A great fuss or disturbance about something very insignificant.” Random House via Dictionary.com

1.2 Spring Integration's support for enterprise integration patterns

Enterprise Integration Patterns describes the patterns used in the exchange of messages, as well as the patterns that provide the glue between applications. Like the diagram in figure 1.1, it's about messaging and integration in the broadest sense, and the patterns apply to both *intra*-application and *inter*application scenarios. Spring Integration supports the patterns described in the book, so we need to establish a broad understanding of the definitions of these patterns and the relations between them.

From the most general perspective, only three base patterns make up enterprise integration patterns: Message, Message Channel, and Message Endpoint. Figure 1.2 introduction.message-channel-endpoint shows how these components interact with each other in a typical integration application.

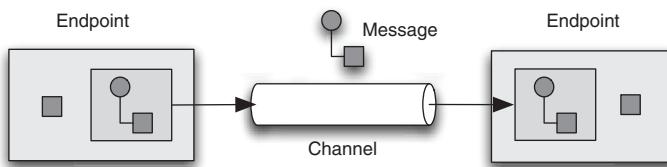


Figure 1.2 A Message is passed through a channel from one endpoint to another endpoint.

There are two main ways to differentiate between these patterns. First, each pattern has more specific subtypes, and second, some patterns are composite patterns. This section focuses on the subtypes so you have a clear understanding of the building blocks. Composite patterns are introduced as needed throughout the book.

1.2.1 Messages

A message is a unit of information that can be passed between different components, called *Message Endpoints*. Messages are typically sent after one endpoint is done with a bit of work, and they trigger another endpoint to do another bit of work. Messages can contain information in any format that's convenient for the sending and receiving endpoints. For example, the message's payload may be XML, a simple string, or a primary key referencing a record in a database. See figure 1.3.

Each message consists of a header and a payload. The header contains data that's relevant to the messaging system, such as the Return Address or Correlation ID. The payload contains the actual data to be accessed or processed by the receiver. Messages can have different functions. For example, a Command Message tells the receiver to do something, an Event Message notifies the receiver that something has happened, and a Document Message transfers some data from the sender to the receiver.

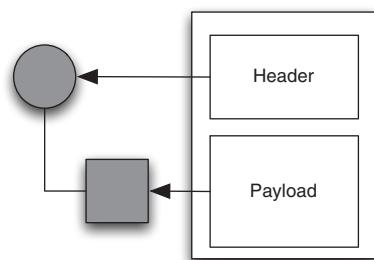


Figure 1.3 A message consists of a single payload and zero or more headers, represented here by the square and circle, respectively.

In all of these cases, the message is a representation of the contract between the sender and receiver. In some applications it might be fine to send a reference to an object over the channel, but in others it might be necessary to use a more interoperable representation like an identifier or a serialized version of the original data.

1.2.2 Message Channels

The *Message Channel* is the connection between multiple endpoints. The channel implementation manages the details of how and where a message is delivered but shouldn't need to interact with the payload of a message. Whereas the most important characteristic of any channel is that it *logically* decouples producers from consumers, there are a number of *practical* implementation options. For example, a particular channel implementation might dispatch messages directly to passive consumers within the same thread of control. On the other hand, a different channel implementation might buffer messages in a queue whose reference is shared by the producer and an active consumer such that the send and receive operations each occur within different threads of control. Additionally, channels may be classified according to whether messages are delivered to a single endpoint (point-to-point) or to any endpoint that is listening to the channel (publish-subscribe). As mentioned earlier, regardless of the implementation details, the main goal of any message channel is to decouple the message endpoints on both sides from each other and from any concerns of the underlying transport.

Two endpoints can exchange messages only if they're connected through a channel. The details of the delivery process depend on the type of channel being used. We review many characteristics of the different types of channels later when we discuss their implementations in Spring Integration. Message channels are the key enabler for loose coupling. Both the sender and receiver can be completely unaware of each other thanks to the channel between them. Additional components may be needed to connect services that are completely unaware of messaging to the channels. We discuss this facet in the next section on message endpoints.

Channels can be categorized based on two dimensions: type of hand-off and type of delivery. The hand-off can be either synchronous or asynchronous, and the delivery can be either point-to-point or publish-subscribe. The former distinction will be discussed in detail in the synchronous vs. asynchronous section of the next chapter. The latter distinction is conceptually simpler, and central to enterprise integration patterns, so we describe it here.

In point-to-point messaging (see figure 1.4), each single message that's sent by a producer is received by exactly one consumer. This is conceptually equivalent to a postcard or phone call. If no consumer receives the message, it should be considered an error. This is especially true for any system that must support guaranteed delivery. Robust

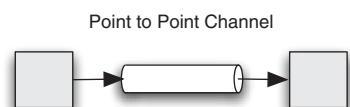


Figure 1.4 A Point-to-Point Channel

point-to-point messaging systems should also include support for load balancing and failover. The former would be like calling each number on a list in turn as new messages are to be delivered, and the latter would be like a home phone that's configured to fall back to a mobile when nobody is home to answer it.

As these cases imply, which consumer receives the message isn't necessarily fixed. For example, in the Competing Consumers (composite) pattern, multiple consumers compete for messages from a single channel. Once one of the consumers wins the race, no other consumer will receive that message from the channel. Different consumers may win each time, though, because the main characteristic of that pattern is that it offers a consumer-driven alternative to load balancing. When a consumer can't handle any more load, it stops competing for another message. Once it's able to handle load again, it will resume.

Unlike point-to-point messaging, a Publish-Subscribe Channel (figure 1.5) delivers the same message to zero or more subscribers. This is conceptually equivalent to a newspaper or the radio. It provides a gain in flexibility because consumers can tune in to the channel at runtime. The drawback of publish-subscribe messaging is that the sender isn't informed about message delivery or failure to the same extent as in point-to-point configurations. Publish-subscribe scenarios often require failure-handling patterns such as Idempotent Receiver or Compensating Transactions.

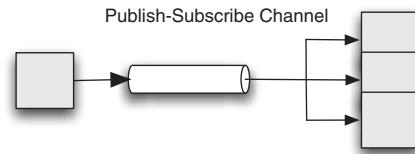


Figure 1.5 A Publish-Subscribe Channel

1.2.3 Message endpoints

Message endpoints are the components that actually do something with the message. This can be as simple as routing to another channel or as complicated as splitting the message into multiple parts or aggregating the parts back together. Connections to the application or the outside world are also endpoints, and these connections take the form of Channel Adapters, Messaging Gateways, or Service Activators. We discuss each of them later in this section.

Message endpoints basically provide the connections between functional services and the messaging framework. From the point of view of the messaging framework, endpoints are at the end of channels. In other words, a message can leave the channel successfully only by being consumed by an endpoint, and a message can enter the channel only by being produced by an endpoint. There are many different types of endpoints. We discuss a few of them here to give you a general idea.

CHANNEL ADAPTER

A Channel Adapter (see figure 1.6) connects an application to the messaging system. In Spring Integration we chose to constrict the definition to include only connections that are unidirectional, so a unidirectional message flow begins and ends in a Channel Adapter. Many different kinds of channel adapters exist, ranging from a method-invoking channel adapter to a web service channel adapter. We go into the

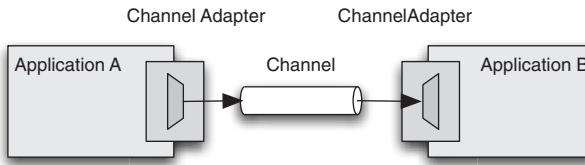


Figure 1.6 Channel Adapter

details of these different types in the appropriate chapters on different transports. For now, it's sufficient to remember that a Channel Adapter is needed at the beginning and the end of a unidirectional message flow.

MESSAGING GATEWAY

In Spring Integration, a Messaging Gateway (see figure 1.7) is a connection that's specific to bidirectional messaging. If an incoming request needs to be serviced by multiple threads but the invoker needs to remain unaware of the messaging system, an inbound gateway provides the solution. On the outbound side, an incoming message is used in a synchronous invocation, and the result is sent on the replychannel. For example, outbound gateways can be used for invoking web services and for synchronous request-reply interactions over JMS.

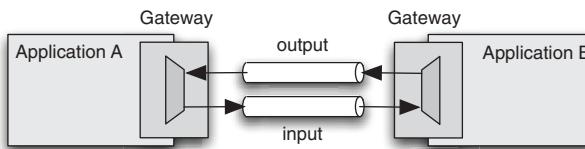


Figure 1.7 Messaging Gateway

A gateway can also be used midstream in a unidirectional message flow. As with the Channel Adapter, we've constrained the definition of Messaging Gateway a bit in comparison to *Enterprise Integration Patterns* (see figure 1.8.).

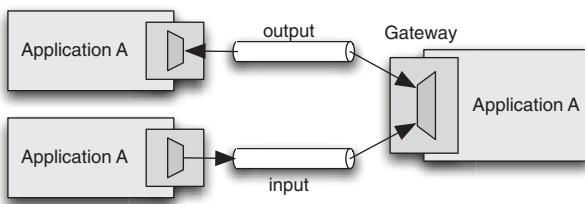


Figure 1.8 Messaging Gateway and Channel Adapters

SERVICE ACTIVATOR

A *Service Activator* (see figure 1.9) is a component that invokes a service based on an incoming message and sends an outbound message based on the return value of this service invocation. In Spring Integration, the definition is constrained to local method calls, so you can think of a Service Activator as a method-invoking outbound gateway. The method that's being invoked is defined on an object that's referenced within the same Spring application context.

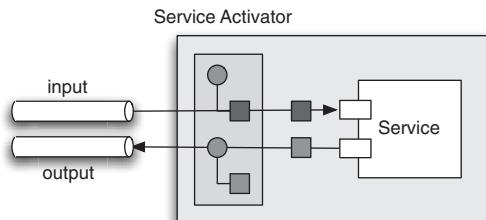


Figure 1.9 Service Activator

ROUTER

A router (see figure 1.10) determines the next channel a message should be sent to based on the incoming message. This can be useful to send messages with different payloads to different, specialized consumers (Content-Based Router). The router doesn't change anything in the message and is aware of channels. Therefore, it's the endpoint that's typically closest to the infrastructure and furthest removed from the business concerns.

SPLITTER

A splitter (see figure 1.11) receives one message and splits it into multiple messages that are sent to its output channel. This is useful whenever the act of processing message content can be split into multiple steps and executed by different consumers at the same time.

AGGREGATOR

An aggregator (figure 1.12) waits for a group of correlated messages and merges them together when the group is complete. The correlation of the messages typically is based on a Correlation ID, and the completion is typically related to the size of the group. Splitter and aggregator are often used in a symmetric setup, where some work is done in parallel after a splitter, and the aggregated result is sent back to the upstream gateway.

You'll see many more patterns throughout the book, but what we covered here should be sufficient for this general introduction. If you paid close attention while reading the first paragraph in this section, you may have noticed that we said Spring Integration *supports* the enterprise integration patterns, not that it *implements* the patterns. That's a subtle but important distinction. In general, software patterns describe proven solutions to common problems. They shouldn't be treated as recipes. In reality,

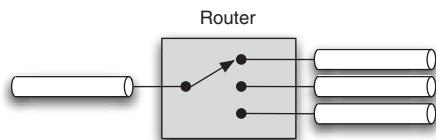


Figure 1.10 Router

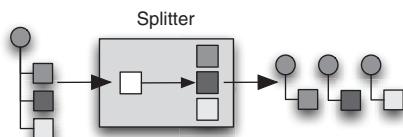


Figure 1.11 Splitter

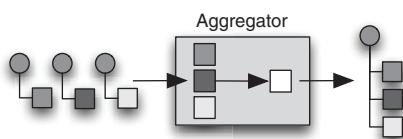


Figure 1.12 Aggregator

patterns rarely have a one-to-one mapping to a single implementation, and context-dependent factors often lead to particular implementation details.

As far as the enterprise integration patterns are concerned, some, such as the Message and Message Channel patterns, are more or less implemented. Others are only partially implemented because they require the addition of some domain-specific logic; examples are the Content-Based Router in which the content is dependent on the domain model and the Service Activator in which the service to be activated is part of a specific domain. Yet other patterns describe individual parts of a larger process; examples are the Correlation ID we mentioned when describing Splitter and Aggregators and the Return Address that we discuss later. Finally, there are patterns that simply describe a general style, such as the Pipes and Filters pattern. With these various pattern categories in mind, let's now see how the concept of Inversion of Control applies to Spring Integration's support for the patterns.

1.3 **Enterprise integration patterns meet Inversion of Control**

Now that we've seen some of the main enterprise integration patterns, we're ready to investigate the benefits that Spring's programming model provides when applied to these patterns. The theme of Inversion of Control (IoC) is central to this investigation because it's a significant part of the Spring philosophy.³ For the purpose of this discussion, we consider IoC in broad terms.

The main idea behind IoC and the Spring Framework itself is that code should be as simple as possible while still supporting all of the complex requirements of enterprise applications. In other words, those complexities can't be wholly ignored but ideally shouldn't have a negative impact on developer productivity or software quality. To accomplish that goal, the responsibility of controlling those complexities should be inverted from the application code to a framework. The bottom line is that enterprise development isn't easy, but it can be much easier when a framework handles much of the grunt work. With that as our definition, we discuss two key techniques that invert control when using Spring: dependency injection and method invocation. We also briefly explain the role of each technique in the Spring Integration framework.

1.3.1 **Dependency injection**

Dependency injection is the first thing most people think of when they hear Inversion of Control, and that's understandable because it's probably the most common application of the principle and the core functionality of IoC frameworks like Spring. Entire books are written on this subject,⁴ but here we provide a quick overview to highlight the benefits of this technique and to see how it applies to Spring Integration.

³ We assume some level of Spring knowledge primarily because we can't cover everything from the basics of Spring to the full spectrum of the Spring Integration framework. If you're new to Spring, you may want to check out some other books, such as *Spring in Action*, Third Edition, by Craig Walls (Manning, 2011).

⁴ *Dependency Injection* by Dhanji R. Prasanna (Manning, 2009).

Object-oriented software is all about modularity. When you design applications, you carefully consider the units of functionality that should be captured within a single component and the proper boundaries between collaborating components. These decisions lead to contracts in the form of well-defined interfaces that dictate the input and output for a given module. When one component depends on another, it should make assumptions only about such an interface rather than about a particular implementation. This promotes the encapsulation of implementation details so that those details can change within the bounds of the interface definition without impacting other code. *What's the big deal?* you may be asking; this is common sense. But it all breaks down as soon as we do something as seemingly harmless as the following:

```
Service service = new MySpecificServiceImplementation();
```

Now the code is tightly coupled directly to an implementation type. That implementation is being assigned to an interface, and hopefully the caller is never required to downcast. But no matter how you slice it, the code is tied to an implementation because interfaces are only contracts and are separate from the implementations. An implementation type must be chosen for instantiation, and the simplest means of instantiating objects in Java is by calling a constructor.

There is another option, and it often follows as a seemingly logical conclusion to this problem. After recognizing that the implementation type leaked into the caller's code, even though that code really only needs the interface, a Factory can be added to provide a level of indirection. Rather than constructing the object, the caller can ask for it:

```
Service service = factory.createTheServiceForMe();
```

This is the first step down the road of IoC. The Factory handles the responsibility that was previously handled directly in the caller's code. The control is inverted in favor of a Factory. The caller gladly relinquishes that control in return for having to make fewer assumptions about the implementation it's using. This seems to solve the problem at first, but to some degree it's just pushing the problem one step further away. It's the programmatic equivalent of sweeping dirt under the rug.

A better solution would remove all custom infrastructure code, including the Factory itself. That final step to full IoC is surprisingly simple: define a constructor or setter method. In effect, that declares the dependency without any assumptions about who's responsible for instantiating or locating it.

```
public class Caller {
    private Service service; // an interface

    public void setService(Service service) {
        this.service = service;
    }
    ...
}
```

In a unit test that focuses on this single component, it's trivial to provide a stub or mock implementation of the dependency directly. Then, in a fully integrated system that may have many components sharing dependencies as well as complex transitive dependency chains, a framework such as Spring can handle the responsibility. All you need to do is provide the implementation type as metadata. Rather than being hard-coded, as in our first example, there is now a clear separation of code and configuration.⁵

```
<bean id="caller" class="example.Caller">
    <property name="service" ref="myService" />
</bean>

<bean id="myService" class="example.MySpecificServiceImplementation" />
```

The Spring Integration framework takes advantage of this same technique to manage dependencies for its components. In fact, you can use the same syntax to define the individual bean definitions, but for convenience, custom XML schemas are defined so that you can declare a namespace⁶ and then use elements and attributes whose names match the domain. The domain in this case is that of the enterprise integration patterns, so the element and attribute names will match those components described in the previous section. For example, any Spring Integration message endpoint requires a reference to at least one message channel (determined by its role as producer, consumer, or both). Here's an example of a simple message splitter⁷:

```
<splitter input-channel="orders" output-channel="items" />
```

Another common case for dependency injection is when a particular implementation of a strategy interface⁷ needs to be wired into the component that delegates to that strategy. For example, here's a message aggregator with a custom strategy for determining when the processed items received qualify as a complete order that can be released:

```
<aggregator input-channel="processedItems"
            release-strategy="orderCompletionChecker"
            output-channel="processedOrders" />
```

Don't worry about understanding the details of the examples yet. These components are covered in more detail throughout the book. The only point we're trying to make so far is that dependency injection plays a role in connecting the collaborating components while avoiding hardcoded references.

⁵ The metadata may alternatively be provided via annotations. For example, the `@Autowired` annotation can be placed on the `setService(...)` method, and the `@Service` annotation could be applied on the implementation class so that XML isn't required. You'll see examples of the annotation-based style throughout the book, but XML was chosen here because it may be easier to understand initially.

⁶ Visit www.springframework.org/schema/integration to explore Spring Integration's various XML schema namespace configurations.

⁷ See the Strategy Method pattern in *Design Patterns: Elements of Reusable Object-Oriented Software* by the Gang of Four: Enrich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).

1.3.2 Method invocation

IoC is often described as following the Hollywood principle: Don't call us, we'll call you.⁸ From the preceding description of dependency injection, you can see how well this applies. Rather than writing code that calls a constructor or even a Factory method, you can rely on the framework to provide that dependency by calling a constructor or setter method. This same principle can also apply to method invocation at runtime.

Let's first consider the Spring Framework's support for asynchronous reception of JMS messages. Prior to Spring 2.0, the only support for receiving JMS messages within Spring was the synchronous (blocking) `receive()` method on its `JmsTemplate`. That works fine when you want to control polling of a JMS destination, but when working with JMS, the code that handles incoming messages can usually be reactive rather than proactive. In fact, the JMS API defines a simple `MessageListener` interface, and the Enterprise JavaBeans (EJB) 2.1 specification introduced message-driven beans as a component model for hosting such listeners within an application server. With version 2.0, Spring introduced its own `MessageListener` containers as a lightweight alternative.

`MessageListeners` can be configured and deployed in a Spring application running in any environment instead of requiring an EJB container. As with message-driven beans, a listener is registered with a certain JMS destination, but with Spring, the listener's container is a simple object that is itself managed by Spring. There's even a dedicated XML namespace:

```
<jms:message-listener-container>
    <jms:listener destination="someDestination" ref="someListener" />
</jms:message-listener-container>
```

The container manages the subscription and the background processes that are ultimately responsible for receiving the Messages. There are configuration options for controlling the number of concurrent consumers, managing transactions, and more.

```
public interface MessageListener {
    void onMessage(Message message);
}
```

It gets even more interesting and more relevant for our lead-up to Spring Integration when we look at Spring's support for invoking methods on any Spring-managed object. Sure, the `MessageListener` interface seems simple enough, but it has a few limitations. First, it requires a dependency on the JMS API. This inhibits testing and also pollutes otherwise pure business logic achieved by relying on the IoC principle. Second, and more severe, it has a void return type. That means you can't send a reply message from the listener method's implementation. Both of these limitations are

⁸ In Hollywood, that probably means "Don't bother us with your calls. On the slim chance that you get the part, we'll call you. But we probably won't, so get over it." In software, we rely on things being more definite.

eliminated if you instead reference a POJO instance that doesn't implement MessageListener and add the method attribute to the configuration. For example, let's assume you want to invoke the following service method:

```
public class QuoteService {
    BigDecimal currentQuote(String tickerSymbol) {...}
}
```

The configuration would look like this:

```
<jms:message-listener-container>
    <jms:listener destination="quoteRequests" ref="quoteService"
                  method="currentQuote"/>
</jms:message-listener-container>

<bean id="quoteService" class="example.QuoteService"/>
```

Whatever client is passing request messages to the quoteRequest destination could also provide a JMSReplyTo property on that request message. Spring's listener container uses that property to send the reply message to the destination where the caller is waiting for the response to arrive. Alternatively, a default reply destination can be provided with another attribute in the XML.

This message-driven support is a good example of IoC because the listener container is handling the background processes. It's also a good example of the Hollywood principle because the framework calls the referenced object whenever a message arrives.

Another common requirement in enterprise applications is to perform some task at a certain time or repeatedly on the basis of a configured interval. Java provides some basic support for this with `java.util.Timer`, and beginning with version 5, a more powerful scheduling abstraction was added: `java.util.concurrent.ScheduledExecutorService`. For functionality beyond what the core language provides, there are projects such as Quartz⁹ to support scheduling based on cron expressions, persistence of job data, and more.

Interacting with any of these schedulers normally requires code that's responsible for defining and registering a task. For example, imagine you have a method called `poll` in a custom `FilePoller` class. You might wrap that call in a `Runnable` and schedule it in Java as in the following listing.

Listing 1.1 Scheduling a task programmatically

```
Runnable task = new Runnable() {
    public void run() {
        File file = filePoller.poll();
        if (file != null) {
            fileProcessor.process(file);
        }
    }
}
```

⁹ www.opensymphony.com/quartz.

```

        }
};

long initialDelay = 0;
long rate = 60;
ScheduledExecutorService scheduler =
    Executors.newScheduledThreadPool(5);
scheduler.scheduleAtFixedRate(task, initialDelay,
    rate, TimeUnit.SECONDS);

```

The Spring Framework can handle much of that for you. It provides method-invoking task adapters and support for automatic registration of the tasks. That means you don't need to add any extra code. Instead, you can declaratively register your task. For example, in Spring 3.0, the configuration might look like this:

```

<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="filePoller" method="poll" fixed-rate="60000"/>
</task:scheduled-tasks>

<task:scheduler id="myScheduler" pool-size="5"/>

```

As you can see, this provides a literal example of the Hollywood principle. The framework is now calling the code. This provides a few benefits beyond the obvious simplification. First, even though the code being invoked should be thoroughly unit tested, you can rest assured that the Spring scheduling mechanism is tested already. Second, the configuration of the initial delay and fixed-rate period for the task and the thread pool size for the scheduler are all externalized. By enforcing this separation of configuration from code, you're much less likely to end up with hardcoded values in the application. The code is not only easier to test but also more flexible for deploying into different environments.

Now let's see how this same principle applies to Spring Integration. The configuration of scheduled tasks follows the same technique as shown previously. The configuration of a Polling Consumer's trigger can be provided through declarative configuration. Spring Integration takes the previous example a step further by actually providing a file-polling Channel Adapter:

```

<file:inbound-channel-adapter directory="/tmp/example" channel="files">
    <poller max-messages-per-poll="10" fixed-rate="60000"/>
</file:inbound-channel-adapter>

```

We should also mention that both the core Spring Framework scheduling support and the Spring Integration polling triggers accept cron expressions in place of the interval-based values. If you only want to poll during regular business hours, something like the following would do the trick:

```

<file:inbound-channel-adapter directory="/tmp/example" channel="files">
    <poller max-messages-per-poll="10" cron="0 * 9-17 * MON-FRI "/>
</file:inbound-channel-adapter>

```

For now, let's move beyond these isolated examples. Thus far, you've seen just a few glimpses of how the IoC principle and the Spring programming model can be applied to enterprise integration patterns. The best way to reinforce that knowledge is by diving into a simple but complete hands-on example.

1.4 Say hello to Spring Integration

Now that you've seen the basic enterprise integration patterns and an overview of how IoC can be applied to those patterns, it's time to jump in and meet the Spring Integration framework face to face. In the time-honored tradition of software tutorials, let's say Hello to the Spring Integration world.

Spring Integration aims to provide a clear line between code and configuration. The components provided by the framework, which often represent the enterprise integration patterns, are typically configured in a declarative way using either XML or Java annotations as metadata. But many of those components act as stereotypes or templates. They play a role that's understood by the framework, but they require a reference to some user-defined, domain-specific behavior in order to fulfill that role.

For our Hello World example, the domain-specific behavior is the following:

```
package siia.helloworld.channel;

public class MyHelloService implements HelloService {

    @Override
    public void sayHello(String name) {
        System.out.println("Hello " + name);
    }
}
```

The interface that `MyHelloService` implements is `HelloService`, defined as follows:

```
package siia.helloworld.channel;

public class MyHelloService implements HelloService {

    @Override
    public void sayHello(String name) {
        System.out.println("Hello " + name);
    }
}
```

There we have it: a classic Hello World example. This one is flexible enough to say Hello to anyone. Because this book is about Spring Integration, and we've already established that it's a framework for building messaging applications based on the fundamental enterprise integration patterns, you may be asking, *Where' are the Message, Channel, and Endpoint?* The answer is that you typically don't have to think about those components when defining the behavior. What we implemented here is a straightforward POJO with no awareness whatsoever of the Spring Integration API. This is consistent with the general Spring emphasis on noninvasiveness and separation of concerns. That said, let's now tackle those other concerns, but separately, in the configuration:

```

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

    <channel id="names" />
    <service-activator input-channel="names" ref="helloService"
        method="sayHello"/>
    <beans:bean id="helloService"
        class="sia.helloworld.channel.MyHelloService"/>
</beans:beans>
```

This code should look familiar. In the previous section, we saw an example of the Spring Framework’s support for message-driven POJOs, and the configuration for that example included an element from the `jms` namespace that similarly included the `ref` and `method` attributes for delegating to a POJO via an internally created `MessageListenerAdapter` instance. Spring Integration’s `ServiceActivator` plays the same role, except that this time it’s more generic. Rather than being tied to the JMS transport, the `ServiceActivator` is connected to a Spring Integration `MessageChannel` within the `ApplicationContext`. Any component could be sending messages to this `ServiceActivator`’s `input-channel`. The key point is that the `ServiceActivator` doesn’t require any awareness or make any assumptions about that sending component.

All of the configured elements contribute components to a Spring `ApplicationContext`. In this simple case, you can bootstrap that context programmatically by instantiating the Spring context directly. Then, you can retrieve the `MessageChannel` from that context and send it a message. We use Spring Integration’s `MessageBuilder` to construct the actual message, shown in the following listing. Don’t worry about the details; you’ll learn much more about message construction in chapter 3.

Listing 1.2 Hello World with Spring Integration

```

package sia.helloworld.channel;

import org.springframework.context.ApplicationContext;
import
    org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.integration.Message;
import org.springframework.integration.MessageChannel;
import org.springframework.integration.support.MessageBuilder;

public class HelloWorldExample {

    public static void main(String args[]) {
        String cfg = "sia/helloworld/channel/context.xml";
```

```

ApplicationContext context = new ClassPathXmlApplicationContext(cfg);
MessageChannel channel =
    context.getBean("names", MessageChannel.class);
Message<String> message =
    MessageBuilder.withPayload("World").build();
channel.send(message);
}
}

```

Running that code produce “Hello World” in the standard output console. That’s pretty simple, but it would be even nicer if there were no direct dependencies on Spring Integration components even on the caller’s side. Let’s make a few minor changes to eradicate those dependencies.

First, to provide a more realistic example, let’s modify the `HelloService` interface so that it returns a value rather than simply printing out the result itself.

```

package siia.helloworld.gateway;

public class MyHelloService implements HelloService {

    @Override
    public String sayHello(String name) {
        return ("Hello " + name);
    }
}

```

Spring Integration handles the return value in a way that’s similar to the Spring JMS support described earlier. You add one other component to the configuration, a gateway proxy, to simplify the caller’s interaction. Here’s the revised configuration:

```

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

    <gateway id="helloGateway"
        service-interface="siia.helloworld.gateway.HelloService"
        default-request-channel="names"/>

    <channel id="names" />

    <service-activator input-channel="names" ref="helloService"
        method="sayHello"/>

    <beans:bean id="helloService"
        class="siia.helloworld.channel.MyHelloService"/>

</beans:beans>

```

Note that the `gateway` element refers to a service interface. This is similar to the way the Spring Framework handles remoting. The caller should only need to be aware of

an interface, while the framework creates a proxy that implements that interface. The proxy is responsible for handling the underlying concerns such as serialization and remote invocation, or in this case, message construction and delivery. You may have noticed that the `MyHelloService` class does implement an interface. Here's what the `HelloService` interface looks like:

```
package sii.a.helloworld.gateway;
public interface HelloService {
    String sayHello(String name);
}
```

Now the caller only needs to know about the interface. It can do whatever it wants with the return value. In the following listing, you just move the console printing to the caller's side. The service instance would be reusable in a number of situations. The key point is that this revised main method now has no direct dependencies on the Spring Integration API.

Listing 1.3 Hello World revised to use a Gateway proxy

```
package sii.a.helloworld.gateway;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

public class HelloWorldExample {
    public static void main(String args[]) {
        String cfg = "sii/a/helloworld/gateway/context.xml";
        ApplicationContext context = new ClassPathXmlApplicationContext(cfg);
        HelloService helloService =
            context.getBean("helloGateway", HelloService.class);
        System.out.println(helloService.sayHello("World"));
    }
}
```

As with any Hello World example, this one only scratches the surface. Later you'll learn how the result value can be sent to another downstream consumer, and you'll learn about more sophisticated request-reply interactions. The main goal for now is to provide a basic foundation for applying what you've learned in this chapter. Spring Integration brings the enterprise integration patterns and the Spring programming model together. Even in this simple example, you can see some of the characteristics of that programming model, such as IoC, separation of concerns, and an emphasis on noninvasiveness of the API.

1.5 Summary

We covered a lot of ground in this chapter. You learned that Spring Integration addresses both messaging within a single application and integrating across multiple

applications. You learned the basic patterns that also describe those two types of interactions.

As you progress through this book, you'll learn in much greater detail how Spring Integration supports the various enterprise integration patterns. You'll also see the many ways in which the framework builds on the declarative Spring programming model. So far, you've seen only a glimpse of these features, but some of the main themes of the book should already be clearly established.

First, with Spring's support for dependency injection, simple objects can be wired into these patterns. Second, the framework handles the responsibility of invoking those objects so that the interactions all appear to be event-driven even though some require polling (control is inverted so that the framework handles the polling for you). Third, when you need to send messages, you can rely on templates or proxies to minimize or eliminate your code's dependency on the framework's API.

The bottom line is that you focus on the domain of your particular application while Spring Integration handles the domain of enterprise integration patterns. From a high-level perspective, Spring Integration provides the foundation that allows your services and domain objects to participate in messaging scenarios that take advantage of all of these patterns.

In chapter 2, we dive a bit deeper into the realm of enterprise integration. We cover some of the fundamental issues such as loose coupling and asynchronous messaging. This knowledge will help establish the background necessary to take full advantage of the Spring Integration framework.



Enterprise integration fundamentals

This chapter covers

- Loose coupling and event-driven architectures
- Synchronous and asynchronous interaction models
- The most important enterprise integration styles

Commercial applications are, most of the time, solutions to problems posed by the business units for which they're developed. It makes little difference whether the problem under discussion is older, and the solution is automating an existing process, or the problem is new, and the solution is an innovation that allows the organization to do business in a way that wasn't possible before.

In some cases, the solutions consist of newly developed components that reuse already-existing applications by delegating functionality to them. This is often the case with legacy applications that implement complex business logic and for which a complete rewrite would be an unjustifiable cost. Other applications are divided from the beginning into multiple components that run independently to get the most of the modern hardware and its high concurrency capabilities. What both these approaches have in common is that they tie together separate components

and applications, sometimes even located on different machines. Such applications are the focus of enterprise application integration.

Each of these concepts discussed in this chapter plays an important role in designing integrated applications. They make sure that the integrated components don't impose needless restrictions on each other and that the system is responsive and can process concurrent requests efficiently. Applying these principles in practice results in several integration styles, each with its own advantages and disadvantages. This chapter shows you how to take them into account when implementing a solution. As a Spring Integration user, you'll benefit from understanding the foundational principles of the framework.

2.1 **Loose coupling and event-driven architecture**

We already discussed building applications that consist of multiple parts (or that orchestrate collaboration between standalone applications). It's important to understand the implications of integration on your design. One of the most important consequences of decomposing applications into multiple components is that these components can be expected to evolve independently. The proper design and implementation of these parts (or independent applications) can make this process easy. Failing to properly *separate concerns* can make evolution prohibitively hard. Our criterion to decide whether a particular design strategy fosters independent evolution or stands in the way is coupling.

In this section we argue why loose coupling is preferable over tight coupling in almost every situation. Because coupling can come in different forms, such as type level or system level, we explore these variants of coupling in more detail. The last part of the section discusses how to reduce coupling in your application by using dependency injection or adopting an event-driven architecture.

2.1.1 **Why should you care about loose coupling?**

Loose coupling within systems and between systems deserves serious consideration, for it has serious implications for design and maintenance. Achieving an appropriate degree of loose coupling allows you to spend more time adding new features and delivering business value. By contrast, tightly coupled systems are expensive to maintain and expand because small changes to the code tend to produce ripple effects, requiring modifications across a large number of interacting systems. It's important to note that this increase in cost isn't the product of a change in conditions but something that could have been avoided at design time.

It's not so much that loosely coupled systems guarantee quality but that highly coupled systems nearly always guarantee complexity in the code and the paths through the code, making systems hard to maintain and hard to understand. Highly coupled systems are also generally harder to test, and often it's nearly impossible to unit test their constituent parts, for example, because their units can't be constructed without constructing the entire system.

IDENTIFYING HIGHLY COUPLED SYSTEMS

How can you identify a system that's highly coupled? Measuring coupling has been the focus of various academic attempts based on various forms of code-level connections. For example, one measure could be how many user types a class references. Where a class is referencing many user types, as shown in figure 2.1, it's considered highly coupled, which is usually a bad sign because changes in those referenced types may lead to a requirement to change the class referencing them. Take, for example, adding a parameter to a method signature: all invokers must provide a value for it.

Using referenced types as a measure of how interconnected classes within a system are will give you an idea whether the system's coupling is high or low. This method works well for individual applications, but as new application design strategies are employed, other forms of coupling have come to prominence. For example, in service-oriented architectures (SOAs), the coupling between the service contract and the service implementation is also taken into account, so that a change in the implementation of the service doesn't create a need to update all service clients. In many SOA scenarios, service consumers may not be under the control of the service implementation. In this case, loose coupling can insulate you from causing regressions that in turn cause unhappy service consumers with broken systems.

A general way of defining coupling is as a measure of how connected the parts of a system are or how many assumptions the components of the system make about each other. For example, systems that directly reference Remote Method Invocation (RMI) for communication in many places are highly coupled to RMI. Or a system that directly references a third-party system in a large number of places can be said to be highly coupled to the third-party system in cause. The connections you create in and between your systems introduce complexity and act as conduits for change, and change generally entails cost and effort.

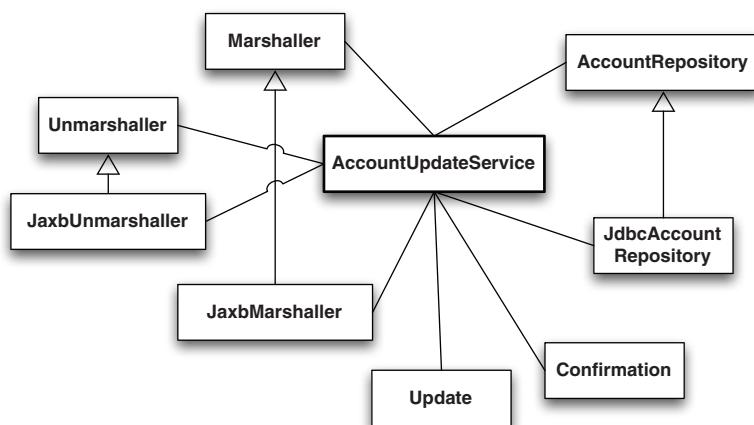


Figure 2.1 A highly coupled system: Components become entangled because of the complex relationships between them.

Coupling

Coupling is an abstract concept used to measure how tightly connected the parts of a system are and how many assumptions they make about each other.

Loose coupling, in the context of integrating systems, is therefore vital in allowing the enterprise to switch between different variants of a particular component (for example, accounting packages) without incurring prohibitive cost due to required changes to other systems. Coupling is an abstract concept, and we show over the next few sections a few concrete examples of different types of coupling, along with approaches using Spring and Spring Integration to reduce coupling and improve code quality through increased simplicity, increased testability, and reduced fragility.

2.1.2 Type-level coupling

Coupling between types is probably the best understood because it's the form of coupling that's most often discussed. The following listing shows a booking service that allows airline passengers to update their meal preferences. The service first looks up the booking from the database to obtain the internal reference, then invokes a meal-preference web service.

Listing 2.1 Airline booking service

```
package siia.fundamentals;

import siia.fundamentals.Booking;
import siia.fundamentals.BookingDao;
import siia.fundamentals.BookingService;
import siia.fundamentals.MealPreference;
import siia.fundamentals.SimpleBookingDao;

import org.springframework.ws.client.core.WebServiceOperations;
import org.springframework.ws.client.core.WebServiceTemplate;
import org.springframework.xml.transform.StringResult;
import org.springframework.xml.transform.StringSource;

import javax.xml.transform.Source;

public class BookingServiceWithStrongCoupling {
    private final BookingDao bookingDao;

    private final WebServiceOperations mealPreferenceWebServiceInvoker;

    public BookingServiceWithStrongCoupling() {
        this.bookingDao = new SimpleBookingDao();
        WebServiceTemplate template = new WebServiceTemplate();
        template.setDefaultUri(System.getProperty(
            "meal.preference.service.uri"));
        this.mealPreferenceWebServiceInvoker = template;
    }

    public void getBooking(MealPreference mealPreference) {
```

```

Booking booking = bookingDao.getBookingById(
    mealPreference.getBookingReference());
Source mealUpdateSource = buildMealPreferenceUpdateRequest(
    booking, mealPreference);

StringResult result = new StringResult();
mealPreferenceWebServiceInvoker.sendSourceAndReceiveToResult(
    mealUpdateSource, result);
}

public Source buildMealPreferenceUpdateRequest(
    Booking booking, MealPreference mealPreference) {
    return new StringSource(
        "<updateMealPreference> " +
        "<flightRef> " +
        booking.getFlightRef() +
        "</flightRef> " +
        "<mealPreference> " +
        mealPreference +
        "</mealPreference> " +
        "</updateMealPreference> ");
}
}

```

The following example shows the `BookingReportingService` used to generate management reports with the business methods omitted.

```

package siiia.fundamentals;

public class BookingReportingService {

    private final BookingDao bookingDao;

    public BookingReportingService(){
        this.bookingDao = new SimpleBookingDao();
    }
    /** Actual methods omitted */
}

```

Because both classes directly reference the `SimpleBookingDao` and the constructor declared by that class, changes to that type may affect both the `BookingService` and the `BookingReportingService`. This type of coupling is known as *unambiguous type coupling* because it's coupled to the a concrete implementation even though it then assigns the instance to members' fields typed as the interface. The `BookingService` also exhibits unambiguous type coupling to the Spring `WebServiceTemplate`. Generally, a system repeating this pattern of coupling to the concrete type can be considered highly coupled because changes in those concrete types will have widespread impact.

The solution to this problem is deferring the creation of concrete instances to the framework, using what we call *dependency injection*, which is our next topic.

2.1.3 Loosening type-level coupling with dependency injection

Standard dependency injection allows you to reduce coupling by addressing unambiguous type coupling. One way to do this is to move the instantiation concern into a single configuration file rather than code. Removing the code that instantiates the collaborators for your services both reduces coupling and simplifies the code. This makes maintenance easier, simplifies testing by allowing tests to inject test-only implementations, and increases the chances of reuse. The following example shows the new version of the BookingService, which now exposes a constructor that accepts instances of its collaborators.

```
package siiia.fundamentals;

import org.springframework.ws.client.core.WebServiceOperations;
import org.springframework.xml.transform.StringResult;
import org.springframework.xml.transform.StringSource;

import javax.xml.transform.Source;

public class BookingServiceWithInjection {

    private final BookingDao bookingDao;

    private final WebServiceOperations mealPreferenceWebServiceInvoker;

    public BookingServiceWithInjection(BookingDao bookingDao,
                                       WebServiceOperations mealPreferenceWebServiceInvoker) {
        this.bookingDao = bookingDao;
        this.mealPreferenceWebServiceInvoker =
            mealPreferenceWebServiceInvoker;
    }

    /*
     * getBooking() and buildMealPreferenceUpdateRequest() remain
     * unchanged
     */
}
```

The Spring configuration to instantiate the BookingService in production looks as follows.

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

    <beans:bean id="bookingDao"
      class="siiia.fundamentals.SimpleBookingDao" />

    <beans:bean id="mealPreferenceWebServiceTemplate"
      class="org.springframework.ws.client.core.WebServiceTemplate">
      <beans:property name="defaultUri"
        value="${meal.preference.service.uri}" />
    
```

```

</beans:bean>

<beans:bean id="bookingService"
    class="siiia.fundamentals.BookingServiceWithInjection">
    <beans:constructor-arg ref="bookingDao" />
    <beans:constructor-arg
        ref="mealPreferenceWebServiceTemplate" />
</beans:bean>

<beans:bean id="bookingReportingService"
    class="siiia.fundamentals.BookingReportingService" />

</beans:beans>

```

This example reduces coupling in the Java code but introduces a new configuration file that's coupled to the concrete implementations. Taking a broader view of the system, you now have that coupling in one place rather than potentially many calls to the constructors, so the system as a whole is less highly coupled because changes to SimpleBookingDao will impact only one place.

With type-level coupling out of the way, it's still possible for the different collaborators to make excessive assumptions about each other, such as about the data format being exchanged (for example, using a nonportable format such as a serialized Java class) or about whether two collaborating systems are available at the same time. We group such assumptions under the moniker of system-level coupling.

2.1.4 System-level coupling

It's possible that collaborators might need to change to address new requirements, but it's almost inevitable that where a large number of systems talk to each other, those systems will evolve at different rates. Limiting one system's level of coupling to another is key in being able to cope easily with changes such as these.

Currently, the booking service is coupled to the use of a web service to contact the meal-preference service as well as the XML data format expected by the service. Things could've been worse: the data format could've been a serialized Java class instead of XML. Serialization would be the right tool to use whenever the expectation is that data will be consumed by the same application at a later time or when data is exchanged between components that are expected to be highly connected to each other (like a client/server application). Exchanging serialized data between independent applications couples them needlessly because it introduces the assumption that both of them have access to the bytecode of the serialized class. It also requires that they use the same version of the class. This in turn demands that new versions of the serialized classes be incorporated into the clients as soon as they're deployed on the server. It obviously assumes that both applications are using the same platform (Java in our case). By using a portable data format like XML, we've just avoided all of this. Of course, the applications are still coupled by the XML format, but in the end, both applications must have a basic agreement on what is and what isn't a correct message. We believe consumers should be as liberal as possible with that.

We still have another problem to deal with: the `BookingService` is also temporally coupled to the meal-preference web service in that we make a synchronous call and therefore the call to the `BookingService` will fail if the meal-preference service is unavailable at the point. Whether that's the desired behavior will depend on the requirements, but it would be nice to make the temporal coupling optional.

One option would be to introduce a new class that encapsulates the call to the meal-preference service, removing the web service concern and the data format concern from the `BookingService`. This solution eliminates some coupling, but it also introduces coupling on a new meal-preference service type and on the signature of the method declared by that type.

By instead replacing the call to the meal-preference service with a new component and connecting the `BookingService` with the meal-preference service, you can further reduce coupling in the `BookingService` without introducing additional type coupling. This can be achieved by replacing direct method invocations with *message passing* over channels.

The following example shows the simplified booking service, which now simply enriches the meal preference passed in with the flight reference.

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

    <beans:bean id="bookingDao"
        class="siiia.fundamentals.SimpleBookingDao"/>
    <channel id="mealPreferenceUpdatesChannel"/>
    <beans:bean id="bookingService"
        class="siiia.fundamentals
            .BookingServiceWithDependencyInjection">           <beans:construct
        or-arg ref="bookingDao"/>      </beans:bean>
    <service-activator input-channel="mealPreferenceUpdatesChannel"
        output-channel="bookingEnrichedMealUpdates"
        ref="bookingService"
        method="getFlightRefForBooking"/>
    <channel id="bookingEnrichedMealUpdates"/>
    <beans:bean id="updateRequestTransformer"
        class="siiia.fundamentals
```

```

    .MealPreferenceRequestTransformer" />
<service-activator input-channel="bookingEnrichedMealUpdates"
                   output-channel="xmlMealUpdates"
                   ref="updateRequestTransformer"
                   method="buildMealPreferenceUpdateRequest" />
<channel id="xmlMealUpdates"/>
<ws:outbound-gateway uri="http://example.com/mealupdates"
                      request-channel="xmlMealUpdates" />
</beans:beans>

```

Focusing on message passing as the main integration mechanism leads us toward the adoption of an event-driven architecture, since each component in the message flow simply responds to messages being delivered by the framework.

2.1.5 **Event-driven architecture**

Event-driven architecture (EDA) is an architectural pattern in which complex applications are broken down into a set of components or services that interact via events. One of the primary advantages of this approach is that it simplifies the implementation of the component by eliminating the concern of how to communicate with other components. Where events are communicated via channels that can act as buffers in periods of high throughput, such a system can be described as having a staged event-driven architecture (SEDA). SEDA-based systems generally respond better to significant spikes in load than do standard multithreaded applications and are also easier to configure at runtime, for example, by modifying the number of consumers processing a certain event type. This allows for optimizations based on the actual requirements of the application, which in turn provides a better usage experience.

The question of whether an application built around the Spring Integration framework is inherently an EDA or SEDA application is open to debate. Certainly Spring Integration provides the building blocks to create both EDA and SEDA applications. Whether your particular application falls into one category or another in a strict interpretation depends on the messages you pass and your own working definition of what constitutes an event. In most cases, this is probably not a useful debate to enter into, so Spring Integration makes no distinction between agents as producers of events and producers of messages, nor are event sinks and consumers distinguished. The one place in which the term *event* is used in Spring Integration is in the implementation of the Event-Driven Consumer pattern, but here the consumer is consuming messages rather than events—it's just that the consumption is triggered by the event of a message becoming available.

Reducing coupling is one of the main concerns when integrating applications. You've just ended a section that shows how this can be done by eliminating assumptions about the concrete types used in the application as well as replacing method and web service invocations with message passing to reduce coupling. With respect to the latter, we mentioned synchronous communication as a way to increase the coupling of two systems, which we now address in more detail.

2.2 Synchronous and asynchronous communication

Another possible assumption made when integrating multiple components is that they're available simultaneously. Depending on whether this assumption is incorporated in the system's design, the components may interact synchronously or asynchronously, and in this section we look at the main differences between the two interaction models as well as their advantages and disadvantages.

From the Spring Integration perspective, you'll see the options that the framework provides in each case and how simple configuration options allow you to switch between synchronous and asynchronous communication without changing the overall logical design of your application.

2.2.1 What's the difference?

In synchronous communication (figure 2.2), one component waits until the other provides an answer to its request and proceeds only after a response is provided. The requests are delivered immediately to the service provider, and the requesting component blocks until it receives a response.

When communicating asynchronously (figure 2.3), the component that issues the request proceeds without waiting for an answer. The requests aren't delivered to the service provider but stored in an intermediate buffer and from there will be delivered to their intended recipient.

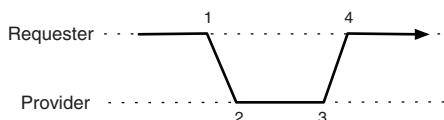


Figure 2.2 Synchronous invocation: The requester suspends execution until it receives an answer.

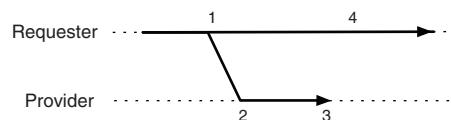


Figure 2.3 Asynchronous invocation: The requester doesn't block and executes in parallel with the provider.

Looking at how different these two alternatives are, the decision to use synchronous or asynchronous communication should be based on their strengths and weaknesses. Let's examine the upsides and downsides of each approach a bit further.

Of the two, synchronous communication is more straightforward: the recipient of the call is known in advance, and the result is received immediately (see figure 2.4). The invocation, processing, and response occur in the same thread of execution (like a Java thread if the call is local or a logical thread if it's remote). This allows you to propagate a wealth of contextual information, the most common being the transactional and security context. Generally, the infrastructure required to set it up is simpler: a method call or a remote procedure call. Its main weaknesses are that it's not scalable and it's less resilient to failure.



Figure 2.4 Synchronous message exchange: The message is received immediately by the provider.

Scaling up is a problem for synchronous communication because if the number of simultaneous requests increases, the target component has few alternatives, such as:

- Trying to accommodate all requests as they arrive, which will crash the system.
- Throttling some of the requests to bring the load to a bearable level.¹

When the load increases, the application will eventually fail, and you can do little about it.

The lack of resilience to failure comes from the fundamental assumption that the service provider is working properly all the time. There's no contingency, so if the service provider is temporarily disabled then the client features that depend on it won't work either. The most obvious situation is a remote call that fails when the service provider is stopped, but this also applies to local calls when an invoked service throws a `RuntimeException`.

DEALING WITH EXCEPTIONS Exception-handling strategies are beyond the scope of this discussion, but it should be noted that a service can attempt to retry a synchronous call if the downtime of the service provider is very short, but it's generally unacceptable to block for a long time while waiting for a service to come up.

Asynchronous communication offers better opportunities to organize the work on the service provider's side. Requests aren't processed immediately but left in an intermediate storage and from there are delivered to the service provider whenever it can handle them (see figure 2.5). This means the requests will never be lost, even if the service provider is temporarily down, and also that a pool of concurrent processes or threads can be used to handle multiple requests in parallel.

Asynchronous message exchange provides control for the behavior of the system under heavy load: a larger number of concurrent consumers means a better throughput. The caveat is that it puts the system under more stress. The point is that, unlike synchronous calls where the load of the system is controlled by the requesting side, in the asynchronous model, the load of the system is controlled by the service provider's side. Furthermore, asynchronous communication has better opportunities for retrying a failed request, which improves the overall resilience to failure of the system.

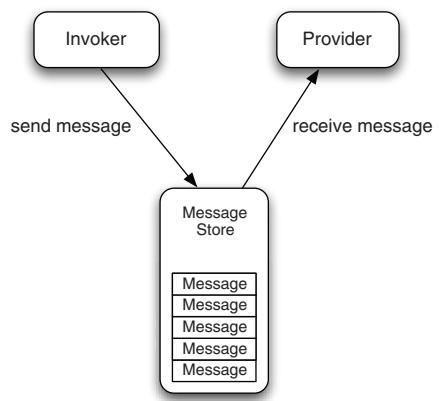


Figure 2.5 Asynchronous message exchange: The message is stored in an intermediate buffer before being received by the provider.

¹ Throttling is the process of limiting the number of requests that a system can accommodate by either postponing some of them or dropping them altogether.

You saw that asynchronous communication has obvious advantages in terms of scalability and robustness, but there's a price to be paid for that: *complexity*. The messaging middleware that performs the message storage and forwarding function is an additional mechanism that must be integrated with the application, which means that instead a simple method call, you must deal with supplemental APIs and protocols. Not only does the code become more complex but also a performance overhead is introduced.

Also, in a request-reply scenario using the synchronous approach, the result of an operation can be easily retrieved: when the execution of the caller resumes, it already has the result. In the case of asynchronous communication, the retrieval of the result is a more complex matter. It can either follow the Future Object pattern (as in `java.util.concurrent.Future`), where after the asynchronous invocation, the invoker is provided with a handle to the asynchronous job and can poll it to find out whether a result is available, or the requester might provide a callback to be executed in case the request has been processed.

After issuing an asynchronous request, you have two components that execute independently: the requester that continues doing its work and the service provider whose work is deferred until later. Concurrency is a powerful tool for increasing the application performance, but it adds complexity to your application too. You have to consider concerns like thread safety, proper resource management, deadlock, and starvation avoidance. Also, at runtime, it's harder to trace or debug a concurrent application. Finally, the two concurrent operations will typically end up executing in distinct transactions. Synchronous and asynchronous communication are compared in more detail in table 2.1.

The advantages and disadvantages of the two paradigms should be carefully considered when choosing one over the other. Traditionally, the integration between enterprise applications is based on message-driven, asynchronous mechanisms.

As the capabilities of hardware systems increase, especially when it comes to execution of concurrent processes and as multicore architectures become more pervasive,

Table 2.1 Synchronous and asynchronous communication compared

	Synchronous	Asynchronous
Definition	Request is delivered immediately to provider, and the invoker blocks until it receives a response.	The invoker doesn't wait to receive a response. Requests are buffered and will be processed when the provider is ready.
Advantages	<ul style="list-style-type: none"> - Simple interaction model - Immediate response - Allows the propagation of invocation context (transaction, security) 	<ul style="list-style-type: none"> - Good scalability under load - Resilience to failure (requester and provider needn't be available at the same time)
Disadvantages	<ul style="list-style-type: none"> - Lack of resilience to failure - Doesn't scale properly under load 	<ul style="list-style-type: none"> - Complex interaction model due to increased concurrency - Hard to debug

you'll find opportunities to apply the patterns that are specific to integrating multiple applications to different components of an individual application. The modules of the application may interact not through a set of interfaces that need to be kept up-to-date but through messages. Various functions might be executed asynchronously, freeing up their calling threads to service other requests. Distinct message handlers, otherwise perfectly encapsulated components, may be grouped together in an atomic operation.

2.2.2 Where does Spring Integration fit in?

While synchronous communication can take place using either procedure calls or messages, by its nature, asynchronous communication is message-driven. We explored the benefits of the messaging paradigm compared to the service-interface approach, and we found the message-driven approach superior, so here's what happens next: Spring Integration allows you to focus on the most important part of your system, the logical blueprint that describes how messages travel through the system, how they are routed, and what processing is done in the various nodes.

This is done through a structure of channels and endpoints through which the information-bearing messages travel to carry out the functionality of the application. It's a very abstract structure. To understand the actions that'll be taken in response to various messages, what transformations they'll suffer, where to send a particular message in order to get certain results, you don't need to know whether the communication will be synchronous or asynchronous.

Let's consider the following example:

```
<si:channel name="input" />

<si:payload-type-router>
    <si:payload-type name="com...Order" channel="orders"/>
    <si:payload-type name="com...Notification"
                      channel="notifications"/>
<si:payload-type-router>

<si:channel name="orders" />

<si:channel name="notifications" />

<si:service-activator ref="ordersProcessor"
    input-channel="applications-channel"
    output-channel="result-channel"/>

<si:service-activator ref="notificationsProcessor"
    input-channel="notifications-channel"
    output-channel="result-channel"/>

<si:channel name="results" />
```

Without getting into too much detail, this configuration means the following:

All requests made to the system are sent to the input channel where, depending on their payload (either Order or Notification), they're routed to the appropriate services. Services in this context are just POJOs, completely unaware of the existence of the messaging system. The results of processing the requests are sent to another channel where they're picked up by another component, and so on. See figure 2.6.

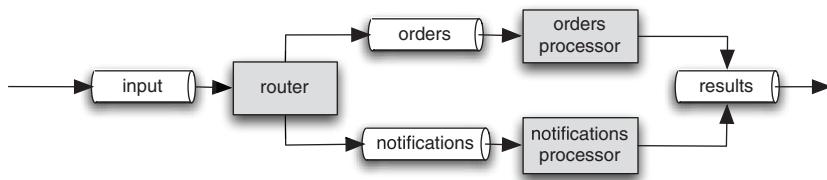


Figure 2.6 Wiring of the order processing system: Channels and endpoints define the logical application structure.

The structure doesn't seem to tell what kind of interaction takes place (is it synchronous or asynchronous?), and from a logical standpoint, it shouldn't. The interaction model is determined by the type of channel that's used for transferring the messages between the endpoints. Spring Integration supports both approaches by using two different types of message channels: `DirectChannel` (synchronous) and `QueueChannel` (asynchronous).

The interaction model doesn't affect the logical design of the system (how many channels, what endpoints, how they're connected). But for improving the performance, you may want to free the message-sending thread (which belongs to the main application) from executing the processing of notifications and applications and do that work asynchronously in the background, as seen in figure 2.7.

SEDA The technical name for this design is staged event-driven architecture, or SEDA, and you saw it in a previous section.

Both types of channels are configured using the same `<channel/>` element. By default, its behavior is synchronous, so the handling of a single message sent to the `input-channel`, including routing, invocation of the appropriate service, and whatever comes after that, will be done in the context of the thread that sends the message, as shown in figure 2.7. If you want to buffer the messages instead of passing them directly through a channel and to follow an asynchronous approach, you can use a queue to store them, like this:

```

<si:channel name="input-channel">
  <si:queue capacity="10"/>
</si:channel>
  
```

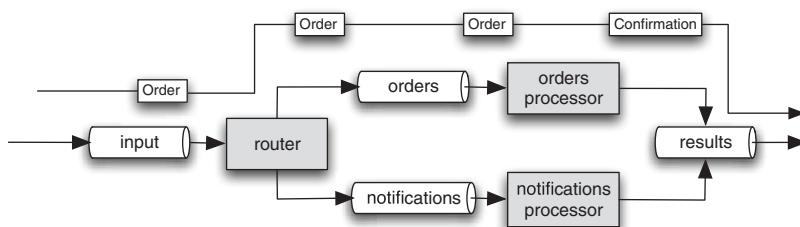


Figure 2.7 Synchronous order processing: The continuous line indicates an uninterrupted thread of control along the entire processing path.

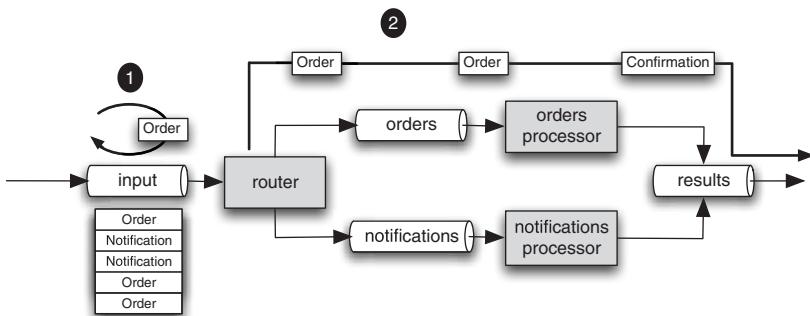


Figure 2.8 Asynchronous order processing: The introduction of a buffered channel creates two threads of control (indicated by continuous line): delivery to router takes place in ① and processing takes place in ②.

Now the router will poll to see if any messages are available on the channel and will deliver them to the POJO that does the actual processing, as shown in figure 2.8. All this is taken care of transparently by the framework, which will configure the appropriate component. You don't need to change anything in your configuration.

The configuration can also specify how the polling will be done or how many threads will be available to process notifications or orders, enabling the concurrent processing of those messages.

In this case, your application is implemented as a pipes-and-filters architecture. The behavior of each component encapsulates one specific operation (transformation, business operation, sending a notification, and so on), and the overall design is message-driven so that the coupling between components is kept to a low level. In some environments, it might be desirable to implement a full traversal of the messaging system as a synchronous operation (for example, if the processing is done as part of an online transaction-processing system, where immediate response is a requirement, or for executing everything in a single transaction).

In this case, using synchronous DirectChannels would achieve the goal while keeping the general structure of the messaging system unchanged. As you can see, Spring Integration allows you to design applications that work both synchronously and asynchronously, doing what it does best: separating concerns. Logical structure is one concern, but the dynamic of the system is another, and they should be addressed separately.

This overview of synchronous and asynchronous communication wraps up our discussion about coupling. Over the years, the effort to reduce coupling and to take advantage of the available communication infrastructure has led to the development of four major enterprise integration styles, which are the focus of the next section.

2.3 Comparing enterprise integration styles

As the adoption of computer systems in enterprise environments grew, the need to enable interaction between applications within the enterprise soon became apparent.

This interaction allowed organizations to both share data and to make use of functionality provided by other systems. Though enterprise application integration can take many different forms, from extract-transform-load jobs run overnight to all-encompassing SOA strategies, all approaches leverage one of four well-known integration styles:

- File-based integration
- Shared-database integration
- Remote Procedure Calls
- Message-based integration

In this section, we provide an overview of these four integration styles and their advantages and disadvantages, noting that although Spring Integration is built on the message-based paradigm, the other three forms are supported by the framework.

2.3.1 Integrating applications by transferring files

The most basic approach is for one application to produce a file and for that file to be made available to another system. This approach is simple and generally interoperable because all that's required is for interacting applications to be able to read and write files. Because the basic requirements for using file-based integration are simple, it's a fairly common solution, but some of the limitations of file systems mean that additional complexity may be created in applications having to deal with files.

One limitation is that filesystems aren't transactional and don't provide metadata about the current state and validity of a file. As a consequence, it's hard to tell, for example, if another process is currently updating the file. In general, to work properly, this type of integration requires some strategies to ensure that the receiver doesn't read inconsistent data, such as a half-written file. Also, it requires setting up a process by which corrupt files are moved out of the way to prevent repeated attempts to process them. Another significant drawback is that applications generally need to poll specific locations to discover if more files are ready to be processed, thus introducing additional application complexity and a potential for unnecessary delay.

Considering these drawbacks, before deciding to use file-based integration, it's always good to see whether any other integration style would work better. Nevertheless, in certain situations, it may be the only integration option, and it's not uncommon to come across applications that use it. It's important in such cases to be diligent and to address the limitations through appropriate strategies, like the ones mentioned previously, and Spring Integration addresses some of them through its support for file-based integration, as discussed in chapter 11.

2.3.2 Interacting through a shared database

Databases are better data storage mechanisms than files and alleviate some of the limitations of the filesystems. They provide atomic operations, well-defined data structures, and mechanisms that provide some guarantees on data consistency.

Shared-database integration consists of providing the different applications with access to the same database. In general, shared databases are used in two scenarios:

- 1 As a smarter form of data transfer, by defining a set of staging tables where the different applications can write data that will be consumed by the receiver applications. Compared with the filesystem style, this approach has the advantage that metadata, validation, and support for concurrent access are available out of the box for the transferred data.
- 2 By allowing different applications to share the same data. This has no direct correspondent to the filesystem style and has the advantage that changes made by one application are made available to everyone else in real-time (unlike the data transfer approach, which involves writing data in the file or staging tables and a certain lag until the recipient application makes use of it).

The challenges of shared-database integration are as follows:

- One-size-fits-all is hardly true in software development, and the compromises necessary to implement a domain model (and subsequent database model) based on the needs of multiple business processes may result in a model that fits no-one too well.
- Sharing the same data model may create unwanted coupling between the different applications in the system. This may seriously affect their capabilities of evolving in the future because changing it will require all the other applications to change as well (at least in the parts that deal directly with the shared model).
- Concurrent systems frequently writing the same set of data might face performance problems because exclusive access will need to be granted from time to time, so they will end up locking each other out.
- Caching data in memory might be an issue because applications may not be aware when it becomes stale.
- Database sharing works well with transferring data between applications but doesn't solve the problem of invoking functionality in the remote application.

When it comes to invoking functionality in the remote application, one possible solution is to use Remote Procedure Calls, which we discuss next.

2.3.3 **Exposing a remote API through Remote Procedure Calls**

Remote Procedure Calls (RPC) is an integration style that tries to hide the fact that different services are running on different systems. The method invocation is serialized and sent over the network, where the service is invoked. The return value is then serialized and sent back to the invoker. This involves proxies and exporters (in Spring) or stubs and skeletons (in EJB).

RPC would presumably allow architects to design scalable applications without developers having to worry about the differences between local and remote invocations. The problem with this approach is that certain details about remoting can't

safely be hidden or ignored. Assuming that it can hide these details will lead to simplistic solutions and leaky abstractions. Dealing with problems properly will violate the loose coupling we've come to enjoy.

RPC requires that parameters and return values of a service are serializable, which means they can be translated into an intermediate format that can be transmitted over the wire. From case to case, this can be achieved in different ways, such as through Java serialization or using XML marshalling through mechanisms such as Java Architecture for XML Binding (JAXB). This not only restricts the types that can be sent, it also requires that the application code deal with serialization or marshalling errors.

We're probably not the first to tell you, but it's worth mentioning again that the network isn't reliable. You can assume that a local method invocation returns within a certain (reasonable) time with the return value or an exception, but with a network connection in the mix, this can take much longer, and worse, it's much less predictable. The trickiest part is that the service that's invoked can return successfully, but the invoker doesn't get the response. Assuming you don't need to account for this is usually a bad idea.

Finally, serializing arguments and return values harms interoperability. Sending a representation of some `Serializable` to an application written in Perl is wishful thinking at best. The need to know about the method name and argument order is questionable. Once this is understood, it's not such a big leap to look for an interoperable representation and a way to decouple the integration concerns from the method signatures on both sides. This is one of the goals of messaging, our final integration style and the one that's the general focus of Spring Integration.

2.3.4 Exchanging messages

Messaging is an integration style based on exchanging encapsulated data packets (messages) between components (endpoints) through connections (channels). As described at www.enterpriseintegrationpatterns.com, the packets should be small, and they should be shared frequently, reliably, immediately, and asynchronously. Potentially this resolves many of the problems of encapsulation and error handling associated with the previous three integration styles. It also provides an easy way to deal with sharing both data and functionality, and overall it's the most recommended integration style when you have a choice.

Message exchanging is the core paradigm of Spring integration, and we discuss in detail in chapters 3 and 4 the concepts behind the integration style and how they're applied in the framework.

2.4 Summary

This chapter explained the fundamental concepts that stand behind enterprise application integration. We provided applicable samples from the framework, and in the rest of the book, you'll see them at work.

First, we discussed coupling as a way to measure how many assumptions two independent systems make on each other. You saw how important it is to minimize coupling to allow the components to evolve independently. There are multiple forms of coupling, but the most important ones are type-level and temporal coupling. Dependency injection helps you deal with type-level coupling, and using a message-passing approach instead of direct calls (either local or remote) helps you deal with temporal coupling.

We presented a detailed overview of the differences between synchronous and asynchronous communication and its implications in coupling, system complexity, and performance. You saw how Spring Integration helps separate the logical design of the system from the dynamic behavior at runtime (whether interaction should be synchronous or asynchronous).

Finally, you got an overview of the four application integration paradigms and their respective strengths and weaknesses. Spring Integration is generally focused on messaging, but it provides support for using the other three types as well.

We introduced Spring Integration from a high level in the first chapter. Then we zoomed in on the fundamental concepts at the root of messaging in chapter 2. In chapter 3, we look in detail at the parts of Spring Integration that enable messaging; starting with Messages and Channels.

Part 2

Massaging

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 of 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 book, most chapters 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. In this chapter we break that pattern by starting with the introduction of API-related concepts because messages and channels are fundamental concepts, and understanding how Spring Integration handles 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 messages as a fundamental enterprise integration pattern and provide an in-depth look at how they’re 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. A *piece of information* is any 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 messages 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 is supposed to 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 this message. For example, every message has an identifier header that uniquely identifies the message instance. In other cases, messages may contain correlation information indicating that various individual messages belong to the same group. In other cases, headers may 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 file system), the message will contain information specific to that source (like the JMS headers or the path to the file in the file system).

The `MessageHeaders` are a `Map` with some additional behavior. Their role is to store the header values, which can be any Java `Object`, and they are 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 are immutable. You might worry 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.

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 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();
```

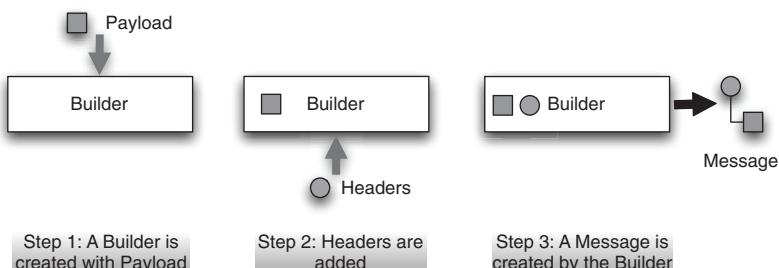


Figure 3.1 Creating a message with the `MessageBuilder`

The MessageBuilder also provides a variety of methods for setting up the headers, including options for setting up values when a given header isn't already set. A more elaborate variant of creating the helloMessage could be:

```
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> helloMessage =  
    MessageBuilder.fromMessage(helloMessage)  
        .setHeader("additional.header", "AnotherValue")  
        .setHeaderIfAbsent("my.custom.header", "replacement")  
        .build();
```

Now you know what messages are in Spring Integration and how you can create one. 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 among 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 of your application by using the right kind of channel for 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 {
    String getName();
    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, described next, 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 message 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 about 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 to ensure 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 correct 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	<ul style="list-style-type: none"> – 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 but 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 in concurrently executing steps, prevent partial failures, or introduce security risks such as leaking permissions to the processing of different messages.
Atomic boundaries	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> – 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	<ul style="list-style-type: none"> - 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 broadcasted 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.

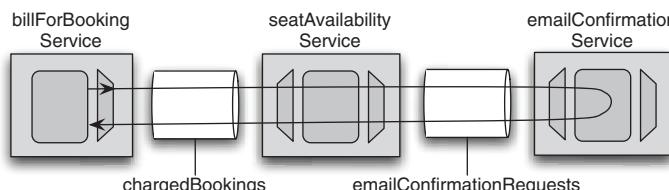


Figure 3.2 Diagram of threading model of service invocation in the airline application

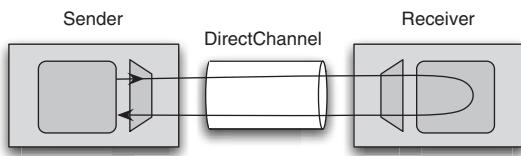


Figure 3.3 Diagram of threading model of service invocation when using a default channel

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/>`.

```

<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" />
  
```

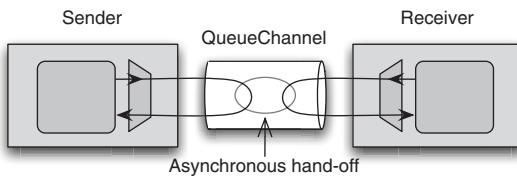


Figure 3.4 Diagram of threading model of service invocation when using a QueueChannel

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

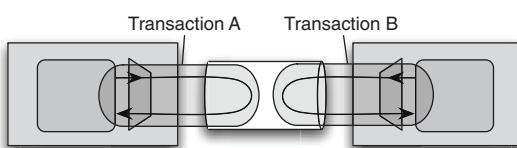


Figure 3.5 Diagram of transactional boundaries when a QueueChannel is used

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="emailConfirmationRequests"
                    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 the means of a publish-subscribe channel. Let's get to the last challenge and emerge victoriously with our drastically 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="emailConfirmationRequests"
                    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" />
```

What you just saw is an example of applying different types of channels for solving the different requirements. Starting with the defaults and working through the example, you 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 recommendable in one use case may not be recommendable in another. We illustrated the decision process with the criteria we find most relevant in every 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 called *channel collaborators*.

We next look at the `MessageDispatcher`, which controls how received messages 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 receiving messages from 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 one 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 `theRoundRobinLoadBalancing`, 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. In this case, 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 before a component calls `receive()` on the 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.¹

```
package sria.business;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.Message;
import org.springframework.integration.MessageChannel;
import org.springframework.integration.channel.interceptor
    .ChannelInterceptorAdapter;

public class ChannelAuditor extends ChannelInterceptorAdapter {

    @Autowired
    private 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="sria.business.ChannelAuditor">
    <beans:property name="auditService" ref="auditService"/>
</beans:bean>

<channel id="chargedBookings">
    ...

```

¹ Spring Integration provides a no-op implementation, `ChannelInterceptorAdapter`, that framework users can extend. It allows users to implement only the needed functionalities.

```

<interceptors>
    <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>
        <wiretap 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.

```

<beans:bean id="typeSelector"
    class="org.springframework.integration.selector.PayloadTypeSelector">
    <beans:constructor-arg value="sia.business.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 concept of messages and channels are the key to the flexibility inherent in applications built on Spring Integration. The ease of swapping channel implementations provides a degree of flexibility in controlling threading models, thread utilization, and latency not seen in any other mainstream Java EAI framework. 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.

Message Endpoints

This chapter covers

- Types of Endpoints and how they differ
- Transaction boundaries around Endpoints
- Endpoints under the hood

In the previous chapter, we covered Message Channels in detail. You now know that some channels accept subscribers to be called in an event-driven way, whereas others require polling consumers. The former enable synchronous invocation of a handler and thereby enable transactions to span both the producer and consumer. The latter offer the flexibility of even more loosely coupled interaction, but break that transaction boundary across separate threads for the producer and sender.

From the programming perspective, the event-driven model is easier to grasp. For example, the `MessageHandler` interface that's central to Spring Integration is as simple as it can be:

```
package org.springframework.integration.core;  
  
public interface MessageHandler {  
  
    void handleMessage(Message<?> message);  
}
```

All message-handling components in Spring Integration, such as Transformers, Splitters, and Routers, implement that interface. Therefore those implementations are relatively straightforward, always reacting to a received message, much in the same way as a Java Message Service (JMS) `MessageListener` would. These components can be connected to any type of channel, but some of the channel types accept event-driven subscribers, whereas others must be polled. Clearly, to connect components to a channel, a certain amount of glue is necessary. In Spring Integration, that glue comes in the form of an adapter that understands how to interact with a given channel while delegating to a `MessageHandler`. The generic term for that adapter is a *message endpoint*.

A few different kinds of adaptation are required depending on the type of channel and the role of the handler component being connected to that channel. If the channel is subscribable, the handler is invoked within the thread of the sender or by the channel's `TaskExecutor` if one is configured. For such a channel, no polling is required. If on the other hand the channel is one that buffers messages in a queue, such as a `QueueChannel`, polling is necessary.

In addition to the distinction between subscription and polling, you must consider whether a reply message is expected. If a reply is expected, would that reply be sent back to a caller or passed to the next component within a linear pipeline? Likewise, when interacting with an external system, you must consider whether a component is unidirectional (a channel adapter) or bidirectional (a gateway). One of the key benefits of a messaging system is the ability to support these different interaction models and to even switch among them by changing configuration rather than directly impacting the underlying components. In other words, it should be trivial to change from a request-reply interaction to a pipeline if you determine that an additional transformation step is necessary before sending a reply.

In this chapter, we explore the different endpoints available in Spring Integration and see how they can be used to decouple message-producing and message-consuming components from the channel type and response strategy. First we describe the difference between polling and event-driven components in detail. Then we look at those differences in the context of inbound and outbound endpoints. Finally we discuss the different response strategies to clarify the distinction between Channel Adapters and Gateways.

SPRING INTEGRATION AN IMPLEMENTATION OF THE ENTERPRISE INTEGRATION PATTERNS?

Spring Integration stays true to enterprise integration patterns (EIP) naming wherever sensible. Not all patterns have their direct counterpart in the API—not every *pattern* maps directly to an *implementation*. Some patterns are more conceptual, and others describe a broad category within which implementations may be classified. The `MessageEndpoint` is an example of such a pattern. The endpoint is too generic to be covered by a single concrete implementation. Instead, we opted for different endpoint implementations in a class hierarchy and explicitly named patterns like `Transformer`, `Splitter`, and `Router` defined in an

XML schema. Likewise, taking into account the various *external* endpoints, the named patterns Gateway and Channel Adapter are defined in an XML schema for each supported type of system (file, JMS, and so on). Other patterns that are supported but not literally implemented include Request-Reply, Selective Consumers, and Return Address. The first two are composite configurations of other components, and the Return Address is supported by a message header that's recognized by any reply-producing message handler that doesn't have an explicit output channel configured.

Before we can talk about endpoints and decide which one is best applied to a given problem, it's important to understand the different properties of endpoints a little better. The next section goes over different characteristics of endpoints that will keep popping up in the rest of the book.

4.1 What can you expect of an endpoint?

As noted in the introduction of this chapter, *endpoint* is a broad term. We need to be precise in our understanding of the properties of a certain type of endpoint before we can have a meaningful discussion about it.

One advantage of Spring Integration is the consistent naming in the namespace support. This section explores what makes endpoints different enough to give them different names. Several characteristics are important:

- Polling or event-driven
- Inbound or outbound
- Unidirectional or bidirectional
- Internal or external

Considering all the possible ways we can combine the switches, we end up with 16 candidates for unique names. Multiply this by the number of supported protocols in case of an external endpoint, and the numbers become dazzling.

Luckily, only certain combinations make sense for a given protocol, which greatly reduces the number of options. Also, Spring Integration can implicitly take care of the polling concerns. You still have plenty of options to consider—polling or event-driven, inbound or outbound (from the perspective of the Spring Integration application), unidirectional or bidirectional (one-way or request-reply respectively), internal or external (with respect to the application context)—so let's look at the most important examples in table 4.1.

As you might have guessed from the table, there are some naming conventions for endpoints. Wherever possible, the names were chosen to stay as close as possible to a pattern. In addition, the properties are signified by consistently applied tokens. For example, a gateway is always capable of bidirectional communication. This doesn't change between HTTP and JMS or between inbound and outbound. In contrast, a channel adapter is always unidirectional: it's either the beginning (inbound)

Table 4.1 Differentiating characteristics of Endpoints.

Endpoint	Polling/Event Driven	Inbound/Outbound	Unidirectional/Bidirectional	Internal/External
<inbound-channel-adapter>	Polling	Inbound	Unidirectional	Internal
<outbound-channel-adapter>	Event-driven	Outbound	Unidirectional	Internal
<gateway>	Event-driven	Inbound	Bidirectional	Internal
<service-activator>	Either	Outbound	Bidirectional	Internal
<http:outbound-gateway>	Either	Outbound	Bidirectional	External
<jms:inbound-channel-adapter>	Event Driven	Inbound	Unidirectional	External

or the end (outbound) of a message flow. Now that you know the choices you can make before selecting an endpoint, it's time to look into the implications of those choices. First we ask what may be the most important question when designing a messaging system.

4.1.1 **To poll or not to poll?**

Those of you who've read *Hamlet* know that Shakespeare got some of it right: the important decision is whether you should be a slave to your invokers or take matters in your own hand. We shouldn't stretch the analogy to suicide, so let's get practical.

When a user of your application is entering a lot of information, you'd like to save that input somewhere as soon as possible. Modern browsers support client-side storage facilities, but in many applications that need to be compatible with lesser browsers, this isn't an option. Even if client-side storage is an option technically, in many cases it isn't an option functionally. Imagine a web application that allows users to collaborate, as they would with an online word processor. This would require changes to flow from one user to another in two directions with the server as a referee in the middle. Although it's important to get the changes from one user to another quickly, it's much more important to get the changes from a single user to his or her client quickly. In other words, if a sentence reaches another user a second after one user typed it, it would be fine. If the cursor were trailing a sentence behind the current user's typing in the client, nobody would use the application.

In most cases when you're typing, an application should respond to a key press within 0.1 seconds; otherwise the application would be unusable. Changes from somebody else may take up to a second before you would notice that they're delayed. You've probably seen a few applications that allow you to collaborate within these boundaries; the rise of this type of collaborative editor came with the rise of Ajax. With the boundaries of network latency, the only option to make an application like this work is to asynchronously send one user's changes to another user's client without making the first user wait for confirmation. As discussed in chapter 3, we use messages as the central concept to transfer data from one place to another.

If messages are handed off asynchronously, this assumes that they should eventually be received by a polling consumer. Because the poller is an active process, it requires a separate worker thread. For this purpose, Spring Integration integrates with any implementation of the core Spring TaskExecutor abstraction. Thread management is then a responsibility of that implementation and can even be delegated to a WorkManager when running in an application server environment that requires threads to be managed by the container. We discuss the details of scheduling and concurrency considerations later in this book. The problem at hand is to make sure the separate worker thread is available and used properly to continue processing messages that have been handed off asynchronously. Even more important, we need to understand how to switch between synchronous and asynchronous invocation.

POLLING ENDPOINTS

A polling endpoint will actively request new data to process. This data may be messages in a QueueChannel, files in a directory, messages in a JMS Destination, and so on. The endpoint needs at least a single thread to perform this polling. You could hand-code an endpoint like this by creating an infinite loop and invoking receive periodically. Although it may be easier to understand the inner workings of an active endpoint because the behavior is self-contained, coding components like this yourself has some serious downsides. First of all, it requires you to write threading code, which is notoriously hard to do right. Second, it makes the component much harder to test in a unit test. Third, it becomes troublesome to integrate the component where a passive component is needed. In Spring Integration, you don't have to write different code depending on whether you want your component to be active. Instead, whether the component should be active or passive is inferred from the configuration and handled by the framework. Configuring the components that will take the responsibility for the polling concerns is still up to you.

EVENT-DRIVEN ENDPOINTS

Asynchronous handoff usually isn't required, and in those cases, an event-driven model should be used. This can be as simple as wrapping a plain old Java object (POJO). Exposing a web service is also a good example. The essential thing about passive, or event-driven, components is that they don't take responsibility for thread management. They're still responsible for proper thread safety, but this could be as simple as not maintaining any mutable state.

4.1.2 *Inbound endpoints*

Just as important as getting information out of the messaging system is getting information into it. This is done through inbound endpoints, typically receiving information in a format that's not native to Spring Integration. Examples are a web service invocation, an inbound email, or a file written to a directory. A Java method invocation is also a plausible inbound integration point. The generic algorithm for an inbound endpoint is shown in figure 4.1.

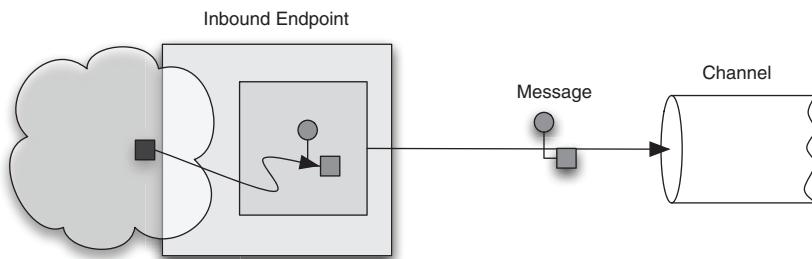


Figure 4.1 Behavior of an inbound endpoint. Input is taken from an external source then converted into a message, which is sent to a channel. This image doesn't show the optional reply message.

POLLING INBOUND ENDPOINTS

Whether an inbound endpoint needs to poll or can be event-driven depends on the architecture external to the message system. Some technical constraints, such as the lack of file system events in Java, affect this decision. But in most cases it depends on the system with which you're integrating. If the external integration point is passive, the inbound endpoint becomes responsible for actively polling for updates.

In Spring Integration, several polling endpoints are provided: inbound channel adapters for file, JMS, email, and a generic method-invoking variant. The latter can be used to create custom channel adapters, such as for querying Twitter or a Really Simple Syndication (RSS) feed.

EVENT-DRIVEN INBOUND ENDPOINTS

Often, an external system actively invokes a service on the messaging system. When that happens, the responsibility of thread management can be left out of the messaging solution and left to the invoker (in the case of a local Java method call) or to the container (in the case of a web service invocation). But as soon as QueueChannels are to be used internally, thread management becomes a concern of the messaging system again.

Examples of event-driven endpoint types supported in Spring Integration include web services (through Spring WS), Remote Method Invocation (RMI), and JMS (where Spring's listener container takes care of the polling). Again, you can generically create a custom event-driven endpoint using the `@Gateway` annotation.

4.1.3 **Outbound endpoints**

At some point in a message flow, you'll likely need to invoke a service that's external to the messaging system. This may be a local method call, a web service invocation, or a message sent on another messaging system such as JMS or Extensible Messaging and Presence Protocol (XMPP). An endpoint is responsible for making this call. The general responsibilities of an outbound endpoint are depicted in figure 4.2.

The algorithm shown in this figure usually must implement details such as transactions and security, which we cover later. The details of conversion and the invocation

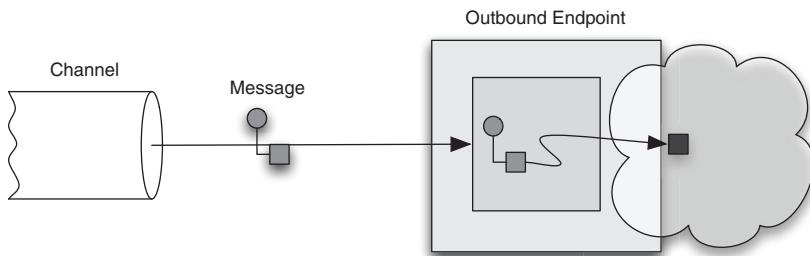


Figure 4.2 Behavior of an outbound endpoint. First a message is received from a channel; then the message is converted into something the external component understands. Finally the external API is invoked. The optional result is ignored in this image.

of external services are also assumed to be implementation details of the specific endpoint. But the reception of the message and whether a result is generated are concepts that belong to the base API.

POLLING OUTBOUND ENDPOINTS

If an outbound endpoint is connected to a `PollableChannel`, you must invoke the `receive` method on its input channel to receive messages. That requires scheduling an active process to do the polling periodically. In other words, step 1 in the algorithm is triggered by the endpoint. Spring Integration automatically wraps any passive components in a polling endpoint if they're configured to be on the receiving end of a `PollableChannel`.

EVENT-DRIVEN OUTBOUND ENDPOINTS

When an outbound endpoint is connected to a `SubscribableChannel`, it can be passive. The channel ensures that a thread exists to invoke the endpoint when a message comes in. In many cases, the thread that invoked the `send` method on the channel is used, but it could also be handled by a thread pool managed at the channel level. The advantage of using a thread pool is that the sender doesn't have to wait, but the disadvantage is that a transactional boundary would be broken because the transaction context is associated with the thread. This is discussed in detail in section 5.2.

4.1.4 Unidirectional and bidirectional endpoints

The previous examples all assume unidirectional communication. Endpoints using this style of communication are called *Channel Adapters*. Unidirectional communication is often enough to establish a working solution, but bidirectional requirements are common. For these scenarios, you can use Gateways. The EIP definition of Gateway isn't as clear as we would've liked, so we did a bit of interpretation. Mainly, the lack of clear distinction between synchronous invocation and asynchronous handoff makes the Gateway concept too broad and widely overlapping with other concepts, such as Channel Adapter. In Spring Integration, a Gateway is synonymous with synchronous two-way communication. If you need an *asynchronous gateway*, you should compose it from other base components (for example, an inbound and outbound channel adapter).

THE RETURN ADDRESS

When a message reaches an endpoint, through polling or otherwise, two things can happen:

- 1 The message is consumed and no response is generated.
- 2 The message is processed and a response is generated.

The first case is where you'd use a Channel Adapter, as shown in the previous sections. The second case becomes more complicated, but luckily Spring Integration has Gateway to help you support it. The complexity lies in that you must do something with the response.

Usually you'll want to either send it further along its way to the next endpoint or send a confirmation back to the original sender when all processing for this message is done. For the first option, you set an output channel, and for the second, you omit the output channel and Spring Integration will use the REPLY_CHANNEL header to find the channel on which the original sender wants the confirmation to go.

At this point you should have a good understanding of the different high-level categories of endpoints. We explored the distinctions between polling and event-driven consumers, and we discussed both unidirectional and bidirectional behavior. In the next section, we add one of the most important pieces to the puzzle: transactions. Unfortunately, although their importance is undeniable, transactions can be confusing for many developers. We'll do our best to remedy that situation, at least in the context of Spring Integration applications. If you're an expert on transaction management, you might like to skip parts of the next section and just skim for Spring Integration particulars.

4.2 ***Transaction boundaries around endpoints***

A common mistake in moving from thread per request to asynchronous handoff is to make incorrect assumptions about transaction boundaries. Another common mistake is to incorrectly assume that the security context is set in the thread that's being handed off to. Both incorrect assumptions occur because security and transaction contexts are traditionally stored in a ThreadLocal. This makes sense for most web applications, and you should practice caution if you want to break this convention. Security contexts shouldn't be shared lightly, and transactions should be kept short.

This section goes into the details of transaction management around Spring Integration endpoints. This isn't a general work on transactions, so we focus on endpoint-related concerns here. Before we explore the technicalities of transaction management and draw parallels to security, we must establish the rationale of thread-bound contexts.

4.2.1 ***Why sharing isn't always a good thing***

Have you ever been working on a shared file and found out that someone else just fixed the same problem you worked on all morning? We bet you have. The first reaction to

a situation like that is usually to request immediate notification of any changes that others plan to make. Or your team decides to plan better or to use a new tool.

Have you ever tried to work on something and been constantly distracted by team members who were trying to work on the same thing and tried to align with you? Have you ever been stuck in planning meetings all day? We bet you have.

These are two sides of the same problem. You must carefully consider the consequences of living with the overhead of either sharing or merging changes. This is exactly the choice you must make when establishing the extent of a transaction. From a traditional web application point of view, you want to give your user a consistent view on reality (*open transaction in view* or even a *conversation*), but you also want to ensure the smallest possible chance of your user running into a conflict with another user's changes.

Good practice is to keep transactions small and to use compensation instead of rollback to recover from mistakes. For this reason, it makes sense to have a transaction context confined to the thread that's servicing a single HTTP request. One commonly suggested approach is to store transaction contexts in a message instead, but that might considerably increase the scope and duration of the transaction, so it's not natively supported by Spring Integration.

We talked about sharing a view of the world from the perspective of an editor, but how about sharing only with people you trust?

Security contexts don't become stale over time or open the door to merge conflicts. Once properly authorized *and kept in sight at all times*, a user doesn't need to re-enter a password. The problem is that it's easy to let someone slip out of sight for a second. If you share a security context with multiple threads, you need to be careful to ensure that the security details are visible only from the context of the user. If multiple users are using the application concurrently, the least error-prone isolation mechanism is to allow access only from the thread currently processing the request.

PASS SECURITY CONTEXT INFORMATION ALONG WITH A MESSAGE Bind contextual information from the security context to a message header before handing off to another thread. Usually, the authentication is no longer needed; just a username will do.

Classic web applications usually don't need to worry about asynchronous handoff to work. Spring Integration comes in when asynchronous handoff becomes relevant, and at that point, you must jump through some hoops. Of course, the first thing a crafty engineer asks before jumping through hoops is whether they can get away with not doing it.

4.2.2 **What are transactions, and can we get by without them?**

Go to the most senior database administrator in your organization and ask this question. (It'll be fun, we promise!) After you've listened carefully to his lecture, remember at least one thing: there's no excuse for being clueless about transactions. That said,

there's no excuse for being a zealot, either. Transactions aren't the answer to all your problems, and overusing them can be detrimental to performance and scalability.

ACID, as we all know, stands for *atomic, consistent, isolated, and durable*. A transaction is supposed to give you all of this. Don't be fooled: with a lot of ifs and buts, a transaction will give you something quite close to ACID, or an exception. The tricky part of ACID is isolation in combination with global consistency. Let's look at two examples of multiple reasonable users in a standard web application.

THE SEAT SELECTION CONFLICT

Jill and Jack must check in for their afternoon flight to New York. They're on the same flight, and they have independently decided to try to find a seat next to the other person. The flight is at 4:20, and they're expecting a cab to the airport at 3:00. It's now 2:49, so they're in a hurry.

Jill hits the seat selection application first. She tags that she wants to travel with a companion and fills in Jack's details. The application renders a page to Jill that shows Jack's position on the plane. Jack is in an aisle seat on row 15. Jill stops to check the make of the plane and the seat arrangement to decide whether she wants the aisle seat opposite to Jack or whether she wants to take the middle seat next to him.

Now Jack hits the site, follows the same steps, and (this is a no-brainer) selects the middle seat next to Jill's window seat on row 8. It won't be as comfortable as the other seats, but at least they'll sit together. Jack's transaction completes fine, and he runs for his cab. Jill is done as well, hits Submit, sees that her seat number has changed, and continues to synchronize her mail before she goes offline.

It doesn't take a lot of mental effort to predict that Jack and Jill are in for a surprise.

The most important thing to take away from this example is that the only way to prevent this scenario is to *stop the world* as soon as a passenger starts seat selection. But how long are you prepared to wait for that passenger to check the seating arrangement before you allow other passengers to get on? Having a globally consistent view of the world is commonly too impractical to work.

Are use cases like the story of Jill and Jack impossible to accommodate? Surely not! Anything is possible: you just need to find an effective way to deal with conflicting changes. We explore several ways next.

FIX THE PROBLEM WHEN THINGS GO WRONG

If the program could figure out that Jill and Jack want to sit next to each other, it could warn them if their effort failed. For example, after Jill submits her change, the application could show an alert warning her that she's moving away from her companion. This will come as a surprise to Jill: in her reality, she was moving toward Bill, not away from him. A properly chosen alert message would make sense to Jill. Let's say the message is, "Your companion has moved in the meantime. Are you sure you want to move away from him/her?" Jill would be able to comprehend this and revise her seat selection.

The scenario just described is a form of *eventual consistency* achieved by *compensating transactions*. After the transaction is committed, stakeholders are invited to make compensating changes based on the new reality.

EXPRESS INTENT IN THE MESSAGE; LEAVE IMPLEMENTATION TO THE MEDIATOR

What if Jill and Jack didn't change their seats but asked to be seated together? Then the scenario would play out in one go. Jack asks to be seated next to Jill first and is moved. Then Jill asks to be seated next to Jack, and the server knows that no move is needed.

The concession here is that Jill and Jack no longer control their locations. Allowing them to do so would require a much more complex application and, most important, would make the application more difficult to use. This isn't always desired, but it can be powerful.

If you take a trivially simple implementation of this system, you can make it work as long as you're prepared to send many requests. Let's say the server understands "Please move me next to my companion," and it blindly executes this directive without any transaction. In this case, conflicting changes such as in the example might occur, but if you keep sending the message until you see the right result as a client, the system also implements eventual consistency.

Eventual consistency is guaranteed through a redelivery policy combined with an *idempotent receiver*.

The two flavors of eventual consistency are both ultimately based on their not having to support a rollback. Rollbacks are the Achilles heel of scalability. If you can get by without them, you can loosen the transactional requirements to allow massive improvements in scalability.

But hold your horses; we're not done yet. If you understand the reasoning behind relaxing ACID constraints, it becomes even more important to understand where you need to replace transactional boundaries with eventual consistency strategies. If you miss a beat here, you'll end up with *lost updates*, *dirty reads*, and all their two-faced friends and relatives. There's no excuse for being clueless about transactions, remember? Let's look at the details of transactions around endpoints.

WHERE IS MY TRANSACTION?

In chapter 3, we looked at transactions around channels. Remember that the transaction boundary is broken as soon as you add a task executor or a queue to a channel. In this section, we look at the beginning and end of the transaction from within the endpoint.

As we learned in the first section of this chapter, there are many different types of endpoints. As much as possible, we've aimed to maintain transactions within the endpoint as a rule. This makes it easier for you, as the user, to identify transaction boundaries on channels. It also allows you to extend a transaction over multiple endpoints by using the right channel in between.

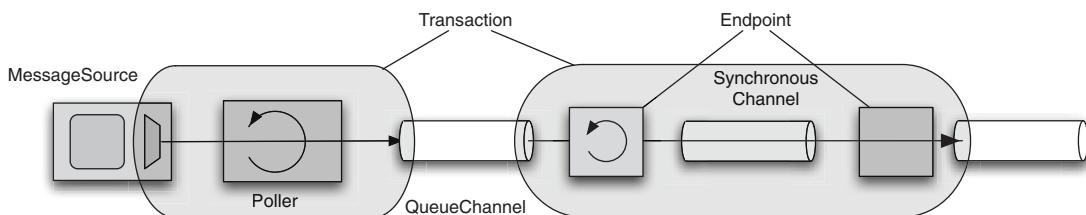


Figure 4.3 Transaction boundaries are determined by the scope of a given thread's responsibility, so the transactional context doesn't propagate across an asynchronous channel.

A transaction is started by a poller just before it pulls a message from a `MessageSource`. To be precise, the poller starts a transaction only if it was configured with a transaction manager. The transaction is then committed as soon as the send method of the channel the poller is sending to returns. If the `MessageSource` is transactional, it participates in the same transaction as the downstream endpoints as long as the boundary isn't broken by an asynchronous handoff.

Similarly, a poller equipped with a transaction manager starts a transaction before pulling a message from a `QueueChannel`. The business transaction in the downstream endpoint is wrapped by a transaction that includes the reception of the message from the channel. It also includes sending the message to the output channel.

Figure 4.3 shows the two standard transaction scopes to keep in mind when designing a system.

When letting a poller take care of the transaction, as shown in the figure, transaction management is simple. The only thing left to keep an eye on is endpoints that can break transactional boundaries around your message. For example, when you use an aggregator, the messages are stored in the endpoint until they're complete or a timeout occurs. This means the message going in doesn't have to continue on the same thread, so it usually doesn't participate in the same transaction.

Whenever transaction management around an endpoint is not related to the poller that invokes it, we give you a heads up in the relevant chapter. In general, this is the case whenever an endpoint can be configured with a task executor.

Although this chapter is relatively technical in nature, much of the discussion has been theoretical. Even when discussing concerns specific to Spring Integration, the examples have been mostly at the namespace-based configuration level. In the next section, we again take a look under the hood so that you can learn more about the implementation details of message endpoints, polling consumers, and event-driven consumers. You'll have a better understanding of the types of components that are created by the namespace parser. If you're not interested in that level of detail, feel free to skip ahead to the summary that follows.

4.3 Under the hood

Early in this chapter, you read that Spring Integration can take care of polling concerns for you. We haven't explained in detail how this works, and in daily life you

don't need to worry about it. If ever you find yourself debugging a Spring Integration application, though, you might benefit from knowing how this works under the hood. Things *will* get technical now, and if you feel you've seen enough at any point, feel free to skip this section. If you're not afraid to look inside, this section is for you.

The base class for all endpoints is `AbstractEndpoint`, and several subtypes exist to take care of the differences between polling and event-driven behavior. `PollingConsumer` wraps a `MessageHandler` and decorates it with a poller so that it can be connected to any `PollableChannel`. `EventDrivenConsumer` wraps a `MessageHandler` and connects it to any `SubscribableChannel`. As you can imagine, the latter is trivial. The `AbstractPollingEndpoint` is more complicated because it has to find a poller and handle exceptions asynchronously. Albeit more complex, this is still nothing more than a wrapper around a `MessageHandler`.

Things become more interesting when we start looking into what happens during the parsing of the application context where the decisions between polling and event-driven behavior are made.

4.3.1 Endpoint parsing

Each XML namespace supported by Spring Integration and the other Spring projects is typically described as a domain-specific language (DSL). Here our domain is enterprise integration. The domain model consists of messages, channels, transformers, routers, and so on. The advantage is that the elements in the namespace are *closer* to the concepts that we're working with than are, say, a simple Spring bean element. You can always drop down to a lower level and configure everything as simple beans as long as you're familiar with the API. But considering most Spring Integration users rely largely on the schemas defined in the various namespaces, we include the role of parsers in our discussion. Let's begin with the following configuration of two transformers and two channels:

```
<channel id="channel1" />

<channel id="channel2">
    <queue/>
</channel>

<transformer id="transformer1"
    input-channel="channel1"
    expression="payload.toUpperCase()"
    output-channel="nullChannel" />
<transformer id="transformer2"
    input-channel="channel2"
    expression="payload.toLowerCase()" />
```

The first thing to notice is that `channel1` has no queue. It's the simplest channel type available in Spring Integration. That element leads to the creation of an instance that implements the `SubscribableChannel` interface, and any endpoint referencing `channel1` as its input channel would be invoked directly when a sender sends a message to that channel. The `channel2` element leads to the creation of an instance that

implements the `PollableChannel` interface, and internally that channel buffers messages that are sent to it. Those messages must then be explicitly received by some active poller. Let's now walk through the result of parsing the associated endpoints.

The endpoints in this case are both Message Transformers. Each has a trivial expression to evaluate, and for purposes of this example, we can assume that the messages will all have a string typed Message payload. Each element supported by a Spring namespace handler is mapped to the implementation of Spring's `BeanDefinitionParser`, which is responsible for working with that element. We avoid going into detail about the internals of the parser implementations, but for those who are curious, feel free to check out the code for `AbstractConsumerEndpointParser`, the common base class for several endpoint parsers in Spring Integration. Its primary role is to connect `MessageHandler` objects to the correct input channels, as defined in the configuration.

At parsing time, a limited amount of information is known. For example, when parsing `endpoint1`, the parser only knows that the channel named in the `input-channel` attribute (`channel1`) should be present within the same context and that the endpoint it creates must be injected with a reference to that channel. The endpoint parser doesn't know yet whether that's a pollable or subscribable channel. As a result, the parser can't know whether to create a polling or event-driven consumer. To handle that limitation, Spring Integration's parsers rely on another common Spring feature: the `FactoryBean` interface. Any object defined in a Spring context which implements that interface is treated differently than other objects. Instead of instantiating the object and adding it to the context, when Spring encounters an object that implements the `FactoryBean` interface, it creates that factory and then invokes its `getObject()` method to create the object that should be added directly to the context. One advantage of that technique is that much more information is available when `getObject()` is called than is available at parsing time. This is how Spring Integration's endpoint parsers avoid the limitation mentioned previously.

The implementation in this case is called `ConsumerEndpointFactoryBean`. As its name suggests, it's a generic creator of consumer endpoints. It can produce either polling or event-driven consumers. Because the input channel instance referenced by name in the configuration is available at the time the factory is invoked, the decision can be made on demand, so if the referenced channel is pollable, `ConsumerEndpointFactoryBean` creates a `PollingConsumer` instance, but if the channel is subscribable, it creates an `EventDrivenConsumer` instance instead. Let's now look at each of those two implementations.

4.3.2 **Endpoint instantiation**

The previous section described the two types of consumer endpoints: `PollingConsumer` and `EventDrivenConsumer`. The distinction between them is based on the type of input channel: `PollableChannel` or `SubscribableChannel`, respectively. This distinction becomes clear when we investigate the constructors for each of these objects. Both take a `MessageHandler` that handle the messages sent by a producer to

the input channel, but the interface of the expected channel is different. The PollingConsumer expects a PollableChannel:

```
package org.springframework.integration.endpoint;

public class PollingConsumer {
    public PollingConsumer(PollableChannel inputChannel,
                          MessageHandler handler) {
        this.inputChannel = inputChannel;
        this.handler = handler;
    }
    ...
}
```

An EventDrivenConsumer expects a SubscribableChannel:

```
package org.springframework.integration.endpoint;

public class EventDrivenConsumer {
    public EventDrivenConsumer(SubscribableChannel inputChannel,
                               MessageHandler handler) {
        this.inputChannel = inputChannel;
        this.handler = handler;
    }
    ...
}
```

You can see that it's not too difficult to create these as simple beans if you don't want to use the namespace support or even the ConsumerEndpointFactoryBean or some reason. For example, here's a bean definition for an EventDrivenConsumer:

```
<beans:bean id="eventDrivenConsumer"
            class=
                "org.springframework.integration.endpoint.EventDrivenConsumer">
    <beans:constructor-arg ref="someSubscribableChannel"/>
    <beans:constructor-arg ref="someMessageHandler"/>
</beans:bean>
```

We've mentioned several times that the primary responsibility of these consumer endpoints is to *connect* the MessageHandler to an input channel, but what does that mean? It has to do with the lifecycle management of these components. As you can imagine, that's also a different issue for PollingConsumers and EventDrivenConsumers because the former requires an active poller and the latter is passive. We end our under-the-hood exploration with a quick investigation of what lifecycle management means in the context of these two consumer types.

The foundation for managing the lifecycle of any components in Spring Integration is the Spring Framework. The core framework defines a Lifecycle interface:

```
package org.springframework.context;

public interface Lifecycle {
    void start();
```

```

void stop();

boolean isRunning();

}

```

As you can see, methods are available for starting and stopping a component. For an `EventDrivenConsumer`, the start operation consists of calling the `SubscribableChannel`'s `subscribe` method while passing the `MessageHandler`. Likewise, the `unsubscribe` method is called from within the `stop` operation. `PollingConsumers` are more complicated. The start operation of a `PollingConsumer` schedules the poller task with the trigger that's set on that consumer. The trigger implementation may be either a `PeriodicTrigger` (for fixed delay or fixed rate) or a `CronTrigger`. The `stop` operation of a `PollingConsumer` cancels the poller task so that it stops running.

You may wonder who's responsible for calling these lifecycle methods. Stopping is more straightforward than starting, because when a Spring `ApplicationContext` is closing, any lifecycle component whose `isRunning` method returns `true` is stopped automatically. When it comes to starting, Spring 3.0 added an extension called `SmartLifecycle` to the `Lifecycle` interface. `SmartLifecycle` adds a Boolean method, `isAutoStartup()`, to indicate whether startup should be automatic. It also extends the new `Phased` interface with its `getPhase()` method to provide the concept of lifecycle phases so that starting and stopping can be ordered.

Spring Integration consumer endpoints make use of this new interface. By default, they all return `true` from the `isAutoStartup` method. Also, the phase values are such that any `EventDrivenConsumer` is started (subscribed to its input channel) before any `PollingConsumer` is activated. That's important because `PollingConsumers` often produce reply messages that may flow downstream to `EventDrivenConsumers`.

4.4

Summary

This chapter dove deeply into Spring Integration internals that you likely won't need to think about on a regular basis. The higher-level components such as `Transformer`, `Router`, and `Channel Adapter` are the typical focal points. But the information you learned here provides a strong foundation for understanding how those higher-level components work. You now know that Message Endpoints break down into either Polling Consumers or Event-Driven Consumers. You also know that regardless of the type of a given consumer, it delegates to a `MessageHandler`. The lifecycle of an endpoint consists of managing the connection between the `MessageHandler` and the `MessageChannel` where messages are received. The way that connection is managed (either by a poller or a simple subscription) is the distinguishing factor for the types of consumer implementation.

Now that we've explored the low-level details of endpoints in this chapter and of messages and channels in the previous chapter, we're ready to move to a higher level. In the chapters that follow, you'll learn about several components that map to the enterprise integration patterns. These include `Routers`, `Splitters`, `Aggregators`, and a

wide variety of channel adapters. We begin this journey by considering the role of Spring Integration in the larger context of a real-world application.

Business concerns are usually the most critical part of an application, and that fact motivates Spring Integration and the Spring Framework to promote a clean separation between integration and business concerns. The goal of these frameworks is to allow developers to focus on implementing the business functionality in their application's domain without spending excessive time on infrastructure. The next chapter provides several examples in the flight management domain of our sample application.

5

Getting down to business

This chapter covers:

- Achieving separation of concerns
- Domain-driven transformations and message-driven services
- Interceptors and gateways
- Chaining endpoints

In the preceding chapters, you were introduced to the big picture of Spring Integration. You learned about the three key components of the framework: Message, Message Channel, and Message Endpoint. Hopefully, you now have a greater appreciation of how those three components promote loose coupling. Now it's time to focus on a related concept: separation of concerns. In this chapter we explore how Spring Integration enables a clean separation of integration concerns from the core business logic of an application.

Separation of concerns is a well-established principle of object-oriented programming and underlies many approaches to software design. Programming to interfaces enforces a separation of the contract from the implementation. Building layered architectures enforces a separation based on roles such as presentation and persistence. Aspect-oriented programming (AOP) enforces a separation between

the domain-specific code and the cross-cutting requirements. The Spring Framework helps developers achieve a separation from the underlying infrastructure so their applications are portable across different environments without requiring code changes.

Spring Integration follows this trend by supporting a clean separation of integration concerns from the particular business domain of an application. It provides a noninvasive framework for supporting message-driven interactions on top of a business service layer. The book so far has focused on the integration concerns. You explored the construction of messages, different types of message channels, and the behavior of both polling and event-driven consumers. You saw a few examples of how these channels are connected to services. Now you're ready to explore those connections in detail.

By the end of this chapter, you should have a thorough understanding of how to transform data between the representations used by external systems and the core domain model of an application. In many cases, such transformation is applied to the format or structure of the data, but in some cases, the content needs to be enriched or headers need to be added to a message. This chapter provides examples of each.

You'll learn how to connect the business services of an application to the messaging components provided by the Spring Integration framework. This is where the Spring programming model is most apparent. There is a clear emphasis on plain old Java objects (POJOs) and Inversion of Control (IoC). You'll also learn how to chain endpoints together to invoke several different business services within a message-driven flow.

Finally, you'll learn about some AOP capabilities provided by Spring Integration. One such capability is the noninvasive interception of application methods to send messages. Another is the framework's ability to dynamically generate proxy implementations of an application interface so that invoking a method will send a message.

5.1 Domain-driven transformation

When building an object-oriented application, one of the most important steps is to design a data model that accurately reflects the business domain of that application. But when integrating enterprise systems, data may be represented in a wide variety of formats. This section covers the ways such data representations can be transformed to and from the domain-driven object model used in the application.

A common requirement for enterprise integration is to transform inbound data into the format expected by the consumer of that data. Syntactic transformations like converting a string representation of XML into a Document Object Model (DOM) object can be handled by a generic transformer that builds on existing libraries. Many structural transformations, such as those that rely on XSLT or XPath expressions, can even be handled by generic transformers. But dealing with *semantic* transformations is different. Semantic transformations require focusing on content within the business domain of an application. Because the consumer of such content is a business service,

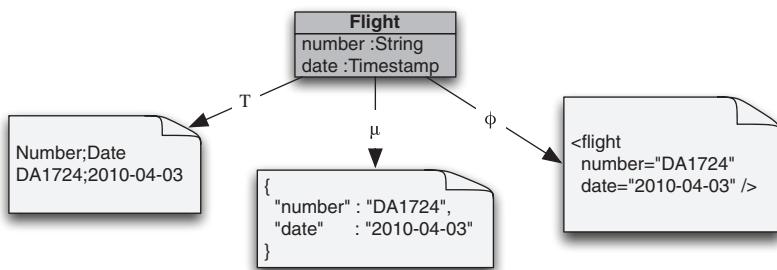


Figure 5.1 Three different representations of the `Flight` object. Separate transformations (`T`, `?`, and `?`) transform a `Flight` domain object into CSV, JavaScript Serialized Object Notation (JSON), and XML representations respectively.

it's likely that the expected format is a domain object rather than something generic like XML or a byte array.

Let's begin with a simple but common case. Objects from a domain model are often represented to the outside world by the simplest means possible. Typically that means the external representation includes the least amount of data sufficient to uniquely identify the object. For example, consider a domain object that maps to a flight. The flight can be uniquely identified by the flight number and its date. That content is simple enough to be represented in text or in a more structured form, such as comma-separated value (CSV) or XML. Figure 5.1 depicts schematically how different transformations can morph the `Flight` domain object into different interoperable formats.

Any of those formats are portable across any number of environments or programming languages. That's a good thing because interoperability is a fundamental concern for enterprise integration. Another benefit of this approach is that it minimizes the amount of data sent over the wire, which means exchanging this data won't have an unnecessary impact on bandwidth or performance.

So far we've talked about transforming *to* a portable format, a process often called *marshalling*. We also need to transform the portable format back to a domain object, which is fittingly called *unmarshalling*. The next section elaborates on both.

5.1.1 Marshalling flight information

Imagine that the ultimate consumer of the flight data is a service that expects a full-fledged Java representation of the flight as defined within the domain model of the application:

```

package sria.business;

import java.util.Date;
import java.util.Map;

public class Flight {
    private String number;
  
```

```

private Date scheduledDeparture;
private String origin;
private String destination;
private Equipment aircraft;
private Crew crew;
private Map<Seat, Passenger> passengers;

// getters, setters omitted
}

```

Not only do you need the ability to transform domain objects into intermediate formats, you also need to transform in the other direction. Typically transformations such as the ones shown in figure 5.1 implement both marshalling and unmarshalling functionality.

The definition of a `Flight` class provides the various properties you'd expect: the flight number, the origin and destination, the scheduled departure and arrival times, the passenger seat assignments, and even a reference to the crew. Together, these objects comprise the *domain* of flight management. Everything you see in the code example is related directly to the business needs of the application. There are no integration concerns mixed in with the domain model, or vice versa. As emphasized in the introduction to this chapter, the main goal is to enforce a clean separation between the business domain and the integration details.

One way that such separation of concerns is often violated in real-world applications is by combining the details of some particular representation or format of the data directly within the class that defines that data. For example, consider the following seemingly innocent addition to the `Flight` class:

```

public class Flight {

    // getters, setters, fields omitted
    public Document toXml() {
        // returns the flight data
        // in the format of an XML document
    }
}

```

That one minor change may seem well justified because, after all, XML is the language of interoperability. Several different users of the flight domain could benefit from the convenience of this addition. Now they can easily create an XML representation of the flight data and send it to another system, write it to a file, and so on.

The problem with that approach is that it's the first step down a slippery slope that leads to tight coupling of the integration concerns within the business domain. Consider that a `fromXml` method is an immediately obvious next step, and then a pair of `toJson` and `fromJson` methods would be justified on the basis of the XML precedent. The next thing you know, your domain model is polluted with so many integration details that it's hard to focus on the business domain.

A much cleaner solution is to isolate the functionality into modular code that is itself easy to test and easy to plug in on an as-needed basis. Some applications might

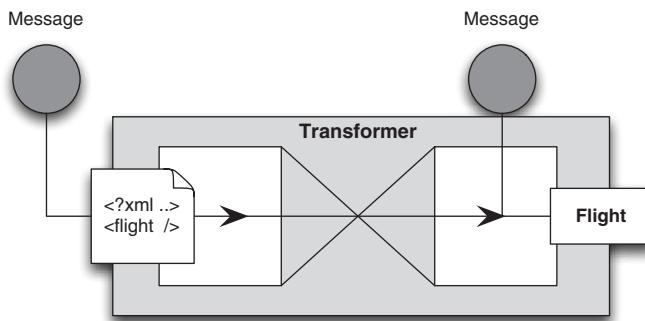


Figure 5.2 A transformer that unmarshals a flight from XML to a domain object

need the XML representation, and others might require JSON (JavaScript Serialized Object Notation). Yet others might need different, more specialized formats, such as a file importer that reads and tokenizes flight information from each line of plain text. By modularizing the XML transformation logic into a component, you can classify it as an integration component providing functionality that is orthogonal to the business domain. Because transformation is a common need in such applications, it's covered by the Enterprise Integration Patterns, specifically the Message Translator. In Spring Integration, the general term for this functionality is a message *transformer*.

Returning to the XML example with this goal in mind, you could isolate that code to a single module with that XML-to-object transformation being its sole function. It might be pragmatic to create a single implementation that handles both marshalling *to* XML and unmarshalling *from* XML, but within any particular message flow, the transformation in one direction may be depicted by a single component. For example, figure 5.2 represents the transformation from XML to our `Flight` object as used in an integration pipeline that begins from some external system, such as a Web Service invocation or Java Message Service (JMS) message listener.

As you can see in figure 5.2, a transformer might be used to unmarshal the object. This way both the sending application and the receiving service can remain oblivious to the other's different representation or interpretation of the domain. This is yet another excellent example of the power of loose coupling.

5.1.2 Using the simplest possible data representation

The XML example captures what is probably the most common approach for representing data in an interoperable way. But even the XML representation might be overkill for particular use cases. The external system that sends flight data to your application might not be concerned with or aware of much of that data, and in such cases, it might be possible to use a far simpler representation of the data involved. As developers, we should always find the simplest solution that effectively and efficiently solves the problem at hand. Besides, when building enterprise integration systems, it's generally a good idea to keep the footprint as small as possible to minimize the bandwidth requirements.

For example, consider a flight delay notification system. All it knows is that a flight with a certain number is going to be delayed by a certain amount of time. In such a system, the data representation might be as simple as this:

SI77+0130

That simple text is sufficient to convey that today's flight SI77 is going to be delayed 1 hour and 30 minutes. On the receiving end, though, the `FlightStatusService` might be expecting a `Flight` object and an actual date representation for the new estimated departure. The interface definition might look like this:

```
package siiia.business;

public interface FlightStatusService {
    void updateStatus(FlightDelayEvent flightDelayEvent);
}
```

The event that's processed by the service might look like this:

```
package siiia.business;

import java.util.Date;

public class FlightDelayEvent {
    private Flight flight;
    private Date estimatedDeparture;

    /* constructor, getters omitted*/
}
```

You may be tempted to add another method to the interface that accepts this simple format from external systems. But, as described earlier with regard to the XML transformation, that type of approach leads quickly to bloated and brittle software by violating the principle of loose coupling and becoming difficult to maintain as a result. An integration framework addresses such concerns so that a clear separation can be maintained between the external representation of the data and the internal domain model. For one thing, external systems may change frequently and new systems may be added that use different formats. What you need is a dedicated component that effectively normalizes the data into the canonical format expected by the business service layer.

```
package siiia.business;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.util.Assert;

import java.util.Calendar;

@MessageEndpoint
public class FlightEventTransformer {
    private final FlightScheduler flightScheduler;

    @Autowired
    public FlightEventTransformer(FlightScheduler flightScheduler) {
        Assert.notNull(flightScheduler,
```

```

        "flightScheduler must not be null");
this.flightScheduler = flightScheduler;
}

public FlightDelayEvent convertToDelayEvent(
    String flightNumberAndDelay) {
String[] splits = flightNumberAndDelay.split("[+]");
Flight flight =
    this.flightScheduler.nextFlightForNumber(splits[0]);
int hours = Integer.parseInt(splits[1].substring(0, 2));
int minutes = Integer.parseInt(splits[1].substring(2));
Calendar cal = Calendar.getInstance();
cal.setTime(flight.getScheduledDeparture());
cal.add(Calendar.HOUR, hours);
cal.add(Calendar.MINUTE, minutes);
return new FlightDelayEvent(flight, cal.getTime());
}
}

```

Figure 5.3 shows how the content enricher collaborates with a repository to convert a simple string to a domain object.

Clearly, this transformer goes beyond syntactical and structural changes. It depends on a `FlightScheduler` service to look up the `Flight` domain object for the given flight number. Assuming you have a well-designed `FlightScheduler` interface to avoid tightly coupling the caller's code to the particular implementation, this transformer would benefit from dependency injection. First, dependency injection makes the transformer code easier to test because you can rely on a stub or mock implementation. Second, it makes it easy to share the same `FlightScheduler` instance that needs to be used by other callers, such as those in the business service layer. You'll see each of these benefits as you continue to develop the example.

5.1.3 **Wiring the components together**

Now that you understand the code, let's turn to the configuration. You may have noticed that the constructor is annotated with `@Autowired`. That annotation was introduced in version 2.5 of the Spring Framework and provides a directive to the container to inject a dependency that matches the expected type of that argument. Assuming your application will have only a single instance that implements the

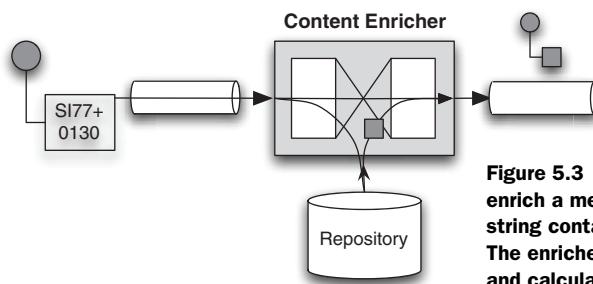


Figure 5.3 A transformer that uses a repository to enrich a message. The incoming message is just a string containing the flight number and the delay. The enricher retrieves the flight from the repository and calculates a date that represents the delay.

FlightScheduler interface at any one time, @Autowired is sufficient for providing that dependency to the transformer.

The @MessageEndpoint annotation at the class level plays a related but slightly different role. It's defined by Spring Integration, but it builds on annotation support of the core Spring Framework. This feature was also added in Spring 2.5 with the introduction of the @Component annotation. @Component's presence in a class allows that class to be automatically recognized by Spring. The result is a bean definition being registered with the container just as if its metadata had been provided in XML.

You may wonder what the @Component annotation has to do with the @MessageEndpoint annotation on the transformer. The answer is that Spring extends the role of @Component to other annotations, which are collectively described as *stereotypes*. The mechanism that drives this is known as *meta-annotations*, and it's a simple technique. Basically, when the @Component annotation is added to another annotation (hence the *meta* designation), components containing that second annotation are recognized by Spring in the same way they would've been if annotated with @Component directly. Rather than annotating all such classes with the generic @Component annotation, a more specific one can be used. The Spring Framework contains even more specific stereotype annotations, such as @Repository and @Service. Specific annotations, like @MessageEndpoint, can better describe the *role* of the annotated component, which is why they're called stereotypes.

In the spirit of test-driven design, let's quickly build an integration test for the transformer. In this case, the *integration* will include an input and output channel and a simple stub implementation of the FlightScheduler. You don't need to worry about the actual production version of the FlightScheduler yet; you just want to make sure the transformer is behaving as expected. The following code depicts a sufficient stub implementation:

```
package siia.business;

import java.util.Date;

public class StubFlightScheduler implements FlightScheduler {

    public Flight nextFlightForNumber(String flightNumber) {
        Flight flight = new Flight(flightNumber);
        flight.setScheduledDeparture(new Date());
        flight.setOrigin("JFK");
        flight.setDestination("LAX");
        return flight;
    }
}
```

Since you're concerned with calculating flight delays, the most important thing to recognize in this stub implementation is that the scheduledDeparture property is set to the current time. In other words, the departure time will be roughly equivalent to the time that the nextFlightForNumber method is invoked.

Now let's create the configuration. You need to define a bean for the preceding stub implementation as well as two channels. You also need to enable the component scanning so the `@MessageEndpoint` annotation will be recognized on the transformer class.

```
<channel id="delayEvents">
    <queue/>
</channel>
<transformer input-channel="flightDelayInput"
             ref="flightEventTransformer"
             method="convertToDelayEvent"
             output-channel="delayEvents"/>

<context:component-scan
    base-package="siaa.business"/>
<beans:bean class="siaa.business.StubFlightScheduler"/>
```

5.1.4 Testing the transformer

Now you can write a JUnit test class that sends a Message to the `flightDelayInput` channel. That message's payload should be a string with the expected format. If all goes well, the `delayEvents` channel should have a message enqueued whose payload is actually a `FlightDelayEvent` instance. The real test is whether that event has an `EstimatedDeparture` property that correctly reflects the delay. The following listing provides the full test class example.

Listing 5.1 Running a test using @Test

```
package siaa.business;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.Message;
import org.springframework.integration.MessageChannel;
import org.springframework.integration.core.PollableChannel;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import java.util.Date;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;

@ContextConfiguration
@RunWith(SpringJUnit4ClassRunner.class)
public class FlightDelayTransformerIntegrationTests {
    @Autowired
    private MessageChannel flightDelayInput;
    @Autowired
    private PollableChannel delayEvents;

    @Test
    public void verifyDelay() {
```

```

        long currentTime = System.currentTimeMillis();
        Message<String> message =
            MessageBuilder.withPayload("SI77+0130").build();
        flightDelayInput.send(message);
        Message<?> transformed = delayEvents.receive(0);
        assertNotNull(transformed);
        Object payload =
            transformed.getPayload();
        assertNotNull(payload);
        assertEquals(FlightDelayEvent.class, payload.getClass());
        Date estimatedDeparture =
            ((FlightDelayEvent) payload).getEstimatedDeparture();
        long secondsFor1hr30min = 90 * 60;
        long delay = estimatedDeparture.getTime() - currentTime;
        assertEquals(secondsFor1hr30min, delay / 1000);
    }
}

```

As you can see, the test confirms that the estimated departure time has been adjusted according to the delay event. The test even provides a few null checks and verifies that the payload type is correct. Those assertions can be helpful because the exact line of the test failure will indicate what the problem was. You don't need to worry too much about the details of this test logic. The main thing to learn from all of this is how to create a simple integration test for a message endpoint that involves sending and receiving messages across message channels. In case you're not familiar with Spring's integration testing support, we provide a quick overview by describing the test class in listing 5.1. If you want to delve into more detail, refer to the Spring Framework reference manual's testing chapter.

The `@Test` annotation is a JUnit 4 annotation and indicates that a particular method provides a test case. This is a nice improvement over the JUnit 3 style, which required methods to begin with `test`. Another nice change in JUnit 4 is that it's no longer necessary to extend a `TestCase` superclass as provided by the framework. Instead, the default test runner strategy knows how to detect the `@Test` annotations. Luckily, it's still possible to extend the testing framework by providing a customized implementation of the test runner. This is how Spring provides its integration testing support for a JUnit 4 environment. In listing 5.2, you can see that Spring's test runner class is provided within JUnit's `@RunWith` annotation.

The other class-level annotation, `@ContextConfiguration`, is defined by Spring's integration test framework, which directs Spring's test runner to locate a configuration file with the same name as the current test class but with a `-context.xml` suffix. For example, the configuration file to be used in conjunction with the test class in listing 5.2 would be named `FlightDelayTransformerIntegrationTests-context.xml`, which should be located in the same package as the test class. That pretty much covers the testing support for this example. The `@Autowired` annotation is standard Spring, and the field name is used to drive resolution based on bean names because a resolution based on type alone would lead to ambiguities in this case.

The previous sections explained how to marshal and unmarshal objects to and from different formats. You also learned how to test these applications using Spring's test context support. The requirement to convert a simple unique identifier to a fully populated instance of a domain object is a common one, but message transformers can address many other common requirements. Let's take a quick look at a few other types of transformation that rely closely on the code from the particular domain model of an application.

5.1.5 Content enricher

Another common requirement is to add content to an existing domain object by invoking a business service. For example, for the flight booking example, you might need to add the email address of a given passenger. Imagine that you could uniquely identify the passenger on the basis of a frequent flyer program number and also look up the email address from the frequent flyer information service. An implementation of this logic in a transformer could be considered a content enricher. The email address is the enriched content being added to the payload of a message that didn't yet include that information.

In fact, if a user is enrolled in the frequent flyer program, you may be able to add much more information beyond the email address. The user's profile may contain preferences, such as meal type, and whether the passenger prefers aisle or window seating. The information would probably also include mileage credits and status.

The main point is that a `Passenger` instance may or may not contain all of this data. The airline doesn't require enrollment in the frequent flyer program, so a minimal `Passenger` instance may contain only the person's name and some form of official identification, such as a passport number, driver's license number, or tax ID. The content enricher in the following listing adds the extra information if available but otherwise passes on unaffected `Passenger` instances.

Listing 5.2 Content enricher for identifying passengers

```
package sria.business;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.integration.annotation.MessageEndpoint;

@MessageEndpoint
public class PassengerProfileEnricher {

    private final FrequentFlyerService frequentFlyerService;

    @Autowired
    public PassengerProfileEnricher(FrequentFlyerService ffService) {
        this.frequentFlyerService = ffService;
    }

    public Passenger addProfileIfAvailable(Passenger passenger) {
        String ffNumber = passenger.getFrequentFlyerNumber();
        if (ffNumber != null) {
            Profile profile =

```

```

        this.frequentFlyerService.lookupProfile(ffNumber);
        if (profile != null) {
            passenger.addProfile(profile);
            return passenger;
        }
    }
    return passenger;
}
}

```

The code in listing 5.3 is fairly straightforward. As with the previous examples, it'd also be easy to unit test without any reliance on the integration framework. The only dependencies are on the domain model objects and services, so test code wouldn't even need to include messages, channels, or endpoints. As time goes on, having such cohesive, dedicated units of functionality proves beneficial to the maintenance effort as well. If something else needs to be added to the passenger profile, it'll be clear where the corresponding code change needs to occur. Likewise, if a new service for profile information is released in the future, it'll be simple to change this single point of access to such information. Once again, this reveals the true value of a loosely coupled solution. The rest of the system would likely adapt to such a change without any rippling side effects.

5.1.6 Header enricher

Business services deal with the payload of a message, and often that payload can be considered a document. Channel adapters and messaging gateways that connect external systems, by contrast, might be interested in information carried in a message header.

As an example, consider an adapter that sends flight delay notifications to passengers who sign up for such notifications. Continuing with the example from the previous section, let's assume the email addresses are available on the payloads, which are instances of the Passenger domain object, because they've been added by the content enricher shown in listing 5.3. Basically, passengers enrolled in the frequent flyer program would have the option of providing an email address as part of their profile as well as enabling notifications for a flight delay.

We won't go into detail on the adapter that sends the emails because we cover email integration in chapter 10. For now, it's sufficient to know that the email-sending adapter expects the target email address to be provided in a header. The email adapter could advertise that the header name should be, for instance, `mail_to`. It's then the responsibility of an upstream component to populate that header with the correct value.

Of course, you could implement this header-enriching transformer in code. All you need to do is receive the payload, which is a `Passenger` instance in this case, then grab the email address, if available, so it can be added to the map of headers. The code might look like the following listing.

Listing 5.3 Header enricher to associate passengers with email addresses

```

package sisia.business;

import org.springframework.integration.Message;
import org.springframework.integration.annotation.Headers;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.integration.mail.MailHeaders;
import org.springframework.integration.support.MessageBuilder;

import java.util.Map;

@MessageEndpoint
public class EmailHeaderEnricher {

    public Message<Passenger> populateEmailHeader(Passenger passenger,
                                                    @Headers Map<String, Object> headers) {

        MessageBuilder<Passenger> responseBuilder =
            MessageBuilder.withPayload(passenger).copyHeaders(headers);
        Profile profile = passenger.getProfile();
        if (profile != null) {
            String emailAddress = profile.getEmailAddress();
            if (emailAddress != null) {
                responseBuilder.setHeader(MailHeaders.TO, emailAddress);
            }
        }
        return responseBuilder.build();
    }
}

```

Again, don't worry about the details for the downstream mail sender; we cover it in chapter 10. The intention of the code in listing 5.4 should be clear nonetheless. If the passenger represented by the current message's payload has an email address on file, it is added to the headers. That way, some generic email-sending adapter that will eventually receive this message doesn't need to know how to extract the email address from every possible type of message payload it may receive. In other words, the adapter specifies a contract: "I will send a mail IF the MailHeaders.TO header contains a valid email address." If the fulfillment of that contract is the responsibility of a separate component, the mail adapter is highly reusable. For example, the same adapter could be used for sending email to banking customers in an application designed for that domain.

You might think this is a lot of code to write merely for the benefit of a reusable mail-sending adapter. That's a valid point, and it highlights one of the key trade-offs involved in enterprise integration and messaging application design. We discussed the benefits of loose coupling at length, but what you see here is the initial indication that loose coupling may have some negative consequences. As with most principles, if taken to extremes, the costs may outweigh the benefits. If every single line of code were modularized into its own class and implementing its own interface, the result would be the ultimate loosely coupled system. But testing and maintaining such a system would be prohibitively difficult. It would probably be much more challenging than testing a tightly coupled implementation where all of the code is in a single class.

With that perspective, consider the code in listing 5.4 once more. And to add fuel to the fire, consider the verbosity of that code in relation to the relatively straightforward goal of the implementation. These days, there seems to be a growing recognition of the benefits of scripting languages and specifically dynamic or functional variants. Despite what some may claim, it doesn't necessarily mean that Java should be avoided altogether. On the contrary, there seems to be a trend toward using multiple languages together in an application, and the Java Virtual Machine (JVM) provides a great environment for exactly that. Deciding which language to use for a particular task should be based on the nature of that task. It so happens that many of these integration tasks are a good fit for dynamic languages. Spring Integration supports Groovy and other scripting languages alongside Java for the implementation of routers, transformers, splitters, filters, and other components.

For now, let's consider a related but even simpler option: Spring's own expression language. As of Spring 3.0 (and hence Spring Integration 2.0), expression language support is provided in the core framework. It's a perfect match for situations like this header enricher. Consider the following example:

```
<mail:header-enricher>
    <mail:to expression="payload.profile?.emailAddress" />
</mail:header-enricher>
```

Believe it or not, this code accomplishes the same thing as the earlier Java code. The special `?.` operator is considered a null-safe property accessor (if the value is in fact null, it won't attempt to access it). Knowing that, you should be able to compare the two implementations and understand exactly how the expression language version works. At this point, you might be thinking, "Wow! I'll just use these expressions for everything." This is a common first reaction, but using the expression language is not always the most sensible thing to do.

Because expressions are stored in strings, and in this case inside the XML configuration, it's all too easy to "program" in XML. Logic stored in this manner is much harder to test and refactor. You should therefore consider using the expression language only for simple, to-the-point expressions. Don't dig deep: `payload.profile?.address?.country?.countryCode` is as bad an idea as it looks. If you're a Java programmer, you're used to type safety and relying on good tools and the compiler to warn you of mistakes. These warnings won't be raised when you use the expression language.

Now that you can consider yourself armed (and warned) with knowledge of various types of transformers, including the expression language support, it's time to really get down to business. Although transformers may be viewed as an extension to the messaging infrastructure you are building, the main goal of an integration application is most likely to invoke the *real* business logic. The next section dives into the details of how to reach that goal with Spring Integration.

5.2 Message-driven services

In the previous section, you converted data from a simple text representation to a `FlightDelayEvent` object that's part of the domain model. You briefly reviewed the `FlightStatusService` interface that would expect such an object. Now let's look at the implementation of that service and see how to invoke it from the integration layer. The component responsible for that invocation is a service activator.

5.2.1 The Service Activator pattern

Recall that you want to invoke the `updateStatus` method and that it expects a `FlightDelayEvent` object. The return value from that method invocation would be a `FlightStatus` object. Therefore, the service activator configuration would look something like this:

```
<service-activator input-channel="flightDelays"
    output-channel="statusUpdates"
    ref="flightStatusService" method="updateStatus" />

<beans:bean id="flightStatusService"
    class="silia.business.SimpleFlightStatusService"/>
```

If you've ever worked with the Spring Framework's JMS support, and specifically with the message-driven POJO feature, this configuration should look familiar. One of the main motivations behind Spring Integration was to provide a more generic model for building message-driven systems with the Spring JMS support serving as a precedent. The feature to be supported is still message-driven POJOs. In this case, the POJO is the Spring-managed `flightStatusService` object. But instead of a JMS message, it's a simple Spring Integration message being mapped to the domain object expected by that service. The source of that message could be any code that sends to that channel or any adapter that's connected to that channel.

You may also have noticed that the service defines a return value, so a corresponding `output-channel` is declared on the service activator configuration element. That output channel might be connected to some external system by a channel adapter, or it might refer to the input channel of another component, such as a router, a splitter, or another service activator.

It's common for the output of one service activator to refer to the input channel of another service activator, and that approach can be used to connect multiple services into a pipeline or *chain*. Figure 5.4 shows three services that are connected in such a chain.

As we discuss later in this chapter, chaining services like this is such a common technique that Spring Integration provides an even simpler way to define such chains without having to explicitly define each channel.

5.2.2 The Return Address pattern

The chained services model works well for *linear* use cases where a sender on one end triggers the message flow but doesn't expect a return value. In such cases, there is an

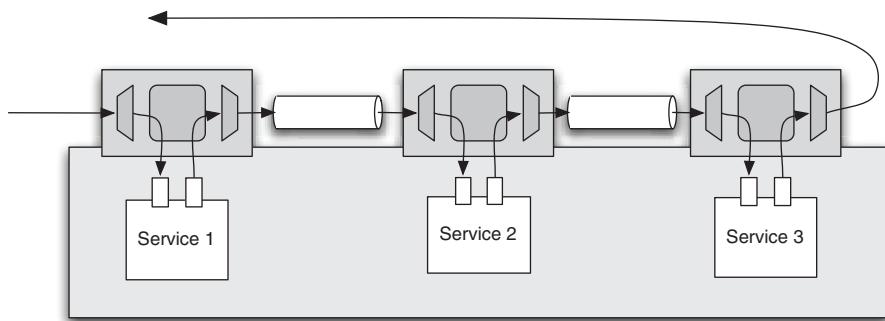


Figure 5.4 Pipeline configuration with three services. A message is deconstructed and reconstructed at each service activator. The payload is fed into the service, and the return value is used to construct a new message. The headers of the message remain the same, so the overall context is preserved throughout the pipeline.

ultimate receiver at the other end of the chain. Either the final service to be invoked doesn't return a value or the chain ends with a channel adapter such as a file writer. In other use cases, the original sender *does* expect a reply. In those cases, a header can be provided on the original message with a reference to a reply channel. In enterprise integration pattern (EIP) terminology, that header represents the *return address*. Here's an example of a sender that provides such a header and then waits for the response:

```
RendezvousChannel replyChannel = new RendezvousChannel();
Message requestMessage = MessageBuilder.withPayload(someObject)
    .setReplyChannel(replyChannel).build();
requestChannel.send(requestMessage);
Message replyMessage = replyChannel.receive();
```

Before going any further with this example, we should point out that you'll probably never write code like this because Spring Integration provides support for the reply channel configuration behind the scenes. You'll see an example of this in the upcoming section on the Messaging Gateway support. For now, we want to consider the implications of the reply channel header for a service activator. If the `requestChannel` in the preceding snippet were connected to the `flightStatusService` object, then the configuration would be similar to the previous service activator example. The key difference would be the absence of an `output-channel`. By leaving that out, you force the service activator to store the service's return value in the payload of a message that it then sends to the channel referenced by the `replyChannel` header. The configuration would look like this:

```
<service-activator input-channel="flightDelays"
    ref="flightStatusService" method="updateStatus"/>

<beans:bean id="flightStatusService"
    class="siiia.business.SimpleFlightStatusService"/>
```

The configuration is almost identical to that in the previous version, and more important, no changes are required to the service code. Switching between a chained-services and a request/reply interaction model is as simple as including or omitting the output channel from the configuration, respectively. Keep in mind that if a service does return a value, and neither the output channel has been configured nor the reply channel header provided, an exception would be thrown at runtime because the service activator component wouldn't know where to send the return value. If a service does return a value, yet you intentionally want to ignore it, you can provide null-Channel as the output-channel attribute's value.

5.3 Message publishing interceptors

Thus far in this chapter, we explored the components that consume messages or, more commonly, just the content of message payloads. In the context of a business service, these consumers may also return values that are then published as response messages. The previous section covered the details of how these response messages can be sent to either the endpoint's output channel or a return address header that's provided with the incoming message for that purpose. But integrating a messaging system with an application's business layer shouldn't be limited to components that react to messages. It's useful to have business services in the application publish messages based on certain events that occur *within* the business services themselves. A typical example is sending a message each time a particular service is invoked. The payload of that message might be the return value of the invocation, it might be an argument that was passed with the invocation, or it might be some combination. In any case, Spring Integration provides support for this by applying Spring AOP. A proxy can be generated to intercept method invocations on services that have no direct awareness of the messaging system. The proxy handles the responsibility of creating the message from the invocation context and publishing that message to a channel. As far as the application developer is concerned, this functionality is configuration driven.

Let's consider an example. Imagine that your system needs to perform some auditing every time a flight's status is updated. Perhaps you're responsible for gathering these statistics to provide data about the percentage of flights that actually depart on schedule. You could write some code that publishes the message directly and then add that code to your `FlightStatusService`, as in the following listing.

Listing 5.4 Gathering statistics when status is updated

```
package siiia.business;

import org.springframework.integration.Message;
import org.springframework.integration.MessageChannel;

public class SimpleFlightStatusService implements FlightStatusService {

    private MessageChannel statisticsChannel;
```

```

public void setStatisticsChannel(MessageChannel statisticsChannel) {
    this.statisticsChannel = statisticsChannel;
}

public void updateStatus(FlightDelayEvent flightDelayEvent) {
    // update the flight information in the database
    Assert.notNull(statisticsChannel,
        "no statistics channel is available");
    Message<FlightDelayEvent> message =
        MessageBuilder.withPayload(flightDelayEvent).build();
    statisticsChannel.send(message);
}
}

```

By now, if we've accomplished our goals in the book so far, you're probably having a Pavlovian response already. Clearly, this is another violation of the principles of loose coupling and separation of concerns. With listing 5.5, even testing the core business functionality of the status update service would require awareness of the message publishing behavior as well.

Fortunately, Spring Integration provides a cleaner way to handle such requirements. As with any application of AOP, the goal is to modularize the functionality so you don't mix cross-cutting concerns, such as messaging, with the core business logic. The key is to apply the noninvasive technique of interception. Also, as with many of the other Spring solutions that rely on AOP, such as transactions, this behavior can be enabled by using a simple annotation.

```

public class SimpleFlightStatusService implements FlightStatusService {

    @Publisher(channel="statisticsChannel")
    public void updateStatus(@Payload FlightDelayEvent flightDelayEvent)
    {
        // update the flight information in the database... and THAT'S ALL!
    }
}

```

As you can see, this is much better because the code is focused only on the business logic. There's no need to worry about the cross-cutting messaging concerns when testing or maintaining the code. Likewise, the message publishing code can be tested independently of the business logic. In other words, it's a cleaner solution all around because it separates the integration and business concerns.

5.4 Domain-driven Messaging Gateways

The noninvasive interceptor-based approach described in the previous section is a powerful yet simple way to send messages based on actions that occur within the business service layer. But interception is only really useful for reactive use cases in which the publication of a message is considered a *by-product* of some other primary action. There are other equally valid use cases in which publishing a message is itself the primary action, and for those cases, a more proactive technique fits well.

Such use cases correspond to the traditional event-driven programming model. Suppose you want to send a general notification that a flight status has been updated. Several parties might be interested in that event. One way to accomplish this is to publish the event to each interested party. The following listing demonstrates a class that accomplishes this in an intuitive but less than ideal way.

Listing 5.5 Publishing event notifications

```
package sisia.business;

public class FlightStatusNotificationPublisher {

    private MailSender mailSender;
    private JmsTemplate jmsTemplate;

    public FlightStatusNotificationPublisher(MailSender mailSender,
                                              JmsTemplate jmsTemplate) {
        this.mailSender = mailSender;
        this.jmsTemplate = jmsTemplate;
    }

    public void publishNotification(FlightStatusEvent event) {
        SimpleMailMessage mailMessage = null; // TODO: create mail Message
        this.mailSender.send(mailMessage);
        this.jmsTemplate.convertAndSend(event);
    }
}
```

Why is this example less than ideal? There are a number of reasons. First, consider what you'd need to do if you encountered a new requirement to add another destination for these events, such as the local file system. You'd need to refactor the code to handle that. The revised implementation might look something like the following listing.

Listing 5.6 Event notifications with added requirements

```
public class FlightStatusNotificationPublisher {

    private MailSender mailSender;
    private JmsTemplate jmsTemplate;
    private File directory;

    public FlightStatusNotificationPublisher(MailSender mailSender,
                                              JmsTemplate jmsTemplate, File directory
    ) {
        this.mailSender = mailSender;
        this.jmsTemplate = jmsTemplate;
        this.directory = directory;
    }

    public void publishNotification(FlightStatusEvent event) {
        SimpleMailMessage mailMessage = null; // TODO: create mail Message
        this.mailSender.send(mailMessage);
        this.jmsTemplate.convertAndSend(event);
        this.writeFile(event);
    }
}
```

```

private void writeFile(FlightStatusEvent event) {
    // convert the event to a String
    // generate a name for the target File
    // write the content to the target File within the directory
}
}

```

Listing 5.7 demonstrates the increased complexity as you try to meet the demands of too many requirements within a single component. Imagine trying to test the publishNotification method in this code. With three different notification targets being handled by a single method invocation, it would be difficult to completely isolate each one in a test. That's a pretty good indicator that you're violating the separation of concerns principle.

We left out the full implementation details of the writeFile method but purposefully provided three comments to describe what such an implementation entails. As you'll see in chapter 11, each step can even be addressed by separate components or strategies when using Spring Integration's file adapters. As for mail sending and JMS publishing, the preceding examples assume use of the corresponding support in Spring. In both cases, there is still a dependency on the mailSender and jmsTemplate objects in the code. As you'll learn later in the book, Spring Integration builds on such support classes provided by the underlying Spring Framework, but with its more generic messaging model, it allows for an even greater degree of separation of concerns. The goal is to remove all awareness of such infrastructure from code in favor of a declarative, configuration-driven model.

For now, don't worry about the details of these channel adapters. As we said, you'll have plenty of exposure in upcoming chapters. The point here is to appreciate the configuration-driven nature and to see how the Messaging Gateway pattern can be supported by a simple proxy. Consider the following configuration:

```

<gateway request-channel="flightStatusNotifications"
         interface-name="siia.business.FlightStatusNotificationPublisher" />
<publish-subscribe-channel id="flightStatusNotifications" />
<mail:outbound-channel-adapter channel="flightStatusNotifications"
                                 mail-sender="mailSender" />
<jms:outbound-channel-adapter channel="flightStatusNotifications"
                               destination="flightStatusQueue" />
<file:file-to-string-transformer
          input-channel="flightStatusNotifications"
          output-channel="flightStatusFilesOut" />
<file:outbound-channel-adapter channel="flightStatusFilesOut"
                               directory="/siia/flightStatus" />

```

Even without knowing any details about the mail, JMS, and file adapters, this configuration should be self-explanatory. The main idea is that multiple adapters are subscribed to a channel, and as a result, the producer of flight status notifications doesn't need to know anything about the individual subscribers. The producer only needs to know about the single channel. Likewise, this configuration-driven approach is

extensible. If you need to add or remove subscribers, it's a matter of inserting or deleting the corresponding element. As far as transformation is concerned, it's also treated as a separate concern. As you can see, the file-to-string-transformer element precedes the outbound file adapter. If you need to transform to a byte array instead, you swap out that element for a file-to-bytes-transformer. Alternatively, if you have custom transformation requirements, it's just as easy to provide a reference to your own transformer implementation instead. You'll learn more about each of these adapters and their associated transformers in upcoming chapters.

The one element we do want to discuss in some detail here is the gateway. As you can see, it declares the fully qualified name of an interface. The result of that element is a generated proxy. If you've ever worked with Spring's support for remoting via Remote Method Invocation (RMI) or Spring's own HTTP invoker mechanism, this concept will be familiar. The element triggers the registration of a Spring FactoryBean that produces a dynamic implementation of the declared interface. In this case, the proxy is backed by the supporting code in Spring Integration that maps the arguments to a message and then publishes that message to the referenced channel.

Earlier we discussed the testing aspects of the nonideal implementation as an indicator that the code was fragile. With that implementation, it was difficult to isolate the various notification subscribers' code for testing purposes. Now the story is much different. For one thing, relying on the framework reduces considerably the amount of inhouse code to be tested. In fact, the only code from the previous example that isn't part of the framework is the interface. This means that all responsibility for testing the internal functionality of the subscription side belongs to the Spring Integration development team. You'd likely want to have some level of integration testing to verify that the mail sender, JMS queue, and file system directory are configured properly. In terms of *unit* testing, though, the focus can shift to the publishing side. Because that's now reduced to the invocation of a method on an interface, it should be easy to build a battery of tests using traditional mock or stub techniques.

5.5 Chaining endpoints

In many cases, messages flow linearly through the system from endpoint to endpoint, separated by direct channels. In these cases, configuration can be unnecessarily verbose if users are required to create channels between each of the endpoints explicitly. Spring Integration allows you to create those channels implicitly using a `<chain>` element (figure 5.5).

As discussed earlier, synchronous channels are used by default to preserve transactional boundaries. In a

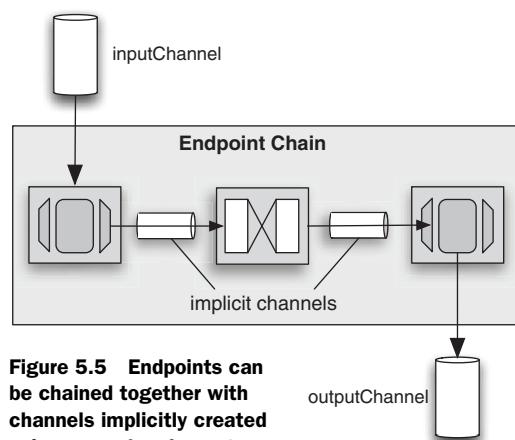


Figure 5.5 Endpoints can be chained together with channels implicitly created using a `chain` element.

chain, the synchronous channels needn't be specified, but there's no way to override the type of channel used. The argument to do things this way is to improve the readability of the configuration files by reducing repetition but not sacrificing the option of a more explicit configuration where needed.

You've seen several examples in this chapter of a single component performing one focused, well-defined task. This is a good thing from the perspective of modular design, but even loose coupling and separation of concerns can be taken to extremes. Consider the following configuration based on this chapter's examples:

```
<channel id="passengers" />
<transformer input-channel="passengers"
             output-channel="passengersWithProfile"
             ref="frequentFlyerService" method="configureProfile" />
<channel id="passengersWithProfile" />
<mail:header-enricher input-channel="passengersWithProfile"
                      output-channel="flightDelays">
    <mail:to expression="payload.emailAddress" />
</mail:header-enricher>
<channel="flightDelays" />
<mail:outbound-channel-adapter channel="flightDelays"
                               mail-sender="mailSender" />
```

This configuration has a lot of unnecessary noise. It has many channels that are used only once and many endpoints configured to connect to those channels. Even if you rely on Spring Integration's creation of default channels for any single component's input-channel attribute, or for the id attribute of a channel adapter, the configuration is still noisy:

```
<transformer input-channel="passengers"
             output-channel="passengersWithProfile"
             ref="frequentFlyerService" method="configureProfile" />
<mail:header-enricher input-channel="passengersWithProfile"
                      output-channel="flightDelays">
    <mail:to expression="payload.emailAddress" />
</mail:header-enricher>
<mail:outbound-channel-adapter id="flightDelays"
                               mail-sender="mailSender" />
```

This configuration allows you to remove the channel elements, but it might be even harder to follow when reading the configuration directly. Now you have to look at the various input-channel values that match with output-channel values. If the endpoints were listed out of order, it would be confusing. The chain element offers a cleaner solution. Whenever default channels are being referenced by only one subscribing endpoint, and the flow of messages across multiple endpoints is a simple linear arrangement, consider the simplification provided by the chain element (see figure 5.6).

```
<chain input-channel="passengers">
    <transformer ref="frequentFlyerService"
                 method="configureProfile"/>
    <mail:header-enricher>
        <mail:to expression="payload.emailAddress"/>
    </mail:header-enricher>
    <mail:outbound-channel-adapter mail-sender="mailSender"/>
</chain>
```

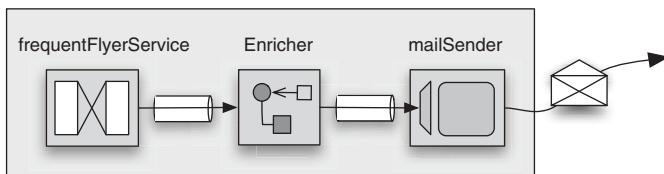


Figure 5.5 The `frequentFlyerService` transformer, the header enricher, and the `mailSender` chained together

When creating a chain, there are a few things to consider regarding the position of endpoints. If an endpoint isn't the last entry in a chain, then it must accept an `output-channel`. Those endpoints that would be playing the role of a terminating endpoint must be placed in the last position. The outbound mail adapter is an example. It couldn't be placed in the middle of the chain because any endpoint added after it would never be invoked.

Likewise, if adding a router to a chain, it must be in the final position. Considering the role of a router is to determine the next channel to which a message should be sent, it wouldn't fit in any other position within a chain because all endpoints before the last one essentially have a fixed output channel.

5.6 Summary

In this chapter, we explored a number of ways Spring Integration may interact with components that are part of a business domain. Typically such components have no awareness of the Spring Integration API. On the contrary, the service objects can be POJOs because Spring Integration provides the necessary adapters to connect those POJOs to message channels. Those adapters also assume the responsibility of mapping between messages and domain objects. This is why we describe it as a *noninvasive* programming model: the methods to be adapted on the POJOs don't need to operate with messages. Instead, those methods can expect domain objects as method parameters and can likewise return domain objects when invoked.

One of the benefits of this model is that pre-existing business services may be connected to the messaging system with little effort. As a result, Spring Integration is easy to adopt incrementally. Even more important, whether connecting to pre-existing services or implementing a completely greenfield application, this model makes it easy to enforce a clean separation of concerns between the business logic and the integration components. The messaging system forms a layer above the business services, and

those services are easy to implement, test, and maintain without having to take any messaging concerns into account.

The message-driven interactions with a business domain featured in this chapter are all relatively simple. Even when multiple steps are necessary, such as transforming content before invoking a service, they're in the form of linear pipelines. You learned how Spring Integration provides support for such a pipeline with the XML configuration of a handler *chain*. But sometimes the interaction with the business layer isn't so straightforward. One of the most common requirements is to add some decision logic to determine where a particular message should go. That decision may be based on some content within the message's payload, or it may be based on a header value that's been added to the message. In chapter 6, we explore Spring Integration's support for addressing such requirements.

The next chapter covers routing and filtering messages. The decoupling provided by Spring Integration can be used to separate business concerns about delivering work to certain services from the processing that happens in the services.



Go beyond sequential processing: routing and filtering

This chapter covers:

- Filtering messages
- Routing messages to one or many consumers
- APIs for filtering and routing

Earlier chapters showed you how to use Spring Integration to create an application from a group of processing units. You learned to choose the format of the messages that are exchanged between components, and to define channels to which messages are published and thus propagated through endpoints to the message handlers that process them.

But there's more to Spring Integration than this basic model. Sure, building an application like this is a simple and efficient (and therefore great) way to reduce coupling. You can also control the responsiveness of the system by adopting an asynchronous model. This works well even for simple, sequential processing models. But this chapter introduces another form of control that goes beyond the

conveyor-belt model of sequential processing: it explains how to selectively process messages and define alternative routes within the system.

First, you'll see how you can limit the scope of what your components will handle by using filters that allow only certain messages to pass through to be processed. Then we'll discuss the benefits of selective processing. When your application provides alternative (or complementary) processing options for your messages, you can use routers to let the system choose the actual destination of a message, thus freeing message producers from the decision process. This technique allows more flexibility in the system because your components become more reusable.

6.1 Do you want to get this message?

The main role of an endpoint is to consume messages from a channel and process them by invoking the associated `MessageHandler`. This means the service is invoked for every message that arrives on the inbound channel.

It's possible that a given consumer isn't interested in all the messages it receives. For example, in a publish-subscribe scenario, different consumers may have different interests in the incoming messages even if they're all potential recipients.

To enable the selective consumption of messages, you can use a special type of message handler, a *message filter*. As you can see in figure 6.1, the filter is a message handler that evaluates messages without transforming them and publishes back to an output channel only the ones that satisfy a given rule.

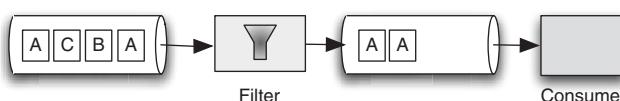


Figure 6.1 A message filter evaluates incoming messages. Only A's are published to the next channel, whereas B's and C's are discarded.

Of course, the decision whether a certain component can process certain types of messages can be made by the component itself. For example, the plain old Java object (POJO) implementation of Service Activator can decide whether messages are valid. Embedding this decision into the component works well if the decision is based on a broad general principle that's applicable everywhere that component is used. Filters are useful wherever the filtering condition is based on an application-specific requirement that's not inherently linked to the Java implementation of the business service.

6.1.1 Filtering out messages

The application can receive cancellation requests through a gateway. The requests are forwarded to a cancellation processing service, and further to a notification service, which sends out confirmation e-mails. Cancellations for different types of reservations may need to be processed differently because the conditions may be different (refund policies, advance notice requirements, and so on).

The example cancellations processor knows only how to process Gold cancellations, so you can discard everything else, as in the following sample:

```

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

    <channel id="input"/>
    <channel id="validated"/>
    <channel id="confirmed">
        <queue/>
    </channel>

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

    <gateway id="cancellationsGateway"
        service-interface="sii.booking.integration.cancellation.
            CancellationsGateway"
        default-request-channel="input"/>

    <filter id="cancellationsFilter"
        input-channel="input"
        ref="cancellationsFilterBean"
        method="accept"
        discard-channel="rejected"
        output-channel="validated"/>

    <beans:bean id="cancellationsFilterBean"-->
        class="sii.booking.integration.cancellation.
            CancellationRequestFilter">
        <beans:property name="pattern" value="GOLD[A-Z0-9]{6}+"/>
    </beans:bean>

    <service-activator
        id="goldCancellationsProcessor"
        input-channel="inbound"
        ref="cancellationsService"
        method="cancel" output-channel="confirmed"/>

    <beans:bean id="cancellationsService"
        class="sii.booking.domain.cancellation.
            StubCancellationsService"/>
</beans:beans>
```

Note the `<filter/>` element used for inserting the filter between the gateway and the service activator that processes the cancellation requests. From a syntax perspective, its definition isn't very different from that of other message handlers discussed so far, such as the transformer and the service activator. It has an input channel for handling incoming messages, an output channel for forwarding messages, and delegates to a Spring bean for the actual processing logic. But it's semantically different. Messages don't suffer any transformation when they pass through this component: the same message instance that arrives on the `inboundCancellations` channel is forwarded to the `validatedCancellations` channel if it passes the filtering test.

In our application, a cancellation request that can be processed by the service must contain a reservation code that corresponds to a Gold reservation. Of course, this tells nothing about whether a reservation with that code exists, and we won't try to do all the validation at this point (many things can be wrong with the request itself, and dealing with such errors is the responsibility of the cancellation service). But, at a minimum, you know that reservation codes have to conform to a certain standard pattern, and you can do a quick test for that. Here's how the implementation of the filtering class looks:

```
package sii.booking.integration.cancellation;

import sii.booking.domain.cancellation.CancellationRequest;
import java.util.regex.Pattern;

public class CancellationRequestFilter {
    private Pattern pattern;

    public void setPattern(Pattern pattern) {
        this.pattern = pattern;
    }

    public boolean accept(CancellationRequest cancellationRequest) {
        String code = cancellationRequest.getReservationCode();
        return code != null && pattern.matcher(code).matches();
    }
}
```

As with the service activator and the transformer, you can implement the logic in a POJO (and we strongly recommend you do so). The filtering logic consists of a method that takes as argument the payload of a message (as in the previous example), one or more message headers (using the @Header/@Headers annotation), or even a full-fledged message, and returns a boolean indicating whether the message will pass through.

WHERE DO REJECTED MESSAGES GO?

In the simplest case, which is also the default, messages are just discarded (or, for a UNIX-based analogy, /dev/null). If you don't want them to be discarded, you have two other options:

- You can specify a channel for the discarded messages. In this case, rejection is more a form of redirection, allowing the application to handle them further as regular messages. From the point of view of the framework, they're still messages and therefore subject to any handling a message on a channel can undergo.
- You can instruct the framework to throw an exception whenever a message is rejected.

The framework allows both options to be enabled at the same time, but from a practical perspective, they're mutually exclusive. Nevertheless, when both options are active, the message is sent on the discarded messages channel before the exception is actually thrown (an important detail when a synchronous channel strategy is in place).

To illustrate, here's a more elaborate variant of the previous example, where rejected cancellation requests are redirected on a specific channel and from there are forwarded to an outbound notification system. Assuming the cancellation request contains enough information about the requester itself, so that a reply message can be sent, you can implement the filter as:

```
<filter id="cancellationsFilter" input-channel="inboundCancellations"
    discard-channel="rejectedCancellations"
    ref="cancellationsFilterBean" method="accept"
    output-channel="validatedCancellations"/>
<bean id="cancellationsFilterBean"
    class="silia.booking.integration
        .cancellation.CancellationRequestFilter">
    <property name="pattern" value="[A-Z0-9]{6}+(">
</bean>
<!-- other definitions -->
<mail:header-enricher header-name="TO"
    expression="payload.requestor?.emailAddress">
<mail:outbound-channel-adapter channel="validatedCancellations"/>
```

Or if you want to throw an exception instead, you can write:

```
<filter id="cancellationsFilter" input-channel="inboundCancellations"
    throw-exception-on-rejection="true" ref="cancellationsFilter"
    output-channel="validatedCancellations"/>
```

USING EXPRESSIONS

As you've seen, it's simple enough to implement the filtering logic as a POJO. But in a lot of common cases, you don't even need to do that. If all the information is to be found in the message class itself, and all you need is to write a logical expression that's computed against the payload or the header values, you can use the Spring 3.0 Expression Language (SpEL) directly. Instead of defining a distinct CancellationRequestFilter, you can get the same result by using the following filter definition:

```
<filter id="cancellationsFilter" input-channel="inboundCancellations"
    discard-channel="rejectedCancellations"
    expression="payload?.code matches 'GOLD[A-Z0-9]{6}+'"
    output-channel="validatedCancellations"/>
```

This way, there's no need for a distinct bean to implement the decision logic. The advantage of using this filter definition is that the filtering logic can quickly be viewed in the context of the message flow. The disadvantage of using SpEL expressions directly is that they're harder to test in isolation from the filter itself, so you should take care to call out to the proper abstractions if the logic gets complicated.

Having two options on hand, when should you create a distinct implementation, and when should you use an inline expression? An inline expression is simple enough, and could be externalized easily (using, for example, a PropertyPlaceholderConfigurer). But it's not flexible or reusable across the application. Our recommendation for making a decision in this case is to use expressions whenever the condition is based on the attributes of the message itself. In more complicated cases,

when the decision involves a sophisticated algorithm, or the sender must consult with other collaborating components, you may want to create a standalone implementation and take advantage of Spring's dependency injection capabilities.

If you really want to have your cake and eat it too, there's a hybrid solution: use a SpEL expression, but delegate (part of) the logic to a Java object. One powerful and simple way to do this is to delegate the decision to the message payload or a header. This requires the message to carry a domain object.

```
<filter id="cancellationsFilter" input-channel="inboundCancellations"
       discard-channel="rejectedCancellations"
       expression="payload.isGold()"
       output-channel="validatedCancellations" />
```

Weighing the pros and cons of each option is something you'll have to do again for each situation. The framework will support whatever decision you make in the end.

Now that you have some tools, let's look at some uses for them.

6.1.2 Using filters for selective processing

Let's get a bit of perspective here: how do messages that don't satisfy the criteria to be processed get on the inbound channel, anyway? Wouldn't it be simpler if their producers didn't bother to send messages that won't be processed?

Often, filters are used in combination with publish-subscribe channels, allowing multiple consumers with different interests to subscribe to a single source of information and to process only the items they're really interested in. Such a solution doesn't preclude multiple components receiving a message at the same time, but the components have complete control over what they can and can't process.

Figure 6.2 shows such an example. Three different components with different interests subscribe to the same channel. It's much easier if a producer knows only about a single destination channel for its messages.

Message filters can decide whether a message will be forwarded to a next channel. If you decide to configure them with a channel for discarding messages, a message

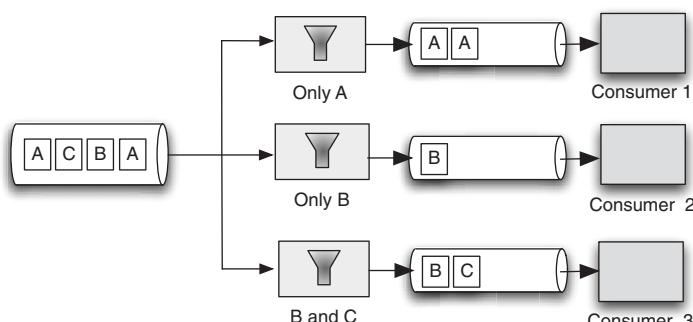


Figure 6.2 A publish-subscribe channel and filters combination for selective processing. The first subscriber is interested only in A's; the second, only in B's; and the third, in B's and C's.

filter will act as a switch, choosing one channel or another depending whether the message is accepted or discarded. In that case, they're a simplified form of a more general type of component that can decide from among multiple destinations where a message should go next. This component is the *message router*, which is the focus of the second part of the chapter.

6.2 **Whose message is this, anyway?**

In Spring Integration, channels play a crucial role in propagating messages across the system. Channels are also, by design, the only components of the framework that are directly referenced either by other components or by external modules of the application. Decoupling producers from consumers means, in this context, that the only information available to a message producer is a reference (or name) to the channel on which it should send a message, whereas the actual destination of the message and its processing sequence are determined by the configuration. For a message to be processed correctly, it should be sent on the correct channel.

As systems become more complex, determining which is the appropriate next channel becomes significantly more complicated. This knowledge is application specific, so it's a good idea to isolate it from the rest of the application.

A well-known aphorism says that “any problem in computer science can be solved by another layer of indirection.”¹ The problem here is to remove as much infrastructure-related knowledge and decision responsibility as possible from the individual components. The solution is a new component—the router—whose role is to choose a next target channel for a message and publish it there. Why wouldn’t components just send each message directly to the appropriate channel, instead of publishing them to an intermediate channel and deferring the routing decision to yet another component? The main benefit of using a message router is that the decision logic is encapsulated in a single component. The message-publishing components needn’t know anything about what follows downstream; the only information they require is which channel they should publish to.

Spring Integration provides an infrastructure for configuring message routing in your application. You can use either one of the routers provided out of the box and use the namespace configuration feature, or you can implement your own routers.

The aphorism we just quoted has a second part, which is usually forgotten: “but that usually will create another problem.” When using a router, its configuration must be aware of all the possible destination channels so it can make correct decisions. This may create another problem, as the saying goes, because the configuration of the router must be updated every time the routing logic changes. In practice, centralizing the configuration in a single place is generally a better choice than spreading the knowledge across the system. If a more dynamic approach is necessary, you can fall back on the alternative of using publish-subscribe channels and filters.

¹ The quote belongs to Butler Lampson, who in turn attributed it to David Wheeler.

6.2.1 Configuring routers

Customers must, at some point, pay for their trips. But many payment methods are available to a customer: online payment by credit card, popular services like PayPal, interbank networks, and so on. Even separate invoicing may be an option to trusted customers. Each payment method requires some specific information (such as an account number, a billing address) and performs a different operation.

Before a reservation is secured, the system must process a payment, which is initiated by the user. The payment process by itself is relatively straightforward: after the user decides on the form of payment, a message is sent to a payment channel, where it is picked up for processing by the system, which completes the booking process. The scenario works equally well if, for example, the user decides to save the reservation and pay for it through online banking. In this case, the payment notification isn't sent by the web application but by an external process (such as a nightly batch-processing transaction).

The individual processing strategies are mutually exclusive alternatives, and it makes sense to provide an individual channel for each of them to ensure that each payment notification is forwarded to the appropriate payment processor. Individual channels make it possible for any module (like the web UI) or a batch process to trigger payments of different types by creating messages with appropriate payloads and sending them on the appropriate channels.

In this case, rather than giving each component access to the complete list of channels and having them decide where to post the next message, it makes more sense to introduce a router. With a router, all payments are sent to a single channel and from there are processed by the router, which then forwards them to the appropriate target destinations. See figure 6.3.

The figure corresponds to the following configuration:

```
<channel id="payments" />
<channel id="invoices" />
<channel id="credit-card-payments" />
<channel id="paypal-payments" />
<router method="routePaymentSettlement" input-
    channel="payments"> <beans:bean
        class="siaa.booking.integration.routing
        .PaymentSettlementRouter"/>
</router>
```

The router receives messages from the payments channel and invokes the `routePaymentSettlement` method for deciding the next destination for the message. In

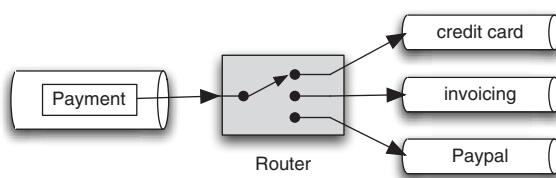


Figure 6.3 The router at work.
Any message published to `payments` is routed to either `invoices`, `credit-card-payments`, or `paypal-payments`

this application, the `PaymentSettlementRouter` can, for example, be implemented as follows (where `CreditCardPayment`, `Invoice`, and `PaypalPayment` extend `PaymentSettlement`):

```
package sisia.booking.integration.routing;

import sisia.booking.domain.payment.CreditCardPayment;
import sisia.booking.domain.payment.Invoice;
import sisia.booking.domain.payment.PaymentSettlement;
import sisia.booking.domain.payment.PaypalPayment;

public class PaymentSettlementRouter {

    public String routePaymentSettlement
        (PaymentSettlement paymentSettlement) {
        String destinationChannel = null;
        if (paymentSettlement instanceof CreditCardPayment)
            destinationChannel = "credit-card-payments";
        if (paymentSettlement instanceof Invoice)
            destinationChannel = "invoices";
        if (paymentSettlement instanceof PaypalPayment)
            destinationChannel = "paypal-payments";
        return destinationChannel;
    }
}
```

The method returns the name of the channel on which the message will be forwarded next, and if it returns null, the message won't be forwarded further.

COULD WE HAVE DONE THIS OTHERWISE?

Yes, we could. A service activator can provide a number of overloaded methods that take the different types as arguments. It looks like this:

```
package sisia.booking.domain.payment;

public class PaymentManager {

    public void processPayment(Invoice invoice) {
        // process payment for Invoice
    }

    public void processPayment(CreditCardPayment creditCardPayment) {
        // process payment for CreditCardPayment
    }

    public void processPayment(PaypalPayment payment) {
        // process payment for PaypalPayment
    }
}
```

Using this class, you can configure a service activator to process the messages:

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

<service-activator input-channel="payments" method="processPayment">
    <beans:bean class="silia.booking.domain.payment.PaymentManager"/>
</service-activator>
```

Spring Integration dynamically invokes the overloaded method that accepts an argument of the type of the payload of the message. You don't need a router as long as the right methods are compiled on the service activator class.

How does this compare with introducing a router? To answer this question, we must remember our primary goals: low coupling and easy extension of the application. The `PaymentManager` is a passable solution, but it works well only if the number of options is fixed and known in advance. If you need to add another payment option, you introduce a new payment type and a new method to handle it. As an example, let's assume you want to handle `DirectDebitPayment` as well. In this case, you have to add a new method to the `PaymentManager`:

```
public class PaymentManager {
    // all methods previously shown and adding:
    public void processPayment(DirectDebitPayment payment) {
        // process DirectDebitPayment
    }
}
```

By comparison, the router-based method allows you to expand the application without modifying anything that exists already (besides the application configuration). What you need to do is add a channel:

```
<channel id="direct-debit-payments"/>
```

And, if you're using the most basic router, shown earlier, you have to modify the routing logic accordingly:

```
package sii.booking.integration.routing;

import sii.booking.domain.payment.CreditCardPayment;
import sii.booking.domain.payment.Invoice;
import sii.booking.domain.payment.PaymentSettlement;
import sii.booking.domain.payment.PaypalPayment;

public class PaymentSettlementRouter {

    public String routePaymentSettlement (
            PaymentSettlement paymentSettlement) {
        String destinationChannel = null;
        if (paymentSettlement instanceof CreditCardPayment)
            destinationChannel = "credit-card-payments";
        if (paymentSettlement instanceof Invoice)
            destinationChannel = "invoices";
        if (paymentSettlement instanceof PaypalPayment)
            destinationChannel = "paypal-payments";
        return destinationChannel;
    }
}
```

If all you do is trade one class change for another, what's the gain here? For one thing, the routing logic is just a thin layer that has no dependency or direct interaction with the business logic, whereas the `PaymentManager` could easily grow into an overly complex entity with too many tasks and too many dependencies.

One of the advantages of using routing is that you can get away without implementing a new router class, as you can see from the first example in this section. Spring Integration comes with a wide variety of routers out of the box, the topic of the next section.

6.2.2 **Routers provided by the framework**

In our example, you saw a router at work. The router in the example decided where to send messages according to the payment option selected by the user, which constitutes the payload type of the payment message. Using the terminology of *Enterprise Applications Integration*, all the routers provided by the framework are *content-based routers*. This means the decision on where to send the message next is based solely on the contents of the message.

DEFAULT ROUTERS

The `PaymentSettlement` router you saw earlier is an example of using a default router. The simplest router is created by defining a `<routable/>` element. When you define a router, bear in mind the following:

- It must describe how the routing decision is made—it must indicate who is responsible for evaluating the message and determining the next channel.
- The decision may come in the form of a channel or channel name; in the latter case, the channel names must be converted to channel instances

When it comes to making a decision, there are two possible variants: either delegate to a method defined on a POJO, or use a SpEL expression. The previous router example showed how to implement a router using a POJO. Now it's time to look at some other variants.

To be usable for the routing logic, a method definition's arguments must comply with the same general requirements as business services and service activators: either take as argument a message or take as argument an object representing the payload and/or a number of @Header-annotated parameters. The return type of the method must be a message channel, a string representing a message channel name, a collection of strings or message channels, or an array of strings or message channels. The latter are supported because routers can return multiple values, and the next section provides an example of how that works.

If the method returns channel names, the names must be converted into channel instances. For that, you need to provide a channel resolver. If you don't want to provide one, the framework, by default, will provide one that looks up channels by their IDs in the application context. Using an explicitly configured channel resolver or not largely depends on how keen you are on using channel names inside your routing logic.

You can always resort to this option for configuring a router, but there are other options that don't require creating a new implementation every time you want to configure a router. The framework provides implementations that cover certain common use cases.

PAYOUT-TYPE ROUTERS

The router you implemented for `PaymentSettlement` instances takes into account the type of the object when deciding the next target channel. In your router implementation, you checked that yourself, but instead of doing that, you could've used an implementation provided by the framework. You could've written:

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

<payload-type-router>
    <mapping type="sia.booking.domain.payment.CreditCardPayment"
        channel="credit-card-payments"/>
    <mapping type="sia.booking.domain.payment.Invoice"
        channel="invoices"/>
    <mapping type="sia.booking.domain.payment.PaypalPayment"
        channel="paypal-payments"/>
</payload-type-router>

<channel id="invoices"/>
<channel id="paypal-payments"/>
<channel id="credit-card-payments"/>
```

As you can see in this example, whenever the next channel can be decided by the type of the message payload, you can use a `<payload-type-router>` out of the box instead of implementing that logic yourself.

HEADER VALUE ROUTERS

If the routing information (for example, the target channel) can be found in one of the headers of the message, you can use a header value router to simplify the configuration.

```
<header-value-router input-channel="payments"
    header-name="PAYMENT_PROCESSING_DESTINATION"/>
```

Here, the `PAYMENT_PROCESSING_DESTINATION` header defines the next destination of the message. The general idea behind routing based on header values is that the routing information can't be easily found in the message itself, but is added to the message earlier in the message flow. A header value router pairs well with a header enricher to populate the relevant header of the message.

```
<header-enricher input-channel="payments"
    output-channel="enriched-payments">
    <header name="PAYMENT_PROCESSING_DESTINATION" ref="enricher"
        method="addProcessingDestination"/>
</header-enricher>

<header-value-router input-channel="enriched-payments"
    header-name="PAYMENT_PROCESSING_DESTINATION">
```

Payload-type routers and header value routers are simplifications that cover particular use cases. Another way to set up a router without writing a new class is to use SpEL expressions.

ROUTING BY SPEL EXPRESSIONS

If the routing decision can be made through a simple evaluation of the message instead of by creating a separate POJO implementation and delegating to it, you can embed a SpEL expression into the router. This works well when the outcome of the expression evaluation matches the names of the target channels.

Let's take another example. As you saw previously, all credit card processing requests are routed to the same channel, `credit-card-payments`. But different credit cards may be processed differently because they're handled by different issuing organizations. You can extend the routing logic by adding another router to deal with credit card payments specifically, as follows:

```
<channel id="credit-card-payments" />

<router input-channel="credit-card-payments"
        expression="payload.creditCardType" />

<channel id="VISA" />

<channel id="AMERICAN_EXPRESS" />

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

This example doesn't use a separate class to implement the routing logic. Instead, a SpEL expression is evaluated against the payload. In this case, you evaluate the `creditCardType` property of the payload (which is of the type `CreditCardPayment`). The outcome of the evaluation may be `VISA`, `AMERICAN_EXPRESS`, or `MASTERCARD`, which are also the names of the potential channels.

In all the examples so far, the routing logic returns the name of the next destination channel. In most situations, this works well, but relying on the channel names to be fixed may prove a problem in the long run. You can introduce another level of indirection by allowing mapping the string values returned by the routing logic to channels from the configuration.

CHANNEL RESOLUTION AND YET ANOTHER LEVEL OF INDIRECTION

A strategy for developing more reusable application components is not to rely on your routing logic to return channel names but instead to use placeholder values that convey the logical significance of the routing process ("Whenever the payment will be settled through VISA credit card payment, send it to the channel for VISA payments, whichever name it has").

In this case, the router can be supplied with a `ChannelResolver` that will take care of translating the string value provided by the routing logic into an actual channel instance.

```
<channel id="VISA-payments" />
<channel id="AMERICAN_EXPRESS-payments" />
<channel id="MASTERCARD-payments" />

<router expression="payload.creditCardType"
        channel-resolver="creditCardPaymentsChannelResolver"
        input-channel="credit-card-payments" />
```

```
<beans:bean id="creditCardPaymentsChannelResolver"
    class="sia.booking.integration.routing.
        CreditCardPaymentChannelResolver"/>
```

In the previous example, the names of the channels had to match the potential results of evaluating the routing expression, but introducing the ChannelResolver gives you more liberty in defining your configuration. Adding a suffix may not seem a significant change to the previous case, but the destination channels have different names than the result of evaluating the expression, and it is the role of the ChannelResolver to bridge the gap.

All the examples so far are based on the assumption that the routing scenario implies a single next destination channel. But routing can also enable a message to be resent to a number of other channels, as discussed next.

6.2.3 **Routers with multiple destinations**

Our payment example showed how to use a router when you have one possible destination out of a few. This situation is the most common situation you'll encounter, but it's also possible to have more than one next destination channel for a message.

Let's consider a notification system that has different ways of notifying customers and different notification types. A simple weather update may be sent through email, but more urgent notifications, such as a cancellation notice, may require sending a Short Message Service (SMS), an email, and placing an automated phone call at the same time to make sure the customer is reached by all the means possible.

Because there is a single source of notifications, the notifications are sent on a single channel. From there, a router distributes them to the channels that correspond to the individual notification strategies: SMS, email, voice. The difference between this notification scenario and the payment scenario is that a notification can be sent by multiple channels simultaneously.

MULTIPLE RETURN VALUES

In the previous POJO-based example, the `routePaymentSettlement` method of the `PaymentSettlementRouter` returned a single value. But sometimes you must handle multiple values. Let's consider the following configuration for notifications:

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

<router input-channel="notifications" ref="notificationsRouter"
    method="routeNotifications">
</router>

<channel id="sms"/>
<channel id="email"/>
<channel id="phone"/>
```

The intent is to contact the client in as many ways the business logic requires. Routing logic must take into account both the notification settings of the user and the urgency of the notification. Receiving an automated phone call in the event of a cancellation is

pretty sensible, but being called every time there's a sale doesn't fall into the same category. Both are notifications, and will be delivered initially on the notifications channel, where they'll be distributed by the router.

```
package sria.booking.integration.notifications;

import sria.booking.domain.notifications.FlightNotification;
import sria.booking.domain.notifications.Priority;

import java.util.ArrayList;
import java.util.List;

public class NotificationsRouter {
    public String[] routeNotification(FlightNotification notification) {
        List<String> notificationTypes = new ArrayList<String>();
        if (Priority.HIGH == (notification.getPriority())) {
            notificationTypes.add("phone");
        }
        if (Priority.MEDIUM == (notification.getPriority())) {
            notificationTypes.add("sms");
        }
        if (Priority.LOW == (notification.getPriority())) {
            notificationTypes.add("email");
        }
        return notificationTypes.toArray(new String[0]);
    }
}
```

In this case, the router sends messages to all the channels returned by the method, and please note that the resulting combination can vary from case to case (both the priority and the user settings are factors to be considered).

If the only job for the router is to dispatch messages to a set of fixed channels, the application can use a recipient list router.

RECIPIENT LIST ROUTERS

A *recipient list router* is slightly different from the routers you've seen so far that make decisions about the next destination channel. Recipient list routers are configured with a group of destination channels and will forward incoming messages to all of them.

You may wonder what such a router is good for. After all, publishing to multiple destinations is the job of a publish-subscribe channel. This may be true when you're designing a system from scratch, but the recipient list router works well when the destination channels are already defined and the target components are already listening to them. Another possible use is for broadcasting a message to multiple channel adapters.

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

<recipient-list-router input-channel="notifications">
    <recipient channel="sms"/>
    <recipient channel="email"/>
    <recipient channel="phone"/>
</recipient-list-router>
```

```
<channel id="sms"/>
<channel id="email"/>
<channel id="phone"/>
```

With such a configuration, any message sent to the notifications channel is automatically forwarded to all the channels defined on the recipient list of the recipient list router.

This overview of the message routers provided out of the box by the framework wraps up the second part of this chapter. Now you know pretty much all that you can do for filtering and routing by using the namespace configuration. If you're interested in learning more about the classes used to implement these features and how they interact, you'll find more information in the next section, where we lift the hood and take a peek at the internal parts of the framework.

6.3 Under the hood

The most common way to use filters and routers is through the namespace configuration. This section dives deeper into details and explains what happens underneath. It's optional reading, because knowing the internals of the framework isn't a prerequisite for using it efficiently.

But there are two valid reasons to investigate what happens inside the framework and what main actors are at play. First, it helps with debugging. Second, the framework provides a few out-of-the-box components that treat the most common use cases and allow you to create problem-specific implementations (such as the ones in chapter 8). Expanding the framework in such a way requires some detailed knowledge about the collaborating classes and the APIs provided by the framework, and this is what we focus on next.

6.3.1 The Message Filter API

The extension point of the Message Filter API is the `MessageSelector`. It's used in several places (including in the `MessageSelectingInterceptor` you saw in chapter 3). It's a method object that computes a boolean value for a given message. The main actor is the `MessageFilter` class, which implements all the filtering process except for decision making, which is left to the `MessageSelector` that's injected into the `MessageFilter`. See figure 6.4.

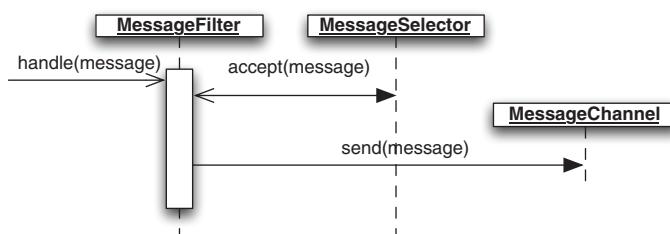


Figure 6.4
Sequence diagram of the `MessageFilter`: a `MessageSelector`'s `accept()` method evaluates the message to decide if it should be sent further.

6.3.2 The Message Router API

Any router is a descendant of `AbstractMessageRouter`, whose basic features are laid out in the following code snippet:

```
public abstract class AbstractMessageRouter
    extends AbstractMessageHandler {
    public void setDefaultOutputChannel (
        MessageChannel defaultOutputChannel) {
        /* ... */
    }
    public void setResolutionRequired(boolean resolutionRequired) {
        /* ... */
    }
    /* ... details left out ... */
    protected abstract Collection<MessageChannel>
        determineTargetChannels(Message<?> message);
}
```

All that's specific to a given routing strategy is encapsulated in `determineTargetChannels`. The interaction is illustrated in figure 6.5. The router invokes the method and retrieves a collection of `MessageChannels`. The list is iterated and the message is sent to every channel from the collection.

The class provides two further configuration options:

- To set up a default output channel, which is used by the router to forward the messages if `determineTargetChannels` returns with no next destination channels
- A flag that forces the router to throw an exception if no next destination channel can be determined (that is, if a resolution is required).

All the namespace elements used to define a router allow for setting these options through XML attributes.

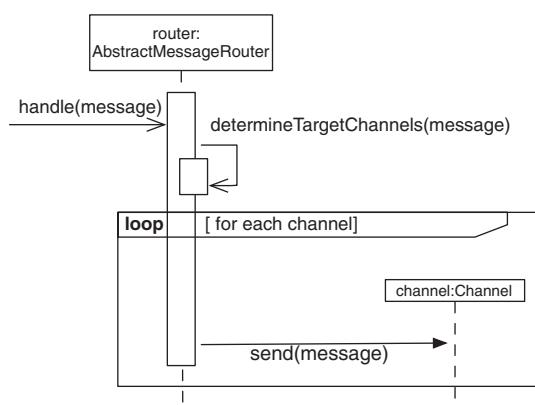


Figure 6.5 Sequence diagram of the `AbstractMessageRouter`: the `determineTargetChannels` method decides the next destinations.

6.4 Summary

In this chapter we moved past the simple sequential model of chaining message handlers and started to deal with more complex configuration problems. You saw how to set up your application to process messages selectively by filtering out the ones that a given component is not supposed to handle. Filters can be used to perform message validation (especially when validity isn't a domain-inherent characteristic but can vary among applications). More than simple validation, this functionality is a powerful complement to publish-subscribe channels, allowing consumers with different interests to subscribe to the same channel but guaranteeing that they'll receive only the messages they're supposed to handle.

To handle a message correctly, a producer must send it to the appropriate channel—a channel that has the intended recipient as a consumer. It's possible to have a number of alternatives to choose from, and in order to have a flexible configuration, and to keep components reusable, such decisions are best left out of the processing components, and set up through the application configuration.

You saw two message distribution alternatives: one using publish-subscribe channels and filters, and one using routers. A router-based solution is both centralized and closed, and by that we mean all the routing configuration is in a single place (the router). You can't add new potential recipients of the message without modifying it. The publish-subscribe channel solution, on the other hand, is de-centralized and open: the routing configuration is emerging as a sum of all the conditions defined on the individual filters, and new subscribers can be added without modifying any existing component. If your scenario calls for adding and removing consumers dynamically, the publish-subscribe channel solution is the best option. Otherwise, the centralized configuration provided by the router allows for easier maintenance.

Ideally, message consumers such as transformers and service activators should be compact, encapsulated, and highly cohesive units. This allows them to be flexible so they can be redeployed in a large variety of scenarios. At the same time, the application may be required to handle more complex messages. The next chapter is dedicated to components such as splitters and aggregators that allow the framework to split messages into groups and deal with the correlated messages.

Part 3

Integrating systems



Splitting and aggregating messages

This chapter covers

- Splitting messages into parts
- Aggregating messages into a composite
- Reordering messages with a Resequencer
- Customizing aggregation

Previous chapters explained how a single message is processed as a unit. You saw channels, endpoints such as service activators and transformers, and routing. All these components have one thing in common: they don't break the unit of the message. If one message goes in, either one message comes out the other end or it is gone forever. This chapter looks at situations in which this rule no longer holds. In some situations, one message goes in and several messages come out (Splitter), and in others, several messages go in before messages start coming out (Aggregator, Resequencer). Examples of endpoints illustrating the various possible scenarios are shown in figure 7.1.

This chapter is the first to introduce *stateful endpoints*. The Resequencer and the Aggregator must hold a state because the outcome of handling a given message

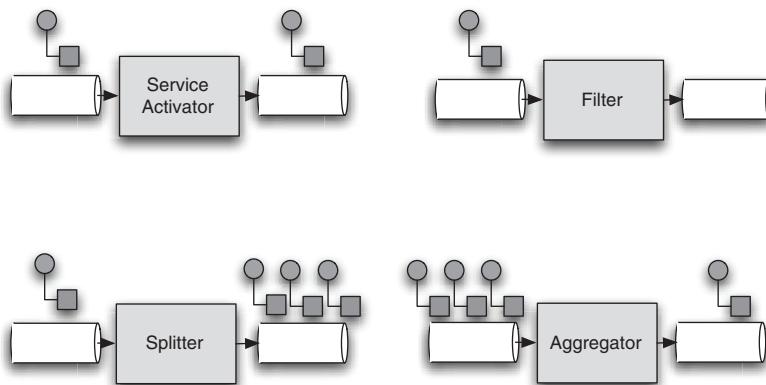


Figure 7.1 Examples of endpoints processing a message in one-to-one, one-to-many, and many-to-one scenarios

depends on the previous messages, which isn't the case for the endpoints described in previous chapters. The fact that these endpoints hold a state for functional reasons differentiates them also from other stateful endpoints that hold a state for improving performance, such as file adapters, which hold a queue of files in memory to prevent costly file listings (see chapter 12). Not all the endpoints in this chapter are stateful: for example, the Splitter, which we introduce for symmetry with the Aggregator, is a stateless component. Generally, the fact that components are stateful or stateless plays an important role in the concurrent and transactional behavior of your application, so it's important to pay close attention to this aspect.

As we discussed in chapter 1, correlating messages is sometimes essential to implementing a certain business requirement. This chapter explains the different options for message correlation, whether your goal is to reassemble a previously split-up array of related data (Splitter-Aggregator), make sure ordering constraints are enforced on messages (Resequencer), or split up work between specialized services (Scatter-Gather). In all these cases, the correlation of the messages is key, so it makes sense to discuss how Spring Integration stores and identifies the correlation of messages and how you can extend this functionality. It's explained in detail in the "Under the hood" section later in the chapter; for now, it's enough to understand the functional concept of correlating messages in a group.

7.1 *Introducing correlation*

The functionality of the components discussed in this chapter—Aggregators, Splitters, and Resequencers—is based on the idea that certain messages are related in a particular way. This section focuses on correlation from a functional perspective, introducing the main concepts behind it.

An Aggregator, such as the one in figure 7.2, waits for messages in a certain group to arrive and then sends out the aggregate. In this particular case, the Aggregator

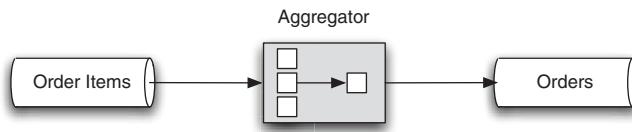


Figure 7.2 An Aggregator is an endpoint that combines several related messages into one message.

waits for all the items of an order to come in, and after they are received, it sends an order to its output channel.

The part we're interested in first is how it determines what messages belong together. Details follow in later sections. Spring Integration uses the concept of a *message group* which holds all the messages that belong together in the context of an Aggregator. These groups are stored with their *correlation key* in a message store. The correlation key is determined by looking at the message, and it may differ between endpoints.

Now is a good time to look at an analogy to help anchor the important terms used in this chapter.

7.1.1 A real-life example

Let's say you're having guests for dinner. Maybe you're not the cooking kind, but you're probably familiar with the concept of home cooking, and that's more than enough. We'll look at the whole process of preparing and serving a meal to see what's involved in automating this process and how it applies in terms of messaging. For the sake of argument, we'll ignore the possibility of ordering take-out (which would greatly decrease the complexity of the setup but would ruin the analogy). Figure 7.3 illustrates the scenario described in the rest of this section.

A recipe is split into ingredients that are aggregated to a shopping list. This shopping list is converted into bags filled with products from the supermarket. The bags are then split into products, which are aggregated to a *mise en place*, which is finally transformed into a meal. The channel configurations are considered trivial and are omitted from the diagram.

Involved in the dinner are the host (that's you), the kitchen, the guests, and the shops. You orchestrate the whole event; the guests consume the end product and turn it into entertaining smalltalk and other things irrelevant to our story. The kitchen is the framework you use to transform the ingredients you get from the shop into the dinner.

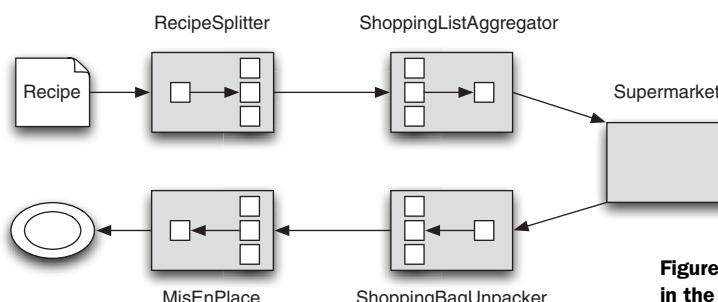


Figure 7.3 The flow of messages in the home cooking example

We're interested in the sequence of events that take place after a date is set.

It starts with you selecting a menu and gathering the relevant recipes from your cookbook or the Internet. The ingredients for the recipes must be bought at various shops, but to buy them one at a time, making a round trip to the shop for each product, is unreasonably inefficient, so you must find a smarter way to handle this process. The trick is to create a shopping list for each shop you must visit. You pick the ingredients from each recipe, one by one, and put them on the appropriate shopping list. You then make a trip to each shop, select the ingredients, and deliver the ingredients back to your kitchen.

With the ingredients now in a central location, each shopping bag must be unpacked and the ingredients sorted according to recipe. Having all the right ingredients (and implements) gathered together is what professional chefs call the *mise en place*. With all the necessary elements at hand, each dish can be prepared, which usually involves putting the ingredients together in some way in a large container. When the dish is done, it's divided among plates to be served.

But what does this have to do with messaging? Perhaps more than you think.

7.1.2 Correlating messages

Let's say the recipe is a message. This message is split into ingredients, which can be sent by themselves as messages. The ingredients (messages) reach an Aggregator that aggregates them to the appropriate shopping lists. The shopping lists (messages) then travel to the supermarket where the shopper turns them into bags filled with groceries (messages), which travel back to the kitchen (endpoint).

The shopping bags are unpacked (split) into products that are put together in different configurations on the counter during the *mise en place*. These groups of ingredients are then transformed into a course in a pan. The dish in the pan is split onto different plates (messages), which then travel to the endpoints that consume them at the table. What we see here is a lot of breaking apart and putting together of payloads. The recipes are split and the ingredients aggregated to grocery lists. The bags are unpacked and the products regrouped for the different courses. The dishes are split and assembled on plates.

Some observations can be made from this analogy that will be helpful to think back on later in the chapter. Splitting is a relatively easy job, but it's important to keep track of which ingredients belong to which recipe. Messages (ingredients in this example) are given a *Correlation ID* to help track them. Most easy examples of splitting and aggregating use a Splitter that takes a thing apart and an Aggregator that turns all the split parts into a thing again. This is a simplistic example, so it's good to keep a simple analogy in mind that does things differently. In a real enterprise, an Aggregator often has no symmetric Splitter.

It now becomes important to think about how we'll know when the aggregate is done. Or to stay with the example, when is the shopping list done? The answer is that it's only done when all the recipes have been split and all ingredients are on their appropriate list. This is still relatively simple but more interesting than to say that a list is done when all ingredients of one particular recipe are on it.

When aggregating is done without a previous symmetric splitting, it becomes harder to figure out which messages belong together, or in this example, which ingredients go on which list. Usually aggregation relies not on a message's native key but on a key generated by a business rule. For example, it could be that all vegetable ingredients go on the greenery list and all meat ingredients go on the butcher list. This assumes that you're not just buying everything from the supermarket, but even if you do, it makes sense to organize your shopping list by type to avoid having to backtrack through the supermarket.

The next sections explain how the components introduced here are used. For those interested in the home cooking example, some code is available on the *Spring Integration in Action* repository (now under <http://github.com/hierynomus/kitchen/>).

7.2 Splitting, aggregating, and resequencing

It's common for domain models to contain high-level aggregates that consist of many smaller parts. An order, for example, consists of different order items; an itinerary consists of multiple legs. When in a messaging solution one service deals with the smaller parts and another deals with the larger parts, it's common to tie these services together with endpoints that can pull the parts out of a whole (Splitter) and endpoints that can (re)assemble the aggregate root from its parts (Aggregator). The next few sections show typical examples and the related configuration of Splitters, Aggregators, and Resequencers.

7.2.1 The art of dividing: The Splitter

The basic functionality of a Splitter is to send multiple messages as a response to receiving a single message. Usually these messages are similar and are based on a collection that was in the original payload, but this model isn't required in order to use the Splitter.

Let's look at an example of splitting in our sample application. Look at flight-notifications.xml to see the starting point of the relevant code. When flight notifications come in, you want to turn them into notifications about trips and send them to impacted users. To do so, you enrich the header of a flight notification with a list of impacted trips. At the end of the chain, you can then use a Splitter that creates a TripNotification for each trip related to the flight.

```
<chain input-channel="flightNotifications"
       output-channel="tripNotifications">
    <header-enricher>
        <header name="affectedTrips"
               ref="relatedTripsHeaderEnricher"
               method="relatedTripsForFlight"/>
    </header-enricher>
    <splitter id="flightToTripNotificationsSplitter">
        <beans:bean class="...FlightToTripNotificationsSplitter"/>
    </splitter>
</chain>
```

Interesting to note here is that the payload isn't chopped up: the splitting is based instead on a list in a header. Spring Integration is indifferent to the splitting strategy as long as it gets a collection of things to send on as separate messages. It's entirely up to you what those things are.

The home cooking example also contains a Splitter, which chops up a recipe into its ingredients:

```
<chain id="splitRecipesIntoIngredients"
    input-channel="recipes"
    output-channel="ingredients">
    <header-enricher>
        <header name="recipe" expression="payload" />
    </header-enricher>
    <splitter expression="payload.ingredients" />
</chain>
```

As you can see, a chain is used around the Splitter to pop references to the original recipe on a header, so you can use it as a correlation key later when you aggregate the products. The Splitter is a simple expression that gets the ingredients (a list) from the recipe payload.

Splitters are relatively simple to configure. You can use a plain old Java object (POJO) or a simple Spring Expression Language (SpEL) expression to retrieve the desired information from the message in the form of a `List`.

7.2.2 How to get the big picture: The Aggregator

We looked at splitting messages using a Splitter. Before we think about doing the reverse, we need to think backwards through the splitting process. The Splitter outputs sets of messages, each generated by a single message received by the Splitter. It's the original message that correlates them (see section 7.2).

In Spring Integration, messages that are correlated through a correlation key can be grouped in certain types of endpoints. These endpoints keep the notion of a `MessageGroup`, discussed in the section "Under the hood" later in this chapter.

Messages can belong to the same group for many reasons. They may have originated from the same Splitter or Publish-Subscribe Channel, or they may have common business concerns that correlate them. For example, the flight notification application could have a feature that allows users to have the system batch the notifications they receive per email on the basis of certain timing constraints. In this case, there's no concept of an original message; you'll often see examples where aggregating isn't used in relation with any splitting logic.

Back in the kitchen, aggregation is also going on. Remember popping the recipe on a header? Now when you aggregate the products from the stores back together, you can use this recipe as a correlation key.

```
<aggregator
    id="kitchen"
    input-channel="products"
    output-channel="meals"
```

```

    ref="cook"
    method="prepareMeal"
    correlation-strategy="cook"
    correlation-strategy-method="correlatingRecipeFor"
    release-strategy="cook" />

```

The Aggregator called kitchen refers to a cook for the assembly of the meal. The cook has a method to aggregate the products.

```

@Aggregator
public Meal prepareMeal(List<Message<Product>> products) {
    Recipe recipe = (Recipe) products.get(0).getHeaders().get("recipe");
    Meal meal = getMealForRecipe(recipe);
    for (Message<Product> message : products) {
        meal.cook(message.getPayload());
    }
    return meal;
}

```

This snippet shows how a group of related products are assembled into a meal, but it doesn't show how these messages are related and when they're released. More details on this are found in the "Under the hood" section. For now, we just show the implementations of correlation strategy (to determine which message belongs to which group) and release strategy (to determine when a group is offered to the cook).

The correlation strategy relates products according to their recipe.

```

@CorrelationStrategy
public Object correlatingRecipeFor(Message<Product> message) {
    return message.getHeaders().get("recipe");
}

```

The release strategy delegates to the recipe to determine if all the ingredient requirements are met by products.

```

public boolean canRelease(MessageGroup group) {
    Recipe recipe = (Recipe) group.getGroupId();
    return recipe.isSatisfiedBy(productsFromMessages(group));
}

```

But gathering related messages and processing them as a group isn't the only use case for correlation. Another component works similarly to the Aggregator and can be used to make sure messages from the same group flow in the correct order: the Resequencer.

7.2.3 Doing things in the right order: The Resequencer

When different messages belonging to the same group are processed by different workers, they may arrive at the end of a message flow in the wrong order. As we saw in chapter 3, there is a priority channel that can order messages internally, but this channel doesn't consider the whole group. Instead, it passes along the message first in line, not caring about gaps in the sequence caused by messages that haven't yet arrived at the channel.

The solution for this problem is a Resequencer. It can guarantee that all messages in the group arrive in exactly the right order on the Resequencer's output channel. This pattern is more like an Aggregator than you might realize at first glance. Like the Aggregator, the Resequencer has to wait for several members of a group of correlated messages to arrive before it can make a decision to send a message to its output channel.

Why ordering should be avoided if possible

Before looking at examples of where a Resequencer might be useful, we offer a word of warning. From an architectural perspective, depending on the ordering of messages is almost without exception a problem when scaling and performance are at stake. The problem arises because the Resequencer is a stateful component, and to guarantee that *all* messages of the sequence arrive in the right order, the only way to clean up the state from a Resequencer is to ensure that all messages of the sequence have been sent to it.

As a rule, you should only depend on resequencing within a single node and only if the whole sequence can reasonably be expected within a short time. What is reasonable and short wholly depends on the characteristics of your application and target environment.

Recovering from message loss or timeouts is far from trivial when you have sequence dependencies. If you can design the system in such a way that messages which are older than the last message processed are simply dropped, this is a fundamentally more robust solution. That said, in some cases resequencing is convenient, so you should understand the concept.

Good recipes give the ingredients in an order that makes sense for the preparation. But as you split the ingredients lists and spread the items out over multiple shopping lists, you end up with the ingredients in a different order than they should be.

When groups of messages are processed concurrently, say, when you start splitting recipes with multiple people or when you let multiple people shop at the same time, it's obvious that you introduce race conditions. As long as no checking is done later and no ordering requirements are presented, you won't see any negative side effects. Remember this rule of thumb: adding concurrency increases the random reordering of messages.

On the shopping list, the ingredients are in semi-random order determined by the order in which the recipes are split. Because stores generally arrange their products by type, it's most efficient for you to organize your list in the same way. This is a traveling salesman problem¹ that can be simplified by assuming the shop has only one possible walking route and you only have to avoid backtracking.

It's important to make sure the shopping list is reordered when it's completed (not before). This is done by the Resequencer, which is much like the Aggregator, but

¹ A traveling salesman problem is the problem of finding the shortest route that visits all destinations from a given set exactly once.

instead of releasing a single message, it releases all messages ordered according to their sequence number or a custom comparator. In the shopping example, the sequence number isn't used, but instead, the ingredients are compared in shopping-list order.

Using a custom comparator, it's not possible to release partial sequences because you can't know in advance whether another ingredient might need to be inserted in the middle of the list.

When doing the *mise en place*, it's a good practice to arrange the ingredients in the order they appear in the ingredient list. The *mise en place* is essentially the resequencing of the correlated ingredients according to the order in which they appear in the recipe. This happens just before they're aggregated into the pan (or bowl or what have you).

For this example of resequencing, you can depend on the sequence number and size set by the recipe Splitter. Therefore, you can release partial sequences. If your ingredients list starts with onions and lists garlic as a second item, you can be sure that if you pull the onions and the garlic out of the shopping bag first, you can prepare them before pulling the next items out.

This should give you some idea of how to think about resequencing in an everyday life scenario. In most complex enterprise applications, it's possible to find an analogy, using your favorite subject, that fits well with what should happen. Car and kitchen metaphors will carry you a long way as an architect.

After reading this section, you should have a good general idea of what splitting, aggregating, and resequencing are and how you can use them in an architecture. The next section elaborates a bit more on some common but nonstandard configurations.

7.3 Useful patterns

Our example cases so far have shown the most obvious correlations between messages: a well-defined group of payloads that are released as a group. This isn't the only possible use for correlation, though, and this section elaborates on two other use cases: timing-based aggregation and the Scatter-Gather pattern. Both demonstrate that aggregation involves much more diverse scenarios than you might first think. The way the different payloads and headers are aggregated is important, and so are what messages belong together and how strong this correlation is from a business point of view.

Many scenarios don't operate with groups of individual messages that can be aggregated together like the order items that belong to a specific order. They operate in a looser fashion: the groups have certain requirements concerning the numbers or kinds of payloads that must be present in the group before release, but, for example, payload instances of the same kind may be interchangeable. Consider an order trading system: the condition for making a trade and releasing a group of messages is to find a match between the buy (long) and sell (short) orders. For example, broker A places a long order for 1000 shares for Glorp Corporation (GLC), and then customer B places a short order for 600 GLC shares, and customer C places a short order for 400

GLC shares. These three orders can be fulfilled against each other, but if a customer D then placed a long order of 500 GLC shares, its order could be also be fulfilled against customer B's short order. The outcome depends greatly on the sequence in which the orders arrive, including timing.

Race conditions like this one are often inevitable because being completely fair is impractical, if not impossible, given the performance requirements. A heuristic approach is a better fit, and various patterns have emerged in an attempt to offer a satisfactory, albeit not ideal, solution. The next two subsections focus on two common patterns in aggregation that don't immediately fall under the straightforward example of taking something apart and putting it back together. The first section explores aggregation based on nothing but timing, and the second section deals with Scatter-Gather.

7.3.1 Grouping messages based on timing

In many Aggregator use cases, completion is based not only on the group of messages but also on external factors such as time. Let's look into such a scenario and see how it's supported by Spring Integration.

REFINING THE SHOPPING LIST AGGREGATION

Let's think back on the dinner example. When, as the host, you're aggregating ingredients on the shopping lists, you can of course wait until all recipes have been split before going to the store, but that might make for a very long list. It might also take a lot of time; say, for example, 10 minutes longer than if you were to give your spouse a partial list so they could leave for a particular shop while you wait for the splitting to finish. Then you can give the next part of your unfinished but long list to a friend, who can also start shopping before the splitting is complete. When the splitting is done, you have three separate lists, two of which are already being worked on. This early completion strategy is useful to ensure all workers are busy in a complex system. Big lists are good for optimization, but making a list infinitely big doesn't help effectiveness.

In terms of Spring Integration's Aggregator support, what should be happening here? First of all, this scenario has a time-based constraint. At a particular time, the aggregated list is sent regardless of whether or not it's complete. Then, of course, the newly arriving messages must still be aggregated, so multiple aggregates, not just one, are sent. In figure 7.4, you can see how this might work in practice.

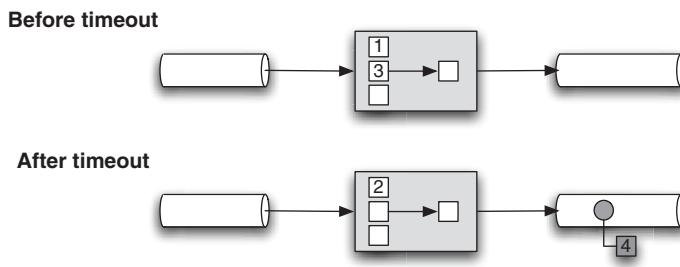


Figure 7.4 Before the timeout, just two out of three messages have arrived. On the timeout, the Aggregator sums 1 and 3 and sends the aggregate (4). A bit later, it receives the missing 2, which it must send out without the rest of the aggregate.

Note on timeouts

Timing out means a separate trigger is fired when the timeout point is reached. This is fundamentally different from normal release because the timeout event is not based on reception of a message. In Spring Integration 2.0 this functionality has been pushed down from the aggregating message handler into the Message Store itself. Because timeout is important to the end user, it's still exposed as a flag on the `<aggregator/>` element. The Message Store will give the Aggregator a callback when it's time to timeout, and when you set the `send-partial-result-on-expiry` flag, the incomplete group will be sent.

The release strategy in a timeout can be unchanged. This means you need to do something else to ensure a partial timeout at some point. It's sometimes possible to modify the release strategy to always release the group when it finds a certain time has elapsed, but the problem with this approach is that the release strategy is only interrogated when a new message arrives. If it takes a while for messages to arrive, the timeout might pass without a release happening.

When an incomplete group is sent on expiry, the remaining group is usually also incomplete. For example, the default strategy of counting the messages and comparing their number with the sequence size will no longer work. Usually in this case, there's a business rule that can tell you whether you received all the messages. In the shopping list example, you can check whether all recipes are split already, and because a direct channel is used for sending ingredients, the last group can be completed without checking the size.

As you can see, aggregation can be based on more than just business keys and even on the same key repeatedly. Next we look at a situation in which the different messages are the result of work done on a different collaborating node: Scatter-Gather.

7.3.2 Scatter-Gather

In the most typical cases, aggregating is based on a list of similar messages and splitting is about cutting up the payload of a message. This isn't always the case, though. In this section, we look at a common use case that doesn't follow this pattern.

ISN'T SCATTER-GATHER THE SAME AS MAP-REDUCE?

The next few paragraphs are about the definition of Scatter-Gather and how it's different from Map-Reduce. (Even if you don't know what Map-Reduce is, you should be fine with the rest of the chapter.)

Scatter-Gather is a name commonly used to refer to a system that scatters a piece of information over nodes that all perform a certain operation on it, then another node gathers the results and aggregates them into the end result. The major difference between it and Map-Reduce is that, in Scatter-Gather, the different nodes might have different functions. You can learn about Map-Reduce from many other resources, and because Spring Integration isn't a Map-Reduce framework, we don't cover it here. It's

important to note that Scatter-Gather and Map-Reduce are by no means mutually exclusive: they are complementary, and a good architect should be able to weigh the applicability of both or either of them against the complexity they inevitably add to the system.

Enterprise Integration Patterns (<http://www.enterpriseintegrationpatterns.com/>) defines Scatter-Gather as follows: “Scatter-Gather routes a request message to the a number of recipients. It then uses an Aggregator to collect the responses and distill them into a single response message.”

This is a broad definition, and it could even be said that Map-Reduce is a subtype of Scatter-Gather. We look at an example where the different nodes have different functions so that we’re forced to stay clear of Map-Reduce concerns.

The home cooking example contains a good candidate for Scatter-Gather. When you split the ingredients over multiple shopping lists, you might find that certain shops offer the same products. You can implement several behaviors that take this into account.

For example, a certain product might often be out of stock in shops. If so, then it’s no problem to stock more than you need (it’s conservable), so you can try to buy it at all the shops. If you serialize the shopping or allow communication between the shoppers, you could decrease the risk of overbuying.

If you’re looking at an expensive product, you can allow shoppers to compare prices with each other when they’re shopping in parallel. This happens at the cost of synchronization overhead. The amount of synchronization needed here depends on how bad it would be if you bought too much or if you bought it at a higher price.

Our example assumes that no synchronization is done and you’ll try to buy the ingredient at all shops. The one that ends up on the *mise en place* is closest to the best-before date. Figure 7.5 presents a schematic overview of a this Scatter-Gather scenario.

To scatter an ingredient after splitting, you need to route it to multiple nodes. You can do this by configuring a router that does its best to route to a single shopping list, but if that fails, it routes to a Publish-Subscribe Channel that all the shopping lists are connected to (so it ends up on all lists instead of one).

Another option is to get rid of the router altogether and use a filter in front of each shop that drops all ingredients which can’t be found at the shop. Yet another option is to let the filtering occur naturally by asking each shop for each product and

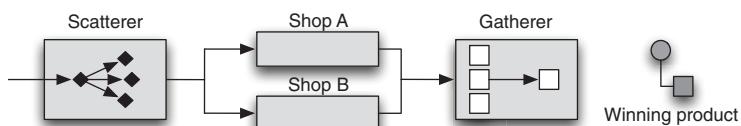


Figure 7.5 The needed ingredient is scattered over all shopping lists and sent to each shop (A and B). The gatherer decides, on the basis of the best-before date (or some other criterion), what is the best product to use and sends that to the *mise en place* (not in this picture).

taking all that are available. It depends on the situation which option is more efficient, and we won't spend time tuning it further here.

Gathering happens when the products come back from the shops. The easiest way is to use whatever comes first out of the shopping bags and to not use unneeded items. The other option is to compare the duplicate products' best-before date when all products for a *mise en place* are complete and store the one that can be conserved longest.

In this section, you saw two examples of Aggregator that differ from the standard usage of reassembling some collection. There's only one thing left to do, and that's to open the black box and look at the machinery of Spring Integration that makes all this tick.

7.4 Under the hood

The best class to start looking at when you want to figure out what Spring Integration does under the hood in terms of correlating messages is the `CorrelatingMessageHandler`. This class is wired by the `AggregatorParser` with collaborators that make it into an Aggregator, as well as wired by the `ResequencerParser` with collaborators that make it into a Resequencer. In this section, we look at the steps the `CorrelatingMessageHandler` performs to group, store, and process messages, then we look at two examples of wiring a `CorrelatingMessageHandler` as an Aggregator and as a Resequencer.

7.4.1 Extension points of the CorrelatingMessageHandler

The `CorrelatingMessageHandler` can process a group of messages in two ways: message in and message out. When the message comes in, it's correlated and stored. When a message group might go out, it's released, processed, and finally marked as completed. Let's look into the details of each of those steps—correlate, store, release, process, complete—as shown in figure 7.6, and introduce collaborators as we go along.

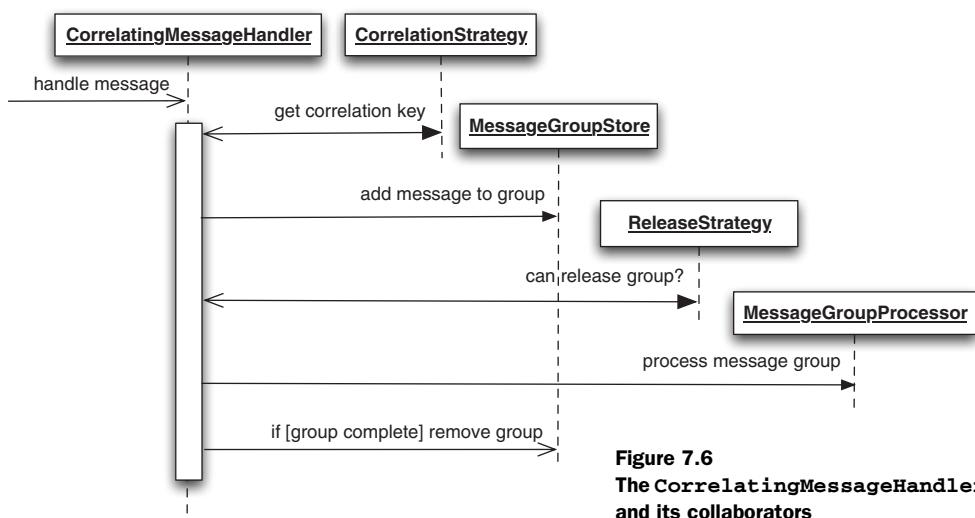


Figure 7.6
The `CorrelatingMessageHandler` and its collaborators

When a message hits the `CorrelatingMessageHandler`, the first thing it needs to do is figure out what `MessageGroup` this message belongs to. The message group is defined by its correlation key (not to be confused with Correlation ID). The correlation key is retrieved from the `CorrelationStrategy`, which defaults to a `HeaderAttributeCorrelationStrategy`. The default strategy picks the Correlation ID from the message headers, but this doesn't have to be your strategy.

After the correlation key is found, the message can be stored with its group. For this you use a `MessageGroupStore`, which defaults to an in-memory implementation. Storage used by the `CorrelatingMessageHandler` can be entirely customized. A `JdbcMessageStore` is available in the framework, but it stands to reason that a NoSQL store is more fitting in many cases. The storage will hold all incomplete message groups, so it's important to consider memory consumption and performance in case of large groups or large numbers of incomplete aggregates.

After the message is stored, that message's group is considered for release. For example, a completed aggregation will be released, and a partially completed group that may contain the first few elements of a sequence may also be released. The release strategy says nothing about the completeness of the group. Its only responsibility is to decide whether the message processor may process this particular group.

Once a group is released, it's handed to the `MessageProcessor` of the `CorrelatingMessageHandler`. This is where the actual operations on the messages are performed. The message processor is handed a template to send messages with and is expected to make all decisions relevant to sending output messages. It's also responsible for marking the messages it has processed in the message group.

The marked messages are then recognizable as processed if the same group hits the message processor later. There is no restriction on the contract that the processor has to fulfill that disallows it from reprocessing marked messages.

In the next few paragraphs, you'll see the implementation of Aggregator and Resequencer as examples of the mentioned strategies. In both Aggregator and Resequencer, correlation and storage are the same (and trivial), so we go into the details of release and processing only.

7.4.2 **How do Resequencer and Aggregator do it?**

The Aggregator, as discussed earlier, takes a group as a whole and forges a new message out of it. We look at the release and processor strategies in detail in the next few paragraphs.

The release strategy of an Aggregator should be to release the group only when the processing is complete (or if it times out). The `SequenceSizeReleaseStrategy` implementation handles this behavior. For this common case, the `MessageGroup` has an `isComplete()` method, the default implementation of which compares the sequence size header to the size of the group. This is convenient if you're implementing a custom release strategy but still are interested in the default completeness of the group.

The message group processor of an Aggregator should turn all the messages of a group into a single aggregated message and send it off to the output channel. The most common implementation used is the `MethodInvokingMessageGroupProcessor`, which wraps around a method. The method should have the following signature (pointcut expression language):

```
* * (List)
```

Similar to other implicit conversions to and from messages in the framework, Spring Integration automatically unwraps the elements in the list if they're not a message. The return value is wrapped in a message if needed and sent to the output channel of the Aggregator.

The Resequencer example follows the same lines as the Aggregator with two main differences. First, the messages from an incomplete group may already be released. Second, the processor is expected to return the same messages that came in.

By default, the release strategy used by the Resequencer is also the `SequenceSizeReleaseStrategy`. In the case of a Resequencer, the `releasePartialSequences` flag can be set. This flag allows the release strategy to release parts of an incomplete sequence that are in the right order to allow for a smoother message flow.

The message processor of a Resequencer takes all the messages in the group, orders them, and then sends all the messages that form a sequence to the output channel. The main customization is to supply a different comparator for the ordering so sequence numbers can be avoided.

In summary, there is one central component, `CorrelatingMessageHandler`, uses several strategies to delegate its work. `CorrelationStrategy` is used to find the Correlation ID of the message group, and `MessageGroupStore` is used to store the message group. To decide when to release the group for processing, a `ReleaseStrategy` is used. A `MessageProcessor` finally deals with the messages. Implementations of these strategies together form the different correlating endpoints.

7.5 Summary

In this chapter you learned to deal with Splitters, Aggregators, and Resequencers. You also saw examples of some nontrivial Aggregator use cases and finally looked at the design that's at the core of Spring Integration. Let's review what you learned about Splitters first.

- One message goes in; many messages come out.
- The output can be based on the payload but also other criteria, such as headers.
- The Splitter will sets a Correlation ID, sequence size, and sequence number for each message.

The chapter also discussed endpoints that group messages together before sending reply messages. Correlation, the basis for both aggregating and resequencing, was examined in detail. The following points are relevant to remember:

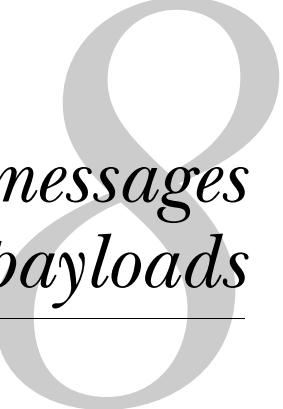
- CorrelationStrategy finds the correlation key, which is based on a message and doesn't have to be the Correlation ID (for example, as set by a splitter).
- MessageGroupProcessor determines what happens in reaction to the release of a group.
- ReleaseStrategy determines when a group is released. A group can be released multiple times.
- MessageGroupStore stores the messages until they are processed.

The Aggregator uses a processor that aggregates the messages. By default, its correlation and release strategies are complementary to the Splitter.

The Resequencer processes messages by reordering them. The correlation and release strategies are similar to those of the Aggregator with the exception of releasing partial sequences. When partial sequences are released, multiple releases of the same group may happen.

Now that you've read this chapter, you should have a clear idea how Spring Integration can help you when you need to split up some work or aggregate the results of some operations back together. Aggregation is a particularly complex use case that often differs subtly from the examples found in this book or online. In some cases it pays to write a custom solution. It's particularly important here to consider carefully the pros and cons of extending the framework versus inventing your own.

This concludes part 1 of the book. You now know how the core of Spring Integration works. We reviewed all the main components in the core and showed you several examples of messaging applications using the components. But this is only the foundation. The interesting part comes when you start integrating with remote systems and look beyond the walls of the JVM. In the first chapter of part 2, you'll work with XML, because it's the ubiquitous language of system integration. Chapter 8 shows you concepts from this and previous chapters reused in the context of XML payloads, such as the XPath splitter and the XPath router. From there, we'll look at many different integration possibilities. Read on!



Handling messages with XML payloads

This chapter covers

- Why you might want to use XML messages
- Transforming between XML and Java
- Transforming XML using XSLT
- Routing and splitting with XPath
- Validating XML messages

Chapter 6 discussed routing, and you saw that the flow of a message through the system, or the chain of events caused by the arrival of a message, is usually dependent on its payload. We discussed in chapter 5 that transforming payloads to and from an intermediate format, such as XML, is essential to integrating different systems with each other.

There are many exchange formats in the wild, but none is as widespread as XML. Virtually all programming languages have some sort of XML support. From JavaScript to PL/SQL (Procedural Language/Structured Query Language), from Smalltalk to Scala, all these languages share knowledge of this common vocabulary.

Integration components generally don't need detailed knowledge of the domain, but they do need to react to the contents of a message. If both input and output are XML messages, and simple message processing is all that's required, marshalling the message (as we discussed in chapter 5) will cause unnecessary complexity and overhead. In such cases it's better to work on the XML directly, using, for example, XPath or *Extensible Stylesheet Language Transformation (XSLT)*. Where more complex processing is required, it's often preferable to convert the XML message to a domain object, so this chapter also looks at the *Object/XML Mapping (OXM)* support provided by the frameworks.

Because XML is the most commonly used interoperable format, it makes sense to equip an integration framework with support for this language. We discuss the details of Spring Integration's XML support, including the different payload types supported by the provided XML endpoints. In common with the rest of the framework, the XML module is rooted in the enterprise integration patterns and provides support for those patterns to make working with XML simple.

This chapter introduces the XML capabilities of the Spring Integration framework using an example how to implement a message flow using the provided XML support. The chapter finishes with a detailed explanation of the approach taken in the implementation of these endpoints along with details of extension points for advanced usage.

8.1 **XML messaging**

In our Spring Integration travel example application, a quotation request for a trip may contain multiple legs. Once the trip is split into legs, the message flow needs to generate quotation requests for the different elements of the trip, such as the hotel accommodation and car rental. To do this, you must separate the constituent parts of the trip relating to hotel, flight, and car rental before transforming the internal canonical representation to one that is understood by the third-party systems. One approach is to maintain a number of Java payload types representing the request formats of each individual system and a set of transformation between the canonical form and the third-party form. This task can become unwieldy as the number of systems with which you integrate grows. The request format of the third-party system is likely to be some form of XML, so the process of generating a quote for each leg of the trip should be implemented as an XML-based message flow.

The trigger for the leg-quote-generation flow is the receipt of a message containing a `LegQuoteCommand`. It contains the requirements for the start and end locations of the leg, the desired dates for travel, and additional specification for hotel and car rental if required. On receipt of this message, the system first converts the message into the canonical XML form of a trip leg. XSLT is then used to add the leg information to each criteria section prior to splitting. Having duplicated the leg information to each criteria section, you can then split the document into separate requests. Since the requests are now self-contained, they can then be processed in parallel, a common

strategy for speeding up the overall processing of the request. In this case the splitting is achieved by the use of XPath, which splits the document into one document per child of the `legQuote` element. The leg quote flow can then process each request in parallel and route it to the appropriate third-party system. Again, this is done using an XPath expression, which routes the message according to the `route` element name. Finally, before making a request to the third-party system, you validate that each request conforms to the schema that defines the third-party request format. This is achieved by using the out-of-the-box support for validating messages against an XML schema definition.

The next section discusses the support for marshalling, which allows you to convert the payload of the message into an XML form ready for subsequent processing and dispatch to the external systems.

8.1.1 Marshalling LegQuoteCommand into XML

To convert your message between XML and objects, you use OXM. This conversion process is commonly called *marshalling* and *unmarshalling*. Spring provides a generic technology agnostic abstraction of the process of mapping between Java and XML in the Spring OXM module. Spring OXM provides a common interface across a number of leading Java OXM solutions. By providing a common abstraction, the OXM module decouples code from the implementation details and provides a consistent exception hierarchy regardless of the OXM technology being used. This decoupling is achieved by encapsulating the mapping process behind implementations of the `Marshaller` interface, which provides mapping from `Object` to `XMLResult`, and the `Unmarshaller` interface, which maps from a `Source` to the `Object` representation. Both these interfaces are shown in the following code snippets and in figure 8.1.¹

```
package org.springframework.oxm;
public interface Marshaller {
    boolean supports(Class<?> clazz);
    void marshal(Object graph, Result result)
        throws IOException, XmlMappingException;
}

package org.springframework.oxm;
public interface Unmarshaller {
    boolean supports(Class<?> clazz);
    Object unmarshal(Source source)
        throws IOException, XmlMappingException;
}
```

¹ Spring OXM was developed as part of the Spring WS project. As of Spring 3.0, the OXM framework is part of the core Spring framework.

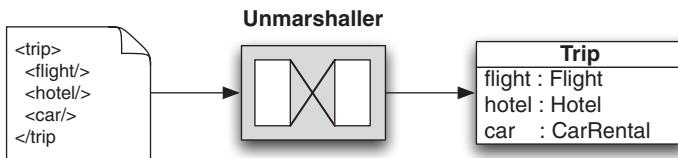


Figure 8.1 The trip unmarshaller implements the **Unmarshaller** interface and can convert trip XML messages to instances of the **Trip** class.

ANNOTATING DOMAIN CLASSES FOR USE WITH JAXB

For the leg quote flow, you can use the Java Architecture for XML Binding (JAXB) v2 OXM technology in conjunction with Spring OXM. Because the sample already contains a set of existing rich domain classes, you can annotate the domain classes with JAXB annotations rather than generate the classes from an XML schema. Following is the annotated LegQuoteCommand, which maps to the root of the canonical XML leg quote document. To indicate that this maps to the root of an XML document, you annotate it with the `@XmlRootElement` annotation. You also specify the name of the XML element with the `name` attribute of the annotation. This contains fields representing the criteria for each of the constituent parts of the leg quote.

```

package sria.booking.domain.trip;

import sria.booking.domain.Command;
import sria.booking.domain.car.CarCriteria;
import sria.booking.domain.flight.FlightCriteria;
import sria.booking.domain.hotel.HotelCriteria;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "legQuote")
public class LegQuoteCommand implements Command {

    private Leg leg;

    private HotelCriteria hotelCriteria;

    private FlightCriteria flightCriteria;

    private CarCriteria carCriteria;

    /**
     * Private constructor for use by JAXB
     */
    private LegQuoteCommand() {
    }

    public LegQuoteCommand(Leg leg) {
        this.leg = leg;
    }

    //setters and getters omitted
}

```

By default, JAXB outputs all fields of the annotated type, although it can only convert standard Java types out of the box. This functionality can be extended by providing `XmlAdapter` implementations capable of mapping types unsupported by JAXB to types supported by JAXB. An example is the `Leg` class, which contains start and end dates and locations. This class uses the Joda Time project (<http://joda-time.sourceforge.net>) to represent dates. The following code shows use of the `@XmlJavaTypeAdapter` annotation to reference a custom adapter capable of converting the `DateTime` instance to a recognized instance of `java.util.Calendar`.

```
package sria.booking.domain.trip;

import sria.booking.domain.Location;
import sria.booking.domain.binding.JodaDateTimeAdapter;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.joda.time.DateTime;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlRootElement
public class Leg {

    @XmlElement
    @XmlJavaTypeAdapter(JodaDateTimeAdapter.class)
    private DateTime startOfLegDate;

    @XmlElement
    @XmlJavaTypeAdapter(JodaDateTimeAdapter.class)
    private DateTime endOfLegDate;

    @XmlElement
    private Location startLocation;

    @XmlElement
    private Location endLocation;

    private Leg() {
    }

    public Leg(DateTime startOfLeg, DateTime endOfLeg,
               Location startLocation, Location endLocation) {
        this.startOfLegDate = startOfLeg;
        this.endOfLegDate = endOfLeg;
        this.startLocation = startLocation;
        this.endLocation = endLocation;
    }

    ...
}
```

The custom adapter is an implementation of `javax.xml.bind.annotation.adapters.XmlAdapter`, shown in the following snippet. This adapter converts between the date types using coordinated universal time (UTC).

```

package sija.booking.domain.binding;

import org.joda.time.DateTime;
import org.joda.time.chrono.ISOChronology;

import javax.xml.bind.annotation.adapters.XmlAdapter;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;

public class JodaDateTimeAdapter extends
        XmlAdapter<Calendar, DateTime> {
    @Override
    public DateTime unmarshal(Calendar cal) throws Exception {
        return new DateTime(cal.getTime(), ISOChronology.getInstanceUTC());
    }

    @Override
    public Calendar marshal(DateTime dt) throws Exception {
        GregorianCalendar cal = new GregorianCalendar();
        cal.setTimeInMillis(dt.getMillis());
        cal.setTimeZone(TimeZone.getTimeZone("UTC"));
        return cal;
    }
}

```

CONFIGURING THE SPRING INTEGRATION MARSHALLER ENDPOINT

The Spring Integration marshalling and unmarshalling support is a fairly thin adapter layer on top of Spring OXM, so you first configure an instance of marshaller or unmarshaller as appropriate. The out-of-the-box implementations all implement both the Marshaller and Unmarshaller interfaces. The following code example shows how to configure an instance of Jaxb2Marshaller, providing a list of the annotated types, in this case just the LegQuoteCommand.

```

<beans:bean id="legMarshaller"
            class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
    <beans:property name="classesToBeBound"
                    value="sija.booking.domain.trip.LegQuoteCommand"/>
</beans:bean>

```

On top of this, you can configure a marshalling endpoint to convert from the LegQuoteCommand to a canonical XML form using the Spring Integration XML namespace, as in the next example.

```

<beans:beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.springframework.org/schema/integration"
    xmlns:xi-xml="http://www.springframework.org/schema/integration/xml"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/
            integration/spring-integration.xsd
        http://www.springframework.org/schema/integration/xml"

```

```
http://www.springframework.org/schema/
    ↵integration/xml/spring-integration-xml.xsd">

<channel id="javaLegQuoteCommands" />

<si-xml:marshalling-transformer input-channel="javaLegQuoteCommands"
    output-channel="xmlLegQuotes"
    marshaller="legMarshaller"
    result-transformer="resultToDocumentTransformer" />

<beans:bean id="resultToDocumentTransformer"
    class="org.springframework.integration.xml.transformer
        ↵.ResultToDocumentTransformer" />
</beans:beans>
```

The Spring Integration marshalling transformer requires an instance of OXM Marshaller. Where it's used outside the scope of a chain, it also requires an input channel and output channel. Looking at the standard Marshaller interface, you can see that the marshaller converts the Object graph to an instance of Result. Because the Spring Integration marshaller's primary purpose is to be a thin adapter layer on top of the OXM support, this behavior is the same behavior as you get with Spring Integration by default. If you were to configure the marshaller with nothing else but an instance of Marshaller and references to the input and output channels, the output of the marshaller would be a message containing a payload of type Result. Looking at the javax.xml.transform.Result interface, it quickly becomes apparent that from the perspective of wanting to process the XML data itself, receiving an instance of Result isn't ideal. In fact, without casting the Result, the only thing you can do is get or set an optional system ID, which may be used for error messages or to relate to a file on the local filesystem. To avoid the proliferation of tests to determine the type of Result and casts to allow something useful to be done with it, most of the provided XML endpoints attempt some form of type negotiation whereby the payload output type is determined by the input. For example, if you pass in a String, most provided XML endpoints will pass back a String. The supported payload types and strategies for determining output type are discussed later in this chapter in the "Under the hood" section. The marshaller is different: because you're passing in an Object and asking for the Object as XML, the marshalling endpoint has no clue how you want the XML. Do you want a String, an org.w3c.dom.Document, or something else? To obey the principle of least surprise, it passes back a Result to be consistent with the underlying OXM marshaller. To make this payload more useful in this case, you provide an optional collaborator of type ResultTransformer to convert the Result to something more useful. You set the optional result-transformer attribute of the marshalling-transformer element with a reference to the ResultTransformer. Out of the box, two implementations of ResultTransformer are provided, one to convert to a Document and the other to convert to a String.

Following is an example of the XML output you'd expect to see for a sample message passed into the marshaller.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<legQuote>
    <carCriteria>
        <carType>Compact</carType>
    </carCriteria>
    <flightCriteria>
        <requiredSeatClass>Business</requiredSeatClass>
        <returnRequired>true</returnRequired>
    </flightCriteria>
    <hotelCriteria>
        <roomType>Double</roomType>
        <smokingRoom>false</smokingRoom>
    </hotelCriteria>
    <leg>
        <startOfLegDate>2010-01-03T00:00:00Z</startOfLegDate>
        <endOfLegDate>2010-01-07T00:00:00Z</endOfLegDate>
        <startLocation city="London" countryCode="UK" />
        <endLocation city="Buenos Aires" countryCode="AR" />
    </leg>
</legQuote>
```

Now that you have the message with an XML payload, you're ready to transform this output into something suitable for splitting and dispatching. The following section explains how the transformation can be achieved using XSLT.

8.1.2 Enriching the leg quote using XSLT

Now that the leg quote is in the canonical form, the next step is to transform the XML document to ensure that it can then be split into three separate requests ready to send to the external systems. These external systems then provide the quotes for car rental, hotels, and flights. Currently, the elements containing information about start and end locations and dates are child element of the root `legQuote` element. Therefore, the next step in the message flow is to use an XSLT to add the leg information to the criteria elements and rename them. To achieve this, you use XSLT with the following XSL.

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="/legQuote">
        <legQuote>
            <flightQuote>
                <xsl:copy-of select="flightCriteria/node()" />
                <xsl:copy-of select="leg/node()" />
            </flightQuote>
            <hotelQuote>
                <xsl:copy-of select="hotelCriteria/node()" />
                <xsl:copy-of select="leg/node()" />
            </hotelQuote>
            <carQuote>
                <xsl:copy-of select="carCriteria/node()" />
                <xsl:copy-of select="leg/node()" />
            </carQuote>
        </legQuote>
    </xsl:template>
</xsl:stylesheet>
```

```

    </legQuote>
</xsl:template>
</xsl:stylesheet>
```

Using the Spring Integration XML namespace, you configure an XSLT endpoint, as shown in the next code example. When configuring the XSLT endpoint, it's possible to provide an instance of the Spring org.springframework.core.io.Resource. The endpoint uses this resource to construct an instance of javax.xml.transform.Templates, which can then be used safely by concurrent threads to create javax.xml.transform.Transformer instances. The alternative is to instantiate the Templates instance directly and pass a reference into the transformer where customization of the template creation is required.

```

<channel id="xmlLegQuotes"/>

<si-xml:xslt-transformer input-channel="xmlLegQuotes"
    output-channel="xslTransformedLegQuote"
    xsl-resource="classpath:/xsl/enrichCriteriaWithLeg.xsl"/>
```

Passing the example message through this transformation produces the following XML, which is now suitable for splitting into the three separate parts of the leg to generate the three quotes (for car rental, hotels, and flights).

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<legQuote>
    <flightQuote>
        <requiredSeatClass>Business</requiredSeatClass>
        <returnRequired>true</returnRequired>
        <startOfLegDate>2010-01-03T00:00:00Z</startOfLegDate>
        <endOfLegDate>2010-01-07T00:00:00Z</endOfLegDate>
        <startLocation city="London" countryCode="UK"/>
        <endLocation city="Buenos Aires" countryCode="AR"/>
    </flightQuote>
    <hotelQuote>
        <roomType>Double</roomType>
        <smokingRoom>false</smokingRoom>
        <startOfLegDate>2010-01-03T00:00:00Z</startOfLegDate>
        <endOfLegDate>2010-01-07T00:00:00Z</endOfLegDate>
        <startLocation city="London" countryCode="UK"/>
        <endLocation city="Buenos Aires" countryCode="AR"/>
    </hotelQuote>
    <carQuote>
        <carType>Compact</carType>
        <startOfLegDate>2010-01-03T00:00:00Z</startOfLegDate>
        <endOfLegDate>2010-01-07T00:00:00Z</endOfLegDate>
        <startLocation city="London" countryCode="UK"/>
        <endLocation city="Buenos Aires" countryCode="AR"/>
    </carQuote>
</legQuote>
```

Having now enriched the message, you're ready to split it into parts representing each of the quotes you need. The next section demonstrates Spring Integration's support for the Splitter pattern using XPath to express how you want to split your XML message.

8.1.3 XPath support

Much of the support for processing XML payloads utilizes XPath expressions within the configuration. Spring Integration builds on the XPath support provided by the Spring Web Services project and adds additional namespace support to make working with XPath-based components easier in the context of Spring Integration. XPath expressions can be defined either within the definition of the endpoint using the expression or, as in the following example, as a top-level bean that can then be referenced in multiple places.

```
<si-xml:xpath-expression id="simpleXPath" expression="//hotel" />
```

Because XML namespaces are commonly used to avoid element name clashes, XPath expressions generally must incorporate a namespace. Namespaces can be incorporated in one of two ways. In the simple case of only one namespace, the namespace can be the URI and prefix, or alternatively, it can be referenced as a map of namespaces.

```
<si-xml:xpath-expression id="simpleXPathNs" expression="//hotel"
    ns-prefix="hb" ns-uri="http://www.example.org/siia/hotel-booking" />

<si-xml:xpath-expression id="moreComplexXPathNs"
    expression="/trip:booking/hb:hotel" namespace-map="namespaceMap" />

<util:map id="namespaceMap">
    <entry key="trip" value="http://www.example.org/siia/trip" />
    <entry key="hb" value="http://www.example.org/siia/hotel-booking" />
</util:map>
```

Given this support for XPath, you can use XPath expressions to split a message into a number of parts for separate processing, as shown in the following section.

8.1.4 Splitting hotel, car rental, and flight quotes

Now you have a quote in a form that allows you to split it into three self-contained requests using an XPath expression that defines how you want to split the document. The provided XPath splitter implementation evaluates the provided XPath expression to create a node set. Each node within the node set then becomes the payload of a new message. To split the leg quote, an XPath expression evaluates to give the direct children of the `legQuote` element, which is simply `/legQuote/*`. Figure 8.2 illustrates the configuration for the XPath splitter that separates the different types of quote request.

```
<si-xml>xpath-splitter create-documents="true"
    input-channel="xslTransformedLegQuote" output-channel="splitQuotes">
    <si-xml: xpath-expression expression="/legQuote/*" />
</si-xml>xpath-splitter>
```

By default, the XPath splitter creates messages, each containing one of the nodes returned by the XPath evaluation. To isolate the message payloads from the split document, the `create-documents` flag can be set. This creates new documents for each of the nodes returned by the XPath evaluation and imports the node to the new document. Importing the node isolates the new document from the source document.

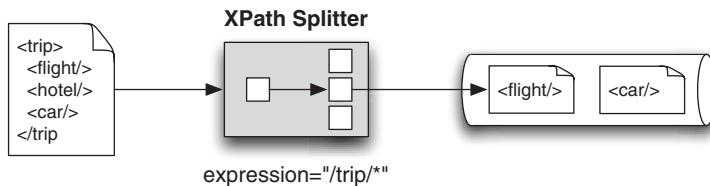


Figure 8.2 Splitting our request into the constituent parts

because the node is copied and becomes the root element of the new document. In this case, the carQuote, flightQuote, and hotelQuote elements in the postsplit messages are no longer children of the legQuote element. This implies that changes made to any of the documents created by the XPath splitter won't be seen in the original document.

8.1.5 Routing messages based on their XML payloads

Now you have your three separate requests; the next thing you need to do is route the requests to the different third-party systems for processing. Since each request type has a well-known root element, you can use the provided XPath router to route according to the root element name, as illustrated in figure 8.3.

The configuration for the XPath router is relatively simple and requires, as a minimum, an input channel and an XPath expression. In the following configuration, you also provide a series of channel mappings. As you learned in chapter 6, if no such mapping is provided, the default strategy is to resolve the value returned by the expression as a bean ID of a channel defined within the application context. It may be possible to use this strategy, but the additional level of indirection provided by the mappings allows the use of meaningful channel names rather than channel names that reflect the structure of the XML messages being processed. Please note that as an alternative to specifying the mappings in the router configuration, you can inject a channel resolver into the router in a similar way to the router element from the core namespace.

```
<si-xml:xpath-router id="quoteRequestRouter"
                      input-channel="splitQuotes"
                      evaluate-as-string="true">
    <si-xml>xpath-expression expression="local-name(/*)"/>
    <si-xml:mapping value="carQuote" channel="carQuoteChannel"/>
    <si-xml:mapping value="flightQuote"
                    channel="flightQuoteChannel"/>
    <si-xml:mapping value="hotelQuote" channel="hotelQuoteChannel"/>
</si-xml:xpath-router>
```

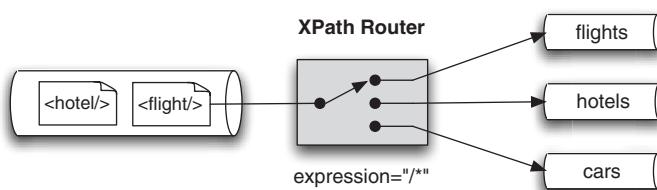


Figure 8.3 Routing the parts with XPath

To ensure the correct operation of your system, you need to apply a somewhat defensive approach to the messages you receive. The next section shows you how to validate an XML message to ensure you can successfully process it.

8.1.6 Validating XML messages

It's important to ensure that only valid requests are sent to third-party systems. As a final step before transmitting the XML, you validate the quote requests produced by the leg quote message flow. To carry out XML validation, you use the provided validating router implementation, which can validate XML payloads against a schema definition provided in either Regular Language for XML Next Generation (RELAX NG) or XML Schema format. The validation support in the XML module takes on the form of a router implementation that routes to either a valid or invalid channel according to the outcome of the schema validation. Following is the XML Schema definition for validating flight quote requests.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema attributeFormDefault="unqualified"
            elementFormDefault="qualified"
            xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="flightQuote" type="flightQuoteType"/>
    <xs:complexType name="endLocationType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute type="xs:string" name="city"/>
                <xs:attribute type="xs:string"
                              name="countryCode"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:complexType name="flightQuoteType">
        <xs:sequence>
            <xs:element type="xs:string"
                        name="requiredSeatClass"/>
            <xs:element type="xs:string"
                        name="returnRequired"/>
            <xs:element type="xs:dateTime"
                        name="startOfLegDate"/>
            <xs:element type="xs:dateTime"
                        name="endOfLegDate"/>
            <xs:element type="startLocationType"
                        name="startLocation"/>
            <xs:element type="endLocationType"
                        name="endLocation"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="startLocationType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute type="xs:string"
                              name="city"/>
                <xs:attribute type="xs:string"
                              name="countryCode"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

```

```
</xs:extension>
</xs:simpleContent>
</xs:complexType>
</xs:schema>
```

Plugging that code into the pipeline is simple, as shown in the following configuration. The schema is loaded as a resource, and references to the input channel and valid and invalid channels are configured.

```
<si-xml:validating-router id="flightQuoteValidator"
    input-channel="flightQuoteChannel"
    valid-channel="validFlightQuoteChannel"
    invalid-channel="invalidRequests"
    schema-location="classpath:xsd/flightQuote.xsd" />
```

XML processing done badly can result in poor performance. Let's look under the hood at the different forms of XML messages that Spring Integration supports and learn how to make more informed choices.

8.2 Under the hood

Spring Integration tries to simplify the work of the application developer by addressing concerns such as the creation of objects that represent XML (for example, `org.w3c.dom.Document` and `javax.xml.transform.Result`). It also takes care of conversion between different representations of XML, such as `String`, `Document`, `Result`, and `Source`. This generally makes the life of the developer easier, but knowing what the framework is doing and how to change the default strategies is useful too. For example, passing around XML as a `String` can be convenient because strings are easy to create and easy to log and debug. In a high-performance application, though, it's important to understand that extensively using XPath evaluation on strings involves a considerable amount of on-the-fly conversion between a `String` and a Document Object Model (DOM) representation.

When dealing with large XML payloads, tree-based parsers (DOM parsers) tend to induce poor performance characteristics in an application. The first thing people do to improve this is to move to a streaming parser (Simple API for XML [SAX], Streaming API for XML [StAX]). Unfortunately, streaming and messaging don't mix well. A stream is a leaky abstraction under messaging, and it'll break or cripple many serialization options. To say either streaming or messaging is the way to go would be wrong, so when you need streams, design to use them without the need for stream references to travel through the messaging system. Spring Integration's XML support steers clear of streaming for this reason.

The following section exposes the internal workings of the Spring Integration XML module with regard to XML payload types. It explains the implications of using different payload types and how to customize the framework behavior if needed.

8.2.1 Supported payload types and return type matching

One of the main challenges when considering the XML module for Spring Integration was deciding which of the many APIs for working with XML in Java to support. It's

possible to interact with data expressed as XML from Java in many different ways, for example, by using SAX events, parsing into a tree representation such as `org.w3c.dom.Document`, or using one of the additional projects (for example, Dom4j or JDom) that aim to make working with XML from Java simpler and more intuitive. Currently, Spring Integration XML support is focused on payloads expressed as `org.w3c.dom.Document`, `java.lang.String`, and some implementations of `javax.xml.transform.Source`, including `org.springframework.xml.transform.StringSource` and `javax.xml.transform.dom.DOMSource`. The development team had a number of reasons for focusing primarily on string representations that could easily be converted into DOM on the fly and on DOM itself. Using JAXP and `org.w3c.dom` packaged classes minimizes the need for third-party libraries and also keeps the implementation relatively simple. In addition, many of the patterns supported, such as Splitter and Router, rely on XPath evaluation, and the fact that the standard implementations of XPath require a fully built document to evaluate against was another reason to initially focus on DOM rather than event-based processing of XML payloads.

The downside of focusing on DOM and string representations exclusively is that processing very large XML documents isn't currently well supported because they result in large in-memory representations. This has clear implications for concurrency and the maximum size of document Spring Integration can support. In the future, support for streaming through StAX and/or SAX may be added to Spring Integration.

8.3 **Summary**

This chapter introduced the capabilities that the Spring Integration XML module provides for working with XML payloads. We used a style that complements the POJO and Spring namespace approach of the framework as a whole. Using XML payloads shouldn't be the default for most projects, but it's a valuable tool for projects in which transforming received XML messages into Java representations is an unnecessary overhead.

This chapter also showed you how to use XPath support to implement content-based routing and filtering. You learned how to split XML messages using XPath expressions.

We also demonstrated Spring Integration's support for XSLT transformations by duplicating information in the XML message before splitting it.

In chapter 9, you'll see how Spring Integration's support for Java Message Service (JMS) makes it trivial to implement request/reply functionality between systems using JMS. Later, in chapter 12, we return to XML, showing support for Web Services that may be more suitable for integration with systems written in other languages.



Spring Integration and the Java Message Service

This chapter covers

- How Spring Integration and JMS fit together
- Sending and receiving JMS Messages
- Gateways and Channel Adapters

For many Java developers, the first thing that comes to mind when they hear “messaging” is the *Java Message Service (JMS)*. That’s understandable considering it’s the predominant Java-based API for messaging and sits among the standards of the Java Enterprise Edition (JEE). The JMS specification was designed to provide a general abstraction over message-oriented middleware (MOM). Most of the well-known vendor products for messaging can be accessed and used through the JMS API. A number of open-source JMS implementations are also available, one of which is *ActiveMQ*, a pure Java implementation of the JMS API. We use ActiveMQ in some of the examples in this chapter because it’s easy to configure as an embedded broker. We don’t go into any specific ActiveMQ details, though. If you want to learn more about it, please refer to *ActiveMQ in Action* by Bruce Snyder, Dejan Bosnjanac, and Rob Davies (Manning, 2011).

Hopefully, by this point in the book, you realize that messaging and event-driven architectures don't necessarily require the use of such systems. We've discussed messaging in several chapters thus far without having to dive into the details of JMS, which reveals that Spring Integration can stand alone as a framework for building messaging solutions. In a simple application with no external integration requirements, producer and consumer components may be decoupled by Message Channels so that they communicate only with messages rather than with direct invocation of methods with arguments. Messaging is really a paradigm; the same underlying principles apply whether messaging occurs between components running within the same process or between components running under different processes on disparate systems.

Nevertheless, by supporting JMS, Spring Integration provides a bridge between its simple, lightweight intraprocess messaging and the interprocess messaging that JMS enables across many MOM providers. In this chapter, you learn how to map between Spring Integration Messages and JMS Messages. You also learn about several options for integrating with JMS messaging destinations. Spring Integration provides Channel Adapters and Gateways as well as Message Channel implementations that are backed by JMS Destinations. In many cases, the configuration of these elements is straightforward. But, to get the most benefit from the available features, such as transactions, requires a thorough understanding of the underlying JMS behavior as dictated by the specification. Therefore, in this chapter, we alternate between the Spring Integration role and the specific JMS details as necessary.

9.1 ***The relationship between Spring Integration and JMS***

Spring Integration provides a consistent model for intraprocess and interprocess messaging. The primary role of Channel Adapters and Messaging Gateways is to connect a local channel to some external system without impacting the producer or consumer components' code. Another benefit the adapters provide is the separation of the messaging concerns from the underlying transports and protocols. They enable true document-style messaging whether the particular adapter implementation is sending requests over HTTP, interacting with a filesystem, or mapping to another messaging API. The JMS-based Channel Adapter and Messaging Gateways fall into that last category and are therefore excellent choices when external system integration is required. Given that the same general messaging paradigm is followed by Spring Integration and JMS, we can conceptualize the intraprocess and interprocess components as belonging to two layers but with a consistent model. See figure 9.1.

Even though we may focus on external system integration when discussing the benefits of JMS, there are benefits to using JMS internally within an application. JMS may be useful between producers and consumers running in the same process because a JMS provider can support persistence, transactions, load balancing, and failover. For this reason, Spring Integration provides a Message Channel implementation that delegates to JMS behind the scenes. That channel looks like any other channel as far as the message-producing and message-consuming components are concerned, so it can be used as an alternative at any point within a message flow as shown in figure 9.2.

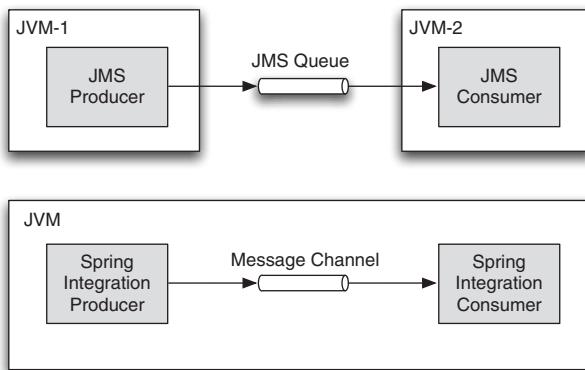


Figure 9.1 The top configuration shows interprocess integration using JMS. The bottom configuration shows intraprocess integration using Spring Integration. Which type of integration is appropriate depends on the architecture of the application.

Even for messaging within a single process, the use of a JMS-backed channel provides several benefits. Consider a Message Channel backed by a simple in-memory queue, as occurs when using the `<queue>` subelement within a `<channel>` without referencing any `MessageStore`. By default, such a channel doesn't *persist* Messages to a transactional resource. Instead, the Messages are only stored in a volatile map such that they can be lost in the case of a system failure. They'll even be lost if the process is shut down intentionally before those messages are drained from the queue by a consumer. In certain cases, when dealing with real-time event data that doesn't require persistence, the loss of those Event Messages upon process termination might not be a problem. It may be well worth the trade-off for asynchronous delivery that allows the producer and consumer to operate within their own threads, on their own schedules. With these Message Channels backed by a JMS Destination, though, we can have the best of both worlds. If persistence and transactions *are* important, but asynchronous delivery is a desired feature, then these channels offer a good choice even if they're only being used by producers and consumers in the same process space.

The main point here is that even though we often refer to JMS as an option for messaging between a number of individual processes, that's not the *only* time to consider JMS or other interprocess broker-based messaging solutions, such as Advanced Message Queuing Protocol (AMQP), as an option. When multiple processes are involved, the other advantages become evident. First among these benefits is the natural load balancing that occurs when multiple consuming processes are pulling

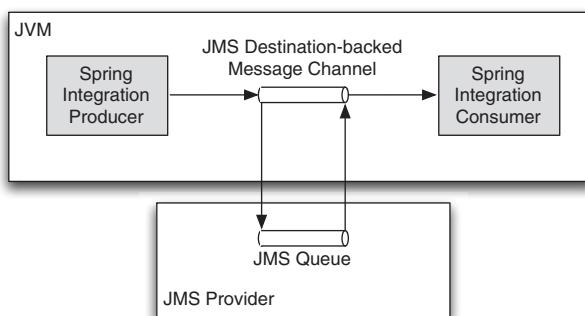


Figure 9.2 Design of the destination-backed channel of Spring Integration. It benefits from the guarantees supported by a JMS implementation, but it hides the JMS API behind a channel abstraction.

messages from a shared destination. Unlike producer-side load balancing, the consumers can naturally distribute the load on the basis of their own capabilities. For example, some processes may be running on slower machines or the processing of certain messages may require more resources, but the consumers only ask for more messages when they can handle them rather than forcing some upstream dispatcher to make the decisions.

The second, related benefit is increased scalability. Message-producing processes might be sending more messages than a single consuming process can handle without creating a backlog, resulting in a constantly increasing latency. By adding enough consuming processes to handle the load, the throughput can increase to the point that a backlog no longer exists, or exists only within acceptable limits in rarely achieved high-load situations that occur during bursts of activity from producers.

The third benefit is increased availability. If a consuming process crashes, messages can still be processed as long as one or more other processes are still running. Even if all processes crash, the mediating broker can store messages until they come back online. Likewise, on the producing side, processes may come and go without directly affecting any processes on the consuming side. This is nothing more than the benefit of loose coupling inherent in any messaging system, applied not only across producers and consumers themselves but the processes in which they run. Keep in mind when we discuss these scenarios where processes come and go, we're not merely talking about unforeseen system outages. It's increasingly common for modern applications to have "zero downtime" requirements. Such an application must have a distributed architecture with no tight coupling between components in order to accommodate planned downtime for system migrations and rolling upgrades.

One last topic we should address briefly here is transactions. We revisit transactions in greater detail near the end of this chapter, but one quick point is relevant to the discussion at hand. In the scenario described previously, where a consuming process crashes or is taken offline while responsible for an in-flight message, transactions play an important role. If the consumer reads a message from a Destination but then fails to process it, such as might occur when its host process crashes, then the message might be lost depending on how the system is configured. In JMS, a variety of options correspond to different points on the spectrum of guaranteed delivery. One configuration option is to require an explicit acknowledgment from the consumer. It might be that a consumer acknowledges each message after it successfully stores it on disk. A more robust option is to enable transactions. The consumer would commit the transaction only upon successful processing of the message, and it would roll back the transaction in case of a known failure. When this functionality is added to the example, not only do the multiple consuming processes share the load, they can even cover for each other in the face of failures. One consumer may fail while a message is in flight, but its transaction rolls back. The message is then made available to another consuming process rather than being lost.

Table 9.1 provides a quick overview of the benefits of using JMS with Spring Integration.

Table 9.1 Benefits of using JMS with Spring Integration

Benefit	Description
Load balancing	Multiple consumers in separate virtual machine processes pull Messages from a shared destination at a rate determined by their own capabilities.
Scalability	Adding enough consumer processes to avoid a backlog increases throughput and decreases response time.
Availability	With multiple consumer processes, the overall system can remain operational even if one or more individual processes fail. Likewise, consumer processes can be redeployed one at a time to support a rolling upgrade.

It's worth pointing out that the benefits listed in table 9.1 aren't limited to JMS. Any broker that provides support for reliable messaging across distributed producer and consumer processes would provide the same benefits. For example, Spring Integration 2.1 adds support for RabbitMQ, which implements the AMQP protocol. Using the AMQP adapters would offer the same benefits. Likewise, although not as sophisticated, even using Spring Integration's queue-backed channels along with a MessageStore can provide the same benefits because that too enables multiple processes to share the work. For now, let's get back to the discussion at hand and explore the mapping of Spring Integration Message payloads and headers to and from JMS Message instances.

9.1.1 Mapping between JMS and Spring Integration Messages

When considering interprocess messaging from the perspective of Spring Integration, the primary role of Channel Adapters is to handle all of the communication details so that the component on the other side of the Message Channel has no idea that an external system is involved. That means the Channel Adapter not only handles the communication via the particular transport and protocol being used but also must provide a Messaging Mapper (<http://mng.bz/FIOP>) so that whatever data representation is used by the external system is converted to and from simple Spring Integration Messages. Some of that data might map to the *payload* of such a Message, whereas other parts of the data might map to the *headers*. That decision should be based on the role of the particular pieces of data, keeping in mind that the headers are typically used by the messaging infrastructure, and the payload is usually the business data that has some meaning within the domain of the application. Thinking of a message as fulfilling the Document Message pattern from Hohpe and Woolf's *Enterprise Integration Patterns* (Addison-Wesley, 2010), the payload represents the document, and the headers contain additional metadata, such as a timestamp or some information about the originating system.

It so happens that the construction of a JMS Message, according to the JMS specification, is similar to the construction of a Spring Integration Message. This shouldn't surprise you given that the function of the Message is the same in both cases. It does mean that the Messaging Mapper implementation used by the JMS adapters has a simple role. We'll go into the details in a later section, but for now it's sufficient to point

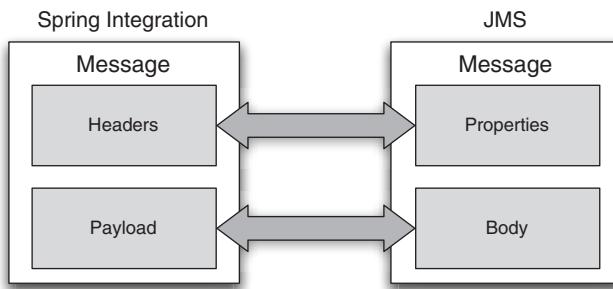


Figure 9.3 Spring Integration and JMS Messages in a side-by-side comparison. The terminology is different, but the structure is the same.

out that there are merely some differences in naming. In JMS, the Message has a *body*, which is the counterpart of a payload in Spring Integration. Likewise, a JMS Message's *properties* correspond to a Spring Integration Message's headers. See figure 9.3.

9.1.2 Comparing JMS Destinations and Spring Integration Message Channels

By now you're familiar with the various Message Channel types available in Spring Integration. One of the most important distinctions we covered is the difference between point-to-point channels and publish-subscribe channels. You saw that when it comes to configuration, the default type for a channel element in XML is point-to-point, and the publish-subscribe channel is clearly labeled as such. The JMS specification uses *Destination* instead of *Message Channel*, but it makes a similar distinction. The two types of JMS Destination are Queues and Topics. A JMS Queue provides point-to-point semantics, and a Topic supports publish-subscribe interaction. When you use a Queue, each Message is received by a single consumer, but when you use a Topic, the same Message can be received by multiple consumers. See table 9.2 for the side-by-side comparison.

Now that we've discussed the relationship between Spring Integration and JMS at a high level, we're almost ready to jump into the details of Spring Integration's JMS adapters. First, it's probably a good idea to take a brief detour through the JMS support in the core Spring Framework. For one thing, the Spring Integration support for JMS builds directly on top of Spring Framework components such as the `JmsTemplate` and the `MessageListener` container. Additionally, the general design of Spring Integration messaging endpoints is largely modeled after the Spring JMS support. You should be able to see the similarities as we quickly walk through the main components and some configuration examples in the next section.

EIP	JMS
Channel	Destination
Point-to-point channel	Queue
Publish-subscribe channel	Topic

Table 9.2 Comparing enterprise integration patterns (EIP) to JMS

9.2 JMS support in the Spring Framework

The logical starting point for any discussion of the Spring Framework's JMS support is the `JmsTemplate`. This is a convenience class for interacting with the JMS API at a high level. Those familiar with Spring are probably already aware of other templates, such as the `JdbcTemplate` and the `TransactionTemplate`. These components are all realizations of the Template pattern described in the Gang of Four's *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et al., Addison-Wesley, 1994). Each of these Spring-provided templates satisfies the common goal of simplifying usage of a particular API. One quick example should be sufficient to express this idea. First, we look at code that doesn't use the `JmsTemplate` but instead performs all actions directly with the JMS API. Note that even a simple operation such as sending a text-based message involves a considerable amount of boilerplate code. Here's a simple send-and-receive echo example:

```
public class DirectJmsDemo {  
  
    public static void main(String[] args) {  
        try {  
            ConnectionFactory connectionFactory =  
                new ActiveMQConnectionFactory("vm://localhost");  
            Connection connection = connectionFactory.createConnection();  
            connection.start();  
            int autoAck = Session.AUTO_ACKNOWLEDGE;  
            Session session = connection.createSession(false, autoAck);  
            Destination queue = new ActiveMQQueue("sia.queue.example1");  
            MessageProducer producer = session.createProducer(queue);  
            MessageConsumer consumer = session.createConsumer(queue);  
            Message messageToSend = session.createTextMessage("Hello World");  
            producer.send(messageToSend);  
            Message receivedMessage = consumer.receive(5000);  
            if (!(receivedMessage instanceof TextMessage)) {  
                throw new RuntimeException("expected a TextMessage");  
            }  
            String text = ((TextMessage) receivedMessage).getText();  
            System.out.println("received: " + text);  
            connection.close();  
        }  
        catch (JMSEException e) {  
            throw new RuntimeException("problem occurred in JMS code", e);  
        }  
    }  
}
```

This code is about as simple as it can get when using the JMS API directly. ActiveMQ enables running an embedded broker (as you can see from the "vm://localhost" URL provided to the `ConnectionFactory`). Many JMS providers would be configured within the Java Naming and Directory Interface (JNDI) registry, and that would require additional code to look up the `ConnectionFactory` and `Queue`. Now, let's see how the same task may be performed using Spring's `JmsTemplate`:

```

public class JmsTemplateDemo {

    public static void main(String[] args) {
        ConnectionFactory connectionFactory =
            new ActiveMQConnectionFactory("vm://localhost");
        JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);
        jmsTemplate.setDefaultDestination(new ActiveMQQueue("sia.queue"));
        jmsTemplate.convertAndSend("hello world");
        System.out.println("received: " + jmsTemplate.receiveAndConvert());
    }
}

```

The code is much simpler, and it also provides fewer chances for developer errors. Any JMS Exceptions are caught and converted into `RuntimeExceptions` in Spring's `JmsException` hierarchy. The JMS resources, such as Connection and Session, are also acquired and released as appropriate. In fact, if a transaction is active when this send operation is invoked, and some upstream process has already acquired a JMS Session, this send operation is executed in the same transactional context. If you've ever worked with Spring's transaction management for data access, this concept should be familiar to you. The idea is roughly the same. If one particular operation in the transaction throws an uncaught `RuntimeException`, all operations that occurred in that same transactional context are rolled back. If all operations are successful, the transaction is committed.

You probably also noticed that the template method invoked is called `convertAndSend` and that its argument is an instance of `java.lang.String`. There are also `send()` methods that accept a JMS Message you've created, but by using the `convertAndSend` versions, you can rely on the `JmsTemplate` to construct the Messages. The conversion itself is a pluggable strategy. The `JmsTemplate` delegates to an instance of the `MessageConverter` interface, and the default implementation (`SimpleMessageConverter`) automatically performs the conversions shown in table 9.3.

The `receiveAndConvert` method performs symmetrical conversion *from* a JMS Message. When used on the receiving end, the `MessageConverter` extracts the JMS Message's body and produces a result with the same mappings shown in the table (for example, `TextMessage` to `java.lang.String`).

Sometimes the default conversion options aren't a good fit for a particular application. That's why the `MessageConverter` is a strategy interface that can be configured

Table 9.3 Default type conversions supported by `MessageConverter`

Type passed to <code>MessageConverter</code>	JMS Message type
<code>java.lang.String</code>	<code>TextMessage</code>
<code>byte[]</code>	<code>BytesMessage</code>
<code>java.util.Map</code>	<code>MapMessage</code>
<code>java.io.Serializable</code>	<code>ObjectMessage</code>

directly on the `JmsTemplate`. Spring provides an Object-to-XML Marshalling (OXM) version of the `MessageConverter` that supports any of the implementations of Spring's `Marshaller` and `Unmarshaller` interfaces within its `toMessage()` and `fromMessage()` methods respectively. For example, an application might be responsible for sending and receiving XML-based text messages over a JMS Queue, but the application's developers prefer to hide the XML marshalling and unmarshalling logic in the template itself. Spring's `MarshallingMessageConverter` may be injected into the `JmsTemplate`, and in turn, that converter can be injected with one of the options supported by Spring OXM, such as Java Architecture for XML Binding (JAXB).

It's also possible to provide a custom implementation of the `Marshaller` and `Unmarshaller` interfaces or even a custom implementation of the `MessageConverter`. For example, you could implement the `MessageConverter` interface to create a `BytesMessage` directly from an object using some custom serialization. That same implementation could then use symmetrical deserialization to map back into objects on the receiving side. Likewise, you might implement the `MessageConverter` interface to map directly between objects and text messages where the actual text content is formatted using JavaScript Serialized Object Notation (JSON).

In this section, you learned how the `JmsTemplate` can greatly simplify the code required to do some basic messaging when compared with using the JMS API directly. The examples covered both sending and receiving, but the receive operations were synchronous. Before we discuss how Spring Integration can simplify things even further, we cover the support for asynchronous message reception in the Spring Framework, which provides the foundation upon which the most commonly used JMS adapters in Spring Integration are built.

9.3 Asynchronous Message reception with Spring

The Polling Consumer and Event-Driven Consumer patterns both make appearances throughout this book. You saw in chapter 3 that even with simple intraprocess messaging in a Spring Integration-based application, each pattern has a role in accommodating various Message Channel options and the use cases that arise from those choices. When dealing with external systems, some transports and protocols are limited to the Polling Consumer pattern. The JMS model enables both polling and event-driven message reception. This section covers the reasons to consider the event-driven approach and how the Spring Framework supports it, ultimately with message-driven plain old Java objects (POJOs) that keep your code simple and unaware of the JMS API.

9.3.1 Why go asynchronous?

Earlier chapters made it clear that receiving Messages is usually more complicated than sending them. Even though the receiving part of the `JmsTemplate` example looks as simple as the sending part, it's important to recognize that in that example, the receive operation is synchronous. The `JmsTemplate` has a `receiveTimeout` property. The `JmsTemplate` receive operations return as soon as a Message is available at the JMS

Destination or the timeout elapses, whichever occurs first. That means that if no Message is available immediately, the operations may block for as long as indicated by the `receiveTimeout` value.

When relying on blocking receive operations, such `JmsTemplate` usage is an example of the Polling Consumer pattern. In an application in which an extremely low volume of messages is expected, polling in a dedicated background thread might be okay. But, as we mentioned earlier in the book, most applications using messaging would prefer to have Event-Driven Consumers.

Support for Event-Driven Consumers could be implemented on top of the simple polling receive calls, but all but the most naive implementations would quickly become complex. A proper and efficient solution requires support for concurrent processing of the received Messages. Such a solution would also support transactions, and ideally, it would accommodate a thread pool that adjusts dynamically according to the volume of messages being received. Those same requirements apply to any attempt to adapt an inherently polling-based source of data to an event-driven one. Obviously, that's a common concern for many components in Spring Integration. To understand the details of how the framework handles those requirements, be sure to read chapters 4 and 15.

As far as the JMS adapters are concerned, the crux of the problem is that the invocation of the JMS receive operation must be performed within the context of the transaction. Then, if the subsequent handling of the message causes a failure, that's most likely a reason to roll back the transaction. Some other consumer may be able to process the message, or perhaps this same handler could handle the message successfully if retried after a brief delay. For example, some system that it relies on might be down at the moment but will be available again shortly. If a JMS consumer rolls back a transaction, then the message won't be removed from the Destination; that's what enables redelivery. But if the Exception is thrown by the handler in a different thread, it's too late: the JMS Consumer has already performed its role, and by passing off the responsibility to an executor that invokes the handler in a different thread, it would've returned successfully after that handoff. It would be unable to react to a rollback based on something that happens later, downstream in the handler's processing of the Message content.

9.3.2 **Spring's `MessageListener` container**

You're probably convinced that implementing an asynchronous message-driven solution isn't trivial. It's the type of generic, foundational code that should be provided by a framework so developers don't have to spend time dealing with the low-level threading and transaction management concerns. Spring provides this support for JMS with its `MessageListener` containers. Let's look at a modified version of the earlier Hello World example. We use `JmsTemplate` for the sending side only. The message is received by a `MessageListener` asynchronously, and the `DefaultMessageListenerContainer` handles all of the low-level concerns.

```

public class MessageListenerContainerDemo {
    public static void main(String[] args) {
        // establish common resources
        ConnectionFactory connectionFactory =
            new ActiveMQConnectionFactory("vm://localhost");
        Destination queue = new ActiveMQQueue("siia.queue");

        // setup and start listener container
        DefaultMessageListenerContainer container =
            new DefaultMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        container.setDestination(queue);
        container.setMessageListener(new MessageListener() {
            public void onMessage(Message message) {
                try {
                    if (!(message instanceof TextMessage)) {
                        throw new IllegalArgumentException("expected TextMessage");
                    }
                    System.out.println("received: " +
                        ((TextMessage) message).getText());
                } catch (JMSException e) {
                    throw new RuntimeException(e);
                }
            }
        });
        container.afterPropertiesSet();
        container.start();

        // send Message
        JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);
        jmsTemplate.setDefaultDestination(queue);
        jmsTemplate.convertAndSend("Hello World");
    }
}

```

The code shows how you can take advantage of asynchronous message reception by depending on Spring's `DefaultMessageListenerContainer` to handle the low-level concerns. Nevertheless, you might be thinking we added a lot more code to the example, and we're back to dealing with some JMS API code directly. For example, we provided an implementation of the JMS `MessageListener` interface, and we're catching the `JMSExceptions` to convert into `RuntimeExceptions` ourselves. In the next section, we take things two steps further. First, we rely on Spring's `MessageListenerAdapter` to decouple our code from the JMS API completely. Second, we refactor the example to use declarative configuration and a dedicated Spring XML schema. In other words, we demonstrate a Message-driven POJO.

9.3.3 **Message-driven POJOs with Spring**

The code and configuration for asynchronous reception can be much simpler. One goal for simplification should be to reduce the dependency on the JMS API. Rather

than having to create an implementation of the `MessageListener` interface, you can rely on Spring's `MessageListenerAdapter` to handle that responsibility. It's straightforward: the adapter implements the `MessageListener` interface, but it invokes operations on a delegate when a `Message` arrives. That instance to which it delegates can be any object. The previous code could be updated with the following replacement for the registration of the listener.

```
container.setMessageListener(new MessageListenerAdapter(somePojo));
```

An even better option is to use configuration rather than code. Spring provides a `jms` namespace that supports the configuration of a container and adapter in a few lines of XML.

```
<listener-container>
    <listener destination="aQueue" ref="aPojo" method="someMethod"/>
</listener-container>
```

Many configuration options are available on both the `listener-container` and `listener` elements, but the preceding example provides a glimpse of the simplest possible working case. The XML schema is well documented if you'd like to explore the other options. Our goal here is to provide a sufficient level of background information so you can appreciate that Spring Integration builds directly on top of the JMS support within the base Spring Framework. At this point, you should have a fairly good understanding of that support. We now turn our focus back to Spring Integration to see how it offers an even higher-level approach.

9.4 *Sending JMS Messages from a Spring Integration application*

Now that you've seen the similarities between Spring Integration Messages and JMS Messages and learned about the core Spring support for JMS, you're well prepared to look at the process of sending JMS Messages from a Spring Integration application. As with most adapters in Spring Integration, a unidirectional Channel Adapter and a request/reply Gateway are available. Because it's considerably simpler, we begin the discussion with the unidirectional Channel Adapter.

Spring Integration's outbound JMS Channel Adapter is a JMS Message publisher encapsulated in an implementation of Spring Integration's `MessageHandler` interface. That means it can be connected to any `MessageChannel` so that any Spring Integration Messages sent to that channel are converted into JMS Messages and then sent to a JMS Destination. The JMS Destination may be a Queue or a Topic, but from the perspective of this adapter implementation, that's a configuration detail.

The main class involved in the one-way outbound JMS adapter is called `JmsSendingMessageHandler`. If you look at its implementation, you'll see that it builds completely on the JMS support of the underlying Spring Framework. The most important responsibilities are handled internally by an instance of Spring's `JmsTemplate`. Most of the code in Spring Integration's adapter handles the various configuration options, of

which there are many. As far as most users are concerned, even those configuration options are handled by the XML namespace support. In most cases, only a small subset of those options would be explicitly configured, but there are many options for handling more nuanced usage requirements. We walk through several of these in a moment, but first let's look at a typical configuration for one of these adapters.

```
<jms:outbound-channel-adapter channel="toJMS"
    destination-name="samples.queue.fromSI" />
```

That looks simple enough, right? Hopefully so, and if you can rely on the defaults, then that's all you need to configure. It's literally adapting the Spring Integration toJMS channel so that it converts the Messages into JMS Messages and then sends them to the JMS Queue named samples.queue.fromSI. If you want to use a Topic instead of a Queue, be sure to provide the pub-sub-domain attribute with a value of true, as in the following example:

```
<jms:outbound-channel-adapter channel="toJMS"
    destination-name="samples.topic.fromSI"
    pub-sub-domain="true" />
```

Sometimes it's not practical to rely on just the name of the JMS Destination. In fact, it's common that the Queues and Topics are administered objects that developers should always access via JNDI lookups. Fortunately, you can rely on the Spring Framework's ability to handle that. Instead of using the destination-name attribute, you could provide a destination attribute whose name is a reference to another object being managed by Spring. That other object could then be a result of a JNDI lookup. For handling that lookup, Spring provides a FactoryBean implementation called the JndiObjectFactoryBean. Although it's perfectly acceptable to define that FactoryBean instance as a low-level bean element, there's XML namespace support for much more concise configuration options, as shown here.

```
<jms:outbound-channel-adapter channel="toJMS" destination="fromSI" />
<jee:jndi-lookup id="fromSI" jndi-name="jms/queue.fromSI" />
```

The functionality would be exactly the same. The difference is limited to the configuration. Access by name is often sufficient in development and testing environments, but JNDI lookups might be required for a production system. In those cases, you can manage the configuration excerpts appropriately by using import elements in the configuration or other similar techniques. The important factor is that you don't need to modify any code to handle those different approaches for resolving JMS Destinations.

Fortunately, the configuration of the ConnectionFactory and Destinations can be shared across both the sending and receiving sides. Likewise, for commonly configured references, such as these Destinations, there is consistency between the inbound and outbound adapters. In the next section, we focus on the receiving side. We begin with the inbound Channel Adapter that serves as a Polling Consumer.

9.5 Receiving JMS Messages in a Spring Integration application

When receiving JMS Messages in a unidirectional way, there are two options. You can define an inbound Channel Adapter that acts a Polling Consumer or one that acts as an Event-Driven Consumer. The polling option is configured with an `<inbound-channel-adapter>` element as defined in the JMS schema. It accepts a `destination-name` attribute for the JMS Queue or Topic. Its default `DestinationResolver` looks up the Destination accordingly, and if you need to customize that behavior for some reason, you can provide a `destination-resolver` attribute with the bean name reference of your own implementation. The `<inbound-channel-adapter>` element also requires a poller unless you're relying on a default context-wide poller element. Here's a simple example of an inbound Channel Adapter that polls for a JMS Message every three seconds.

```
<int-jms:inbound-channel-adapter id="pollingJmsInboundAdapter"
    channel="jmsMessages" destination-name="myQueue">
    <int:poller fixed-delay="3000" max-messages-per-poll="1" />
</int-jms:inbound-channel-adapter>
```

Like the outbound version, if you're referencing a Topic instance rather than a Queue, you should also provide the `pub-sub-domain` attribute with a value of `true`. If instead you want to reference a Queue or Topic instance, you can use the `destination` attribute in place of `destination-name`. This is a common practice when defining this adapter alongside Spring's JNDI support, as shown previously in the outbound Channel Adapter examples. The following is an example of the corresponding inbound configuration:

```
<int-jms:inbound-channel-adapter id="pollingJmsInboundAdapter"
    channel="jmsMessages" destination="myQueue">
    <int:poller fixed-delay="3000" max-messages-per-poll="1" />
</int-jms:inbound-channel-adapter>

<jee:jndi-lookup id="myQueue" jndi-name="jms/someQueue"/>
```

As mentioned earlier, polling is rarely the best choice when building a JMS-based solution. Considering that the underlying JMS support in the Spring Framework enables asynchronous invocation of a `MessageListener` as soon as a JMS Message arrives, that's almost always the better option. The only exceptions might be when you want to configure a poller to run infrequently or only at certain times of the day. If the poller is limited to run at certain times of the day, you'd most likely use the `cron` attribute on a poller element. Other than in those rare situations, the responsiveness will be better and the configuration will be simpler if you stick with Spring Integration's message-driven Channel Adapter. The basic configuration will look the same, but there's no longer a need to define a poller:

```
<int-jms:message-driven-channel-adapter id="messageDrivenAdapter"
    channel="jmsMessages" destination-name="myQueue" />
```

It may seem odd that, unlike most adapters you've seen, the element doesn't include `inbound` in its name. Considering it's message-driven, it should be relatively clear

that this Channel Adapter is reacting to inbound JMS Messages that arrive at the given Queue or Topic. It sends those messages to the channel referenced by the channel attribute.

9.6 Request-reply messaging

The discussion and examples in this chapter have thus far focused on unidirectional Channel Adapters. On the sending side, we haven't yet discussed the case where we might be expecting a reply, and on the receiving side, we haven't yet discussed the case where we might be expected to send a reply. We saw that Spring Integration's inbound JMS Channel Adapter can receive messages with either polling or message-driven behavior. On the other hand, the outbound Channel Adapter can be used to send messages to a JMS Destination, be it a Queue or a Topic. In both cases, the Spring Integration Message is mapped to or from the JMS Message so that the payload as well as the headers can be preserved in the JMS Message's body and properties respectively.

This section introduces Spring Integration's bidirectional Gateways for utilizing JMS in a request-reply model of interaction. Much of the functionality, such as mapping between the JMS and Spring Integration Message representations, is the same. The difference is that these request-reply Gateway adapters are responsible for mapping in both directions. As with the unidirectional Channel Adapter discussions, we begin with the outbound side. Whereas earlier we could describe the outbound behavior as solely responsible for sending Messages, in the Gateway case, there is a receive as well, assuming that the JMS reply Message arrives as expected. The simplest way to think of the outbound Gateway is as a send-and-receive adapter.

9.6.1 The outbound Gateway

In the simplest case, the outbound configuration will look similar to the outbound-channel-adapter configuration we saw earlier:

```
<int-jms:outbound-gateway request-channel="toJMS"
    reply-channel="jmsReplies"
    request-destination-name="examples.gateway.queue" />
```

The request-channel and reply-channel attributes refer to Spring Integration MessageChannel instances. Any Spring Integration Message that's sent to the request channel will be converted into a JMS Message and sent to the Gateway's request Destination (in this context, *Destination* always refers to a JMS component, and *channel* is the Spring Integration channel). In this example, the Destination is a Queue. If it were a Topic, the request-pub-sub-domain attribute would need to be provided with a value of true. Because the Gateway must manage sending and receiving separately, many of its attributes are qualified as affiliated with either the request or the reply. The reply-channel is where any JMS reply Messages are sent after they're converted to Spring Integration Messages.

You may have noticed that we didn't provide a reply-destination-name. That attribute is optional, but it's common to leave it out. The Gateway implementation

provides the `JMSReplyTo` property on each request Message it sends to JMS. If you don't provide a specific Destination for that, then it'll handle creation of a temporary queue for that purpose. This assumes that wherever these JMS Messages are being sent, a process is in place to check for the `JMSReplyTo` property so it knows where to return a reply Message. We discuss the server-side behavior in the next section when we cover the inbound Gateway. For the time being, we discuss this interaction with a hypothetical server side where we assume such behavior is in place. The `JMSReplyTo` property is a standard part of the Message contract and is defined in the JMS specification. Therefore, it's commonly supported functionality for server-side JMS implementations that accept request Messages from a sender who is also expecting reply Messages. You must be sure that you're sending to a Destination that's backed by a listener implementation with that behavior. The inbound Channel Adapter we discussed earlier would *not* be a good choice because, as we emphasized, it's intended for unidirectional behavior only. On the other hand, the inbound Gateway we discuss in the next section would be a valid option for such server-side request-reply behavior. The core Spring Framework support for Message-driven POJOs also supports the `JMSReplyTo` properties of incoming Messages as long as the POJO method being invoked has a nonvoid and non-null return value.

9.6.2 **The inbound Gateway**

Like the outbound Gateway, Spring Integration's inbound Gateway for JMS is an alternative to the inbound Channel Adapters when request-reply capabilities are required. Perhaps the quickest way to get a sense of what this means is to consider that this adapter covers the functionality we described in abstract terms as the server side in the previous section. The outbound Gateway would be the client side as far as that discussion is concerned. The inbound Gateway listens for JMS Messages, maps each one it receives to a Spring Integration Message, and sends that to a Message Channel. Thus far, that's no different than the role of an inbound Channel Adapter. The difference is that the Message Channel in this case would be the initiating end of some pipeline that's expected to produce a reply at some point downstream. When such a reply message is eventually returned to the inbound Gateway, it's mapped to a JMS Message. The Gateway then sends that JMS Message to the reply destination. That particular JMS Message fulfills the role of the reply message from the client's perspective.

The reply destination is an example of the Return Address pattern. It may have been provided in the original Message's `JMSReplyTo` property, and if so, that takes precedence. If no `JMSReplyTo` property was sent, the inbound Gateway falls back to a default reply Destination, if configured. As with the request Destination, that can either be configured by direct reference or by name. The attribute used for a direct reference is called `default-reply-destination` (again, it's the default because the `JMSReplyTo` property on a request message takes precedence). If configuring the name of the reply Destination so that it can be resolved by the Gateway's `DestinationResolver` strategy, use either the `default-reply-queue-name` attribute or the `default-reply-topic-name` attribute. If there's neither a `JMSReplyTo` property

on the request Message nor a configured reply Destination, then an exception will be thrown by the Gateway because it would have no way of determining where to send the reply.

That description of the inbound Gateway's role probably sounds like it involves a complex implementation. Keep in mind that it builds directly on top of the underlying Spring JMS support that we described earlier. Now you can probably appreciate why we went into considerable detail about that underlying support. As a result, you already have a basic understanding of how the inbound Gateway handles the server-side request-reply interaction. As with any Spring Integration inbound Gateway, once it maps to a Spring Integration Message, it's sending that message to a Message Channel. What makes each Gateway unique is *what* it receives and *how* it maps what it receives into a Spring Integration Message.

Now that you understand the role of the inbound Gateway for JMS, let's look at an example. We start with the simplest configuration options:

```
<jms:inbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"/>
```

The request-channel indicates where the inbound messages should be sent after they're created by mapping from the JMS source message, and the request-destination-name is where those JMS Messages are expected to arrive. You may have noticed that no reply-channel attribute is present. This attribute is an optional value for inbound Gateways in general. If it's not provided, then the Gateway creates a temporary, anonymous channel and sets it as the replyChannel header for the Message that it sends downstream. As with the <inbound-channel-adapter> for JMS, the connection-factory attribute is also optional, as long as a JMS ConnectionFactory instance is named connectionFactory in the application context. Add that attribute if for some reason the JMS ConnectionFactory you need to reference has a different bean name.

As you might expect, knowing that we're building on top of Spring's Message-Listener container, a number of other attributes are available. Many of them are passed along to that underlying container. For example, you might want to control the concurrency settings. Following is an example that indicates five core consumers that should always be running, but when load increases beyond the capacity of those five, the number of consumers can increase up to 25. At that point, each of those extra consumers can have up to three idle tasks—those where no message is received within the receive timeout of 5 seconds, at which point the consumer will be cleared. The end result of such configuration is that the number of consumers can dynamically fluctuate between 5 and 25 consumers based on the demand for handling incoming messages.

```
<jms:inbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"
```

```
concurrent-consumers="5"
max-concurrent-consumers="25"
idle-task-execution-limit="3" />
```

This example shows that various settings of the underlying `MessageListener` container can be configured directly on the XML element that represents the Spring Integration gateway. The preceding attributes are a small subset of all the configurable properties of the container. When defining your elements in an IDE with good support for XML, such as the SpringSource Tool Suite, you can easily explore the entire set of available attributes.

We've now covered the various Spring Integration adapters. You saw how such adapters can be used on both the sending side and the receiving side. You saw the unidirectional channel adapters as well as the bidirectional gateways that enable request-reply messaging. Next, we consider the scenario in which the JMS messaging occurs between two applications that are both using Spring Integration.

9.7 ***Messaging between multiple Spring Integration runtimes***

In the previous sections, you saw the inbound Gateway and the outbound Gateway. Both play a role in supporting request-reply messaging, but they were discussed separately thus far. That's because each can be used when you're limited in the assumptions you can make about the application on the other side. As you might expect, the two Gateways can work well together when you have Spring Integration applications on both sides. Figure 9.4 captures this situation by reusing one of the diagrams from the introductory chapter, this time labeled specifically for JMS.

In the scenario depicted in the figure, it will obviously be necessary to map between the Spring Integration Messages used in each application and the JMS Messages that are being passed between the applications. We saw several examples of how the adapters use Spring's `JMS MessageConverter` strategy to convert to and from JMS Messages. So far, the examples have mapped between the JMS Message body and the Spring Integration Message payload. Likewise, the JMS properties have mapped to and from the Spring Integration Message headers. These are by far the most common usage patterns for message mapping with JMS, and they make minimal assumptions about the system on the other side of the message exchange.

In a particular deployment environment, though, it might be well-known that Spring Integration-based applications exist on both sides of the JMS Destination. One Spring Integration application would act as a producer, and the other would act as a consumer. The interaction may be unidirectional, using the Channel Adapters we saw earlier in this chapter, or the interaction might involve request-reply exchanges

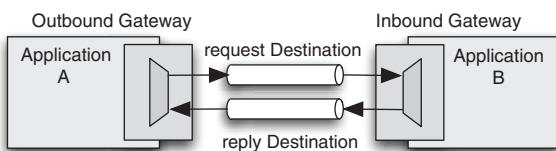


Figure 9.4 A pair of gateways, one outbound and the other inbound. Each is hosted by a separate Spring Integration application. Those two applications share access to a common JMS broker and a pair of Destinations.

wherein one of the applications contains an outbound Gateway and the other contains an inbound Gateway (keep in mind that a Gateway acts as both a producer and a consumer). If that's the nature of the deployment model, it may or may not be desirable to pass the entire Spring Integration Message as the JMS Message body. The default mapping behavior would obviously work in such an environment, but if you want to "tunnel" through JMS instead, for some reason, then you can override the default configuration. To send a Spring Integration Message as the actual body of a JMS Message, provide a value of `false` to the `extract-request-payload` property of an outbound Gateway.

```
<jms:outbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"
    extract-request-payload="false" />
```

When passing the Spring Integration Message as the JMS Message body, it's necessary to have a serialization strategy. The standard Java serialization mechanism is one option, because Spring Integration Messages implement the `Serializable` interface. One thing to keep in mind when choosing that option is that nonserializable values that are stored in the Message headers won't be passed along because they can't be serialized with that approach. An even more important factor to keep in mind is that Java serialization requires that the same class be defined on both the producer and the consumer sides. Not only must it be the same class, but the version of the class must be the same. JMS facilitates this option by providing support for `ObjectMessages`, where the body is a `Serializable` instance.

At first, it seems convenient to pass your domain objects around without any need to think about conversion or serialization, but it's almost always a bad idea in reality. By requiring exactly the same classes to be available to both the producer and the consumer, this approach violates the primary goal of messaging: loose coupling. Even if you control both sides of the messaging exchange, the fewer assumptions one side makes about the other, the more flexible the application will be. As any experienced developer knows, the most regrettable assumptions are those made about the future of an application. For example, if an application needs to evolve to support multiple versions of a certain payload type, reliance on default serialization to and from a single version of a class will be a sure source of regret.

With these twin goals of reducing assumptions and increasing flexibility in mind, let's consider some other options for serializing data. Probably the most common approach in enterprise integration is to rely on XML representations. Spring Integration provides full support for that option with the Object-to-XML Marshaller and XML-to-Object Unmarshaller implementations from the Spring Framework's `oxm` module (see chapter 8 for more detail). Another increasingly popular option for serializing and deserializing the payload is to map to and from a portable JSON representation. The advantage of building a solution based on either XML or JSON instead of Java serialization is that the system can be much more flexible. It's not necessary to have the same version on both sides. In fact, assuming the marshaller and unmarshaller

implementations account for it, the producer and consumer sides may even convert to and from completely different object types.

Regardless of the chosen serialization mechanism, you must configure a `MessageConverter` on the Gateway any time you don't want to rely on the default, which uses Java serialization. You might choose the `MarshallingConverter` provided by Spring for object-to-XML conversion, or you might implement `MessageConverter` yourself. Either way, define the `MessageConverter` within the same `ApplicationContext`, and then provide the reference on the outbound Gateway or Channel Adapter's configuration.

```
<jms:outbound-gateway id="exampleGateway"
    request-destination-name="someQueue"
    request-channel="requestChannel"
    extract-request-payload="false"
    message-converter="customConverter" />

<bean id="customConverter" class="example.CustomMessageConverter"/>
```

Generally, we recommend avoiding the tunneling approach because having the mapping behavior on both sides promotes loose coupling. Even then, it's worth considering the serialization strategy. If a Spring Integration payload is a simple string or byte array, then it'll map to a JMS `TextMessage` or `BytesMessage` respectively when relying on the default `MessageConverter` implementation. The default conversion strategy also provides symmetric behavior when mapping from JMS. A `TextMessage` maps to a string payload, and a `BytesMessage` maps to a byte array payload. But if your Spring Integration payload or JMS body is a domain object, then it's definitely important to consider the degree of coupling because the default `MessageConverter` will rely on Java serialization at that point.

Spring Integration provides bidirectional XML transformers in its XML module (again, see chapter 8 for more details), and it provides bidirectional JSON transformers in the core module. Both the XML and JSON transformers can be configured using simple namespace-defined elements in XML. You can provide the object-to-XML or object-to-JSON transformer upstream from an outbound JMS Channel Adapter or Gateway, and you can provide the XML-to-object or JSON-to-object transformer downstream from an inbound JMS Channel Adapter or Gateway. One advantage of relying on the transformer instances is that you can reuse them in multiple messaging flows. For example, you might also be receiving XML or JSON from an inbound file adapter, and you might be sending XML or JSON to an outbound HTTP Gateway.

If such opportunities for reuse aren't relevant in your particular application, you may prefer to encapsulate the serialization behavior. You can rely on any implementation of Spring's `MessageConverter` strategy interface. As mentioned earlier, the `MarshallingMessageConverter` is available in the Spring Framework. A similar JSON-based implementation will likely be available in the future but wouldn't be difficult to implement in the meantime. You can provide any other custom logic or delegate to any other serialization library that you choose. If going down that path, you would define the chosen `MessageConverter` implementation as a bean and then reference it from the Channel Adapter or Gateway's `message-converter` attribute.

9.8 Managing transactions with JMS Channel Adapters and Gateways

Typically, the requirements of an application that utilizes a messaging system include such terms as “guaranteed delivery” and provide details about retry policies in light of potential failure scenarios. Those requirements usually support one key concern: messages can’t be lost even if errors occur while processing them. In the worst case, after a number of attempts, a message should be delivered to a dead letter queue. Proper transaction management is essential when confronting such requirements. The primary goal when managing transactions in general is to ensure that data are always accounted for. No party should relinquish responsibility until another party has assumed responsibility. In a messaging interaction, the two parties that may assume responsibility at any given time are the producer and the consumer.

Therefore, when working with JMS, it’s important to understand exactly how and where to apply transactions on the producer side as well as on the consumer side. It’s also important to consider whether the unit of work that should be transactional includes not only interaction with the JMS message broker but also some other transactional resource, such as a database. Finally, it’s important to understand the advanced options (those that may be outside of the JMS specification) that are available with a given broker, such as the retry policies. Such options are beyond the scope of this book, but the JMS provider’s documentation is a good place to start. If you’re using ActiveMQ, its options are covered in detail in *ActiveMQ in Action*. Here, we cover the general aspects of JMS transactions and specifically how they can be configured on Spring Integration’s JMS Channel Adapters and Gateways.

9.8.1 JMS transaction basics

Let’s revisit the creation of a JMS Session, the object that represents a unit of work in the JMS API. As you saw at the beginning of this chapter, the Session acts as a factory for creating the Messages as well as the producer and consumer instances. When creating a Session from a Connection, you must decide whether that Session should be transactional. Here’s a simple example using the JMS API direction (without Spring):

```
Connection connectionFactory = connectionFactory.createConnection();
int autoAck = Session.AUTO_ACKNOWLEDGE;
Session session = connectionFactory.createSession(true, autoAck);
```

That boolean value (true in this example) indicates that the Session should be transactional. Note that the second argument indicates the acknowledge mode for the Session. The two values are mutually exclusive. If you provide a value true to indicate that you want a transactional Session, the acknowledge mode value will be ignored. On the other hand, if you provide false to indicate that you don’t want a transactional Session, then the value will play a role. We briefly return to the acknowledge modes in a moment, but first let’s discuss the publishing side where things are straightforward.

When publishing JMS Messages, the transactional boolean value passed when creating the Session plays an important role. The Session is used as a factory for creating any `MessageProducer` instances, and the transactional characteristics of those producers would reflect that boolean flag. If the Session isn't transactional, then publishing Messages will truly be a fire-and-forget operation, and there will be no way to undo the publish operation or to group multiple operations into a single unit of work. In contrast, when the transactional setting is enabled on a Session, any messages published with that Session are only made available within the broker upon committing the transaction. That means that if something goes wrong, any messages published with that Session can be rolled back instead. The `commit()` and `rollback()` methods are available directly on the Session.

When using a transactional Session, similar behavior is available on the receiving side. A `MessageConsumer` created from a transactional Session won't permanently take its messages from the broker until the Session's transaction is committed. If something goes wrong after receiving a message, calling `rollback()` on the Session will undo the reception. That means any messages received within the scope of the rolled-back transaction may be eligible for redelivery, potentially to a different consumer.

As mentioned previously, when the transactional boolean value is `false`, the acknowledge mode plays a role in the message reception. The valid integer values for acknowledge modes are defined as constants on the `Session` class. When using any Spring or Spring Integration JMS support, `AUTO_ACKNOWLEDGE` is the default. You can set the value to `CLIENT_ACKNOWLEDGE` instead, which indicates that you expect the client consuming the JMS Messages to explicitly invoke the `acknowledge()` method on those Messages. An `acknowledge()` call on a single Message will acknowledge *all* previously unacknowledged Messages that have been consumed on that same Session. A third option, called `DUPS_OK_ACKNOWLEDGE`, is similar to `AUTO_ACKNOWLEDGE` in that no explicit calls are required of the consuming client. It differs in that the acknowledgments may be sent lazily, allowing for the chance that duplicate Messages might be received in the case of certain failures within the window of vulnerability. The details of acknowledge modes are covered in the JMS specification. You can also find some information in Spring's JavaDoc and XML schema documentation.

After our detailed discussion earlier in this chapter, you know that the Spring Framework's JMS support includes a component that serves as a `MessageListener` container. Session management for message consumption is one of the responsibilities of such a container. In consideration of the mutual exclusivity between the boolean value to indicate a transactional Session and the acknowledge mode to be used otherwise, the schema-based configuration of the listener container rolls these options into a single `acknowledge` property. It can take a value that corresponds to any of the three acknowledge modes defined by the JMS specification, *or* it can be set to `transacted`. Here are two examples to illustrate:

```
<!-- Transactional MessageListener Container -->
<listener-container acknowledge="transacted">
```

```

<listener destination="someQueue" ref="someListener"/>
</listener-container>

<!-- Non-transactional auto-ack MessageListener Container -->
<listener-container acknowledge="auto">
    <listener destination="someQueue" ref="someListener"/>
</listener-container>

```

When configuring Spring Integration's inbound Channel Adapter or Gateway for JMS, you'll see the same attribute. It accepts any value from the same enumeration: `auto`, `client`, `dups-ok`, or `transacted`. Here are a couple examples.

```

<int-jms:message-driven-channel-adapter id="exampleChannelAdapter"
    channel="jmsMessages"
    destination-name="queue1"
    acknowledge="transacted"/>

<int-jms:inbound-gateway id="exampleGateway"
    request-destination-name="queue2"
    request-channel="jmsMessages"
    acknowledge="dups-ok"/>

```

One important final note: if no value is provided, the default will be `auto` acknowledge mode, and hence Sessions won't be transactional. Transactions add overhead, so if you're sending nonessential event data where occasional message loss isn't a problem, the default might be fine. But if your messages are carrying document data to be processed, be sure to consider this setting carefully.

9.8.2 A note about distributed transactions

The Spring Framework provides an abstraction for transaction management that makes it easy to switch between different strategies for handling transactions within an application. In most cases, only configuration changes are necessary. Most likely, that change is limited to replacing a single bean because the code depends on the abstraction. Furthermore, because transaction support is typically handled by interception within a proxy, end user code is almost never affected directly.

The key interface in Spring that provides this abstraction is the `PlatformTransactionManager`. It defines the methods you'd expect for managing transactions regardless of the details of the underlying system.

```

public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException;
}

```

As far as transaction management strategies are concerned, broadly speaking, there are two main choices: local or global transaction management. What we mean by *local* is that transactions can be managed directly against the transactional resource without

any need for a mediator. In the case of JMS, the local transactional resource would be the JMS Session instance. As you saw earlier, a Session can be created with a simple boolean flag to indicate whether it should be transactional. The JMS Session class also defines commit and rollback methods.

Another example that might clarify this terminology is the JDBC Connection class. It's also a local transactional resource because it provides commit and rollback methods. Whereas the JMS Session is transactional upon creation (assuming the flag is true), the JDBC Connection has a `setAutoCommit()` method such that passing a value of `false` will disable auto-commit mode. Explicit commit (or rollback) is then required, so the Connection is then *transactional* in that multiple operations can be handled within the scope of a single unit of work. A JMS Session and a JDBC Connection are both good examples of resources that enable local transaction management. The way the respective APIs expose the functionality is slightly different, but that's where Spring's `PlatformTransactionManager` can step in to provide a consistent view. In fact, implementations are available out of the box for these two cases: `JmsTransactionManager` and `DataSourceTransactionManager`.

Earlier we mentioned a common case where a unit of work spans a JMS Message exchange and one or more database operations. In those cases, it might be tempting to move toward a global transaction solution. You could rely on XA-capable resources and a Java Transaction API (JTA) implementation that supports two-phase commit as the transaction manager, but it's not always necessary to go that far. A distributed transaction in the more general sense means more than one resource is involved. Rather than assuming you need to use XA and two-phase commit along with the extra overhead they bring, you should consider all of the options.

It might be possible to find some middle ground such that you can maintain the simplicity of the local transaction management but chain multiple local resources together. The trade-off is a requirement to carefully consider the order of operations and how to manage certain rare failure scenarios. One of the most common patterns when dealing with messaging is to have a process that begins with a received JMS Message but then invokes a database operation based on that message's content. For example, imagine an order being received and then inserted into the database. The JMS Session is one local transactional resource, and the JDBC Connection is another. If a failure occurs while processing the message content or inserting the database record, then an exception could trigger a rollback of both the JDBC and JMS transactions. The JMS Message would be eligible for redelivery, and the database would remain unchanged. If everything goes smoothly, the database insert is committed, and the JMS Message is permanently removed from its Destination.

The rare situation that could lead to a problematic result is if the database transaction commits, but for some reason, such as a system-level failure, the JMS transaction rolls back afterward. In that case, the JMS Message could potentially be redelivered, and so the application must be able to handle such duplicates. If the database operation is idempotent, then there might not be any problem at all. In an idempotent

operation, a record would be ignored if the same data already exists, likely based on a where clause condition. If the operation isn't inherently idempotent, there might be more work involved to add similar logic at a higher level of the application. Perhaps any message where the `JMSRedelivered` property is true can be checked against the database prior to invoking the normal insert operation. If it's recognized as a duplicate that has already been handled successfully, the message could be ignored, and its JMS transaction could be committed so it won't be redelivered anymore. Keep in mind that while that sounds like added overhead, such redeliveries would likely be extremely rare. Depending on the particular application structure, these types of solutions might be simpler and add less overhead than a full XA two-phase commit option. For much more detail on these types of distributed transaction patterns and the trade-offs involved, refer to "Distributed Transactions in Spring, with and without XA" by Dave Syer: (available at <http://mng.bz/9DF4>). Dave is the lead of the Spring Batch project and a committer on several other Spring projects, including Spring Integration.

9.9 **Summary**

We covered a lot of ground in this chapter. Considering the central role that JMS plays in many enterprise Java applications, we wanted to make sure it was clear where Spring Integration overlaps with JMS and where the two can complement each other. You saw the relationship between the two message structures and how to map between them. You also learned how the underlying Spring Framework provides base functionality that greatly simplifies the use of JMS and how Spring Integration takes that even further with its declarative configuration and higher level of abstraction. After walking through both the unidirectional adapters and the request-reply Gateways, we dove into a bit of depth regarding transactions. At this point, you should be well prepared to build Spring Integration applications where none of the code is directly tied to the JMS API while still benefiting from the full power of whatever underlying JMS provider you choose. In chapter 10, we cover Spring Integration's support for yet another type of messaging system, one that you most likely use on a daily basis: email.

10

Going e-postal

This chapter covers

- Email as an enterprise application integration mechanism
- Sending and receiving emails with Spring Integration
- Design strategies for email-based applications

If JMS is the first thing that comes to your mind when associating Java and messaging, then chances are that doing the same for messaging and Internet will make you think of email. We all know it well. For decades, email has been the primary method of exchanging digital messages, and has made its way from being the killer app of the nascent internet to becoming an indispensable personal and business communication tool.

But this isn't a book about email etiquette, nor do we want to settle the age-old question of top-posting versus bottom-posting: we're interested only in the implications of using email as an enterprise application integration tool. Besides its primary role as a means of communication between people, email is also a useful interaction medium for applications, either with users—by sending notifications or receiving requests—or with other applications as well. Email is a complex messaging mechanism, supporting broadcast, data delivery as attachments, store-and-forward transmission with message relaying, and allowing a choice of message receiving

protocols, either event-driven (IMAP push) or pull (POP, IMAP pull). Also, email is continuously evolving in response to changes in technology and challenges such as spam and fraud, incorporating more sophisticated mechanisms for secure data transmission, authentication, and authorization, to name just a few areas of interest.

In this chapter, we provide an overview of the most significant use cases that require email support, and you'll learn how you can use Spring Integration to add it to your application. We show you how to send and receive email, the various options for doing so, and how to choose among them. Let's start with sending emails, the more typically encountered use case.

10.1 **Sending email**

Event-driven interaction isn't something that happens only between applications and systems: humans have been exchanging messages since the dawn of history, and electronic mail is a way of using modern technology for implementing an ancient communication pattern. But incorporating it in working applications opens the way for interaction workflows that improve and transform the user experience. For example, you may be waiting for a shipment (maybe your print copy of *Spring Integration in Action*), but you don't know the exact date and time when you have to be home to receive it. You may try to find out when you're supposed to be home by repeatedly checking the shipper's site, but this can become a tedious exercise, and often a futile one, because the information may change frequently. "Hold on," you say, "I'd rather get a notice when you *know* something," and you subscribe to receive a notification when the status changes. Now you may carry on with your own business, assured that you'll be informed without having to repeatedly check the shipper's site for a status report.

You can find many other examples where a notification system is a valuable add-on to an application, and for a notification system to be efficient, it must use a mechanism that allows easy access on the recipient's side. This assumption holds true for email: it's widely available, either as a public service or as part of a corporate communication infrastructure, and because mobile devices have become so popular, email is accessible even when you're on the go.

Although email is a widely used notification method, there are other ways of sending short notifications to users, such as Short Message Service (SMS) and social networking media, such as tweet and chat. Email differentiates itself from the other such personal messaging systems through features such as the ability of carrying attachments, broadcast, store-and-forward delivery, delivery status availability, and failure notification, which make it interesting for use as an intersystem message exchange transport. Email isn't a typical first choice for integrating applications hosted on the same machine or deployed in the same local network, where specifically designed messaging middleware, accessible via Java Message Service (JMS) or Advanced Message Queuing Protocol (AMQP), is a far better fit when considering any relevant criterion (performance, throughput, transaction capabilities, delivery guarantees), but emails can be used as an integration mechanism in other situations, for example

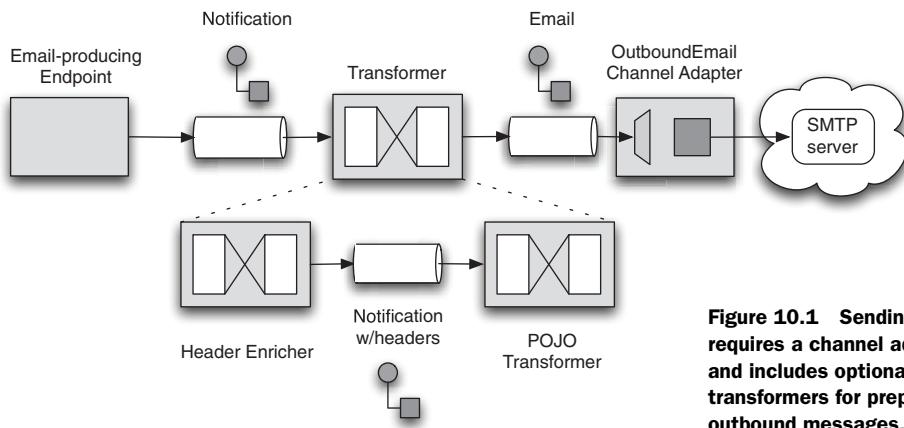


Figure 10.1 Sending email requires a channel adapter and includes optional transformers for preparing outbound messages.

for systems communicating over the Web. They also don't require any special setup for remote access, like exposing ports and addresses, as a message broker may normally require.

The process of sending email from within an application can be divided in two logical steps: (1) preparing an email message by populating its subject, body, attachments, and recipients list and (2) the operation of sending it. A complete solution involves a chain of Spring Integration endpoints with specific roles, typically similar to what you can see in figure 10.1.

Because the key component of this chain (and the only one whose presence is mandatory) is the channel adapter, we look at that first.

10.1.1 The Outbound Channel Adapter

The operation of sending email messages from your application is fairly straightforward and intuitive, considering that it's based on a pattern you're already familiar with: the Outbound Channel Adapter. As a first step, you must configure such an adapter, which can be as simple as in the following code snippet, where the username and password attributes describe the login credentials for using the Simple Mail Transfer Protocol (SMTP) services provided by the host:

```
<mail:outbound-channel-adapter channel="outboundMail"
    host="${host}" username="${user}" password="${password}" />
```

What you send in a message can be, in the simplest case, equivalent to the following snippet: the outbound channel adapter composes a message with the payload as the body and the recipients lists (to, cc, bcc) and subject as header values.

```
Message<String> message = MessageBuilder("Hello World!")
    .setHeader(MailHeaders.TO, "recipient1@examplehost.com")
    .setHeader(MailHeaders.SUBJECT, "Greeting")
    .build();
outboundMailChannel.send(message);
```

We said *equivalent* because sending mail in a programmatic fashion isn't the only way to send it and perhaps isn't even the most common: email messages are usually produced by upstream endpoints or by a publisher interceptor. The end result, regardless of what led to it, is that a Spring Integration message with a mail-specific payload and mail-specific headers is sent to a channel to which an outbound email channel adapter listens then composes an email message based on payload and header content and sends it.

One aspect to consider when composing messages for the email outbound channel adapter is that it supports only a limited number of payload choices. A string is the most straightforward alternative and can be either a predefined message or content generated from a template. Another type of payload accepted by the channel adapter is a byte array, in which case the resulting email message adds the binary content as an attachment. You can use additional headers for describing the attachment (filename, content type, and so on). A complete list of the message headers and how they're used for constructing the outgoing message is found in table 10.1. The header name refers to string constants defined in the `MailHeaders` helper class.

To understand more advanced ways of composing email messages and configuring outbound email channel adapters, we need to look closer at the innards of the mail-sending process.

Table 10.1 Spring Integration mail message headers

Header name	Meaning
<code>MailHeaders.FROM</code>	The sender of the message
<code>MailHeaders.TO</code>	List of recipients
<code>MailHeaders.CC</code>	List of carbon-copy (CC) recipients
<code>MailHeaders.BCC</code>	List of blind carbon-copy (BCC) recipients
<code>MailHeaders.SUBJECT</code>	Email subject
<code>MailHeaders.REPLY_TO</code>	Reply-to address
<code>MailHeaders.MULTIPART_MODE</code>	Indicates whether the message supports alternative texts, attachments, and inline content (used only for byte array messages)
<code>MailHeaders.CONTENT_TYPE</code>	Can be used to set an alternative content type for message body (for example, to send HTML messages instead of plain text)
<code>MailHeaders.ATTACHMENT_FILENAME</code>	Filename of the attachment (used only for byte array messages)

10.1.2 Advanced configuration options

Under the hood, the Spring Integration email outbound channel adapter is based on the Java Mail API and on additional utilities provided by the Spring framework.

The underlying usage of Java Mail means that you can configure the channel adapter by inserting additional properties through the `javaMailProperties` attribute of the `<mail:outbound-channel-adapter>` element. This mechanism is used for finely tuning the configuration parameters, adding connectivity features such as Secure Sockets Layer (SSL) support for sending messages, as shown in the following snippet. For a complete list of the configuration options and their meanings, please consult the Java Mail documentation.

```

<int-mail:outbound-channel-adapter channel="outboundMail"
    host="${host}" username="${user}" password="${password}"
    java-mail-properties="javaMailProperties"/>

<util:properties id="javaMailProperties">
    <prop key="mail.imap.socketFactory.class">
        javax.net.ssl.SSLSocketFactory</prop>
    <prop key="mail.imap.socketFactory.fallback">false</prop>
    <prop key="mail.store.protocol">imaps</prop>
    <prop key="mail.transport.protocol">smtpls</prop>
    <prop key="mail.smtps.auth">true</prop>
    <prop key="mail.smtp.starttls.enable">true</prop>
</util:properties>

```

For applications that need to send email (especially the ones that don't use Spring Integration), using the Java Mail API directly can become a cumbersome exercise, which involves operating directly with the low-level components of the API and direct manipulation of resources. To simplify the general process of sending and receiving email, the Spring framework provides a higher-level API for sending emails, the `JavaMailSender`.

In a nutshell, sending or receiving email requires interacting with instances of the type `javax.mail.internet.MimeMessage`, which is the Java Mail representation of an email message. When receiving messages, dealing directly with such objects is fairly simple because they provide access to all the properties of the email (as explained in the following section). But when sending messages, creating and manipulating `MimeMessage` instances requires access to a low-level resource, the `javax.mail.Session` that represents the connection to the mail server. Out of the box it's impossible to separate concerns in a good manner and create a service layer that's responsible for composing an infrastructure-independent email, which is a problem when considering, for example, the need for unit testing such code. Spring therefore provides the `JavaMailSender` abstraction, an intermediate layer that uses a Spring-specific object, the `MailMessage`, as an input and interacts with the infrastructure for sending messages, as shown in figure 10.2. For more details, we encourage you to read the email section of *Spring in Action* by Craig Walls (Manning, 2011).

Using the `JavaMailSender` has two consequences. On one hand, users can inject a `JavaMailSender` object directly in the outbound channel adapter, which can also act as a general configuration for sending emails (rather than, for example, individually configuring each channel adapter). On the other hand, the outbound channel adapter accepts two other types of object as payload: the raw Java Mail `MimeMessage`

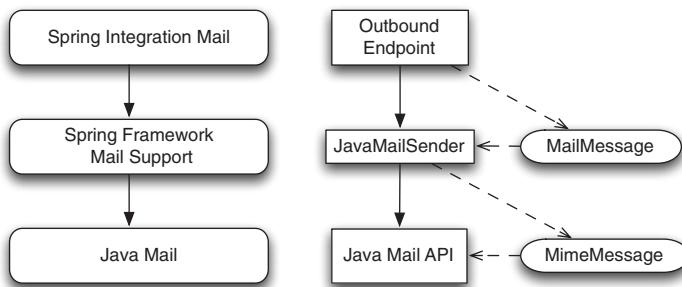


Figure 10.2 Spring Integration uses the mail sending layer provided by the Spring Framework proper. The components from each layer, as well as the mail message abstractions, are shown on the right side

and the Spring-specific `MailMessage`, which allows developers to compose messages using the Spring Framework utilities and send them through the channel adapter.

Sending email is usually part of a larger chain of events, and to illustrate that, let's consider a business example. You saw in chapter 6 that as soon as the application is notified of a flight schedule change, it must inform all the customers who are booked on that flight. The upstream service adapter takes care of searching for accounts and producing one or more `Notification` objects.

10.1.3 Transforming outbound messages

What can you do with messages whose payloads are not strings, byte arrays, or `MailMessage/MimeMessage` instances? If you can't deal elegantly with domain objects, the adapter will serve only a limited set of corner cases. One of the strengths of Spring Integration is decoupling between components, so it may not seem too far a stretch that a service adapter won't want to produce dedicated `MailMessage` instances, but domain-specific notifications, leaving it to the downstream components to translate them into email-ready instances. You may also need this functionality when supporting more than one kind of notification type by the application: besides email, you may have SMS messages or even direct phone calls. In such cases, an upstream component will create a generic `Notification` instance, which is sent to a publish-subscribe channel, with the email-sending component being just one of the subscribers.

The pipes-and-filters architecture of Spring Integration has a simple solution for this: all you need to do is include a message transformer that converts the message with a plain old Java object (POJO) payload into one of the four compatible payloads, as shown in figure 10.1.

Generally speaking, it's not just about the payload. If the transformer logic is only dealing with the payload (for example, creating a string message based on the content of the `Notification` object), you can use a dedicated header-enricher component for populating the message headers with appropriate values. A header enricher can use static values or Spring Expression Language (SpEL) expressions evaluated against the message to be enriched. In general, the best practice is to insert the header enricher first because the transformer may remove information that's required for populating the headers; for example, the recipient of the message may be one of the properties

of the `Notification` object but may not be contained in the message body payload. A complete example, including the transformer and the optional header enricher, is shown in the following snippet. To simplify the configuration, the endpoints are chained together.

```
<chain input-channel="notificationRequest"
       output-channel="outboundMail" />
<service-activator ref="notificationService" />
<header-enricher >
    <to expression="payload.email" />
    <from value="${settings.email.from}" />
    <subject value="Notification" />
</header-enricher>
<transformerref="templateMessageGenerator" />
</chain>
```

This concludes the section dedicated to sending email. It's time now to discuss how to handle inbound email messages.

10.2 ***Receiving email***

Sending notification emails is the most common use case, but a significant number of applications also support an email-driven application flow. Sending documents as email attachments can be an alternative to uploading them on a web page, especially when you don't want to put up with creating a dedicated website. Also, emails carry the identity of the sender, which works pretty well in request-reply scenarios, because the identity of the sender and the reply-to information is communicated as part of the email. And, as we mentioned in the previous section, email can serve as transport for carrying data over the Web when using a message broker would create too many logistical and infrastructural issues.

For example, your application may offer users the opportunity to perform flight status checks through email. The company provides an email address to which customers send an email with the keyword STATUS and the flight number in the subject, and the application replies with the flight information for today.

Our application acts as an email client, reading the contents of the mailbox and processing every incoming message as a request. The general structure of an email-receiving application is shown in figure 10.3. The inbound channel adapter creates messages with a Java Mail `MimeMessage` payload, and the messages are sent on the adapted channel for further processing downstream. For separating concerns in the application, the `MimeMessages` are converted to domain objects downstream, typically using a transformer.

The first step for creating such an application is deciding how to receive emails—what kind of a channel adapter you'd like to include. You can either poll the mailbox or be notified when new messages arrive, so let's examine these two options before we discuss how to process inbound messages.

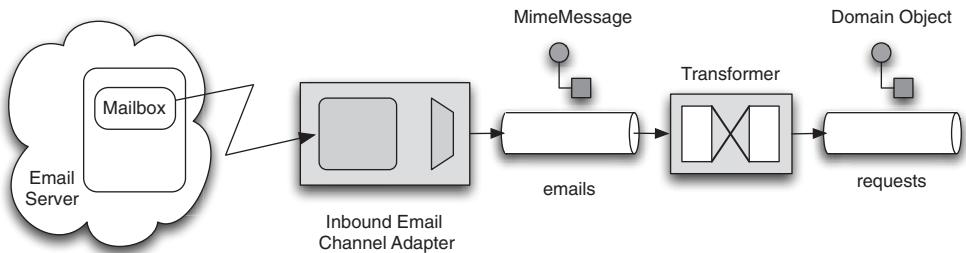


Figure 10.3 Receiving emails requires a channel adapter and includes optional transformers for extracting message content.

10.2.1 Polling for emails

Polling is the most basic email-receiving strategy, and it's supported by all application servers. A client connects to the mailbox and downloads any unread messages that have arrived since the last polling cycle.

A polling channel adapter can be configured as in the following example:

```

<mail:inbound-channel-adapter id="mailAdapter"
    store-uri="imaps://[username]:[password]@imap.gmail.com:993/INBOX"
    java-mail-properties="javaMailProperties"
    channel="emails"
    should-delete-messages="true"
    should-mark-messages-as-read="true">
    <int:poller max-messages-per-poll="1" fixed-rate="5000"/>
</mail:inbound-channel-adapter>
    
```

This example describes an Internet Message Access Protocol (IMAP)-based channel adapter, but Spring Integration supports both of the two major email access protocols, Post Office Protocol 3 (POP3) and IMAP, through the same channel adapter element. Whether the adapter uses one or the other protocol is indicated in the `store-uri` attribute, which in this example indicates that secure IMAP (`imaps`) is in use, and additional Java Mail properties are set on the adapter through the `java-mail-properties` attribute. The protocol that will be used in a particular case depends on the setup of the mail server where the incoming mailbox is located, and it's not typically left to you as an option.

Though considering the differences between protocols may not help you choose one over the other, it may help you understand how the protocols work and what consequences configuration options such as `should-delete-messages` have on the application.

POP3 is a protocol designed for downloading messages locally. Mailboxes are set up as a temporary storage, and the assumed permanent destination of the messages is the local store provided by the email client. This means the server is somewhat aware that a message has been read, but only for the duration of a session. In Spring Integration, an email client session is started when the application starts and is closed on shutdown, so any messages that aren't deleted from the server before shutdown may

be received multiple times. To solve the issue of duplicate messages, an application can choose to delete messages from the server as soon as they are read by setting `should-delete-messages` to `true`, but destructive reading carries its own risks as well: because there's no such concept as a transaction, a failure that occurs between the moment the email has been read and the moment it has been processed may result in the message being lost. Moreover, using the mailbox in a multiple-client, publish-subscribe scenario would be problematic if one of the subscribers started deleting emails. Because there's no single best choice here, Spring Integration forces you to state explicitly which one of the two parameters you want to set, essentially deciding between providing a contingency on loss or duplication.

IMAP is a protocol designed for storing messages remotely. Mailboxes are the permanent storage for emails, and email clients, by default, provide a view of the remote mailbox state rather than retrieving content locally. Because the remote mailbox has a state, deleting messages isn't necessary to avoid duplicate reception. In the case of IMAP, you can also decide whether you want to mark processed messages as read. Setting this option on a POP3 adapter is possible, but the value will be ignored because POP3 doesn't provide an updatable persistent mailbox state.

IMAP provides a more robust alternative to POP3 in polling scenarios, but its true advantage is that, under certain circumstances, you can replace polling with event-driven reception.

10.2.2 *Event-driven email reception*

Although polling is available for any email server, for particular server configurations, it's also possible to set up an event-driven inbound channel adapter for receiving emails. This functionality is available when connecting to IMAP servers that support IMAP-Idle, a feature that allows clients to request real-time notifications on mailbox changes. As a result, clients don't have to poll the server for new messages: they're notified when new messages are added to the mailbox.

This approach is lighter on resource consumption and traffic because the application doesn't have to actively check the server for new messages. It also improves responsiveness because applications are notified as soon as a message arrives, without the lag introduced by a polling cycle. But it's important to note that the IMAP-Idle command has a timeout of, at most, 30 minutes, which means that clients have to reissue it periodically.

For using the IMAP-Idle message retrieval strategy, all you need to do is use a different type of channel adapter.

```
<mail:imap-idle-channel-adapter id="mailAdapter"
    store-uri="imaps://[username]:[password]@imap.gmail.com:993/INBOX"
    channel="emails"
    auto-startup="true"
    should-delete-messages="false"
    should-mark-messages-as-read="true"
    java-mail-properties="javaMailProperties"/>
```

Whether it's polling or event-driven, the inbound email channel adapter will create a message with a payload of the JavaMail's Message type. To handle these messages in the application, you must transform them to a more neutral format, such as a string or a POJO.

10.2.3 Transforming inbound messages

Once you configure your inbound channel adapter, the next step is to process the received messages downstream. For handling the lower-level Java Mail Message objects in the domain-aware services, it's typical to include a transformer before the objects enter the application flow.

The simplest solution is to use a SpEL-based transformer that's evaluated against the Java Mail Message payload of the inbound message. For example, the transformer in the following snippet extracts the message body and creates a new Spring Integration message, using it as a payload. You can write more complex SpEL expressions to include other properties of the message, such as the subject, sender, or date:

```
<chain input-channel="emails"
    ↗ output-channel="handled-emails">
    <transformer expression="payload.content"/>
    <service-activator ref="emailHandlingService"/>
</chain>
```

Spring Integration provides an out-of-the box transformer you can use for extracting the email body or the binary content into a string payload. One of the benefits of using it is that it also extracts the other email properties as message headers, mirroring what happens when you send a message. Review table 10.1 for a reminder of the mail property/message headers mapping. Here's an example where it replaces the previous SpEL based transformer:

```
<chain input-channel="emails"
    ↗ output-channel="handled-emails">
    <mail:mail-to-string-transformer />
    <service-activator ref="emailHandlingService"/>
</chain>
```

If you want to customize the transformation rather than extracting the payload as a string, you can replace the `<mail:mail-to-string-transformer>` here with a regular transformer that accepts an inbound `javax.mail.Message` payload. Extend the `AbstractMailMessageTransformer` class provided by the framework and implement the logic for extracting the message content, as in the following snippet:

```
public class MyMailMessageTransformer
    ↗ extends AbstractMailMessageTransformer<InputData> {
    @Override
    protected MessageBuilder<InputData>
        ↗ doTransform(javax.mail.Message mailMessage) throws Exception {
        InputData inputData = new InputData();
        // populate inputData from message
        return MessageBuilder.withPayload(inputData);
    }
}
```

Then you can add your implementation to the chain using a regular custom transformer setup:

```
<chain input-channel="emails"
    ↵ output-channel="handled-emails">
    <transformer>
        <beans:bean class="example.MyMailMessageTransformer"/>
    </transformer>
    <service-activator ref="emailHandlingService"/>
</chain>
```

The resulting transformer will also extract the mail properties as header values, as in the case of the string transformer. One of the advantages of the latter two approaches over the SpEL transformer is that they automatically provide access to entries, such as the originator of the message and the reply address, which may be useful when handling the message and are important if you want to implement an ad hoc gateway that sends back an email response to an inbound email request. For implementing such a round-trip solution, all you need to do is create a sequence of endpoints that starts with an inbound channel adapter and ends with an outbound one.

10.3 Summary

Email is a widely used modern communication technology that can be used for user interaction, for allowing applications to send notifications or receive requests via email from users, and as a transport mechanism, especially between remote applications. On criteria such as performance and reliability, email can't compete with dedicated messaging middleware, but it has some qualities that make it attractive in Internet communication: it's widely available and accessible to users, it can transfer information over the internet without requiring firewall access, and it supports small to mid-sized attachments.

Spring Integration provides support for sending and receiving emails through dedicated channel adapters, which cover all the major connectivity options and email transfer protocols (POP3, IMAP, SMTP). It provides out-of-the box support for handling the most typical use cases, such as text messages (including HTML) and file attachments, and special situations, such as complex email structures and conversion to POJOs, can be easily addressed by including customized transformers in the processing chain.

This wraps up our introduction to email support in Spring Integration, and it's time to look at our next topic: file support.

Filesystem integration

This chapter covers

- Picking up files from the filesystem
- Scanning and filtering directories
- Writing messages to the filesystem
- Dealing with concurrent writing and reading

The book *Enterprise Integration Patterns* defines four basic integration styles: file transfer, shared database, Remote Procedure Call (RPC), and messaging. This chapter deals with file-based integration. No different from the other integration options, file-based integration is all about getting information from one system into another. In file-based integration, this is done through one system writing to disk and another reading from it. The particular details of the implementation (for example, whether both systems stream to the same large file, or multiple files are being moved around) aren't part of the definition. After reading this chapter, you'll have a solid understanding of the details of file-based integration and how to implement it cleanly using Spring Integration's file handling support.

In the previous two chapters, we looked at messaging through Java Message Service (JMS) and email. In many ways, messaging-based integration is desirable over file-based integration. Unfortunately, many legacy systems still offer file transfer as their only integration option, but when comparing the simplicity of sharing a file to

more modern solutions like JMS or web services, file-based integration is also tempting in many new applications. Having one process write a file in a directory and another read from that directory is often the *simplest thing that might possibly work*. If a simple solution just works, it has earned a right to stay in business.

Enterprises have been trying hard to move away from file-based integration in favor of *service-oriented architecture*, or *SOA* (with various definitions), but most of them have ended up keeping some of the legacy file-based integration points around.

Building new applications that have to be integrated into an architecture that has grown under these circumstances necessitates interaction with the filesystem. In this chapter, you learn how to do this with Spring Integration, and we provide some generic pointers on designing interaction with the filesystem. The chapter explores reading and writing, different ways of dealing with files, how the different components work under the hood, and how to handle several advanced problems you might encounter.

Before going into how to interact with the filesystem, let's carefully examine *why* it makes sense to do so in the first place. An example is in order too.

11.1 Can you be friends with the filesystem?

Most Java developers dread dealing with the filesystem like they dread dealing with concurrency. If you consider yourself a good developer and you really like those things, you might have noticed that applying yourself to using them doesn't make you friends in your team. In the introduction, we associated file-based integration with the simplest thing that might possibly work, but in many cases, dealing with files properly is far from simple. Simplicity is the best friend you can have in your programming career. More often than you'd expect, things like the filesystem can be safely ignored in favor of a memory solution.

A component as common as the filesystem can't be all bad. In UNIX, for example, everything is a file, and working with intermediary files is often the simplest solution. In an object-oriented realm such as the Java Virtual Machine (JVM), dealing with the filesystem requires an extra abstraction, which adds complexity. Table 11.1 lists the advantages and disadvantages of using files.

Table 11.1 Advantages and disadvantages of interacting with the filesystem

Advantage	Disadvantage
Files survive an application crash.	Extra complexity dealing with resources, open streams, locking.
The disk has much more room than memory, so running out of room is less likely.	Filesystem access is much slower than memory access.
Files are easier than more advanced data stores to integrate in other applications.	Filesystem has no atomicity, consistency, isolation, durability (ACID) or Representational State Transfer (REST) semantics, and scalability is more complicated.

If you can get away with the simple solution of using objects on the heap, go for it. But if it doesn't *just work*, look at the filesystem seriously before you jump into even more complex solutions. We look at an example that can best be implemented using file sharing. The rest of the chapter teaches you how you can implement it using Spring Integration's file support.

11.1.1 A file-based collaborative trip diary editor

One of the best examples of filesystem integration is an editor. In this chapter, we talk about a web-based editor for trip details. It would be great fun to elaborate on a vector graphics editor or something like that, but that would be over the top. To keep things simple, let's discuss a plain text editor.

The editor uses a plain text file as its output format, and that would be that if we didn't have to worry about undo history and live collaboration. For this example, we make it a point to worry about those things. In the flight-booking application, users can connect with friends and keep a diary of the trip online. You expect heavy usage, you don't want to bombard the database with updates, and you don't want to keep everything in memory either. Put it this way, the filesystem seems like a decent middle road. To keep the undo history, you create small files with changes and store them alongside the base file. Whenever a change is made in the editor, a file containing that change is created.

Now when the editor is closed, you can reload the file and the full undo history at startup, but the organization used here also allows you to do something even better. You can have multiple editors open, perhaps at different terminals, that all edit the same piece of text. If one editor writes a change to the directory, the other editors pick up the file and update their screen.

The examples discussed here can easily be extended to a rich text editor, a spreadsheet, or something else. The only restriction is that you can define a base file format and a file format for the changes. In figure 11.1 you can see the high-level design of the application.

Let's explore the responsibilities of the components in figure 11.1 in more detail. First, a single client creates a new document to work on, which results in a directory for the document, an empty file as a starting point. After each change, a file with the change is created.

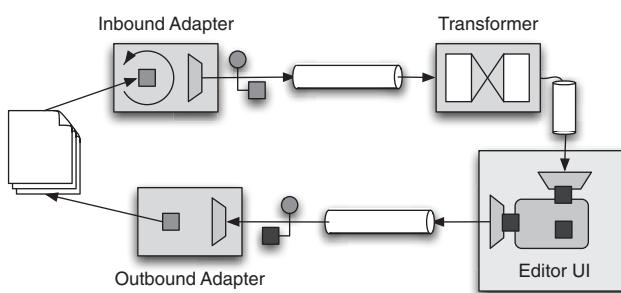


Figure 11.1 A file is picked up by the inbound adapter; it then flows through the transformer. The UI interprets the change and sends changes made by its user back to the working directory.

The file: namespace

To ease the configuration of file-related components, Spring Integration comes with a dedicated file: namespace. To use this namespace, add the following to the root element of your configuration file.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:i="http://www.springframework.org/schema/integration"
    xmlns:file=
        "http://www.springframework.org/schema/integration/file"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation=
        "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/integration/
            spring-integration.xsd
        http://www.springframework.org/schema/integration/file
        http://www.s...org/schema/integration/file/
            spring-integration-file.xsd">
```

Note in particular the addition of `xmlns:file="http://www.springframework.org/schema/integration/file"`. This is where you add the file namespace.

The file: namespace allows you to use the shorthand for file-specific inbound and outbound channel adapters and various file-oriented message transformers. All the different components can also be configured using plain old `<beans/>`. If you need to do more advanced customizations to the configuration, it could be useful to understand the bean configuration. Look at the “Under the hood” section at the end of this chapter.

You might want more fine-grained modifications later on, perhaps to update other clients on a key-by-key basis, but for this example, the solution we outlined in the previous paragraphs will suffice.

If another client joins in now, things get interesting. If both clients watch the directory with the base file and changes, they'll see new files appearing in the directory as the other client edits the file. Each client can then apply these changes to the in-memory model and refresh the screen so that the user is aware of the change. With an appropriate scan rate and change size, a seamless collaboration between different users can be achieved.

We chose this example because it requires the right interactions with the filesystem to be suitable as an illustration of Spring Integration's file support. The first thing you should remember about file support in Spring Integration is the namespace. You don't have to use the namespace to be able to configure the various elements, but typically the namespace is an excellent tool to hide unwanted complexity. Only in advanced scenarios does it become important to consider the underlying plumbing in detail.

In the next section, we use the namespace to configure the file-writing leg of the application.

Change

A change is a consistent set of manipulations to the document. It's recorded between two cursor positions.

11.2 Writing files

Writing a file to the local filesystem is a simple job, so let's start there. To be able to write a file, your application needs several things: write access to the directory where the file should be written, a byte array or a string to write to the file, and a filename to write this data into. If you tried to do this using the raw File API of the JDK, you'd have to write some less-than-elegant boilerplate code that you've probably reproduced dutifully on countless occasions:

```
Writer output = new BufferedWriter(new FileWriter(aFile));
try {
    output.write("This is written to the file in default encoding");
}
finally {
    output.close();
}
```

Spring Integration contains a component that does all these things (see figure 11.2), as detailed in the next few sections. The only thing left to do is wire them up in your application context.

Some concerns related to the intended reader of the file must be taken into account. What do you do about encoding? If writing a file takes a long time, how do you prevent readers from picking up incomplete files? This section focuses on the standard solution to this problem, and at the end of the chapter, we go into more complex scenarios.

The easiest way out of a concurrent read/write bug is to write to a file until it's done and only then move it to a place where the reader can pick it up. This is exactly what the outbound adapter described in the next section does by default.

11.2.1 Configuring the file-writing endpoint

To set up an outbound channel adapter, you use the `<file:outbound-channel-adapter/>`. This element creates a file-writing component that writes the payloads of messages to a directory of your choosing. You can configure several things for this element. Table 11.2 lists the attributes you can use to do this configuration.

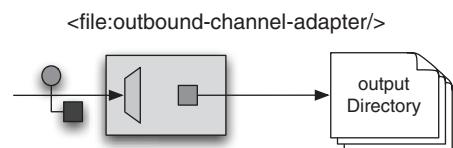


Figure 11.2 Schematic design of the Spring Integration File Outbound Channel Adapter.
The outbound adapter is a passive component that responds to incoming messages by writing their payloads to a file.

Table 11.2 Attributes of the <file:outbound-channel-adapter/> element

Name	Description
id	The ID of the endpoint or the implicit channel leading to it (see channel).
channel	The channel from which the messages that should be converted to files are received. If no channel is specified, a channel is created automatically as with any other channel adapter.
output-directory	The directory to which the files should be written. This directory is a resource, but the default loading rule is always from the filesystem, no matter in what type of ApplicationContext this bean is loaded.
file-name-generator	Allows custom naming strategy; the default strategy is to use a header or the name of the File instance in the payload.
delete-source-files	If delete-source-files is set to true and the payload of the messages is a File, the original file is deleted.

Of these attributes, only id and output-directory are required. In keeping with good practices in Spring Integration components, reasonable defaults are included for the other properties.

WHAT DOES THIS DO?

When a file is received from the input channel, the file channel adapter opens it and looks at the payload. Several payloads, such as byte[], File, and String, are supported. If the payload can be written to a new file, a *temporary* file is opened. This file has a suffix so that readers can prevent opening files that aren't ready yet. Then the adapter writes the contents to the file, and if all goes well, it moves the file to its final name. This name is provided by the `FileNameGenerator` that you can wire up as a bean and inject using the `file-name-generator` attribute.

ALTERNATIVE: <FILE:OUTBOUND-GATEWAY/>

In many cases, you must handle an input file after it's written to a directory. See also section 11.4.8. You can configure a `<file:outbound-gateway/>` for this task. This component behaves similarly to the channel adapter, but it allows you to send the created file to another endpoint immediately. This option is an excellent choice if you must notify another service when the file is written.

WHAT COULD GO WRONG?

Several potential problems must be considered when you write files. You might encounter an `IOException`, for example, because you suddenly lost the ability to write to the disk. In this case, a `MessageDeliveryException` is thrown containing the original `IOException`. There is nothing special about this failure, but Spring Integration allows you to let the exception bubble as a `RuntimeException`.

It's also possible to overwrite an existing file unintentionally. Be careful to implement a custom `FileNameGenerator` in such a way that it doesn't generate the same

filename for files that should be named differently. The default name generator ensure that within the same application context the names are unique, but you'll probably need more control over the filenames. It's also important to avoid overwrites between multiple runs of the application, whether in parallel or in sequence.

Now that we have the tools, let's try them. With the outbound adapter, you can implement the save feature of the collaborative editor.

11.2.2 Writing increments from the collaborative editor

In the sample application introduced earlier, we defined the requirement to track changes incrementally. The files should be easily identifiable and orderable. To make this work, the files are named after the running application key (which is unique for each time the application context is loaded) and a timestamp. Later in this chapter, you learn how to use this information to make sure changes are applied to the other applications in the right order.

Each file will contain a change in a format which allows an endpoint that's processing it to apply it to a string. The details of applying the changes are tricky, but lucky for us they're not relevant to this book. It's not the responsibility of the channel adapter to ensure that the content of the files is in an appropriate format for the reader. This responsibility lies with the application itself. See figure 11.3 for a graphical representation of the writing leg of the application.

At this point it's good to inspect the code related to file writing.

```
<file:outbound-channel-adapter
    channel="outgoingChanges" directory="#{config.diary.store}"
    auto-create-directory="true"
    filename-generator="nameGenerator" />

<bean id="nameGenerator"
    class="com.manning.siia.trip.diary.ChangeFileNameGenerator">
    <constructor-arg value="#{config.processId}" />
</bean>
```

Now you can write files to the work directory, but first you must learn to read files.

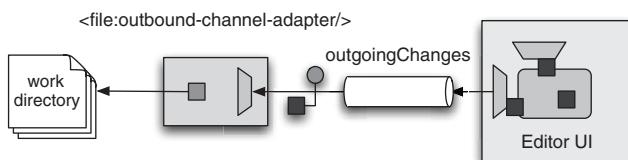


Figure 11.3 The writing leg of the collaborative editor: a String (blue) is passed to the `<file:outbound-channel-adapter/>` as a Message. The String payload is then converted to a file in the work directory.

11.3 Reading files

Reading files is more complex than writing them, but once it's clear which file should be read, it's as trivial as writing to a file.

```
BufferedReader input = new BufferedReader(new FileReader(aFile));
StringBuilder contents = new StringBuilder();
try {
    String line = null;
    while ((line = input.readLine()) != null){
        contents.append(line);
        contents.append(System.getProperty("line.separator"));
    }
} finally {
    input.close();
}
```

This snippet contains some boilerplate, and judging from countless examples of where people have messed this up, it's quite error prone.

AUTOMATIC RESOURCE MANAGEMENT IN JAVA 7 Closing resources properly has become easier in Java 7 with Automatic Resource Management.

Although it's interesting to discuss the low-level correctness of filesystem access, the primary goal of this chapter is to address with the functional complexities of dealing with files. Back to the example.

The Trip Diary module must read files from a directory to update the displayed diary with changes from other editors.

The complexity lies in determining which file to open and when. When a reading process is watching a directory, it has to periodically list all the files in a directory and decide which ones are new. This task may seem simple, but it's tricky. Imagine you're looking at a directory that someone else is dropping files in. How can you know which files to pick up and which files you've already processed? How do you ensure that no one is still writing to that file?

By default, Spring Integration's file inbound channel adapter (depicted in figure 11.4) reads all files in a certain directory, but once it's read, a file isn't picked up again. As a user, you can customize the behavior by specifying which files should be read using `FileListFilters`. More on this later. First let's make sure you understand Java's file API a little better.

11.3.1 A File in Java isn't a file on your disk

Central to the filesystem interaction in Java is the `File` class, which is immutable. This means that once created, it can't be changed. This is a good thing, because as mentioned earlier, immutable payloads make your life much easier once you start

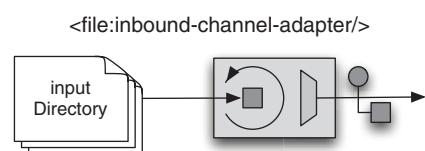


Figure 11.4 Design of the Spring Integration Inbound Channel Adapter

processing messages concurrently. Having a `File` object as a message payload is perfectly safe from a concurrency point of view. But don't be lulled to complacency too soon. We all know that files change all the time, so there must be something more to it.

WHAT'S IN A FILE?

Looking at the `File` source code reveals where the problem is hidden. The `File`'s path is effectively immutable, but most operations on the `File` don't access immutable fields; instead, *they access the filesystem*. You can understand that getting the last modification date requires accessing the filesystem to check whether the file was changed lately. This means you have to be careful to distinguish between operations that access the filesystem and operations that don't. If you need to look at the modification date or the file size to determine if a file should be read, you assume the responsibility to account for concurrent filesystem access. Good, lazy programmers shun responsibility.

KEEPING IT SIMPLE

A common tactic in dealing with this problem is to make sure that while files are being written, reading processes don't see them yet—for example, by writing in a directory that isn't visible to the reading process, or, as we do automatically in Spring Integration, writing to files with an agreed suffix. Then when you're ready writing move the file to where the reading process can see it. By holding the writing process partly responsible, the readers can be much simpler. Exclude in-progress files using a `FileListFilter` and let Spring Integration deal with the visible files with the default behavior.

But how can you make Spring Integration do all that? The next section shows you the details of the configuration.

11.3.2 Configuring the file-reading endpoint

The `<file:inbound-channel-adapter/>` element creates a file-reading adapter. The attributes for this element are listed in table 11.3.

Table 11.3 Attributes of the `<file:inbound-channel-adapter/>` element

name	description
<code>id</code>	The identifier of the endpoint or the implicit channel leading to it (see <code>channel</code>).
<code>channel</code>	The channel on which messages for the files in the input directory should be placed. If no channel is specified, a channel is created automatically as with any other channel adapter.
<code>directory</code>	The directory from which this adapter should read files. This directory is a resource, but the default loading rule is always from the filesystem, no matter what type of <code>ApplicationContext</code> this bean is loaded in.
<code>filter</code>	Specifies a custom filter to determine which files are to be picked up. This attribute overrides the default filters.

Table 11.3 Attributes of the <file:inbound-channel-adapter/> element (continued)

name	description
scanner	Specifies a custom scanner in case the whole scanning logic should be overridden. Because the filter is embedded in the default scanner, this attribute is mutually exclusive with a filter, although a custom scanner can easily delegate to a filter that's injected into it. Look at <code>RecursiveLeafOnlyDirectoryScanner</code> for an example implementation.
comparator	Custom comparator used to determine the order in which files will be received.
filename-pattern	Ant-style pattern used in a filter that's applied to the files in the directory. This filter is <i>added</i> to the default <code>AcceptOnceFileListFilters</code> so that no duplicates are picked up. This option should not be used with the <code>filter</code> attribute.
regex-pattern	Similar to <code>filename-pattern</code> ; <code>regex-pattern</code> interpreted as a regular expression.

Of these attributes, only `id` and `directory` are required. In addition, the `<file:inbound-channel-adapter/>` uses a poller, which can be configured as a sub-element, as with a `<service-activator/>`.

WHAT DOES THIS DO?

You have quite a few options here, but because only `id` and `directory` are required, an inbound channel adapter configuration can be as simple as this:

```
<file:inbound-channel-adapter id="fileGuzzler"
    directory="work" />
```

This configuration would pick up files written to the `work` directory and send them to a channel called `fileGuzzler` wrapped in a message. It would also make sure they're picked up only once during the lifetime of the application.

When neither `filter` nor `filename-pattern` is specified, the `FileReadingMessageSource` that's created will use an `AcceptOnceFileListFilter` by default. This filter remembers all files it has seen before and doesn't let them pass again. The default filter works well in combination with, for example, a `SimplePatternFileListFilter` that excludes unfinished files as agreed with the writing process. If the writing isn't under your control, though, you might have no choice but to write a more elaborate (and more stateful) `FileListFilter` by either extending `AbstractFileListFilter` or implementing `FileListFilter` directly.

When the poller invokes `receive()` on the internal `FileReadingMessageSource`, it lists the files in the directory, adds them to an internal queue, and lets the poller iterate over the files until it satisfies its `maxMessagesPerPoll`. The internal queue is prioritized according to the `Comparator`, if provided. Otherwise, the files come out in natural order.

In the next section, we again apply the tools to the example application. The tricks you just learned will help you implement the reading leg of the application.

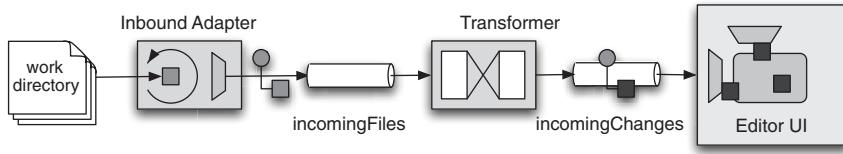


Figure 11.5 Inbound components of the collaborative editor: a file is picked up from the working directory and sent on the `incomingFiles` channel wrapped in a message. It's then transformed into a `String` and passed along to the editor.

11.3.3 From the example: Picking up incremental updates

Now that you know how to read files, let's look at reading the incremental updates from the directory. First, a quick outline of the algorithm.

Given two running editors on the same document, editor A and editor B, files will be written into the directory by both A and B. Editor A wants to read the files from B, or more precisely, the files not written by A itself. See figure 11.5.

If you distinguish each editor uniquely by a key generated on startup, restarting A will result in a running editor C, where $C \neq A$. This is convenient because a newly started editor can read all the changes to the file, and even a restarted editor can read all the files written by the previous editor run by the same user. As mentioned before, you could use a combination of a timestamp and process ID to prevent file duplication.

```

<file:inbound-channel-adapter
    channel="incomingChanges" directory="#{config.diary.store}"
    filter="onlyNewChangesFilter"/>

<bean id="onlyNewChangesFilter"
    class="org.springframework.integration.file.CompositeFileListFilter">
    <constructor-arg>
        <list>
            <bean class="org.sfw.int.file.AcceptOnceFileListFilter" />
            <bean class="...RefuseWrittenByThisProcess">
                <constructor-arg value="#{config.processId}" />
            </bean>
        </list>
    </constructor-arg>
</bean>
  
```

You learned in this section how to configure a file-reading component, and we discussed the more elaborate details of the possible configuration options. In the examples, you can see that the typical boilerplate code is replaced by somewhat more readable declarative XML. The collaborative editor now has both legs to stand on.

Great. Now you can read and write files. But what are you going to do with them inside the application?

11.4 Handling file-based messages

When a file is read and wrapped into a message, you need to do something with it that makes sense within the context of your application. Rarely is a `File` meaningful within

the application domain. If it isn't, it must be converted into something that makes sense, typically an object from the domain. The process consists of two parts: first, the contents of the file are read into memory, and second, the raw data is converted into domain objects. The conversion process is called *unmarshalling*, and plenty of frameworks, such as Spring OXM (discussed in chapter 8), can do it. Spring Integration file support doesn't care how you do your unmarshalling, so even though it's an important concern, it's not discussed in detail in this chapter. The first part of the process, reading the files into memory, is an important responsibility of the file support. We discuss it from several angles in this section.

11.4.1 Transforming files into objects

Many different types of content can reside in a file, but from a high-level view, it's all just bytes. The built-in transformers of Spring Integration can transform a `File` object into either a `byte[]` or a `String`. You can use a `FileToByteArrayTransformer` or a `FileToStringTransformer` respectively for transformations. Transforming a file means your system is now going to deal with it. At this point, you might choose to delete the file, but if the system fails to deal with the file's content successfully and you've already deleted the file, you may lose the data.

Imagine you're writing new products in a directory for your online shop. These products will be dropped in a directory as XML messages and picked up by a process that inserts them into the database backing your web application. First, the XML must be transformed into a `String`, and then an unmarshalling step will prepare the objects destined for persistence through a repository. If the transformer deletes the file, it does so after it finishes the transformation, but it's not unthinkable that subsequent requests are separated by asynchronous handoff. In this case, obviously, the file will be deleted before the objects are persisted, but even if there is no asynchronous handoff, problems can occur because the transaction boundaries are broken. If the process crashes before the file is deleted, but after the inserts are committed, restarting the process might cause duplication.

The next section outlines some common file-handling scenarios and discusses the pros and cons of each approach.

11.4.2 Common scenarios when dealing with files

Managing files on disk from several applications can be done in several ways. A lot of passionate discussion takes place on how this file management should work. In the end, it depends on individual circumstances. Let's look at a few different strategies: (1) letting the file sit in the input directory, (2) deleting the file as soon as it's consumed by a transformer, and (3) simulating a transaction that attempts to make delivering the message and deleting the file an atomic operation.

JUST LEAVE IT

If no harm is caused by picking up the file again when the application is restarted, or if you have a clear way to filter out old messages, it's often best to leave the files in the

input directory. If dealing with a file is a nondestructive operation, it's much easier to recover from a bug or failure that causes the file to be dealt with improperly. The downside of this approach is that from the outside it's impossible to determine, on the basis of its name or location, whether a file was processed. If a new file is created that can be correlated to the input, or if the application can be queried to determine whether a file was processed, this problem can be worked around.

If the files are left on the filesystem, it's usually easy to devise a cleanup strategy after the application is operating for a while and you start running out of space. For example, a script that deletes all files older than a week is not hard to write. It's important, though, to think of how much space you need for the files in advance and also think about purging strategies. If these strategies are complicated, you shouldn't postpone implementing them.

DELETE ON CONSUMPTION

Sometimes incoming files aren't valuable, and dropping a file or two is no big deal. For example, if a weather service gets forecast updates every minute, dropping one when it crashes won't be noticed when the service restarts, so deleting the file from the transformer makes perfect sense. But of the three options listed, this one is the most likely to delete a file prematurely and never deliver the contents to a downstream component. Avoid this solution if you can't afford to lose some data.

SIMULATE TRANSACTION

Some files are too big to just leave them, but you can't afford to lose a single one. In this case, you can't delete the file from the transformer because you could lose the data if the JVM is killed, but you have to delete the file in process. A file-reading endpoint will set the original file on the message as a header. This header can be used to delete a file after it's fully processed. Usually, the process should write a file to a processed directory when processing is complete. You can do this with a `<file:outbound-channel-adapter ... delete-source-files="true" />`, but it's not a real transaction. If the filesystem is full or the process crashes midstream, input files could sit in the input directory even though they're already processed. To mitigate this risk, you can implement an idempotent receiver in the endpoint that processes the file. Then again, it might be easier to just leave the file and write a separate cleanup routine.

11.4.3 Configuring file transformers

The `file:` namespace contains a few more elements: the `<file:file-to-bytes-transformer/>` and the `<file:file-to-string-transformer/>`. Each supports the same attributes, so the only difference is the type of the payload of the outgoing message. Table 11.4 shows a short description for the allowed attributes.

The configuration of this element is almost trivial. Only `input-channel` and `id` are required; in a chain, the element can even be used without any attributes. Setting the `delete-files` attribute to `true` is equivalent to deleting on consumption. This attribute defaults to `false` for a good reason: you wouldn't happy if your application deleted a file even though its content wasn't processed correctly. In some use cases, it's

Table 11.4 Attributes of the <file:file-to-string-transformer/> element

Name	Description
id	The identifier of the endpoint.
input-channel	The channel from which the file messages that should be transformed are received. Can be omitted only if the transformer is part of a chain.
output-channel	The channel to which output of the transformation is sent. When omitted, the result is sent to the reply channel set as a message header.
delete-files	Boolean flag that determines whether the file payloads of incoming messages will be deleted by this component. Defaults to false.

reasonable to have no strict guarantees about picking up files, then the overhead of having a separate cleanup strategy might not be worth the extra security.

By now you're probably ready to see the application work, so we put the ends together in the next section.

11.4.4 Applying incoming changes to the collaborative editor

In the example collaborative editor, the incoming changes contain snippets of text and a position at which they should be inserted in the overall text. We don't dive deeply into the tricky domain at this point, but if you're interested in the details, check out the sources and play around with them for a while. As far as Spring Integration support is concerned, applying changes is only about reading the files and passing the resulting strings along to the editor. The transformer is blissfully unaware of whatever tricks the editor needs to do with the supplied strings.

You can refer back to figure 11.5 to see how the transformer fits in the overall flow of the application.

There is little else to the API for reading files. On the surface, it looks simple, and if the file interaction you're planning from your application is as simple as this, there's no more to it than configuring inbound and outbound adapters and a transformer. As mentioned in the introduction of this chapter, though, the devil is in the details. If the details become important, the next section will help you find your footing when dealing with issues such as locking or files being picked up multiple times or before they're ready.

11.5 Under the hood

This section deals with the nitty-gritty details of filesystem integration. It's interesting for people who need to understand the source code of the spring-integration-file project, but it can safely be ignored without missing out on the power of filesystem integration support. If you need to implement more elaborate file-related use cases, you can avoid painful debugging by understanding the underlying concepts and rules. Reading this section in advance is highly recommended in such cases. Here we discuss the details of ordering and locking when reading files. These concerns are caused by

writing files but surface on the reading end in particular. This section therefore focuses on the `FileReadingMessageSource`.

11.5.1 `FileReadingMessageSource`

The `FileReadingMessageSource` is the core of the `<file:inbound-channel-adapter>`. It's a completely passive component that should be polled on its `receive()` method. The endpoint that the component is wrapped in by Spring Integration takes care of the polling transparently.

When a poll occurs for the first time, it follows the steps depicted in figure 11.6. On subsequent `receive()` invocations, the internal list is synchronized with the file-system only if it's empty or if the `scanEachPoll` flag is set to true. This can have implications on the internal ordering, as discussed next.

ORDERING FILES

In some cases, the order in which files are read is important. The collaborative editor, for example, is highly dependent on the order in which files are processed. The internal workings of the `FileReadingMessageSource` should be understood before assumptions are made about ordering. Before we dive into the code, first we need to understand the problem better.

The file listing returned by the `File`'s `listFiles()` method is a `File[]`, so it's ordered, but the order isn't well defined on different operating systems and under different configurations. To keep a long story short: it isn't a good idea to rely on the ordering of this array. The `FileReadingMessageSource` doesn't. Because ordering is required in some cases, as mentioned, it keeps the files in a `PriorityBlockingQueue` internally to ensure that files *in the queue* are returned in order. As usual, a `Comparator` can be used to control the ordering.

The problem now arises that when files aren't written in order, the queue might have to be reordered after a file was wrapped in a `Message` and sent a channel by a

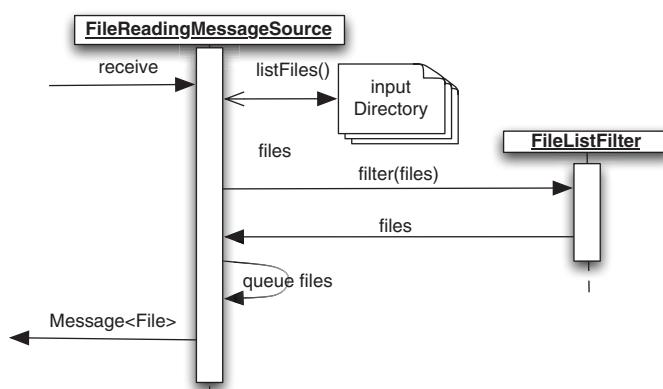


Figure 11.6 Internal behavior of the `FileReadingMessageSource` when polled: (1) get a listing of the input directory, (2) filter the resulting list, (3) add the listed files to the internal queue, and (4) return the head of the queue wrapped in a message.

poller. If the file sent should've come after the file added to the queue just after the ordering was broken, the `FileReadingMessageSource` can do nothing to fix it, because the incorrectly reordered subset is already sent downstream. We wouldn't be telling you this if you couldn't fix it, though.

If you write the files into the directory in the order in which you want them to be received *and* you provide a comparator that puts the files in the desired order, the files will be received in that order. If you do something else, like writing the files in the right order without a comparator, the files might be received in the right order if you're lucky. If you tweak the concurrency settings, this might change. It's never a good idea to depend on message order implicitly. If you can't avoid depending on order, you should use a Resequencer. This makes the system more robust and easier to maintain. Another problem that we already touched on in the introduction of this chapter is the risk of reading unfinished files. Let's consider it in some more detail.

READING UNFINISHED FILES

As mentioned in section 11.3, it's usually important to pick up files only after the writing process is done writing to them. For example, the built-in transformers assume that a file is finished and return a `String` or `byte[]` containing whatever was in the file *at the moment they reached the end of it*. As long as the file isn't opened, there's no problem passing around a reference while other processes are writing to it, but once a transformer (or another endpoint) opens the file for reading, all bets are off.

The best way to avoid premature reading of a file is to have the writing process decide when the file can be picked up. This can be done through a move at the moment the writing is finished or through a barrier that holds file messages until the writer says they're done. The move-when-ready process has a few caveats. One is that the move operation isn't always transactional. For example, an application using this strategy might start failing unpredictably when a system administrator configures a different device for the input directory than the directory where the file is created. This is typically configured from the outside in a properties file, so the type of directories that should be used now must be part of the documentation of the application.

If the writing process is out of the control of the reader, it might still be possible to determine whether a file is ready. For example, if an XML file is being written linearly, the only thing the reader has to do is postpone files that don't have the last end tag. Many formats have similar terminator sequences that can help prevent reading unfinished files.

The problem with reading terminator sequences is that they're located at the end of the file, so you must open the file and read the last part of it. This approach obviously performs a lot worse than the move-when-ready strategy.

Another tricky problem arises when the writing process is writing in multiple parts of the file concurrently. The last part could be done while the middle part is still under construction. BitTorrent uses this strategy, as do other fields, such as image processing. In cases where a file isn't moved when it's done, and looking for terminators is either too inefficient or not possible, you have yet another option: file locking.

File locking works differently on different operating systems, but Java has an abstraction over file locking in the `java.nio` libraries. Spring Integration's file support was expanded to include support for filtering based on `java.nio` in version 2.0. You inject a `NioFileLocker` and make sure the writing process respects locks too. You can also work with lock files that are created, moved, and deleted according to a protocol agreed upon by both writer and reader.

A word of warning is in order here. Locking might seem like a great idea at first, but it's more complex than you might think. If you can find a way to solve your problem without locking, you're doing yourself a huge favor. And with that, we've told you enough about dealing with files. It's time to round up and move on to the next challenge.

11.6 Summary

In this chapter you learned how to work with files using Spring Integration. We discussed the basics of writing files first with the `<file:outbound-channel-adapter/>`. This adapter provides the most common options needed to create output files in a configurable directory. Using the file-writing components of Spring Integration frees you from the responsibility of opening `FileOutputStreams` or `FileWriters` and closing them again. We also discussed the outbound gateway, which is useful if you need to postprocess the written file.

You saw how to read files from the filesystem, prevent duplication, and deal with pattern-matching filters. We discussed the `FileListFilter` extension point that allows you to filter incoming files on any criterion of your choosing. You can go even further than that and implement your own scanner.

When files are read from the system, there are several things you might need to do with them; the most common tasks are implemented in the Spring Integration file transformers. You can easily apply what you learned to implement a file transformer that better suits the needs of your business case.

Finally, we looked under the hood of the components we discussed. We examined ordering and locking in the `FileReadingMessageSource` and urged you to avoid as much as possible relying on these complex options. In the final section, we explored reading unfinished files in more detail, so you have a better idea of the advanced problems you may encounter when implementing file-based integration with or without Spring Integration.

After reading this chapter, you should have a good understanding of how Spring Integration helps you tie your messaging infrastructure into the filesystem. With the adapters described here, it's simple to convert between messages and files. As a final note, remember that the file support in Spring Integration, though convenient, doesn't compete with frameworks that focus on the interpretation and processing of files, such as Spring Batch and other extract, transform, and load tools. It's a healthy idea to use Spring Integration to create the events that drive the specialized process that deals with the files.

In chapter 12, we go into integration through web services, which in many cases is a more robust alternative than the file-based solution.

12

Spring Integration and web services

This chapter covers

- POX- and SOAP-based gateways
- Simple HTTP-based integration

In the previous chapter, we saw Spring Integration’s support for integration through the exchange of files. Although file-based exchanges between systems are still common, enabling two systems to interact through the filesystem can be painful for a number of reasons. Most systems, for example, don’t share a filesystem because they’re likely to be on separate hardware, potentially in different parts of the world. With separate hardware in remote locations, file transfers over the network would have to facilitate this form of communication. Differences in file format and character encoding may create additional challenges.

Using the network directly for message exchange is an appealing alternative. TCP networks offer a ubiquitous transport layer over which calls can be made between systems without many of the downsides of using the filesystem. Because of this capability, web service use has grown to the point where it’s now the default approach for many forms of intersystem communication. This chapter discusses

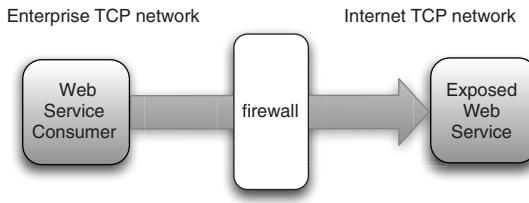


Figure 12.1 Integration over HTTP is simple because the network exists and HTTP traffic passes relatively easily in and out of enterprise networks.

the support offered by Spring Integration for both exposing and consuming web services and the different flavors of web services supported (see figure 12.1).

Let's first make clear what we mean when we talk about web services. There is no clear consensus on a concise definition, so we have to come up with our own. Wikipedia offers a fairly loose definition of a web service: "Web services are typically application programming interfaces (API) or web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services." Using this definition, any form of messaging that includes a network hop can be considered a web service.

With Spring Web Services (Spring WS), Arjen Poutsma followed the good practice of decoupling transport and message. Spring Integration follows in this tradition. Before Spring Integration was released, there was support for web services over Java Message Services (JMS), email, and Extensible Messaging and Presence Protocol (XMPP) built on top of Spring WS. In this book, separate transport mechanisms are addressed in separate chapters.

Most people think of web service as communication using XML messages sent with HTTP. Even though the definition is much broader, this is the form of web service that we focus on in this chapter.

Many people also equate web services with SOAP (Simple Object Access Protocol). SOAP provides a standard for web services defined as follows: "A Web Service is a software component that is described via WSDL and is capable of being accessed via standard network protocols such as but not limited to SOAP over HTTP." WSDL (Web Service Description Language) provides a standard mechanism to describe a web service, but its use is by no means universal, so this definition doesn't reflect the reality of web services even when considering only those using HTTP.

Attempts to standardize web services gained great popularity in the 1990s. Most large software vendors were happily selling a broad range of tools and generating an increasing number of add-ons to the specs. Known as the WS-* specifications, these add-ons covered everything from security to business activities.

The relatively heavyweight and complex approach to web services dictated by SOAP left many people looking for alternatives. One popular alternative is to resort to plain XML, commonly called *Plain Old XML* or *POX*. POX payloads are typically exchanged over HTTP. An increasingly popular option today is to use simpler formats that can be customized for the task at hand. Simpler forms of web services can use HTTP as more than just a transport. The biggest growth in this form of web service has come through

the popularity of RESTful web services, which apply Representational State Transfer (REST) principles.

Spring Integration provides support for exchanging XML-based web service messages over HTTP by building on the Spring WS project for both SOAP-based and POX-based web services. We look at POX support in the first section of this chapter.

Spring Integration provides adapters for working with HTTP directly as well, which can be useful for non-XML or RESTful services. We look at the HTTP adapters in the second section of this chapter. Let's first look at the most common case using XML.

12.1 XML web services with Spring WS

Spring WS is a Spring portfolio project that focuses on contract-first web services with XML payloads. As mentioned before, not all XML web services use SOAP. Spring WS provides support for POX-based web services among others. One of the strongest points of Spring WS is that it decouples application code from the relatively complex requirements of SOAP. Developers can code agnostic to the fact that SOAP is being used except where the application requires control over those details. Without the use of an abstraction like Spring WS, the creation of even a simple SOAP message, as in the following example, is relatively complex; it requires a good understanding of XML APIs and namespace usage.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Body
        xmlns:m="http://www.example.org/stock">
        <m:GetStockPrice>
            <m:StockName>IBM</m:StockName>
        </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

This example (borrowed from W3 schools) shows both the HTTP headers and XML payload for a simple SOAP request. We already need to define a minimum of two elements in the SOAP namespace before we get to the message we want to pass. It's also worth noting that SOAP always uses the POST HTTP method to make requests even in the case of a read-only request for data that arguably is better suited to the use of the HTTP GET method. This is because SOAP uses HTTP as a mechanism for passing data and makes no real use of the HTTP protocol. You'll see later in this chapter that this contrasts with the more HTTP-centric alternatives to SOAP.

The Spring WS project provides a simple programming model that insulates the developer from much of the complexity of web service development. This is

achieved by combining Spring configuration with simple interface-based or annotated endpoints.

Spring Integration extends the simple programming model offered by Spring WS. It adds the power and simplicity of messaging combined with the established enterprise integration patterns (EIP) implementations.

Web service implementation can be complex, but using EIP and messaging generally produces a simpler, easier-to-implement solution. For example, an insurance quote comparison site might need to split the quote request into a number of requests in different formats, invoke a number of different quote providers using different technologies, and then process a number of responses into a single web service response. This can be achieved by using Spring WS alone, but it's much less work to implement using a combination of the two, and that's what the web service support in Spring Integration is intended to do.

To get started, you need to make sure the application can send and receive external messages through Spring WS. In the next section, we look into wiring the beans needed to expose an inbound gateway to do this.

12.1.1 Exposing a Spring WS-based inbound gateway

The inbound web service gateway allows for a POX- or SOAP-based endpoint to be exposed with requests, resulting in the creation of a Spring Integration message that's then published to the channel. When receiving web service requests over HTTP, a front controller servlet must be deployed and configured to pass requests to the Spring Integration inbound gateway. Just as when using Spring WS directly, the front controller is an instance of `org.springframework.ws.transport.http.MessageDispatcherServlet` mapped to a URI on which requests will be received using a standard `web.xml` configuration file.

```
<servlet>
    <servlet-name>si-ws-gateway</servlet-name>
    <servlet-class>org.springframework.ws.transport...
                  http.MessageDispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>WEB-INF/si-ws-gateway-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>si-ws-gateway</servlet-name>
    <url-pattern>/quoteservice</url-pattern>
</servlet-mapping>
```

The `DispatcherServlet` requires a bean implementing the Spring WS strategy interface `org.springframework.ws.server.EndpointMapping` to be provided by the application context configured via the `contextConfigLocation` parameter name. This strategy interface provides support for mapping a given request received by the servlet to a particular endpoint. Spring WS provides built-in support for mapping on a variety

of characteristics of the request from the URI invoked by the caller to the name of the root payload element in the request payload. When using the Spring Integration inbound gateway, it's more common to delegate all received requests to one gateway instance with additional routing being carried out by using the built-in Spring Integration support, for example, the XPath routing capabilities. The following configuration delegates all received requests to a channel named ws-requests.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:int="http://www.springframework.org/schema/integration"
    xmlns:int-ws="http://www.springframework.org/schema/integration/ws"
    xsi:schemaLocation="http://www.springframework.org/...
        schema/integration/ws
    http://www.springframework.org/schema/...
        integration/ws/spring-integration-ws-2.0.xsd
    http://www.springframework.org/schema/integration
    http://www.springframework.org/schema/...
        integration/spring-integration-2.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean class="org.springframework.ws.server...
        endpoint.mapping.UriEndpointMapping">
    <property name="defaultEndpoint" ref="ws-inbound-gateway" />
</bean>

<int-ws:inbound-gateway id="ws-inbound-gateway"
    request-channel="ws-requests"/>

<int:channel id="ws-requests" />
```

By default, the inbound gateway publishes an instance of Spring Integration Message, the payload of which will be the payload of the web service request in the form of an instance of javax.xml.transform.Source. In some cases, it may be desirable to publish the whole web service message rather than just the payload. This is achieved by setting the extractPayload flag false, as follows.

```
<int-ws:inbound-gateway id="ws-inbound-gateway"
    request-channel="ws-requests"
    extract-payload="false" />
```

12.1.2 Calling a web service with the outbound gateway

Invoking a web service from Spring Integration is also extremely simple. The following snippet shows how to configure an example in which the payload of the message received on the requests channel becomes the payload of the message sent to the w3schools temperature-conversion service. The payload of the web service response is then published to the configured reply channel.

```
<int-ws:outbound-gateway
    uri="http://www.w3schools.com/webservices/tempconvert.asmx"
    request-channel="requests" reply-channel="responses" />
```

12.1.3 Marshalling support

Marshalling versions of both the inbound and outbound gateways are provided and are configured by configuring the inbound or outbound gateway with a marshaller and/or unmarshaller to allow automatic conversion between Java and XML for both the inbound and outbound messages.

12.2 Simple HTTP endpoints

SOAP- and POX-based endpoints aren't the only form of web service supported by Spring Integration. It has become increasingly popular to provide web services that don't conform to web service standards such as SOAP and the plethora of additional WS-* specifications. Of particular interest to many is the use of the architectural style known as *REST*, which stands for *Representational State Transfer*. In a RESTful approach, HTTP is used as the service protocol. The term REST originates from the doctoral dissertation of Roy Fielding, one of the authors of the HTTP specification—hence the close relationship between RESTful services and HTTP.

A REST request works as follows: on receipt of an HTTP request using, for example, the `DELETE` HTTP method and a URI of `http://www.example.org/book/123`, the request would be interpreted as a request to delete book number 123.

```
DELETE /books/123 HTTP/1.1
Host: www.example.org
```

But receiving a `GET` request as follows, where only in the HTTP method is different, would result in retrieval of a representation of *book 123*.

```
GET /books/123 HTTP/1.1
Host: www.example.org
```

In HTTP-based communication, for example, it's common not to make use of all the HTTP methods and instead to use `GET` or `POST` for a multitude of purposes, including requests which are really deletions of some sort. This is technically not REST, but sometimes tricks are used to stay as close as possible to the intent of REST without blocking browsers that only support `GET` and `POST`.

Spring 3.0 brought in extensive support for RESTful web services as an extension to the existing Model-View-Controller (MVC) framework for web applications. This support focused on mapping controller methods to requests using both the URI and the HTTP methods while also making it easy to map parts of the URI to controller method parameters. The following code shows an example controller configured to process requests for the retrieval of books:

```
@RequestMapping(value="/hotels/*/books/{book}",
    method=RequestMethod.GET)
public ModelAndView getBook(@PathVariable("book") long bookId) {
    ...
}
```

Spring Integration 2.0 uses Spring 3.0 REST support through a set of HTTP adapters that provide inbound and outbound support for HTTP-based services. Because REST is

a style rather than a specification, HTTP-based services may or may not conform to the REST style. Therefore, although it's possible to use Spring Integration to expose services in a RESTful fashion, the HTTP can also be used to expose services that aren't considered RESTful. Support for all HTTP methods is something that may not exist in some technologies, such as Asynchronous JavaScript + XML (Ajax), that use, almost exclusively, GET and POST HTTP methods and rarely use URI schemes that reference resources.

12.2.1 Processing HTTP inbound requests

The HTTP inbound gateway and channel adapter endpoints allow for the consumption of an HTTP message, which is then used to construct a Spring Integration message for processing. The inbound endpoints, by default, attempt to convert the received message according to a strategy. The default strategy for conversion is based on the content type of the request and converts the received payload to an appropriate Java message. To start processing inbound HTTP requests, you first need a servlet to act as the entry point for requests. You configure this servlet using an `HttpRequestHandlerServlet` mapped to the appropriate URL via the standard `web.xml`. You configure it with a Spring application context containing the Spring Integration inbound gateway. In Spring Integration, this can be configured using the HTTP namespace.

```
<servlet>
    <servlet-name>httpTripInboundGateway</servlet-name>
    <servlet-class>org.springframework.web.context.support.
        HttpRequestHandlerServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/http-ws-servlet.xml
        </param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>httpTripInboundGateway</servlet-name>
    <url-pattern>/httpquote</url-pattern>
</servlet-mapping>
```

The contents of the file `http-ws-servlet.xml` would then be as follows, defining which Spring Integration channel should be used for received HTTP requests in the form of Spring Integration messages. In this example, a reply channel is provided, but it isn't required, and when it's not provided, a temporary channel is used to receive responses.

```
<si-http:inbound-gateway id="httpTripInboundGateway"
    request-channel="tripQuoteRequestsChannel"
    reply-channel="tripQuoteResponseChannel"/>
```

The inbound gateway will configures an instance of either `HttpRequestHandlingMessagingGateway` or `HttpRequestHandlingController`. Which one is instantiated at runtime depends on whether the Spring MVC view technology is being used to render

the response. If an attribute specifying a view name is provided, then the controller version is created; otherwise, the gateway implementation with its own response-rendering logic is used. When the controller-based gateway is used, the HTTP response can be rendered by one of the large number of view technologies supported by Spring MVC, including Java Server Pages, FreeMarker, or a custom implementation of the View interface.

Apart from specifying the view, several other attributes can be set. Table 12.1 shows an overview of the attributes and their functions.

The ability to provide a view gives a great deal of flexibility in generating a response.

Table 12.1 Additional configuration attributes for HTTP inbound gateway namespace

Attribute	Description
view-name	Name of the view to use to render responses. If view-name is provided, a controller class is used to provide the inbound request-processing functionality. If not, the gateway class is used.
extract-reply-payload	Boolean flag indicating whether the type of the reply should be a Spring Integration message or the payload of the message should be extracted. Default is true. Supported by gateway and controller.
reply-key	Key to use for the response message when adding it to the model map that will be passed to the view. Supported by controller.
reply-timeout	Timeout for waiting on a response. Supported by controller and gateway.
message-converters	List of <code>HTTPMessageConverters</code> used to convert the HTTP message.
supported-methods	HTTP methods allowable for this gateway. Supported by controller and gateway.
convert-exceptions	Indicates whether the provided message converters should be used to convert any exception thrown to an HTTP message. Defaults to false. Supported by gateway.
request-payload-type	Indicates to the message converters the desired Java type to convert the request to. Supported by controller and gateway.
error-code	Code to be used to indicate failure in any <code>Errors</code> object passed to the view. Supported by controller.
errors-key	Key to use for an <code>Errors</code> instance containing binding errors if an exception is thrown during request processing. Supported by controller.
header-mapper	Optional reference to a bean implementing <code>HeaderMapper<HttpHeaders></code> . This strategy converts the HTTP headers to and from Message headers. If not specified, the default strategy is used. This maps all headers prefixing the HTTP header name with X-. Supported by controller and gateway.
name	For example, /getQuote. When used with a Spring Dispatcher-Servlet and the BeanNameUrlHandlerMapping, this attribute provides the path prefix for requests that this endpoint will handle. Supported by controller and gateway.

12.2.2 Inbound-only messages using inbound-channel-adapter

For processing HTTP requests which require only confirmation that the request was successfully accepted, Spring Integration provides an inbound channel adapter. The implementation for the channel adapter uses exactly the same classes, `HttpRequestHandlingMessagingGateway` and `HttpRequestHandlingController`, but they are configured not to wait for a reply message. Internally, this is done using `MessageTemplate.send()` for the channel adapter and `MessageTemplate.sendAndReceive()` for the gateway.

Here's an example of using an inbound channel adapter. The `view-name` attribute is specified, so the resolution of the view and the creation of the response is handled as usual in Spring MVC.

```
<int-http:inbound-channel-adapter
    id="httpSmsSubscriptionInboundChannelAdapter"
    channel="smsSubscriptionsChannel"
    supported-methods="GET, POST"
    name="/subscribe"
    view-name="confirmReceived" />
```

Note that in this example, the name of the bean is assigned as a path. This allows the bean to be used with the default behavior of Spring's `DispatcherServlet`, as configured in the following snippet.

```
<servlet>
    <servlet-name>sms-update</servlet-name>
    <servlet-class>org.springframework...
        web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            classpath:http-applicationContext.xml
        </param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>sms-update</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

Now you know how to receive incoming messages over HTTP, but that's only one side of the story. Next up is the outbound part of it.

12.2.3 Outbound HTTP requests

In addition to processing inbound HTTP requests, Spring Integration provides an outbound HTTP gateway capable of transforming a Spring Integration message and optionally publishing the HTTP response as a Spring Integration message. Following is a simple example using the HTTP namespace support. This example uses the URI variable capabilities of the REST template used to make the HTTP request, allowing the

evaluation of an expression on the message to provide a value for insertion into the URI used for the HTTP request. This example uses the RestTemplate to make an HTTP request. The URL can contain a variable, which is filled at runtime. In this case, we expect a location to be contained within the message payload, and it's used to request the weather for that location from Google.

```
<http:outbound-gateway
    url="http://www.google.com/ig/api?weather=${location}"
    request-channel="weatherRequests"
    http-method="GET"
    expected-response-type="java.lang.String">
    <http:uri-variable name="location" expression="payload" />
</http:outbound-gateway>

<si:channel id="weatherRequests" />
```

The HTTP response is provided as a String even though the content type of the response is `text/xml`. Changing the gateway to specify the response type as a Source will facilitate processing as XML.

```
<http:outbound-gateway
    url="http://www.google.com/ig/api?weather=${location}"
    request-channel="weatherRequests"
    http-method="GET"
    expected-response-type="javax.xml.transform.Source">
    <http:uri-variable name="location" expression="payload" />
</http:outbound-gateway>
```

In the previous two sections, we have talked about synchronous messaging. This is natural because HTTP is an inherently synchronous protocol. But there's no reason why we can't use a synchronous protocol for asynchronous communication architectures. The next section discusses the outbound channel adapters that can be used to do this.

12.2.4 Outbound channel adapters

An outbound channel adapter is much like an outbound gateway except that it doesn't send a response on a reply channel. Behind the scenes, the HTTP outbound channel adapter uses the RestTemplate from the Spring Framework. Therefore, the `ResponseErrorHandler` interface is implemented to determine what is an error. The error handler can be provided by setting an attribute on the outbound channel adapter element. If no custom error handler is provided, then an instance of `DefaultResponseErrorHandler` is used. This error handler may not give you the expected behavior. It treats only HTTP response codes of `4xx` and `5xx` as exceptional. This can lead to problems, for example, with a `POST` request for which the response is `301` (moved permanently). This response won't be treated as exceptional, and messages will be silently dropped as the HTTP server discards them without any indication of a problem at the Spring Integration end. Following is an example configuration that posts messages to the configured URL.

```
<http:outbound-channel-adapter id="blogAdapter"
    url="http://www.blogger.com/feeds/5800805938651950760/posts/default"
    channel="flightUpdates"
    http-method="POST"
    charset="UTF-8" />
```

12.3 Summary

Integration over the Internet is becoming increasingly important. It allows easy integration of content and services from a wide variety of sources and support for a large number of different consumers of that content with different format requirements. Although standards such as SOAP are still favored in large corporations, many people are moving to more flexible approaches to web services. The de facto standard on the web is REST because it fits so well with the HTTP paradigm. To integrate systems that take different approaches to web services, you need flexibility in your integration solution. Spring Integration provides this flexibility.

In this chapter, you learned that Spring Integration provides a number of options for web service integration. Spring Integration covers common forms of web service, such as SOAP, REST, POX, and other HTTP-based forms of integration. The namespaces simplify the development of applications that expose or consume web services. Combining Spring Integration with Spring WS and the REST support provided by Spring MVC helps the developer to use EIP patterns when exposing or using web services.

Now that we covered the enterprise type of service integration, we can spend the next chapter looking at more frivolous integration options. As it does email, Spring Integration supports other messaging options suitable for human interaction, such as Twitter, XMPP, and others.

Part 4

Advanced Topics

Chatting and tweeting

13

This chapter covers

- Messaging with Extensible Messaging and Presence Protocol (XMPP)
- XMPP status updates
- Messaging with Twitter

We covered a lot of Spring Integration adapters so far. We saw how to invoke both Simple Object Access Protocol (SOAP)-based and Representational State Transfer (REST)-based web services over HTTP. We saw how to send and receive Messages using the Java Message Service (JMS) API. Those adapters, and most of the others that we covered up to this point, are focused on well-established technologies that are commonly encountered in the enterprise development environment. HTTP is a ubiquitous protocol that provides the foundation of the World Wide Web, and JMS is included among the Java EE specifications. In this chapter, we shift our focus to two different technologies: XMPP and Twitter. Both are well on their way to reaching mainstream status, if they aren't already there. Many enterprise Java developers probably use both technologies daily, but XMPP and Twitter have relative newcomer status in the realm of enterprise Java applications. That trend is likely to shift, and in this chapter, you learn how easy it can be to introduce these technologies into your applications via Spring Integration's corresponding adapters.

13.1 XMPP

XMPP stands for *Extensible Messaging and Presence Protocol*. It defines a fairly generic model for packets containing content and metadata which can be passed along with that content to describe it. As indicated by the name, the two categories of content are messages and presence notifications. Even though XMPP can be applied to many different domains, its most widely recognized role is in the implementation of instant messaging (IM) services. Even if you've never heard of XMPP, you might use it on a daily basis. Several popular IM services are built on it, including Google's chat, Apple's iChat, and many more. The original server implementation is known as *Jabber*.

XMPP supports full-duplex communication. That's a fancy way of saying that the messaging can occur in two directions at the same time. This concept is easy to understand by considering a common case of messages crossing in a real-time chat scenario. Figure 13.1 demonstrates a chat session in which both parties are busy typing a message at the same time rather than taking turns. You probably experience this on a daily basis.

Spring Integration provides adapters for sending and receiving both message content and presence notifications with XMPP. The potential use cases for exchanging message content are relatively obvious. The applicability of presence notifications is less obvious, but it's a distinctly powerful feature of the protocol. Consider the roster features of a typical IM client. When you use an XMPP-based IM client and you see the members of your friends list who are online, that functionality is driven by presence notifications. Likewise, when someone updates their status, your client receives that update in your roster. Those presence notifications are separate from but parallel to the message content. Let's look at the adapters, beginning with the sending of simple message content.

13.1.1 Sending XMPP messages

We start with what's probably the most common use case for XMPP in a Spring Integration application: sending messages. The concept of chat has many practical uses beyond the simple IM scenarios between friends. Two machines can exchange messages as a way to pass data or notify each other about important events. Likewise,

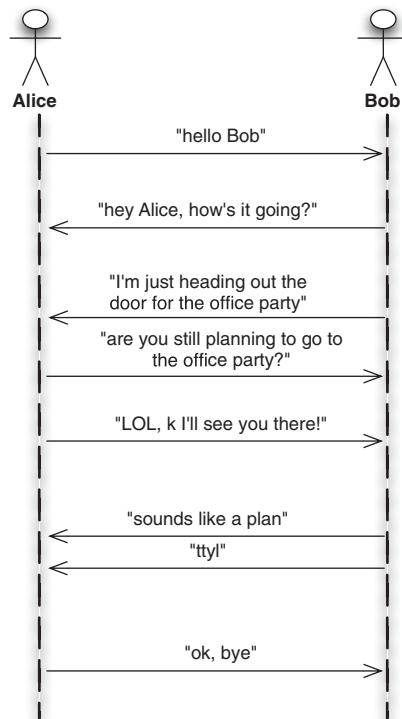


Figure 13.1 XMPP supports full-duplex messaging, meaning that communication occurs bidirectionally, even simultaneously.

machines could send messages to a central chat room. The clients reading messages in that chat room might be a combination of other machines and human users. The topic of such messages might be system-related notifications, such as “server X has encountered a problem,” or they might be events within the business domain, such as “order X has just been processed and all items are available in the warehouse.” Presence notifications could play a role as well, but we explore those adapters later.

The first step to using any of the XMPP adapters is to set up an account and its associated credentials. Once the account is activated, it can be used to create a connection to an XMPP server. Rather than configuring the account information every time you define an adapter, Spring Integration allows you to configure the connection separately so that it can be reused across multiple inbound and outbound adapters. The following is an example of such a connection:

```
<int-xmpp:xmpp-connection id="xmppConnection"
    user="johndoe"
    password="bogus"
    host="somehost.org"
    service-name="mychat"
    port="5222" />
```

It’s generally recommended that you externalize the connection settings in a simple text file as key-value pairs. Using the same example, you could accomplish that by creating a properties file, such as `xmpp.properties`:

```
xmpp.username=johndoe
xmpp.password=bogus
xmpp.host=somehost.org
xmpp.service=mychat
xmpp.port=5222
```

Then, you could associate placeholder variables with the keys from that properties file. To load the properties, you’d include a single `property-placeholder` element in the configuration file:

```
<context:property-placeholder location="classpath:xmpp.properties" />

<int-xmpp:xmpp-connection id="xmppConnection"
    user="${xmpp.username}"
    password="${xmpp.password}"
    host="${xmpp.host}"
    service-name="${xmpp.service}"
    port="${xmpp.port}" />
```

Once the connection is defined, you can set up the adapter. Because we’re focusing on the outbound adapter that publishes XMPP messages, the element to use is the `outbound-channel-adapter`. We already provided all of the connection details with our XMPP connection, so this configuration is trivial. The only thing that’s required is the name of the Spring Integration channel where the application will send messages intended for the XMPP chat.

```
<int-xmpp:outbound-channel-adapter channel="chatChannel" />
```

If for some reason you named the XMPP connection something other than `xmpp-Connection`, then you must provide its name explicitly via the `xmpp-connection` attribute on the `outbound-channel-adapter` element.

```
<int-xmpp:outbound-channel-adapter channel="chatChannel"
    xmpp-connection="myXmppConnection"/>
```

Hopefully, that configuration seems straightforward. Now you can turn to the use of these adapters from application code.

USING XMPP ADAPTERS IN AN APPLICATION

As with any other channel adapter, the point of the abstraction is to enable the application to operate only in terms of the simple generic messages. Nevertheless, for the adapter to construct an XMPP message from a Spring Integration message requires one additional piece of information beyond the chat content. The adapter needs to know to whom the message should be sent. The way to pass that information is through a Spring Integration Message header. The key of that header is provided as a constant value: `XmppHeaders.TO`. The following code provides an example of sending a message along with that header:

```
Message<String> message = MessageBuilder("Hello XMPP!")
    .setHeader(XmppHeaders.TO, "myFriend")
    .build();
chatChannel.send(message);
```

As you know by now, it's not necessary to work directly at the API level. Less invasive alternatives promote a separation of concerns. One of those alternatives is to use the aspect-oriented programming (AOP)-based technique that's driven by the `@Publisher` annotation. Let's look at an example of sending a chat message every time an order is processed.

```
@Publisher("processedOrders")
String processOrder(Order order) {
    this.updateInventory(order);
    this.createInvoice(order);
    this.scheduleShipping(order);
    return this.generateConfirmationId(order);
}
```

This example would work perfectly if you intend to send the chat messages as a byproduct of the service method invocation. That method is already being called as part of your application logic. You're intercepting that normal call in order to also send a chat message each time. Under the covers, Spring Integration is using AOP, where a proxy is created to decorate the service. The only thing you need to add to your configuration to enable such proxies is a one-line directive with an element defined in the core Spring Integration namespace:

```
<annotation-config/>
```

You can even add the default channel to use for any @Publisher annotation that doesn't include one explicitly. That works well if you have a simple application and can rely on a single channel for all interceptor-driven messages.

```
<annotation-config default-publisher-channel="orders"/>
```

Another option is to use a gateway proxy. Even though this technique also relies on a proxy, it's best suited for a different type of use case. We already saw that the intercepted application service method treats messaging as a by-product, and the primary responsibility of the code is still handled by the application service. With the gateway proxy, there's no underlying service implementation to intercept. Rather than having messages as a by-product, the primary purpose of the gateway is to send a message. For that reason, you only need to provide an interface, not an implementation. Such an interface might look something like this:

```
public interface ChatGateway {  
    void chat(String text);  
}
```

Then, the configuration points to the fully qualified name of that interface.

```
<gateway id="chatGateway" service-interface="example.ChatGateway"  
        default-request-channel="chat"/>
```

In this case, Spring Integration creates a proxy that implements the interface. You can then reference that code, typically using dependency injection to make it available from client code. You might wonder why go through this hassle when you could just as easily inject the channel directly or, for more control, use a `MessagingTemplate` instance. The advantage of using the gateway proxy is that it's noninvasive. It's the same justification for using the @Publisher annotation rather than requiring the calling code to have a direct dependency on the Spring Integration API.

In both cases, the underlying implementation provided by the proxy will handle the API-level concerns so that you don't have to write that code. Also, in both cases, you can easily leave this configuration out of the picture for simple unit-level testing. In the gateway case, you can test the calling code by swapping a mock-driven implementation rather than the gateway proxy. Hopefully, you can recognize two of the main Spring themes here: (1) Inversion of Control in the sense that the framework handles the messaging responsibilities for you, and (2) test-driven development enabled by the use of noninvasive techniques based on coding to interfaces rather than implementations.

You may have noticed that we're missing one important detail in the recent examples. We're passing the chat message payload text, but we left out the recipient's user-name. Obviously, that information must be provided so that the messages can be sent to someone. Therefore, we need to provide the TO header. Again, we'd like to keep this noninvasive.

Probably the simplest option, if it'll work from the perspective of the calling code, is to rely on a method argument. Here's a slightly modified version of the ChatGateway interface to demonstrate such an argument:

```
public interface ChatGateway {
    void chat(@Header(XmppHeaders.TO) String username, String text);
}
```

The unannotated argument is used as the payload of a newly created Spring Integration Message, and the argument annotated with `@Header` is added to that message with the annotation's value as the header name. This example relies on the `XmppHeaders.TO` constant because that's what the downstream XMPP adapter expects.

If you're not willing or able to include the username in the method invocation, another option that's even less invasive is to add a header-enricher to the message flow. Assuming the gateway is sending to the channel named `chatChannel`, the following configuration would work nicely:

```
<int-xmpp:header-enricher input-channel="chatChannel"
    output-channel="xmppOut">
    <int-xmpp:chat-to value="johndoe" />
</int-xmpp:header-enricher>

<int-xmpp:outbound-channel-adapter channel="xmppOut"
    xmpp-connection="myXmppConnection"/>
```

One obvious limitation of that particular configuration is that the header value is statically defined. If you need to determine the username dynamically, you can add an expression instead. Imagine a situation where the customer `Account` object is stored in a header already. Perhaps the `Account` instance was passed as an annotated method argument instead of just the username. Here we show just the header-enricher's `chat-to` subelement in isolation to focus on this option.

```
<int-xmpp:chat-to expression="headers.account.chatUsername" />
```

A slightly more advanced option would use an expression that relies on some other bean that's defined in the application context. For example, you might have the `Account` object, but it doesn't provide the user's IM username. At the same time, imagine you have a simple lookup service that can be used to find the chat username from the account number. The following would connect that service to the header-enrichment step.

```
<int-xmpp:chat-to
    expression="@usernameMapper.findUsername(headers.account.number)" />
```

This expression is one example, but the point is that combining the Spring Expression Language (SpEL) with the header-enricher functionality provides considerable flexibility in how you derive the necessary information from the message context at runtime. That in turn enables the dynamic behavior often needed when sending messages in a chat application. Next, we turn to the receiving side.

13.1.2 Receiving XMPP messages

In the previous section, we covered all the details of sending XMPP messages via a Spring Integration channel adapter. Considering that XMPP is a protocol for chatting, there must be a receiver as well as a sender. As we described earlier, XMPP is designed for full-duplex communication, meaning that both parties involved in a chat can play the role of sender and receiver simultaneously. In Spring Integration, each direction of communication is handled by a distinct channel adapter. The outbound channel adapter represents the role of sending, because a chat message is sent *out* from the perspective of the Spring Integration application. Now, we turn to the corresponding inbound channel adapter. Its responsibility is to receive messages that are coming *into* the Spring Integration application.

The inbound adapter has much in common with the outbound adapter. Not only do they rely on the same protocol and the same underlying API for negotiating that protocol, but they both rely on the same connection configuration to use that API. That means that any of the `xmpp-connection` element examples displayed in the previous section would be equally valid on the receiving side. Otherwise, a simple inbound adapter looks like a mirror image of the outbound adapter:

```
<int-xmpp:inbound-channel-adapter channel="chatChannel" />
```

This example assumes the `XmppConnection` instance is defined as a bean named `xmpp-Connection`. Each Spring Integration Message sent to `chatChannel` would contain the body of the received XMPP message as its payload. That body contains text content and is therefore represented as a `String` instance. If you have a reason to use the entire XMPP message instance as the Spring Integration Message payload, then the `extract-payload` attribute can be provided on the channel adapter XML element with a value of `false`.

That pretty much covers the essentials of sending and receiving basic chat messages. In the next section, we explore the Spring Integration support for the other major feature of XMPP: presence messages.

13.1.3 Sending and receiving presence status updates

You now know how to send and receive chat messages, and obviously those are the most important features of a chat service implementation. Nevertheless, when you consider your typical usage of an IM application, you may realize that you also depend heavily on the ability to know which of your friends are available. In fact, most IM applications support status messages that provide more detail than whether someone is online or offline. Your friends are listed on the roster, or buddy list, and some may have status messages that indicate, for instance, “do not disturb,” or they may have a custom message such as “in San Fran until Tuesday.” XMPP supports these types of messages as well, and they’re known as *presence notifications*. If you want to know when your friends become available online, go offline, or change their status message, you can listen to their presence notifications in addition to their chat messages. XMPP

differentiates the two, and therefore Spring Integration provides separate channel adapters for presence notifications. As it has for the chat message support, Spring Integration has a pair of inbound and outbound adapters for presence.

Here's an example of a simple inbound adapter for receiving presence notifications:

```
<int-xmpp:presence-inbound-channel-adapter channel="presenceChannel"
    xmpp-connection="xmppConnection"/>
```

Note that the presence adapter can be defined with an element from the same `xmpp` namespace. The difference is that this presence adapter is qualified as `presence-inbound-channel-adapter`, whereas chat messages are received on the unqualified `inbound-channel-adapter`. The reasoning is that the more common feature is the chat messaging. As with the chat adapters, the `xmpp-connection` attribute is optional. As long as the name of the `XmppConnection` bean is `xmppConnection`, the attribute is unnecessary.

To send presence status updates, you can define the mirror image outbound adapter as shown here:

```
<int-xmpp:presence-outbound-channel-adapter channel="presenceChannel"
    xmpp-connection="xmppConnection"/>
```

You may decide to use the same channel for chat and presence messages, but by providing distinct adapters for each, the Spring Integration support allows you to make that decision on a case-by-case basis. If you were to implement a chat service, you'd most likely maintain the separation of concerns. One channel can be dedicated to inbound chat messages, and another can be dedicated to outbound chat messages. Likewise, you can provide one inbound and one outbound channel for transferring the presence notifications. Here again, the `xmpp-connection` attribute is unnecessary if the referenced bean is named `xmppConnection`.

As you've seen thus far in this chapter, XMPP is a useful protocol that spans a variety of interesting use cases. It caters well to building a chat server, but that only scratches the surface. Considering the trend toward highly distributed applications and event-driven architectures, XMPP can fill an important gap where applications, not just people, need to communicate with each other and notify each other when they come online or go offline. The Channel Adapters provided by Spring Integration make it easy to add such behavior to your own application without having to dive into the depths of XMPP or code directly against any XMPP client library.

In the next section, we discuss Twitter. We also define different types of messages, or *tweets*. Some are public; others are private. Spring Integration again provides corresponding Channel Adapters so that you can easily integrate the entire spectrum of Twitter features into your application.

13.2 Twitter

The first thing most people think of in relation to Twitter is the timeline. When someone tweets, they're posting a status update or, more generally, some comment that will then appear in that user's timeline. Likewise, if you use a Twitter client application,

your standard view is most likely the combined timeline that displays the tweets of all the users' you follow as well as your own tweets. Twitter provides additional features through its API. You can perform a search for all tweets that include some interesting text, regardless whether those tweets were posted by users you follow. Similarly, you can find all tweets that mention your own username, and those results also may include tweets from users whom you don't yet follow. Finally, Twitter provides support for sending and receiving direct messages, which are private alternatives to the more common broadcasting that's visible to everyone. Because Twitter provides so many different options, Spring Integration provides a wide variety of Twitter adapters. Table 13.2 shows the name of each Twitter adapter element that can be configured in XML and the corresponding role of that adapter in terms of how it uses the underlying Twitter API.

Table 13.1 Spring Integration Twitter adapters

XML element name	Twitter role
outbound-channel-adapter	Adds a Twitter status update for the authenticated user
dm-outbound-channel-adapter	Sends a Twitter direct message to a single recipient
inbound-channel-adapter	Reads the Twitter timeline for the authenticated user
dm-inbound-channel-adapter	Reads the Twitter direct messages for the authenticated user
mentions-inbound-channel-adapter	Reads any tweets that mention the authenticated user
search-inbound-channel-adapter	Reads any tweets returned by the provided query

The remainder of this chapter focuses on the adapters listed in the table. You'll see that Spring Integration covers the whole spectrum of Twitter functionality. With minimal configuration and no direct dependency on any Twitter API, you can easily add a Twitter search feature, or direct messaging, or plain old tweeting of status updates.

We start with the simplest case: Twitter search. The reason search is simpler than most of the other adapter types is because the underlying API enables search capabilities without requiring authentication. Let's begin this tour of Twitter adapters with the ability to convert search results into inbound Spring Integration Messages.

13.2.1 Receiving messages from a Twitter search

One of the most common uses of Twitter in a Spring Integration application is to perform a search periodically and send the results downstream in Messages. The element to define in XML-based configuration is the `search-inbound-channel-adapter`. It also happens to be simple in its configuration because a Twitter search doesn't require authentication. We explore other Twitter adapters that do require authentication because they're focused on a particular user, but first let's take a look at an example with the search adapter.

```
<int-twitter:search-inbound-channel-adapter channel="tweets"
    query="#springintegration">
    <int:poller fixed-rate="60" time-unit="SECONDS"/>
</int-twitter:search-inbound-channel-adapter>

<int:logging-channel-adapter id="tweets"/>
```

As you can see, the poller subelement dictates that the Twitter search should be performed every minute. The results are passed to the channel named tweets, and in this case, that channel is created by a logging-channel-adapter. Each Message passed to that channel will contain a single tweet from those search results as its payload. The actual instance type is a Tweet class defined by Spring Social. That class defines a number of properties, listed here. This Tweet type is common to all of the adapters you see in this chapter. The following excerpt shows all of the instance variables defined on the Tweet object:

```
private long id;
private String text;
private Date createdAt;
private String fromUser;
private String profileImageUrl;
private Long toUserId;
private Long inReplyToStatusId;
private long fromUserId;
private String languageCode;
private String source;
```

Knowing those properties, you can control the content of downstream messages by using SpEL. Consider the previous logging example. You could add the expression attribute to have more control of the message to be logged.

```
<int-twitter:search-inbound-channel-adapter channel="tweets"
    query="#springintegration">
    <int:poller fixed-rate="60" time-unit="SECONDS"/>
</int-twitter:search-inbound-channel-adapter>

<int:logging-channel-adapter id="tweets" expression="payload.fromUser +
    ' : ' + payload.text + ' [' + payload.createdAt + ']'"/>
```

You might be doing other things downstream besides logging. A more general approach would be to define a transformer element. Then you can reuse that transformer in different message flows.

```
<int-twitter:search-inbound-channel-adapter channel="searchResults"
    query="#springintegration">
    <int:poller fixed-rate="60" time-unit="SECONDS"/>
</int-twitter:search-inbound-channel-adapter>

<int:transformer input-channel="searchResults" output-channel="tweets"
    expression="payload.fromUser + ' : ' + payload.text +
        ' [' + payload.createdAt + ']'"/>

<int:service-activator input-channel="tweets"
    ref="searchResultAnalyzer"
    method="addResult"/>
```

At this point, you know everything you need to know to periodically retrieve Twitter search results as Spring Integration Messages. Another popular use for Twitter adapters on the inbound side is to read your Twitter timeline. It requires more configuration because you must provide the information necessary for configuration of authentication details. Whereas the search operation can be handled with an anonymous Twitter template instance, many other operations, including reading your timeline, require authentication. We cover the timeline reading adapter shortly, but first we discuss how you should provide the configuration necessary for creating a Twitter template instance that can pass OAuth credentials.

13.2.2 OAuth configuration for the Twitter template

The `TwitterTemplate` is provided by Spring Social. It implements a `Twitter` interface, and that interface provides access to a variety of operations instances which define the methods that all of these Spring Integration Twitter adapters depend on to interact with the Twitter API. As with other Spring templates, it provides a higher level of abstraction to simplify invocation of the underlying operations. In the case of Twitter, it means you can configure it once, share it across your application, and invoke those Java methods rather than performing REST requests directly against the Twitter API.

In the earlier search adapter examples, you didn't see the `TwitterTemplate` instance because it isn't a required reference attribute on the `search-inbound-channel-adapter` element. If not provided explicitly, it's created behind the scenes. But if you were to look into the search adapter's code, you'd see that it does delegate to the `TwitterTemplate`. The reason it doesn't need to be created explicitly is because the search adapter doesn't require authentication. To perform any Twitter operation that does require authentication, a `TwitterTemplate` must be configured in the Spring configuration so that the authentication credentials can be provided. The authentication mechanism used by this template is OAuth, and its credentials consist of a consumer key and an access token as well as their associated secrets. To learn more about OAuth, visit <http://oauth.net/>. If you're looking for information specific to the use of OAuth by Twitter, then read through the Twitter OAuth FAQ list at http://dev.twitter.com/pages/oauth_faq.

To create a `TwitterTemplate` with OAuth capabilities, you must first register your application at <http://dev.twitter.com>. Upon registration, you receive a consumer key and an associated secret. On the developer site, you should see a link to register your application. When filling out the form, be sure to specify the Application Type as Client. You must also click on the Access Token button to receive the access token and its associated secret. Be careful to store all of the keys and secrets in a safe place.

In the examples throughout this chapter, the OAuth configuration properties are stored in a file. Such a file can be protected so that your application has read access but other applications don't. The properties file might be named `oauth.properties`, for example, and would look something like this:

```
twitter.oauth.consumerKey=abC012Def3G45abC012Def3G45  
twitter.oauth.consumerSecret=abc012Def3G45abC012Def3G45
```

```
twitter.oauth.accessToken=12345678-abC012Def3G45abC012Def3G45
twitter.oauth.accessTokenSecret=abC012Def3G45abC012Def3G45
```

Then, the TwitterTemplate can be configured as a bean definition using property placeholders.

```
<bean id="twitterTemplate"
    class="org.springframework.social.twitter.api.impl.TwitterTemplate">
    <constructor-arg value="${twitter.oauth.consumerKey}" />
    <constructor-arg value="${twitter.oauth.consumerSecret}" />
    <constructor-arg value="${twitter.oauth.accessToken}" />
    <constructor-arg value="${twitter.oauth.accessTokenSecret}" />
</bean>
```

TWITTERTEMPLATE AND SPRING SOCIAL Spring Social 1.0 was released during the development of Spring Integration's version 2.1. As of that version, Spring Integration builds on the TwitterTemplate included in the Spring Social library. The examples in this chapter feature that template. But if you're using Spring Integration 2.0, there's a Twitter4jTemplate instead, and as its name suggests, it builds on the Twitter4J library. The configuration is almost identical. Both take the OAuth configuration values as constructor arguments, and they even appear in the same order. That means the only necessary change is in the fully qualified classname in the template's bean definition.

Now that you know how to configure a TwitterTemplate instance that uses OAuth configuration properties, let's discuss some of the adapters that require authentication. We begin with the inbound Twitter adapter for reading your timeline.

13.2.3 Receiving messages from your Twitter timeline

Along with the search functionality you saw earlier, reading the timeline is the other major use case for an inbound adapter accessing Twitter. Unlike searching, though, timeline reading requires an authenticated connection. You have to be logged in because the timeline is associated with your user account. As you saw in the preceding section, OAuth is the mechanism used to establish an authenticated connection.

If you've created a TwitterTemplate instance that has the required OAuth properties, as shown in the earlier excerpt, then you can provide a reference to it when creating a timeline-reading Twitter adapter. Here's what the adapter looks like:

```
<int-twitter:inbound-channel-adapter id="timelineReadingAdapter"
    twitter-template="twitterTemplate" channel="inboundTweets">
    <int:poller fixed-rate="10" time-unit="SECONDS"
        max-messages-per-poll="25" />
</int-twitter:inbound-channel-adapter>
```

That adapter has a poller, and it's configured to poll every 10 seconds. With each poll, it attempts to read up to 25 tweets from the timeline of the user whose OAuth credentials are provided to the template. One thing you might notice is that the element defined in the namespace for this adapter is named inbound-channel-adapter, whereas the adapter that performed a Twitter search was more explicitly named

search-inbound-channel-adapter. The rationale is that reading the timeline is the most obvious thing you can do with Twitter. If you have a Twitter client on your desktop or phone, when you start that client application, the default view is most likely the timeline. In the next section, we explore what's probably the most popular use of Twitter for *outbound* adapters: updating your Twitter status.

13.2.4 Sending messages to update your Twitter status

In the previous section, you saw how to read the Twitter timeline with an inbound adapter. If you hadn't already known, you learned that the Twitter timeline is a collection of status updates, typically called tweets, from all of the Twitter users whom you follow. In this section, we explore the other side of that relationship. When you update your status by posting a tweet, all of your followers should soon see the update in their timelines.

As with reading the timeline, you must be authenticated with valid Twitter account information before you can send status updates. The configuration for the OAuth credentials is the same for all adapters that require authentication. In fact, each of these adapters delegates to the underlying TwitterTemplate instance, and your job is to ensure that the TwitterTemplate is configured properly, including the authentication credentials if needed. Only the Twitter search adapter can be used without authenticating, so in all other cases, the TwitterTemplate should be configured with the full OAuth credentials. Because the TwitterTemplate can be used concurrently, you'd most likely define only a single TwitterTemplate instance for a given Twitter user account. Then you can reference that template from each of the adapters that you might be using in an application as long as those adapters are intended to represent that account.

Once the TwitterTemplate is defined, the status updating adapter is trivial to define. Because updating your status is generally synonymous with tweeting, the adapter on the outbound (or sending) side that's used for updating status has the unqualified element name `outbound-channel-adapter`. It's the mirror image of the timeline-reading adapter on the inbound side.

```
<int-twitter:outbound-channel-adapter id="statusUpdatingAdapter"
                                      twitter-template="twitterTemplate"
                                      channel="outboundTweets" />
```

Like Twitter itself, that's pretty simple. Also like Twitter, your Message text should be within the limit of 140 characters. In the next section, we look at another interesting use case that's common with Twitter: the ability to receive any tweets that reference your own username.

13.2.5 Receiving messages from Twitter retweets, replies, and mentions

Status updates can be retweeted or replied to by those who follow you and see the status update in their timeline. Perhaps more interesting is that someone who doesn't follow you but happened to notice your tweet, perhaps in a search result, might

decide to retweet it. Likewise, they might decide to start following you and reply to the tweet. In some respects, a tweet is similar to a message sent to a chat room when using an IM application or XMPP, except for two key differences. First, the chat room is much larger and less restrictive, basically the Internet itself. Second, you may or may not be posting a tweet in an attempt to solicit responses.

Even if you don't expect or intend to provoke responses, others might be inspired to extend your isolated tweet into a dialog. You might be tweeting about something you find interesting without any intention of starting a discussion, but it could grow into an interesting discussion that pulls in people whom you had never known. On the other hand, you might initiate such a discussion more intentionally by posting an interrogative or poll message, such as "Which one of these laptops should I buy?" Compared to an IM or a chat room, a dialog via Twitter might be an unintended but welcome consequence and may include people who happened to stumble upon one of the tweets in that dialog. Twitter doesn't organize tweets into threaded discussions: any tweet may have an in-reply-to value. It's less structured than a chat room, but that lack of structure is what differentiates Twitter as an open-ended form of messaging among a massive community. In the next section, we walk through the process of monitoring Twitter for any mentions of your username in retweets or replies.

When it comes to messaging styles, you can think of Twitter as a publish-subscribe scenario in which the person posting a tweet is the publisher and all of that person's followers are the subscribers. Considering that all of the tweets are visible to anyone who performs a search or visits the person's public timeline, the pool of subscribers extends far beyond just the followers.

Once a tweet is posted, those subscribers may decide to retweet if they want to reiterate the point of the original tweet and make sure that their followers see the tweet. If they want to provide some commentary of their own or an answer to a question, then they can reply instead. The difference is subtle, but all retweets and replies share one thing in common: they contain the original publisher's username, including the @ symbol that precedes all Twitter usernames.

In addition to retweets and replies, someone may refer to someone else's username in a Twitter message, such as "I just had dinner with @somebody." If you want to find all mentions of yourself, you could perform a search on your username, but because it's such a common operation, the Twitter API provides explicit support for reading all mentions of the logged-in user's username. Likewise, Spring Integration provides an inbound channel adapter for reading those mentions. It's configured almost identically to the basic timeline-reading adapter, but with a different element name. Like the timeline-reading adapter, it requires a template instance injected with the user's OAuth configuration properties.

```
<int-twitter:mentions-inbound-channel-adapter id="mentionsAdapter"
    twitter-template="twitterTemplate" channel="mentions">
    <int:poller fixed-rate="10" time-unit="SECONDS"
        max-messages-per-poll="25"/>
</int-twitter:mentions-inbound-channel-adapter>
```

Before taking this discussion too far, we should also consider that Twitter provides an alternative for direct messaging. Unlike the open-ended internet-as-chatroom approach of posting a status update that potentially starts a dialog, with direct messages, you send to a single explicitly specified recipient. This goes to show that Twitter provides both publish-subscribe and point-to-point messaging semantics. The difference between the two in Twitter terms is extreme: either broadcast to the internet or send a message to exactly one person. Let's now look at the latter option.

13.2.6 Sending and receiving direct messages via Twitter

We mentioned that Twitter, in general, supports a broad publish-subscribe style of messaging. Although it's probably not as popular, Twitter also supports a point-to-point style via direct messages. The concept is simple. If you want to send a message to someone who follows you, and you want only that person to receive it, then use direct messages instead of updating your status.

When considering the classification of communication options, direct messaging with Twitter seems to sit between email and chat. Just as some people refer to Twitter public timeline posting as *microblogging*, the direct message functionality may be thought of as *micromailing*. In both cases, the essential characteristic of the Twitter alternative is its enforced brevity. From a cultural perspective, it's interesting to see the wild success of a technology that seems to be defined by a limitation. Even though the content and size of messages may have more in common with those of an IM chat session, the request-and-reply interaction typically feels more like email. To some degree, those who frequently use direct messaging with Twitter embrace the asynchronous nature of communication. They accept the reality that even if other parties are currently available, they may prefer to respond when it's more convenient for them. Using a phone-call analogy, it's the same reason voicemail is so commonly used even on a mobile phone that's most likely in the presence of its owner most of the time. When you send a colleague a message like "Want to grab a beer after work?" you probably don't need to know the answer immediately. Likewise, the receiver will likely perceive it as more polite that the sender acknowledges the question as a lower priority than one that would necessitate a synchronous phone call. It's as if Twitter gives the *sender* the option to go straight to voicemail. Enough philosophizing. Let's take a look at the Spring Integration Channel Adapters that support direct messages.

THE OUTBOUND CHANNEL ADAPTER FOR DIRECT MESSAGING

The outbound channel adapter for sending direct messages via Twitter looks similar to the status updating channel adapter. The key difference is that its element name is qualified with the `dm-` prefix. Because sending direct messages requires authentication, you also need to provide a reference to a fully configured `TwitterTemplate` instance. That configuration must include the OAuth credentials.

```
<twitter:dm-outbound-channel-adapter id="dmOutAdapter"
    twitter-template="twitterTemplate"
    channel="dmOutChannel" />
```

There's one more requirement when sending a direct message via Twitter. You must somehow specify the user to whom you'd like to send the message. As you'd likely expect by now, the target user information should be specified in a message header. There's a constant defined to represent that header's name: `TwitterHeaders.DM_TARGET_USER_ID`. Here's an example of programmatically sending a Spring Integration Message that ultimately triggers the sending of a Twitter direct message. This example assumes the configuration of the `dmOutChannel`, as shown in the earlier `dm-outbound-channel-adapter` example.

```
Message<String> message = MessageBuilder.withPayload("hello")
    .setHeader(TwitterHeaders.DM_TARGET_USER_ID, "someuser")
    .build();
dmOutChannel.send(message);
```

This code would send a direct message whose content is `hello` to the user whose name is `someuser`. Let's look at the receiving side next.

THE INBOUND CHANNEL ADAPTER FOR DIRECT MESSAGING

The receiving side is simpler because the target user ID is a responsibility of the sender. All you need to do on the receiving side is make sure you've configured a `TwitterTemplate` with the proper user information and OAuth credentials. Then, you can reference that `TwitterTemplate` from the `dm-inbound-channel-adapter` element. Any direct messages sent to that user will be received by that channel adapter. Here's an example.

```
<twitter:dm-inbound-channel-adapter id="dmInAdapter"
    twitter-template="twitterTemplate"
    channel="dmInChannel" />
```

The inbound channel adapter that receives direct messages is practically identical to all of the others that Spring Integration provides for receiving tweets. The element names are different, but the template reference configuration looks the same across all of them, and nothing else is required. With the preceding information and examples provided, you'll be able to work with not only the status update Tweets that appear in the timeline but also with search results, mentions, and direct messages.

13.3 Future directions

In this chapter, we covered Spring Integration's support for any XMPP-based chat service and support for the full range of Twitter options: search, timeline status updates, mentions, and direct messages. Spring Integration will continue to evolve to support additional chat and IM protocols, such as Internet Relay Chat (IRC) and Short Message Peer-to-Peer Protocol (SMPP). In fact, some prototypes have already been created in the Spring Integration sandbox repository for both of those.

Likewise, Spring Integration will evolve to provide many more channel adapters in the social media realm. The relatively new Spring Social project provides support for a number of different social media technologies, and it's relatively easy for Spring Integration to build adapters on top of those underlying APIs. The supporting classes provided in the Spring Social project take the form of templates, strategies, and other

common idioms of the Spring programming model. Twitter is one of the supported technologies, and as of version 2.1, Spring Integration relies completely on that library's `TwitterTemplate`.

Other technologies supported by the Spring Social project will be represented by new channel adapters added in future versions of Spring Integration. In most cases, the template implementations for the various social media sites will be built on top of Spring's `RestTemplate`, and many of them will depend on common OAuth configuration just as you saw with Spring Integration's current Twitter support. You can expect to see support for Facebook, LinkedIn, GitHub, Gowalla, and more. In the meantime, you can rely on any of the Spring Social project's template implementations from within simple service-activator elements defined in a Spring Integration configuration. We highly encourage such usage now because even when new channel adapters are added and the corresponding namespace support is provided, the underlying functionality will be the same. The new Spring Integration adapters will be invoking the same templates that you can go ahead and start using in the meantime.

13.4 Summary

XMPP and Twitter are aimed at different use cases. One distinction is that XMPP focuses on bidirectional messaging in real time. That *presence* earns one of the letters in the protocol acronym shows that *being online* and hence available for chatting is an important aspect of the intended usage. With Twitter, on the other hand, people tend to "catch up" when they have free time. When updating your status, you don't expect people to respond immediately and likely don't sit there waiting for that to happen (at least we hope you don't). In terms of expected response time, even direct messages in Twitter are typically considered more like miniature email messages than chat messages.

Despite those differences, XMPP and Twitter also have similarities. Both require that you register with a username, and both provide ways to keep track of your friends. As you saw throughout this chapter, Spring Integration also treats them as consistently as possible and provides header-enricher components for both so that you can dynamically determine the intended recipient of a message. Even more important, as with all of the adapters you've seen so far in the book, those described in this chapter enable you to easily interact with XMPP and/or Twitter without having any direct dependency on the underlying APIs. You can configure the Channel Adapters and then connect the associated Message Channel to other components such that you're dealing only with the text content of the messages. Your code can be extremely simple, depending only on strings—a prime example of that clean separation of concerns we emphasized so early in the book, which is the essence of the Spring Integration framework.

This chapter concludes part 3 of the book. You have been presented with the most typical integration options. Other options can be implemented as extensions of Spring Integration or integrated as generic Service Activators, Gateways, or Channel Adapters. In part 4, we discuss advanced topics such as concurrency, performance, and monitoring. The upcoming chapters build on the knowledge you've gained thus far and give you the tools to help you maintain solid integration solutions.

Monitoring and management

This chapter covers

- Tracking message history
- Adding wire taps to message channels
- Using Spring Integration's JMX support
- Invoking operations via the Control Bus

When it comes to monitoring message-driven applications, meaningful statistics can be gathered from two perspectives: (1) component-centric data can be acquired by monitoring the general message traffic at the level of a Message Channel or Message Endpoint; (2) message-centric data can be acquired by recording every Message Channel or Message Endpoint that a particular Message passes. The former is useful for detecting trends in throughput. The latter is useful for tracking the detailed history of a given Message even though it traverses a pipeline of loosely coupled components, possibly spanning many different threads of control along the way. In this chapter, we discuss both of these perspectives. We also look at Spring Integration's support for the Wire Tap and Control Bus patterns as described in *Enterprise Integration Patterns* (Hohpe and Woolf, Addison-Wesley, 2003).

14.1 Message history

To a large degree, software architecture is the art of evaluating trade-offs. Almost every advantageous characteristic of a given design has a cost. For example, a higher level of abstraction may offer the advantage of a simpler developer experience, but its cost might be extra overhead. One of the most important roles of an architect is to navigate such trade-offs and determine if the advantage gained is worth the cost.

When it comes to a message-driven architecture, the most significant advantageous characteristic is the loose coupling of components. That brings a number of benefits, as we've hopefully made clear throughout this book. Channel Adapters can be swapped across a wide variety of transports and protocols. Service Activators can be reconfigured to point to different objects providing business functionality. Scripts can be refreshed dynamically at runtime. And so on.

If loose coupling is the advantage, what is the trade-off? In the course of its flow, each message may traverse a number of different components. The fact that those components are decoupled means that no single component plays a centralized role of tracking where the message has been. As any developer knows, it won't be long before monitoring requirements make their way into the high-priority list for any application. With that in mind, Spring Integration provides a mechanism for tracking the flow of each message without compromising loose coupling or resorting to a single, centralized resource. This mechanism records not only *where* the message has been but also *when* it was there. The key to this strategy is that the historical data is written directly to the Message. When enabled, a header is dedicated to this purpose.

To enable such tracking, you add an element from the core namespace anywhere within the configuration:

```
<message-history/>
```

The following example begins with an inbound adapter, passes through a transformer, and ends with an outbound adapter. Two channels are involved as well. The configuration could be more concise if you relied on implicit channel creation based on the adapters' id attributes or the transformer's input-channel attribute, but the goal here is to make the example as clear as possible. For that reason, the configuration defines all channels and endpoints explicitly. In the logging-channel-adapter, we specify that the full message should be logged, which ensures that the history header value is included in the log output.

```
<message-history/>

<inbound-channel-adapter id="poller" channel="transformChannel"
    expression="'foo'">
    <poller fixed-delay="10000"/>
</inbound-channel-adapter>

<channel id="transformChannel"/>

<transformer input-channel="transformChannel"
    expression="payload.toUpperCase()" output-channel="logChannel"/>
```

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

<logging-channel-adapter id="logger" channel="logChannel"
    expression="headers.history"/>
```

To run this example, use the following main method. Because the inbound channel adapter is a Polling Consumer, its task is scheduled automatically. The bootstrap process only requires instantiating the application context.

```
public static void main(String[] args) {
    new ClassPathXmlApplicationContext("context.xml", Demo.class);
}
```

After running for about 30 seconds, the output looks like this:

```
INFO : org.springframework.integration.handler.LoggingHandler -
    poller,transformChannel,logChannel,logger
INFO : org.springframework.integration.handler.LoggingHandler -
    poller,transformChannel,logChannel,logger
INFO : org.springframework.integration.handler.LoggingHandler -
    poller,transformChannel,logChannel,logger
```

Earlier, we mentioned that the history-tracking mechanism informs you not only where the Message has been but also when it was at each component. From the output, you can see that it records the name of each component it passes, but the time tracking seems to be lacking. What's happening here is that only the result of the history header value's `toString()` method is shown. To avoid a lot of extra noise, in typical logging scenarios, that method is implemented in a purposefully simple way. The actual value of the history header is essentially a list of key-value pairs. The keys are `name`, `type`, and `timestamp`. In other words, in addition to the name of the component, it holds the type of the component and the exact time in milliseconds at which it was tracked at that component. The following example provides a bit more information. This time, the inbound channel adapter invokes a Groovy script rather than a Spring Expression Language (SpEL) expression. Everything else in this configuration is the same. The `logging-channel-adapter` was just replaced by the `service-activator` that invokes a Groovy script.

```
<message-history/>

<inbound-channel-adapter id="poller" channel="transformChannel"
    expression="'foo'"
    <poller fixed-delay="10000"/>
</inbound-channel-adapter>

<channel id="transformChannel"/>

<transformer input-channel="transformChannel"
    expression="payload.toUpperCase()"
    output-channel="logChannel"/>

<channel id="logChannel"/>

<service-activator id="logger" input-channel="logChannel">
    <groovy:script>
```

```
headers.history.each { println it }
println '=' * 80
</groovy:script>
</service-activator>
```

After running this revised example for 30 seconds, you see the following output:

```
[name:poller, type:inbound-channel-adapter, timestamp:1304966962510]
[name:transformChannel, type:channel, timestamp:1304966962510]
[name:logChannel, type:channel, timestamp
:1304966962516]
[name:logger, type:service-activator, timestamp:1304966962516]
=====
[name:poller, type:inbound-channel-adapter, timestamp:1304966973309]
[name:transformChannel, type:channel, timestamp:1304966973309]
[name:logChannel, type:channel, timestamp:1304966973309]
[name:logger, type:service-activator, timestamp:1304966973309]
=====
[name:poller, type:inbound-channel-adapter, timestamp:1304966983312]
[name:transformChannel, type:channel, timestamp:1304966983313]
[name:logChannel, type:channel, timestamp:1304966983313]
[name:logger, type:service-activator, timestamp:1304966983313]
=====
```

As you can see, when logging the individual entries of the history header value, both the component type and timestamp information are available in addition to the component name. Let's take a quick look at these timestamps, which are recorded in milliseconds, to learn how the system is performing. First, we compare the timestamp of each subsequent invocation of the logger component. Only the last five digits are significant because the rest of the digits are exactly the same. Those significant values in order are 62516, 73309, and 83313. As you can see, these are roughly 10 seconds apart (depicted here as 10,000 milliseconds). That's to be expected because the poller has a fixed delay of 10 seconds. The differences between those values would likely be closer to 10 seconds if the `fixed-delay` attribute were changed to a `fixed-rate` attribute on the poller element.

Another thing you see in this example is that not much time is spent in the flow. The middle of the three flow executions shows no quantifiable elapsed time at the millisecond level of granularity, and the difference between start and end times in the last execution is only 1 millisecond. The first execution shows 6 milliseconds, but that's most likely a result of priming the system. After running the example several times, a similar difference is consistently recorded for the first execution, and the subsequent executions always show negligible overall times. If some CPU-intensive processing occurred within the flow, such as on methods being invoked from Service Activators, these history timestamps would be helpful for determining where in a flow the time is being spent. Most important, because the history data is recorded and stored with the Message, those timestamps are available regardless of how many thread context boundaries may have been crossed during the message flow.

14.2 Wire Tap

In chapter 3, we covered the `ChannelInterceptor` interface and the methods it provides for receiving callbacks before and after sending messages on a Message Channel as well as methods that are called before and after receiving if that Message Channel is pollable. The ability to add interceptors to a channel is one of the key advantages of a message-driven architecture. It goes hand in hand with the loose coupling and further promotes separation of concerns in a similar way to using aspect-oriented programming (AOP). As with AOP advice, the interceptors enable the addition of cross-cutting concerns, such as logging or security, to be added to any channel in a reusable way without requiring direct modification to that channel. Another important cross-cutting concern is monitoring, and because that's the topic of this chapter, we now revisit the role of a `ChannelInterceptor` that's specifically dedicated to that concern.

You can easily implement the `ChannelInterceptor` interface or extend its abstract convenience base class, `ChannelInterceptorAdapter`, to add any type of statistics gathering and logging you'd like. But, sometimes you might want to reuse functionality available from the integration framework for the monitoring. For example, you might only be interested in gathering information about messages whose payload contains certain data, and obviously a Message Filter could be useful in such a situation. Likewise, a Message Router might be a convenient way to send messages to different monitoring services dedicated to certain types of message content. For these reasons, Spring Integration provides an out-of-the-box interceptor that supports the `Wire Tap` pattern.

Imagine a banking application in which messages are published every time a debit is requested. Then consider the requirement to monitor all debits whose transaction amount exceeds a certain threshold, such as \$10,000 dollars. This could be implemented in the main message flow by inserting either a publish-subscribe channel or Recipient List Router that passes the messages along to a logger, which is preceded by a filter that checks the balance of the transaction represented by the message payload. The flow would look something like that shown in figure 14.1.

Even though it gets the job done, the flow pictured in the figure may not be the best way to achieve the goal. It's somewhat subjective and needs to be decided on a case-by-case basis, but the key factor in making this decision is whether the role of that

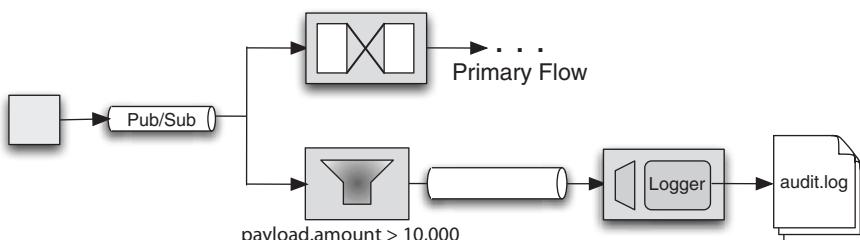


Figure 14.1 A publish-subscribe channel enables multiple downstream message flows but doesn't make any clear distinction between secondary and primary flows.

logging component is really considered part of the main flow. Perhaps this is a mandatory compliance-driven auditing function, and the logging component is a first-class component of the application. Or, it might be a cross-cutting concern, or what some may call a *nonfunctional requirement*. For example, if the main purpose of the logging is for monitoring the message flow for data that's covered by some service-level agreement (SLA), then even though it's important for the application, it's fulfilling more of an infrastructural than a business concern. In such a case, the Wire Tap pattern may be more appropriate. Figure 14.2 shows the subtle but important difference.

The choices are without limit when it comes to the downstream flow initiated by such a Wire Tap. That's obvious when you consider that it's just like any other message flow that happens to run perpendicular to the main flow in a logical sense. All of the components that Spring Integration provides, and all of the options for invoking methods on Spring-managed objects or evaluating expressions, are available to that tangential flow as well. Here is the configuration that corresponds to the Wire Tap diagram.

```
<channel id="debitChannel">
    <interceptors>
        <wire-tap channel="auditChannel"/>
    </interceptors>
</channel>

<service-activator input-channel="debitChannel" method="process">
    <beans:bean
        class="org.springframework...samples.wiretap.DebitService"/>
</service-activator>

<filter input-channel="auditChannel"
    expression="payload.amount > 10000"
    output-channel="logger"/>

<logging-channel-adapter id="logger" expression="'auditing debit: ' +
    payload"/>
```

Because both the filter and logging-channel-adapter use SpEL expressions, the only Java code is in the `DebitService` and the `Debit` domain object. For this example, the

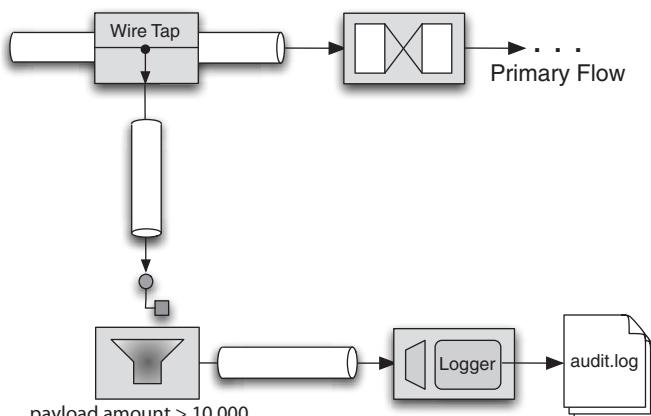


Figure 14.2 A Wire Tap connected to a channel enables a flow with cross-cutting behavior to be added while maintaining a clean separation from the primary message flow.

Debit domain object contains just two properties: amount and accountId. Along with a constructor to set those properties and some methods to retrieve them, it provides a `toString()` method so that the logging Channel Adapter's output provides useful information. Here is the Debit class definition:

```
public class Debit {

    private final BigDecimal amount;
    private final String accountId;

    public Debit(BigDecimal amount, String accountId) {
        this.amount = amount;
        this.accountId = accountId;
    }

    public BigDecimal getAmount() {
        return this.amount;
    }

    public String getAccountId() {
        return this.accountId;
    }

    public String toString() {
        return
            "[amount=" + this.amount + ", accountId=" + accountId + "]";
    }
}
```

Then there is the DebitService code, and for purposes of this example, we just print a message to the standard output:

```
public class DebitService {

    public void process(Debit debit) {
        System.out.println("processing debit [" +
            debit.getAmount() + "] from account: " +
            + debit.getAccountId());
    }
}
```

Finally, we need to send a few Messages through the `debitChannel` to demonstrate the Wire Tap functionality. The following code shows a main method that sends two different debit Messages. The first one is for an amount under the \$10,000 threshold we set for the required auditing. The second Message exceeds that amount.

```
public class Demo {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext("context.xml", Demo.class);
        MessageChannel debitChannel =
        context.getBean("debitChannel", MessageChannel.class);
        Message<Debit> message1 =
        MessageBuilder.withPayload(
            new Debit(new BigDecimal(5000), "SMALL")).build();
        Message<Debit> message2 =
```

```

        MessageBuilder.withPayload(
            new Debit(new BigDecimal(25000), "BIG")).build();
        debitChannel.send(message1);
        debitChannel.send(message2);
    }
}

```

When executing that `main()` method, the output would look like the following. As you can see, both debits were processed by the `DebitService`, but only the one whose amount exceeded the \$10,000 threshold was logged by the Channel Adapter. The other Message, with the \$5,000 debit amount, didn't make it past the filter's SpEL expression and was therefore prevented from continuing in the audit flow.

```

processing debit [5000] from account: SMALL
INFO : org.springframework...handler.LoggingHandler - auditing debit:
[amount=25000, accountId=BIG]
processing debit [25000] from account: BIG

```

The key point is that the Wire Tap can initiate a Message flow that uses the same Spring Integration endpoint types that are available to the primary flow. At the same time, the Wire Tap enables a clear separation for that secondary flow. It's easy to disable that flow by removing the Wire Tap interceptor from the Message Channel. Likewise, you can move that Wire Tap to another channel or apply it to more than one. You can even configure the Wire Tap as a global channel interceptor by defining it as a top-level element. If you do that, be sure to provide a channel name pattern to restrict the Wire Tap to the set of channels that should be intercepted. It might be a good idea to rely on naming conventions within your application to facilitate that. For example, to intercept credit transactions as well as debit transactions and potentially others, those channels could be named `accountDebitChannel`, `accountCreditChannel`, and so on. The following configuration demonstrates such a naming convention along with a top-level Wire Tap element applied globally based on pattern matching against channel names.

```

<wire-tap id="globalWireTap"
    channel="auditChannel" pattern="account*Channel"/>

<channel id="auditChannel">
    <dispatcher task-executor="someExecutor" />
</channel>

<channel id="accountDebitChannel"/>
<channel id="accountCreditChannel"/>
<channel id="accountTransferChannel"/>

```

One other thing you may have noticed in the configuration is that the `auditChannel` contains a `dispatcher` subelement that provides a reference to an `Executor` instance. That `Executor`-driven dispatcher on a Message Channel supports a nonblocking handoff to a thread within a pool. In other words, the flow initiated by this Wire Tap can be asynchronous. That means the primary flow isn't held up by this secondary flow. The only disadvantage of such a configuration is that the Wire Tap's flow won't

participate in any transaction that may have been in progress because the thread boundary is crossed and the transactional context doesn't propagate to the new thread. In many cases, that's the desired behavior, especially if a failure in the Wire Tap flow isn't important enough to worry about rolling back the primary flow.

You might sometimes need the Wire Tap flow to be transactional. Perhaps the auditing is required to meet compliance regulations, and any transaction that isn't audited is in violation of those regulations. In those cases, a nice middle ground solution that offers the best of both worlds is to use a channel with a queue subelement where the referenced queue is backed by a transactional Message Store implementation, such as the JDBC implementation that Spring Integration provides. That way, the Message is still handed off without blocking the primary flow, but it's persisted within a transaction so that failures in the Wire Tap flow don't result in lost data. Of course, if the Wire Tap flow is important, you might consider whether it's better to think of it as a parallel primary flow after all. In other words, it might be better to use a publish-subscribe Channel or Recipient List Router, as discussed earlier.

It's worth recognizing the flexibility that the choice of Message Channel types provides in these various Wire Tap scenarios. It may at first seem like every Wire Tap flow should be asynchronous by default, but when you encounter a scenario such as compliance-driven auditing, it's nice to know the choice of channel types exists. It might be common to have asynchronous Wire Tap flows, but doing so is as simple as adding the dispatcher subelement demonstrated earlier. Once again, the same capabilities of the framework are available to Wire Tap flows as are available to the primary flow, even down to the choice of Message Channel type.

Now that you know how to add a Wire Tap where you have the full power of the framework available to build some custom flow for monitoring purposes, we turn to another powerful tool in the monitoring toolbox. In the next section, we explore the ways you can take advantage of Java Management Extensions (JMX) in your message flows. You learn how to monitor certain attributes of the Message Channels as well as the Message Endpoints within the application context.

14.3 **JMX support in Spring Integration**

JMX is a standard way of exposing runtime information about a running Java application to allow monitoring and runtime management of the application. Attributes exposed via JMX can either be read-only, such as the number of messages on a queue, or read-write, such as the maximum number of database connections allowed in a connection pool. As well as exposing attributes, JMX allows operations to be exposed. Each operation relates directly to a Java method invocation with zero or more parameters. In addition to operations and attributes, JMX has a concept of notifications, which are essentially events emitted by a component. These are typically used to notify listeners of some sort of problem rather than a more general form of interprocess or intercomponent communication. One of the advantages of using JMX is the tooling support: many monitoring systems support JMX, allowing operations teams to use the tooling they may already use to monitor hardware to also monitor Java applications that

expose information via JMX. Where JMX is not directly supported, it's also possible to map some of the JMX concepts to other monitoring technologies; for example, JMX notifications can be mapped to SNMP (Simple Network Monitoring Protocol) traps.

JMX is supported in Spring Integration in two ways. First, we look at the out-of-the-box support for exposing information about message channels, message sources, and message handlers. We then look at Spring Integration's support for adapting JMX concepts to Spring Integration messages, and vice versa.

14.3.1 Monitoring channels and endpoints with JMX

The out-of-the-box JMX support means that if you add the following configuration, a wide range of information about the runtime behavior of your application is made available. In the case of channels, the implementation type of the channel determines the amount of data to be monitored: a default set of information is made available for all channels, additional information is made available for pollable channels, and yet more information is made available for channels that are both pollable and backed by a queue. The type of monitoring applied is determined by checking which of the interfaces `MessageChannel`, `PollableChannel`, and `QueueChannel` are implemented by beans during the initialization phase of the application.

```
<jmx:mbean-export default-domain="quote" />
```

The default message channel implementation exposes the attributes shown in table 14.1. Where an attribute relating to rate or ratio over time is exposed, it's calculated using an exponential moving average. This allows for the calculation of rates and ratios that give a higher weight to more recent data items yet precludes the need to constantly recalculate or to maintain a long list of data points. In applications requiring high performance or throughput, it is worth considering that there will be some effect from observation. The "Under the hood" section explores this overhead in more detail. Where the return type in the table is shown as `org.springframework.integration.monitor.Statistics`, a call to get the attribute will return an instance of this class containing count, mean, max, min, and the standard deviation of the attribute.

Table 14.1 Default set of metrics exposed for message channels

Name	Type	Description
Send count	int	Number of messages sent to this channel. Includes unsuccessful message sends that result in errors. Resending the same message a number of times will increment this value repeatedly even where it isn't successfully processed.
Send rate	Statistics	Provides statistics relating to the rate of messages being sent on this channel. Includes a mean number of messages per second using the previously mentioned exponential moving average. Retrying failed message sends results in multiple entries for the purposes of this statistic.

Table 14.1 Default set of metrics exposed for message channels (continued)

Name	Type	Description
Time since last send	double	Time in seconds since last successful or unsuccessful send.
Mean send rate	double	Calculated mean number of messages sent per second.
Send duration	Statistics	Statistics related to successful send durations in seconds.
Min send duration	double	Minimum recorded time for a successful send in milliseconds.
Max send duration	double	Maximum recorded time for a successful send in milliseconds.
Mean send duration	double	Mean milliseconds per successful send.
Standard deviation of send duration	double	Standard deviation of measured milliseconds per successful send.
Send error count	int	Number of sends that resulted in an error.
Mean error rate	double	Calculated mean per second of messages sent that resulted in an error.
Error rate	Statistics	Statistics relating to message sends resulting in an error.
Mean error ratio	double	Mean per second ratio of messages causing an error where 1 indicates no successful sends and 0 indicates no errors.

The attributes listed in the table relate only to sending because the `MessageChannel` interface doesn't define any methods related to receiving messages. Channel implementations that cater to asynchronous receive operations where the send doesn't result in a direct pass to the receiver additionally implement the `PollableChannel` interface. Where this interface is detected, additional details related to receive operations are exposed, as detailed in table 14.2.

Table 14.2 Additional metrics exposed for pollable channels

Name	Type	Description
Receive count	int	Calls to receive that returned a non-null result and did not result in an error
Receive error count	int	Calls to receive that resulted in an error

Where a channel is backed by a queue, it's often useful to know how many messages are currently queued and what the remaining capacity of the queue is. Classes that implement `QueueChannel` additionally expose the metrics shown in table 14.3.

Table 14.3 Additional metrics exposed for channels backed by a queue

Name	Type	Description
Queue size	int	Current number of messages queued and waiting to be received
Queue remaining capacity	int	Number of messages that can be queued before queue is full

Monitoring channels is useful for checking throughput and error rates, but other important statistics can be tracked by monitoring Spring Integration components such as routers, transformers, and adapters that act as message sources and handlers.

Components that act as handlers implement the `MessageHandler` interface in some form. It may not be obvious when writing a plain old Java object (POJO) router that it will effectively implement this interface, but there will always be either an adapter or a superclass that implements this interface somewhere at runtime. Table 14.4 details the `MessageHandler` metrics exposed for monitoring purposes.

Table 14.4 Metrics exposed for message handlers

Name	Type	Description
Handle count	int	The number of calls that have been made to the handle-Message (<code>Message<?> message</code>) method
Error count	int	The number of calls to the handle message method that resulted in an error
Mean duration	double	Mean duration of handle calls in milliseconds
Min duration	double	Min duration of handle calls in milliseconds
Max duration	double	Max duration of handle calls in milliseconds
Standard deviation of duration	double	Standard deviation for duration of handle calls
Active count	int	The number of calls to handle messages currently in process

In addition to channels and handlers, the built-in support also exposes message sources. These are typically pollable sources of messages, such as inbound JMS channel adapters whereby a scheduled task periodically calls `receive` on the message source and any non-null result is then published to a channel as a message payload. In the case of the message sources, a count of messages received in response to calls to the `receive` method is maintained and exposed as a JMX attribute.

In addition to exposing attributes, the `mbean-export` element in a Spring Integration configuration triggers the exposure of a number of operations. All of the monitored components expose a `reset` operation that allows all metrics, such as counts and rates, to be reset to zero. In addition, active components (components having some scheduled periodic behavior, such as polling a channel) generally implement the Spring `Lifecycle` interface, which defines `stop`, `start`, and `isRunning` methods. These methods allow active components to be stopped individually without the need to stop the whole application or application context. For convenience, `MessageHandlers` and `MessageSource` instances that implement the `Lifecycle` interface have these methods exposed as JMX operations.

14.3.2 Integration using JMX adapters

In this section, we look at the support offered by Spring Integration for JMX notifications, operations, and attributes in turn. The JMX support offered by Spring Integration covers both inbound and outbound adapters. Outbound adapters are primarily there to make it easy to manage and monitor Spring Integration applications. The inbound adapters allow the use of Spring Integration to carry out the management and monitoring of an application via JMX. It's also common to use Spring Integration both for the core application functionality and the management and monitoring of the core application. This self-monitoring approach is discussed in the next section.

One of the most common monitoring requirements for an application is that it should produce notifications when things go wrong so problems can be addressed early rather than waiting until a small unnoticed problem escalates to a show stopper. Using JMX, the best way to achieve this requirement is through notifications that allow an application to send data that may indicate a problem to subscribed listeners. The concept of a notification maps well to a message, so the support for JMX notifications takes the form of channel adapters that map between notifications and Spring Integration messages.

The inbound channel adapter is simple to set up and requires a channel name and a JMX object name. The following configuration assumes a Spring bean named mBeanServer is available.

```
<jmx:notification-listening-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=publisher" />
```

Where not all notifications will be of interest to an application, an instance of javax.management.NotificationFilter can be provided to the inbound channel adapter. The following example demonstrates use of an MBean server bean with a nonstandard name.

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

The outbound equivalent is also straightforward. The following example shows how to configure a notification-publishing channel adapter that publishes Spring Integration messages as JMX notifications.

```
<context:mbean-export/>

<jmx:notification-publishing-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=publisher" />
```

When it comes to JMX operations, Spring Integration provides support for operation invocations where no result is expected through a channel adapter implementation and support for operation invocations where a response is of interest through a

gateway implementation. In both cases, the invocation of the JMX operation is triggered by the receipt of a Spring Integration Message that's used to determine the parameters to pass the operation if any are required. Both the gateway and channel implementations use the same strategy for mapping the inbound trigger message to operation invocation parameters. Where the payload of the message is a Map, the payload is assumed to be a set of key-value pairs. Where a single parameter is expected, the payload is assumed to be the parameter. Where the JMX invocation doesn't expect a parameter, the payload is ignored in all cases.

Given that the adapter carries out the parameter mapping, all that's required is the JMX object name, the operation name, the request channel name, and the name of the MBean server if it's not the default of `mbeanServer`.

```
<jmx:operation-invoking-channel-adapter id="adapter" channel="requests"
    object-name="example.domain:name=TestBean"
    operation-name="ping" mbean-server="myMbeanServerRef" />
```

The operation invoking gateway looks almost exactly the same except it allows for the configuration of a reply channel.

```
<jmx:operation-invoking-outbound-gateway request-channel="requests"
    reply-channel="replyChannel"
    object-name="org...jmx.config:type=TestBean,name=testBeanGateway"
    operation-name="methodWithReturn" mbean-server="myMbeanServerRef" />
```

The final JMX adapter allows for periodically polling an attribute exposed over JMX. The attribute value then becomes the payload of the resulting message.

```
<jmx:attribute-polling-channel-adapter id="adapter"
    channel="channel"
    object-name="example.domain:name=someService"
    attribute-name="requestCount">
    <si:poller max-messages-per-poll="1" fixed-rate="500"/>
</jmx:attribute-polling-channel-adapter>
```

14.4 Control Bus

As described in *Enterprise Integration Patterns*, the idea of a Control Bus is to use the same messaging components that are used for the application to manage the messaging system itself. In other words, you should be able to invoke management operations on endpoints and other manageable resources in the system by sending Control Messages to a Message Channel. That channel essentially plays the role of an operation channel.

Spring Integration supports this notion of Control Bus by allowing you to send messages whose payload represents a command to be executed. The target of that execution can be any Spring-managed object that happens to be manageable. Two alternative syntaxes are supported for the command: SpEL and Groovy. We look at both in this section. First, let's explore Spring's support for annotations that label managed resources. This is a good starting point because Spring Integration relies on that same mechanism to determine valid targets of command execution.

14.4.1 Spring's support for management annotations

The Spring Framework defines a handful of annotations in its JMX support to facilitate building applications whose components may be monitored and managed. This is yet another example where having a container that's aware of the objects defined within its context, such as Spring being aware of its beans, allows other behavior to be added to those objects. It's the same general idea that we've described elsewhere on the book of using AOP to noninvasively add cross-cutting concerns. In this case, the cross-cutting concerns are monitoring and management.

The class-level annotation that Spring defines is `@ManagedResource`, and its role is to indicate a candidate for monitoring and management. When it comes to monitoring, the corresponding concept in JMX is a managed attribute—basically a property of an object, with a getter and/or setter, that can be read or written via JMX. Here's a simple example of a class annotated with `@ManagedResource` that indicates a single property that can itself be managed. The annotation for that property can be applied at method level to the getter and/or setter, and its name is `@ManagedAttribute`.

```
@ManagedResource
public class NumberHolder {

    private final AtomicInteger number = new AtomicInteger();

    @ManagedAttribute
    public int getNumber() {
        return this.number.get();
    }

    @ManagedAttribute
    public void setNumber(int value) {
        this.number.set(value);
    }
}
```

You can see that the numeric value should be managed in terms of both read and write operations. If you'd like to expose other operations that don't map to simple JavaBean-style getters and setters, then you can add `@ManagedOperation` annotations to such methods. The example can easily be extended to demonstrate that. You just add an increment operation.

```
@ManagedResource
public class NumberHolder {

    private final AtomicInteger number = new AtomicInteger();

    @ManagedAttribute
    public int getNumber() {
        return this.number.get();
    }

    @ManagedAttribute
    public void setNumber(int value) {
        this.number.set(value);
    }
}
```

```

@ManagedOperation
public void increment() {
    this.number.incrementAndGet();
}

}

```

Now, an instance of `NumberHolder` configured within a Spring application can easily be exposed for JMX monitoring and management. To export those attributes and operations from the Spring context requires only a single configuration element from Spring's context namespace:

```
<context:mbean-export/>
```

Spring Integration takes advantage of these same annotations. They are present on the `MessageChannelMetrics` interface and the various classes that implement that interface, such as `QueueChannelMetrics` and the others that you learned about in the previous section about JMX monitoring. As you will learn in the next section, those same annotations are used in Spring Integration to determine what operations may be invoked on components through the Control Bus. The Control Bus's awareness of invocable operations is not limited to the annotated classes in Spring Integration's own API. You can use those annotations anywhere in your code and then have the annotated methods available as operations on the Control Bus. The next two sections present a few examples of invoking such operations. We cover two different options for the syntax of the command messages sent to the Control Bus channel: SpEL and Groovy.

14.4.2 Using SpEL for control messages

The Control Bus is a Spring Integration component that accepts messages on its input channel much like a Service Activator, Transformer, or other type of Message Endpoint. The key difference with the Control Bus is that the payload of a message it receives is expected to indicate an invocable operation. The Spring Integration core namespace provides an implementation that evaluates SpEL expressions for those operations. To enable such a Control Bus in your application, add the following element to its configuration.

```
<control-bus input-channel="controlChannel"/>
```

The Message payloads sent to the Control Bus should contain SpEL expressions referencing a bean that's the intended target of the control operation. The `@` symbol is used for that bean reference. The following example shows how to send a control message that increments the value of the `NumberHolder` instance discussed in the previous section.

```
MessageBuilder.withPayload("@numberHolder.increment()").build();
controlChannel.send(message);
```

That example would work because the `NumberHolder` class contains the `@ManagedResource` annotation, and the `increment` method contains the `@ManagedOperation`

annotation. Knowing that, you can provide any number of beans within your application context that will be valid targets for Control Bus operation messages.

A major use of the Control Bus is to manage the Spring Integration components. Probably the most common use case is to start and stop endpoints. Polling Consumer endpoints can start and stop polling on demand. Even Event-Driven Consumer endpoints can be controlled this way because the start operation will correspond to subscribing, listening, or simply activating such an endpoint. To accommodate these and similar use cases for a large number of other Spring components, the valid operations also include those methods defined on the `Lifecycle` interface, even if the `@ManagedOperation` annotation isn't present. Here's an example of stopping an endpoint by sending a message to the Control Bus:

```
MessageBuilder.withPayload("@filePollingAdapter.stop()").build();
controlChannel.send(message);
```

Obviously, those lifecycle operations can be exposed on your own Spring-managed objects as well if their classes implement Spring's `Lifecycle` interface. Finally, you can also invoke certain operations on the configurable thread pools that Spring defines. For example, imagine that you have the following configuration, utilizing Spring's task namespace, to create a thread pool `Executor` instance within your application context.

```
<task:executor id="myExecutor" pool-size="5" />
```

This element generates a bean whose type is `ThreadPoolTaskExecutor` with a pool of 5 threads. Certain properties of that thread pool are configurable, such as the core and max pool size values. The following example shows a Control Bus operation that sets the max size of that thread pool:

```
MessageBuilder.withPayload("myExecutor.setMaxPoolSize(25)").build();
controlChannel.send(message);
```

Now that you've seen a number of possibilities enabled by the Control Bus, let's take a quick look at an alternative implementation, one that accepts Groovy syntax rather than SpEL in the Control Message payloads.

14.4.3 Using Groovy for control messages

The Groovy Control Bus is basically the same as the SpEL implementation included in the core module. As long as you have the `spring-integration-groovy` JAR on your classpath, you can use the `groovy` namespace and its Control Bus alternative as follows.

```
<groovy:control-bus input-channel="controlChannel" />
```

This implementation compiles each control message's payload into a Groovy script. It then makes any managed bean in the Spring context available as a variable to that script. The beans that are eligible to be managed in this way are chosen according to the same criteria described in the previous section. Any instance of a class with `@ManagedResource`, any `Lifecycle` implementation, and Spring's thread pool classes

are all eligible to be accessed as variables in the Groovy script payload of a message sent to the Groovy Control Bus. The variable name would match the bean name.

14.5 Under the hood

We mentioned earlier that monitoring channels and endpoints with JMX may cause some small performance overhead. As with all potential performance issues, don't panic: wait until you know for sure that you have a problem because premature optimization can introduce as many problems as it solves, or more. If you are interested in what happens under the hood, this section provides a detailed view of the implementation of statistics gathering for channels and endpoints.

To determine the overhead, it's worth running a small benchmark on your system and keeping in mind that the overhead relative to the request processing is probably more important than the absolute figure for the overhead. For example, where you're hitting a database, an overhead of a few hundredths of a millisecond will be insignificant. To give an idea of the sort of overhead we're talking about, running a simple test that sends 10,000 messages with a String payload to a queue channel showed an increase of about 0.0012 milliseconds per message when JMX statistics were enabled on the channel. This was run on an early 2011 quad-core MacBook Pro. In almost all cases, such overhead is insignificant in the context of the processing that's occurring. In our test, there was no processing taking place, so the 0.0012 milliseconds represented an increase of about 40% per message sent.

It's helpful to understand what the Spring Integration framework is doing under the hood when you include the `mbean-export` tag from the Spring Integration JMX namespace. The first thing that needs to happen is that the framework must start monitoring when certain methods are called on beans implementing the key Spring Integration interfaces. This is achieved by using the standard Spring strategy of a `BeanPostProcessor`. The `BeanPostProcessor` receives a callback for each bean at two points in the bean's lifecycle: before any initialization has occurred and after initialization. The signature for the after-initialization method is shown here for reference. The interesting thing about the message signature is that it has a nonvoid return to allow something other than the bean passed as an argument to be returned, effectively changing the instance that's available in the application context under this bean name. The most common use of this approach is to return a wrapper of some sort. That is how Spring enables AOP, which allows you to apply cross-cutting concerns such as security and transaction management by wrapping the bean and intercepting calls to its methods.

```
Object postProcessAfterInitialization(Object bean, String beanName)
    throws BeansException
```

Spring generally supports two implementations of AOP: one using AspectJ and the other using the AOP Alliance project. Many people prefer AspectJ because they regard it as more powerful, but it comes with some additional setup complexity. Its use requires some form of weaving, either postcompile or at class load time. Although

that complexity has decreased in recent years, in part with help from Spring, most of the Spring-based AOP still relies on the AOP Alliance library, which uses built-in JDK dynamic proxies to create a wrapper instance of a specified interface at runtime.

When you add `mbean-export`, the framework adds a bean that is an instance of the `IntegrationMBeanExporter` class to the application context. This bean implements `BeanPostProcessor`, which means it gets a callback for each bean defined in the application context. This callback tests to see if the bean passed in implements `MessageChannel`, `MessageSource`, and `MessageHandler`. If the bean implements one of these interfaces, the post-processor uses an instance of `NameMatchMethodPointcutAdvisor` to build advice, which intercepts any of the methods of interest, such as `send` or `receive`. This advice incorporates one of the custom metrics classes, such as `PollableChannelMetrics`, which is responsible for updating the metrics on the intercepted component in response to one of the method calls of interest.

This is where the overhead comes from: each call to send on a message channel passes through an interceptor, which decides if the method call is of interest and then updates the held metrics if appropriate.

14.6 **Summary**

In this chapter, we looked into the monitoring and management aspects of a messaging application and how they can be implemented using Spring Integration. We distinguished between monitoring the messages and monitoring the endpoints. We used `MessageHistory` support for the former and `Wire Tap` and `JMX` support for the latter.

We introduced the Control Bus to manage the application. This bus can control the application through `JMX` operations. It can be invoked through control messages using either `SpEL` or `Groovy` to specify the operation and its target bean.

Now that you know how to monitor an application, it's time to go deeper into concurrency and performance characteristics presented in chapter 15.

15

Managing scheduling and concurrency

This chapter covers

- Scheduling tasks and controlling polling frequency
- Processing messages asynchronously
- Managing concurrent tasks
- Scheduling and concurrency APIs under the hood

In the previous chapters, you learned about the core and specialized components and how to put them together to build a successful application. But there's more to building an application than finding all the parts you need and putting them together into a structure. This chapter takes a turn to discuss application building from a different perspective: dynamic configuration.

Dynamic configuration requires shifting focus from the logic of the application to the expected behavior at runtime. Configuring runtime behavior involves controlling the schedule of various timed events in the application, such as the frequency with which external message sources are polled, and also configuring various parts of the application to run concurrently, making optimal use of system resources, and increasing the rate at which messages are being processed. You learn to manage scheduling and concurrency in this chapter.

A final word before we start: sometimes, fine-tuning the dynamic performance of an application is more art than science because no two applications are exactly the same. No prescription can tell you precisely what to do and when, and often you must experiment to find what works best in your case. But engineering is an art too, so we provide you with the options and guidelines on best practices, leaving it to you to determine what works best for your application.

15.1 Controlling timed events

As the previous chapters made abundantly clear, communication is the predominant point of interest of enterprise integration applications. To integrate properly, the components must communicate effectively. Factors such as the format of data that's being exchanged, the processing services, ordering, and most important, timing play critical roles in achieving this goal.

Integration applications often must satisfy timeliness requirements, which can be of various types. You might have responsiveness requirements, defining how quickly the system should respond when a new message becomes available, or scheduling requirements, prescribing how often or at what time the application should interact with its counterparts. This section focuses on the framework features that control the timed behavior of the application.

Poller configuration is the most common time-related concern. Pollers are active components that pull new data from external message sources or trigger the handling of the existing messages stored in system's queues. Such an asynchronous mode of operation, in which the responsibility of acknowledging the existence of new data falls on the service provider, increases the overall robustness of the system, but it also introduces a lag that's ultimately related to how aspects such as frequency, timeout, or number of messages processed per polling cycle are configured. Timed events are not just about polling, though, so we show you how to set up a scheduled producer for invoking existing beans and sending out messages at specified times.

15.1.1 Pollers and their configuration

You encountered polling in previous chapters. An analysis from chapter 5 explained that the main distinction between event-driven and polling components is in how they receive messages that they have to process. Event-driven components receive messages directly from their invokers (see figure 15.1), and polling components must reach out and consult a message store to find out whether any new data is available for them. In Spring Integration, polling is used internally by endpoints for receiving messages transmitted asynchronously through queue channels (see figure 15.2) and also by channel adapters for reading data from a message source, which is typically used for retrieving external data.

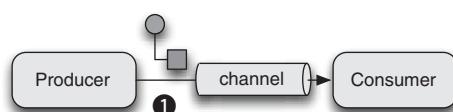


Figure 15.1 Event-driven components receive messages directly from their invokers.

Whether an application needs to use polling at all is often a matter of choice; for example, you might deliberately introduce a queue channel for handing off messages asynchronously between components. It can also be a matter of necessity: some external systems can integrate in an event-driven fashion and push messages into an application, but the application often has to reach out and grab the data. For example, receiving up-to-date weather information about airports may require accessing an external web service that is exposed publicly and can be invoked by the interesting parties, as opposed to receiving notifications from a subscriber service. In the former case, polling the web service is the only option available.

Poller configuration shouldn't be new to you either. If you followed the chapters in order, you saw `<poller/>` elements whenever we configured a channel adapter or a polling endpoint. Because polling is a sensitive topic, and its configuration has a strong impact on performance, no default is provided by the framework. Any polling component in your application must have a polling configuration, describing the polling frequency, timeouts, whether the component can process more than one message per polling cycle, or whether the polling task should be deferred to a task executor.

Polling component definitions don't have to include an explicit poller configuration: they can also rely on the existence of an applicationwide default configuration, which has to be provided by the developer. This makes it possible to simplify component definitions in general (because the default is usually a configuration that applies to the majority of the polling components). At the same time, this scheme allows you to customize the configuration wherever the default doesn't fit the needs.

In both cases, a `<poller>` element is used to define a set of polling configuration parameters. The following listing shows you how to configure two different polling endpoints using the default and individual configuration.

Listing 15.1 Configuring polling endpoints

```

<poller id="defaultPoller"
    fixed-delay="500" default="true" />

<channel id="input1">
    <queue capacity="10" />
</channel>

<channel id="input2">
    <queue capacity="10" />
</channel>

```

1 Default poller configuration

```

<service-activator id="activator1"      ← ② Endpoint with default poller
  input-channel="input1"/>

<file:inbound-channel-adapter id="inputDirPoller"
  directory="${java.io.tmpdir}"
  filter="filter"
  comparator="testComparator"
  auto-startup="false">
  <poller fixed-delay="10000"/>
</file:inbound-channel-adapter>

```

The definition of the default poller configuration is shown at ①. The service activator, defined at ②, uses this configuration. The channel adapter, defined at ③, defines a different policy because it needs to consult the message source much less frequently. In general, it's good practice to create a default configuration to be used for queue channels and polling endpoints: it simplifies the overall configuration, and channel adapters are likely to need a customized configuration anyway, so they rarely rely on the default.

Note in the listing that the numeric values are configured in the `fixed-delay` and `fixed-rate` attributes of the poller elements. They're used for configuring the polling frequency, and we look at them next.

15.1.2 Controlling the polling frequency

The frequency at which pollers operate has a strong impact on the speed with which messages travel across the system and on the responsiveness of the system. The more frequently the application polls a message source or a queue channel, the more likely the polling event is to occur close to the moment when the message becomes available, and so the application will be able to respond faster by processing it. Over time, this leads to a shorter lag on average. But not all types of messages are equally important, and not all messages require immediate attention, so every application should set up a polling scheme that's consistent with the timeliness requirements of the application, which often define what is an acceptable delay. As a consequence, the polling frequency should be configured to balance performance with resource usage.

Because an application must be highly responsive, you might consider setting the frequency to the highest possible values. But every polling operation consumes computing resources that may be needed elsewhere. This is appropriate in terms of resource usage if the effort of checking that a new message is ready is minimal and if the messages are likely to arrive at a relatively high rate. In general, if polling operations are expensive or blocking for a relatively long time (such as occurs with remote accesses), it's a good idea to perform them less frequently; otherwise, the system will be busy all the time performing checks that may not produce any result.

Let's look at the poller definitions from listing 15.1. The default polling interval is set to a fixed delay of 0.5 seconds; all numeric values are in milliseconds. This means that after each polling operation, the system waits half a second before trying again. By contrast, the channel adapter uses a fixed-rate frequency of 10 seconds, which means the system executes the polling operation every 10 seconds, regardless of whether the previous polling operation is finished.

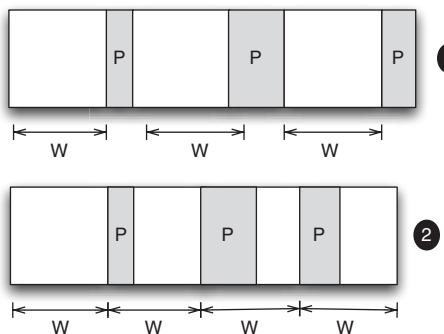


Figure 15.3 Fixed-interval polling (1) versus fixed-rate polling (2). A fixed-rate scenario guarantees that polling will happen at the same preset interval, but the gap between polls may vary. A fixed-interval scenario guarantees that the interval between polls is constant, but the timing of the polling event may drift over time.

Figure 15.3 captures the differences between the two strategies. Note how fixed-interval polling causes the timing of the polling event to drift over time, but fixed-rate polling takes place at predictable moments.

The two approaches are similar, and the differences are barely noticeable when the polling operation is extremely short. A fixed-rate setup guarantees that polling will happen at definite moments, which in turn makes the application behave in a more predictable way. A fixed-interval strategy allows a period of blackout between two successive polling operations. Table 15.2 shows you a few points to consider when deciding between the two.

Table 15.1 How to decide between fixed-rate and fixed-interval strategies

Scheduling type	What to consider
Fixed-rate	<ul style="list-style-type: none"> – Ensures that messages are read at well-defined moments – Works best when the duration of the polling operation is fixed and short relative to the polling period; otherwise it may force the system into a busy loop
Fixed-interval	<ul style="list-style-type: none"> – Guarantees a minimum silent interval between successive reads – Works best when the duration of the polling interval is variable or when the resource needs exclusive access (the silent interval guarantees that other components can also access it)

15.1.3 Scheduling jobs at precise times

Fixed-rate scheduling produces a predictable sequence of polling events, but they aren't entirely deterministic. Let's consider an email client that synchronizes with the server every 10 minutes. In this case, it doesn't matter when the 10-minute period starts as long as the interval is respected. But there are other operations for which the start time is important. For example, batch operations such as payment transactions processing and bulk reference-data changes are usually processed outside the normal application operation hours. (We cover batch operations in chapter 16, and we also encourage you to read Spring Batch in Action by Arnaud Cogoluegnes, Thierry Templier, Gary Gregory, and Olivier Bazoud [Manning, 2011]). Batch operations are resource intensive too, and running them while other users are accessing an application will slow down operation.

CRON expressions

CRON expressions are schedules tied to the system clock and calendar (say, “every day at 5:00 p.m.”) that were originally used in UNIX systems. The CRON expression variant used by Spring is a string comprised of six fields separated by space, representing (in order) second, minute, hour, day, month, weekday. Wildcards and special expressions can be used for describing wider range of values. For details, please consult the Spring reference documentation.

For batch operations, instructions such as *execute this task every 24 hours* aren’t precise enough. You must specify, for example, *execute this job every day at midnight*. Neither fixed-rate scheduling nor fixed-interval scheduling have the capacity to describe this type of configuration. To provide this type of configuration, you must use a different type of trigger, CRON-expression-based triggers, as in the following example:

```
<poller id="defaultPoller" default="true" cron="0 0 * * *" />
```

The frequency and polling strategies—fixed rate, fixed interval, and predefined times—are the most frequently used settings for customizing the polling behavior. Polling performance is also affected by other parameters, such as timeouts and the number of messages processed per cycle.

15.1.4 Advanced configuration options

Optimizing the polling strategy doesn’t stop at finding the best frequency of the polling cycle. Consider a case where you need to poll a message source for which a read operation is time-consuming, regardless of whether data is available, which happens often when doing invocations across the network. The polling operation will be blocked until the read operation completes, but it can’t wait there indefinitely: it prevents the application from performing other useful jobs. You must place a limit on the time a poller can spend waiting for a message to arrive. You can configure the value of the timeout as follows:

```
<poller id="defaultPoller" fixed-rate="10000" default="true"
       max-messages-per-poll="1" receive-timeout="2000" />
```

With this setting, the timeout is set to 2 seconds. If a message arrives after 2 seconds, it’s picked up by the next polling operation. Setting a reasonable timeout for polling operations is important for the performance of your application. Setting a large timeout value without good reason won’t help performance. Not getting a response beyond the acceptable waiting time can mean a lot of different things but should be interpreted as a sign that either a message is not available or the resource from which data is read is unavailable, so there’s little chance of reading any data from it. But the timeout value shouldn’t be set too low either, because long-running read operations must be allowed to complete. Generally speaking, timeouts should be set as low as possible but not lower than the longest duration of a normal read operation plus some contingency added in. For example, a timeout in the range of a few tens of seconds for a remote web service

call could be reasonable, whereas reading from a queue channel would be capped at a few seconds at most. This is an example and your settings should be based on how your application works.

Let's consider the case of a poller that runs every second. Let's also assume that not one but two messages become available on the message source. With what we configured so far, here's what will happen: when the first polling cycle begins, the poller reads the first message and starts processing by delivering it to an endpoint. The second message will be left unprocessed until the next polling cycle.

Timeouts

Timeouts should be set as low as possible but not lower than the longest duration of a normal read operation plus some contingency added in. They are a way of saying, "I've been waiting for too long. It's not going to happen."

You can immediately see how this configuration can become a limitation: the rate at which messages can be consumed is limited by the frequency of the polling cycle. Besides polling the message source more frequently, another option can help in this case: receiving more than one message per polling cycle. You can do so by setting the `max-messages-per-poll` attribute of the poller as follows:

```
<poller id="defaultPoller" fixed-rate="5000"
    default="true" max-messages-per-poll="2" />
```

With this setting in place, both messages will be received in the same polling cycle without the second message having to wait. You can choose any maximum value for the number of messages that you can receive in a cycle, but setting the value too high isn't a good idea either because it may increase the time that the poller spends reading messages beyond an acceptable threshold. Figure 15.4 illustrates the differences between the two approaches.

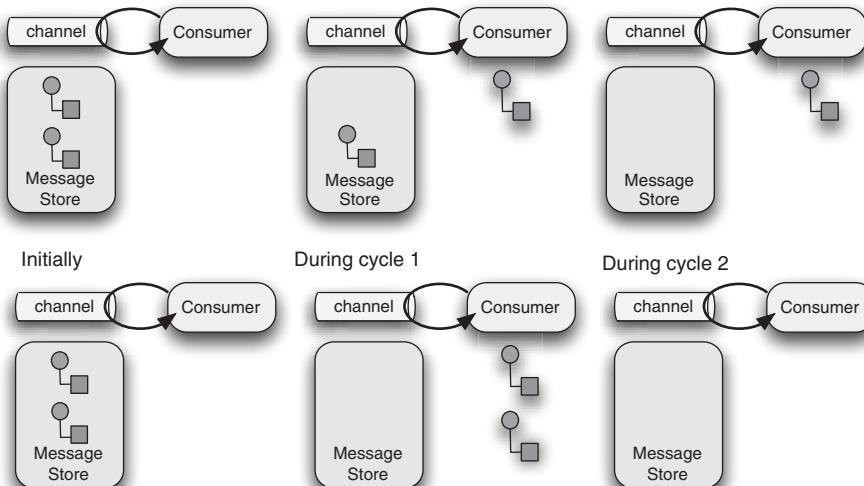


Figure 15.4 The effect of using `max-messages-per-poll`: The messages are processed one by one, and the second message has to wait for a complete polling cycle in order to get processed (top). The messages are processed as part of the same polling cycle (bottom).

This concludes our overview of poller configuration as a way of controlling the time-related aspects of the application. But there's another mechanism you can use for triggering timed events: the scheduled publisher.

15.1.5 Publishing messages according to a schedule

Besides polling, another type of timed event can take place in your application. Spring Integration allows you to configure a special type of component that publishes messages regularly using the same periodic schedule configuration attributes.

This component can be used for triggering periodic tasks and works as an alternative to polling. Suppose you want to execute a periodic cleanup task on your application's database. The following listing shows a possible configuration.

Listing 15.2 Regularly cleaning the database

```
<inbound-channel-adapter
    expression="${cleanup.expiration.threshold}"
    channel="clean-input">
    <poller fixed-rate="600000"/>
</inbound-channel-adapter>

<service-activator id="cleanup" input-channel="clean-input"
    ref="cleanupService" method="clean"/>
```

Every 10 minutes, a message is sent by the publisher, triggering the invocation of the cleanup service (and in turn cleaning up the database). The power of this setup is that the service activator can be triggered outside the polling schedule by sending a message on the clean-input channel. The payload of this message may contain configuration data, so the parameters for each execution of the target service can be parametrized.

This component can also be used to turn a service activator into a channel adapter. Consider the case when the application needs to send out reminder or notification emails. Such examples may be necessary for notifying customers that they have a scheduled flight on the same day. This is usually a multistep process, which begins with searching the database for such reservations, generating notification messages, and sending them as emails.

The message publisher simplifies the implementation of such a solution because the payload of the published messages can be set up in a SpEL expression, which in turn can invoke a search Data Access Object (DAO). The complete solution is shown in the following example.

```
<inbound-channel-adapter
    expression="${reservations.findUpcoming()}"
    channel="notifiable-reservation-list">
    <poller fixed-rate="600000"/>
</inbound-channel-adapter>

<splitter input-channel="notifiable-reservations"
    output-channel="transformer-input"/>
```

```
<transformer ref="notificationTransformer"
            method="reservationsToMessages"
            input-channel="transformer-input"
            output-channel="emails"/>

<mail:outbound-channel-adapter channel="outboundMail"
                                mail-sender="mailSender" />
```

In this case, the process is started by the publisher component that generates a list of reservations which need notification, a transformer that creates messages out of the reservations, and an email outbound channel adapter that sends the messages created by the transformer. A polling solution would've required the implementation of a `MessageSource`, which would've delegated to the reservations services, making the process more complicated.

The publisher component wraps up the polling section. You now know how to incorporate time-related concerns in your application's design. Polling and timeliness are also closely related to concurrency management, because poller tasks and asynchronous operations in general are concurrent operations. Let's focus on concurrency next.

15.2 Managing concurrency

Enterprise applications can increase their performance and responsiveness by executing concurrent operations. Concurrency may be a requirement (for example, the application may be required to be able to handle multiple incoming requests at once, or it could be a way of taking advantage of the capabilities of the hardware) especially since multicore and multiprocessor systems have become mainstream.

In either case, the problem of creating an application that executes tasks in parallel can be split into two parts, with orthogonal concerns. You first must find an application design that allows operations to execute in parallel, mainly by splitting multistep operations into finer-grained parts that get executed asynchronously. One important counterpoint is that single-threaded operations can propagate context information, such as the transactional and security context, across endpoints, and this decision also involves a trade-off. The first subsection walks you through the mechanics of breaking down single-threaded processes into multiple asynchronous tasks.

Once you have the right application design, you must allocate resources (threads) to execute the parallel operations. Again, this involves a number of trade-offs between maximizing the number of concurrent operations and making sure the system's resources are used efficiently. In the second subsection, we describe the thread management infrastructure of Spring Integration and how it allows changing the configuration parameters (and implicitly the concurrency degree of the application) in a transparent fashion and without affecting the application design.

15.2.1 Breaking down the thread

An enterprise integration application is an assembly of message-processing endpoints connected by channels. A message sent to a channel undergoes a series of transformations while it travels through this structure. From a purely logical standpoint, you can

start by ignoring whether this series of transformations happens within the scope of a single thread or spans multiple thread contexts: the processing that takes place within each individual endpoint and the end result will be the same. In fact, Spring Integration applications may work just fine by adopting a single-threaded approach, which is also the default. Without any additional configuration options, processing a message from end to end takes place within the scope of a single thread.

Single-threaded processing may work well when the goal of using Spring Integration is to encapsulate a multistep process that includes invocations across multiple services and systems. In these situations, you don't need to worry about concurrency. Multistaged processing pipelines are good candidates for parallelization if it's acceptable for the various stages to execute in separate thread contexts.

To illustrate the concept, let's begin with a simple example: a car assembly operation. Building a car goes through two stages: assembling and painting. Each stage takes 1 second, so producing a complete car requires 2 seconds. The simplest possible setup can be seen in the following listing.

```
<channel id="input"/>
<channel id="paint"/>
<channel id="output"/>

<gateway id="supplyInput"
    service-interface="com.manning.siaia.pipeline.SupplyInput"
    default-request-channel="input"/>

<service-activator id="assemblyLine" input-channel="input"
    output-channel="paint">
    <beans:bean
        class="com.manning.siaia.pipeline.AssemblyLine"/>
</service-activator>

<service-activator id="paintShop" input-channel="paint"
    output-channel="output">
    <beans:bean class="com.manning.siaia.pipeline.PaintShop"/>
</service-activator>

<service-activator id="exitCounter" input-channel="output"
    ref="counter"/>

<beans:bean id="counter"
    class="com.manning.siaia.pipeline.Counter"/>
```

A PieceKit payload is delivered to the AssemblyLine where it's assembled into a car. The car is painted in the PaintShop and sent to the final counter, which tells how many cars were built in a given time frame. An overview of the whole process is shown in figure 15.5.

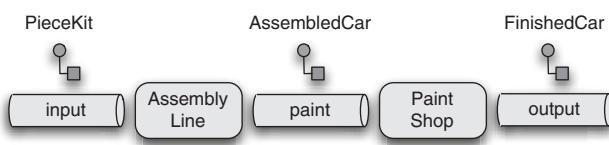


Figure 15.5 The car-assembly operation consists of assembling and painting. The two stages can be executed in the same thread or asynchronously.

In the default case, because of using direct channels, the whole process of building a car is single-threaded. Trying to build 10 cars in a loop, you discover that the process will take 20 seconds.

You can save time if you allow for creating cars in parallel by keeping the current configuration and sending messages in separate threads. This type of scenario would happen, for example, in web applications, where different HTTP requests are processed in separate threads. What we want to focus on is how to use the framework for handling parallelization internally. Instead of executing both stages in the same thread, let's execute them in parallel: after being assembled, the car is handed over asynchronously to the PaintShop, and the invoking thread returns, making room for another PieceKit to be sent to AssemblyLine. The AssembledCar is painted in a separate thread. The differences between the two scenarios is shown in Figure 15.6. Single-threaded processing produces a car every 2 seconds, but breaking the thread boundary and doing part of the processing in a separate thread produces a car every 1 second.

With Spring Integration, switching between scenarios is a matter of choosing different types of channels without modifying the logical structure of the application. You can hand over messages asynchronously by using queue channels or by using an executor channel.

When you change the input to a queue channel rather than a direct channel, the processing of the message by the endpoint doesn't execute in a message-sending thread, which loses control over the message once it's sent to the queue. Instead, the message is processed in the scope of a new task initiated by the poller of the endpoint that's connected to the channel. The message-sending thread returns to the loop and is now free to send another new message to the channel, and so on, blocking only when the queue of the channel is full. Meanwhile, the poller repeatedly removes elements from the queue and processes them.

```
<i:channel id="paint">
    <i:queue capacity="10"/>
</i:channel>
```

You can get a similar result by using an executor channel. This is a subscribable channel connected to a task executor, which provides a thread in the context of which the application will process the new message. The task executor is the main backing abstraction for thread allocation in Spring. Instead of creating a new thread directly for executing a concurrent task, the application components pass on the task that needs to be executed asynchronously to a task executor, which encapsulates the thread allocation strategy. The same abstraction is used for polling.

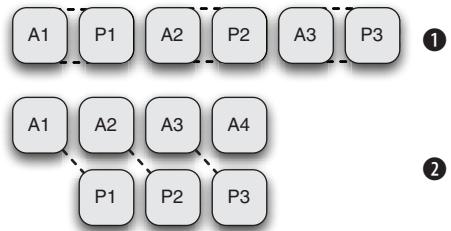


Figure 15.6 The difference between the single-threaded and concurrent approach. Scenario 1 is single-threaded. Scenario 2 is concurrent: the assembly and paint take place in parallel, which increases the processing speed.

```
<i:channel id="paint">
    <i:dispatcher task-executor="executor"/>
</i:channel>
```

From a purely concurrency-focused perspective, the queue channel and the executor channel work in a similar way by interrupting the thread context of the message-sending thread and executing the processing of the message asynchronously. There are differences between them that make each of them more suitable for different scenarios, as you can see in table 15.4.

Table 15.2 Queue channel versus executor channel: What you need to consider

Channel type	Features
Buffered channel	<ul style="list-style-type: none"> – Wide choice of queuing and storage strategies – Allows advanced features such as prioritization or persistence – Need for polling introduces a lag, influenced by the polling strategy – Support for distributed message processing if the queues are accessible from different virtual machines
Executor channel	<ul style="list-style-type: none"> – No polling lag: messages are processed as soon as thread is available – Less robust in face of an increased load – Unprocessed messages may be lost – Lacks support for distributed message processing: messages must be executed from within the same virtual machine

In general, asynchronous handoff is a solution for processing multiple messages concurrently, which is a common occurrence in scatter-gather and publish-subscribe scenarios. As in the previous example, you can set the output of the splitter to be either a queue channel or an executor channel, and every resulting message will be processed independently.

If you’re using a publish-subscribe channel, you can define a task executor at the channel level too, allowing each subscriber to process the message in a separate thread.

```
<publish-subscribe-channel id="notifications"
    ↗ task-executor="taskExecutor" />
```

After clarifying the design decisions that allow for the concurrent execution of tasks, we discuss how to set up the backing infrastructure for asynchronous execution and review in more detail the setup of the task executor.

15.2.2 Configuring the infrastructure

There’s a distinction between designing your application so that it has parallel execution capabilities and whether they actually execute concurrently. The distinction comes from the fact that no matter how much you desire to run things in parallel, the application needs to allocate an appropriate number of virtual machine threads. In the previous section, we talked about the task executor, which is the main component that allows the asynchronous execution of tasks. Now, we provide an example of how to configure such a task executor. As you’ll see, it’s simple. To clarify the motivations behind it, though, we first look at the main concerns regarding thread allocation.

THREAD ALLOCATION CONCERN

Thread allocation is an infrastructure concern best kept separate from the business logic to allow an easy switch between environments—for example, scaling up and down or even running on different platforms. It addresses the main concerns regarding thread management, which can be summarized as follows:

Only a limited number of tasks may execute at once. Concurrent applications make better use of the system's resources, but too much of a good thing may be bad, and executing too many tasks at once may backfire. Each thread consumes its own share of memory, CPU, and other resources, and as consumption increases, a large number of highly concurrent activities may perform worse than the same number of activities executing with moderate concurrency (for example, by limiting the number of threads that can execute concurrently and forcing some of those activities to wait until their predecessors have completed).

Threads are expensive resources to create. Even if you limit the number of possible concurrent activities in the system, creating new threads is an expensive process, so destroying a thread after you're done with it would be a waste. As with any expensive resource, recycling is a better solution: instead of creating a new thread every time you need to launch a new concurrent process, you can use a pool of precreated threads, which can be borrowed as necessary and returned upon completion.

Threads are expensive resources to maintain. The number of concurrent tasks executing at once can be only as large as the number of threads in the pool. One impulse could be to improve performance by increasing the number of threads in the pool. Setting the size of the pool too low can force concurrent activities to execute sequentially, but setting it too high can make the application reserve a large amount of resources that it doesn't even use. Threads in the pool consume resources such as memory even when they're not running, and this may affect other processes executing on the same machine. Often, the solution is to allow the number of threads in the pool to fluctuate between a lower and an upper bound. This elasticity allows for the concurrency to increase when the system is under heavy load and to decrease when things return to normal.

Managed environments don't allow the direct creation of new threads. Managed environments, such as Java EE containers, are even more restrictive than other runtimes: application developers are explicitly forbidden to create new threads in the system. In such cases, the system exposes an API that application developers can use to launch concurrent activities.

CONFIGURING THE TASK EXECUTOR

The preceding concerns are mainly addressed by the Spring task executor abstraction. Whenever a component needs to execute a task asynchronously (a task in this case being to send a message or to poll a message source), it sends it to the task executor, which allocates a thread for its execution. Its most important characteristic is that the thread allocation mechanism is completely isolated from the application itself. The most common implementation of the task executor implementation is backed by a pool of threads, and the executor's job is to allocate a free thread or keep track of

the tasks that must be executed and to launch them when a thread becomes available. Adding such a component to your application is easy and is supported by a Spring XML namespace. In its simplest form, it can be defined as follows:

```
<task:executor id="executor"/>
```

The first example of using a task executor in a concurrent scenario is polling. Poller tasks are launched automatically by the framework and execute concurrently, using a scheduling mechanism configured transparently by the framework. A polling operation includes reading a message from a message source or buffered channel and sending it to the adapted channel or polling endpoint for further processing, a process which may include traversing other channels and endpoints as long as the handoff is synchronous. This means that a polling operation can take a long time to execute, keeping the internal scheduler thread. Depending on how many polling operations need to be executed at the same time, the internal scheduler may run out of threads, and other polling operations may have to wait. This is never a good thing, so one solution is to delegate the execution of the polling operation to an application-configured task executor, as in the following example. This configuration option applies to pollers configured inside endpoints or channel adapters as well.

```
<poller id="defaultPoller" fixed-delay="1000"
        default="true" task-executor="executor"/>
```

With this setup, the internal scheduler thread is kept busy for a short time, and it delegates the execution of the polling operation to the task executor. This approach doesn't increase the degree of concurrency in the application, but introduces a more sensible way of managing threads for the existing concurrent tasks.

In your application, you can use one of its different implementations, of which the most common is the one based on the J2SE 5.0 concurrency API. It's intended to be used by default and is supported by a Spring namespace. You previously saw a definition for it, and here's how you can also adjust the pool size.

```
<task:executor id="taskExecutor" pool-size="10"/>
```

A more sophisticated configuration can involve setting up a task executor with a variable pool size and a limit of, for example, 100 waiting tasks.

```
<task:executor id="taskExecutor"
        pool-size="5-25"
        queue-capacity="100"/>
```

A more detailed description of the API as well as other task executor implementations can be found in the next section, which takes a look under the hood.

15.3 Under the hood

In the previous sections, we provided a higher-level view on how to configure the task executors and schedulers that back up polling and concurrent task execution in Spring Integration to adjust performance and resource usage and to ensure that the application meets its timing requirements. For a typical user, this perspective is

enough, because the namespace support and abstraction implementations for task execution and scheduling covers the typical use cases.

This section focuses on the needs of more special cases that aren't covered by the defaults. It introduces you to the Spring API for task scheduling and execution and shows you how to provide your own implementations that customize thread allocation or timed event triggering.

15.3.1 The TaskExecutor API

As you saw earlier, the Spring Framework provides its own abstraction for executing concurrent activities:

```
public interface TaskExecutor extends Executor {  
    void execute(Runnable task);  
}
```

By default, an application can use an implementation based on the J2SE 5.0 Executor API, but in some situations, such as when running in managed environments that forbid the creation of new threads by Java EE components (including by using the J2SE 5.0 Executor API), the J2SE 5.0 Executor can't be used directly. Such environments are WebSphere and WebLogic, and, in general, any Java EE application server discourages the creation of such threads. In such cases, the application server exposes its own API in the form of a Java Naming and Directory Interface (JNDI)-bound WorkManager. Although not configurable directly by the application developer, its purpose is similar to that of the J2SE 5.0 Executor: an application can execute a new concurrent task through it.

This abstraction has various implementations, so depending on the environment, you can choose between one that leverages the J2SE 5.0, an adapter around the CommonJ WorkManager, and you can also implement this interface on your own, if needed.

By contrast, in a managed environment, you can use an environment-specific TaskExecutor:

```
<bean id="taskExecutor"  
      class="org.springframework.jca.work.WorkManagerTaskExecutor" />
```

From a strictly logical point of view, all three definitions produce equivalent results: a TaskExecutor that can be used to execute concurrent activities. Their behavior is different, though, reflecting the fact that you're using different implementation.

The TaskExecutor allows the launch of concurrent activities, and, generally speaking, the assumption is that it launches the activities as soon as possible (as soon as a thread is available to execute it). Also, it's assumed that the Runnable encapsulates a one-off activity that executes and finishes within a reasonable time frame (so that the thread on which it executes is freed up and another Runnable can be picked up for execution). But an application may need to schedule periodic activities as well as tasks that execute at specific moments in time, and the TaskExecutor isn't enough for that. For covering those cases, the Spring Framework provides a richer abstraction, the TaskScheduler.

15.3.2 The TaskScheduler API

We talked about task executors and how to delegate the execution of an operation to them. But to understand the scheduling mechanism, we need to look at a different mechanism: task scheduling. The work of a task executor is relatively simple: once a task (`Runnable`) is received, it tries to acquire the first available thread for it and, as soon as a thread becomes available, it executes it. But *as soon as possible* isn't sufficient when tasks need to be executed at precise moments. You need a more sophisticated contract.

```
public interface TaskScheduler {
    ScheduledFuture schedule(Runnable task, Trigger trigger);
    ScheduledFuture schedule(Runnable task, Date startTime);
    ScheduledFuture scheduleAtFixedRate(Runnable task, Date startTime,
                                       long period);
    ScheduledFuture scheduleAtFixedRate(Runnable task, long period);
    ScheduledFuture scheduleWithFixedDelay(Runnable task, Date startTime,
                                          long delay);
    ScheduledFuture scheduleWithFixedDelay(Runnable task, long delay);
}
```

As you see, the methods provided by the `TaskScheduler` allow you run a task at a specific moment or periodically (at either a fixed rate or a fixed interval). The role of the `ScheduledFuture` is to give the invoker a handle for inquiring about the remaining time until the task is scheduled to execute and the time when it will actually execute. You can also implement your own schedule by implementing the `Trigger` interface, shown next, along with the associated `TriggerContext` that allows an implementer to calculate the next scheduled time based on the previous history.

```
public interface Trigger {
    Date nextExecutionTime(TriggerContext triggerContext);
}

public interface TriggerContext {
    Date lastScheduledExecutionTime();
    Date lastActualExecutionTime();
    Date lastCompletionTime();
}
```

The `nextExecutionTime()` method returns the next moment when the task should run. A custom implementation of the `Trigger` interface is useful when the already provided implementations (fixed interval, fixed delay, and cron-expression-based) don't cover the needs of your use case—a rare occurrence, but a possible one. Suppose you have a task that needs to run every 10 minutes during working hours and every 2 hours outside that interval. None of the out-of-the-box triggers can provide this functionality, so you need a custom trigger, as in the following example:

```
public class CustomTrigger implements Trigger {

    public Date nextExecutionTime(TriggerContext triggerContext) {
        // ... implement
    }
}
```

This custom trigger can be used by a poller as follows:

```
<poller id="defaultPoller" trigger="1000" task-executor="taskExecutor"/>
```

As with the `TaskExecutor`, the Spring Framework provides special implementations for environments in which the J2SE 5 abstractions aren't applicable. Similar to the `WorkManagerTaskExecutor` from the previous section, a `TimerManager`-based scheduler can be used in a managed environment.

This wraps up the overview of the internal task execution and scheduling API. Although not a common need in day-to-day usage, providing your own implementations of those abstractions may come in handy in special cases. Also, now you have a better understanding of how the abstractions work, and you can make better decisions on what to use and when. It's time to wrap up the chapter, so let's take a quick look at what we covered.

15.4 Summary

You learned about the dynamic configuration options of Spring Integration. They're complementary to the component definitions you learned about in previous chapters. You didn't learn about a new type of component but about how to configure the existing ones.

Pollers allow you to control the timing aspects of your application. By adjusting the frequency rates, you can adjust its responsiveness to changes, such as data becoming available for processing in an external source or an internal message handoff that takes place over a message queue. Polling strategy, though, means not only the frequency of reading data but also other parameters such as the timeout and number of messages that are read in a cycle. Besides polling, you also learned how to use publishers to trigger operations that must take place at a given frequency or at specific times.

Applications must be able to do multiple things at once. But the degree of concurrency in an application, a factor that plays an important role in determining its throughput, is controlled in two ways. On one hand, you can design your application to allow for parallelizing a large number of its processing operations by breaking single-threaded flows that span multiple processing endpoints into smaller sequences and using asynchronous handoff to allow their execution on multiple threads. On the other hand, you need to allocate the resources (threads) so that these operations can run in parallel. Both aspects can be addressed by using the configuration options of the Spring Integration framework. There's a trade-off too, because important information, such as transactional context, is lost in the process of handing off messages asynchronously.

By wrapping up the handling of timeliness and concurrency, we can move further and look at a particular type of enterprise application integration: massive dataset processing and batch operations.

Batch applications and enterprise integration

This chapter covers

- Quick introduction to batch applications and Spring Batch
- Using batch application in enterprise integration
- Combining the functionality of Spring Integration and Spring Batch

An important niche in the landscape of software solutions is occupied by batch processes—applications that run independently with no user interaction. Akin to living fossils, they’re a glimpse into the history of computing. At the beginning of the computing era, collecting data and feeding it to programs that ran on mainframes was the only efficient option. Access to the actual system, let alone the idea of a user interface, was restricted to a limited number of operators.

Technology has evolved since then, and progress in user interface and communications technology has led to their gradual replacement with more interactive solutions. But, resorting again to an analogy with evolutionary biology, the secret of the survival of batch applications is the specific set of traits that ensured that they remain a superior solution in specific use cases.

By the end of this chapter, you'll be familiar with the most significant features of batch applications, recognizing the main differences between them and other paradigms, such as online transaction processing. In what concerns this book, batch applications are important for their role in enterprise integration.

Because this book focuses on Java enterprise software development, we highly recommend one of Spring Integration's siblings, Spring Batch, as an implementation solution (and if you take a strong interest in batch job implementation, we also recommend you read *Spring Batch in Action* to learn about it in detail). In this chapter, you get a quick glimpse of how to use it for implementing a typical batch job example. Apart from that, you'll find no sibling rivalry here: the two frameworks complement each other well to produce sophisticated, event-driven, and highly scalable applications, and learning about the various integration opportunities will conclude our foray into batch application development.

Let's begin by looking at what makes batch jobs so special.

16.1 Introducing batch jobs

Before introducing you to Spring Batch, we provide some background on the typical features of batch applications. This section is a must-read if batch applications are new or unfamiliar to you, and it's a recommended read for everyone else. It focuses on the main concepts and concerns that drive the particular design and implementation of batch jobs in general, laying the foundation for next section's insight into the specifics of Spring Batch.

A quick definition of batch processing would describe it as the execution of a set of programs (called *jobs*) with little to no manual intervention. Their working model is that the input data is collected in advance and provided up front in a machine-readable format, such as files or database entries, and the output is, in counterpoint, machine-consumable as well. As a result, these applications aren't user-driven, and the set of operations that need to be performed manually is limited to starting a job and introducing its input parameters (such as the location of the input file or other values relevant to a particular execution of the program) as well as checking the results and perhaps restarting failed jobs if necessary. Currently, this application model faces strong competition from *online transaction processing*, which was made possible by advances in user interface technology and communications infrastructure.

16.1.1 Online or batch, that's the question

To understand the differences between the two approaches, as shown in figure 16.1, consider a payment-processing application. The online transaction-processing approach would take the payments as soon as they're submitted, contact the buyer's and seller's financial institutions, execute the debit and credit operations, and return a confirmation immediately. The batch-processing approach bundles the payments and processes them later. We stress the importance of bundling as opposed to simple deferral. Batch processing isn't the same as asynchronous processing whereby

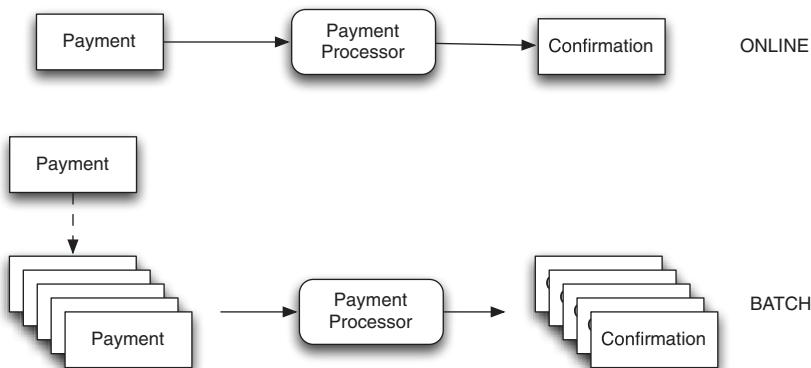


Figure 16.1 Online versus batch processing: Requests are processed individually or as a group respectively.

requests are buffered until the system can process them; in such cases, the input is still granular and consists of a single request at a time. What's characteristic for batch applications is that multiple requests (and we talk about of an order of magnitude of millions of items) are provided as input at once.

In some use cases, such as airline reservations, using a batch-processing approach makes little sense. You could collect the travel information and process requests overnight, returning an answer with the available options on the next day. It's an absurd example, but it serves to explain why highly interactive and online transaction-processing systems gained so much traction. In such situations, users want to receive answers immediately and be able to act on them: What if there's limited seating on the flight of their choice? But there are specific cases when processing items in a batch is necessary or at least preferable.

16.1.2 Batch processing: what's it good for?

For one thing, some business processes are based on the concept of gathering data during the day and processing them overnight. For example, processing credit card transactions in batches as opposed to in real time is preferred because of the increased security (when the list of transactions is sent once a day, it's easier to monitor and authorize than if interactions take place randomly, and this approach also simplifies logistics).

In other cases, batch processing is used for importing data outside the normal usage hours of the application. This may be necessary for avoiding data inconsistency: consider an online store that needs to update its products catalog every month: the store needs to be offline while the catalog is updated. On the other hand, this may be done for performance reasons—for example, a large import that involves writing a massive amount of data to the database can seriously impair the performance of a running application that uses the same storage system because of the high resource consumption or frequent table locking that may need to take place during the import.

Extract, transform, and load (ETL) processes (see figure 16.2) are frequently used in enterprise integration for transferring data between applications.

This is a fairly generic description of a batch job, and at a first glance, it looks simple—a lot of the batch jobs are implemented as scripts or as simple stored procedures that are executed periodically. A closer look reveals that they have a complex set of requirements, which in certain cases require a significantly more sophisticated approach. Let's look at an example to guide us through the intricacies of a batch job implementation.

16.1.3 Batch by example

Consider the case of a batch payment-processing system. At the end of each day, the day's transactions are submitted in a file and, overnight, the accounts in the system must be updated. The transactions are transmitted as a text file, each line containing the payer and payee accounts and the amount that's transferred between them, as well as the transfer date, as in the following listing. This application must record the payments and update the payer and payee accounts accordingly.

Listing 16.1 A input file for a batch job; each row represents an individual item

```
1,2,2.0,2011-01-03
3,4,10.0,2011-01-03
5,1,22.0,2011-01-04
1,2,21.75,2011-01-04
...
4,2,9.55,2011-01-05
```

The general outline of the batch job is shown in figure 16.3.

INPUT DATA MUST BE STREAMED

Each row of the input file represents a payment, and this application must parse this data and convert it into data structures. Because of the data volume, it's impractical to assume that the whole file will be loaded in memory and parsed from there. The transaction log contains millions of records, and the sheer amount of memory necessary to

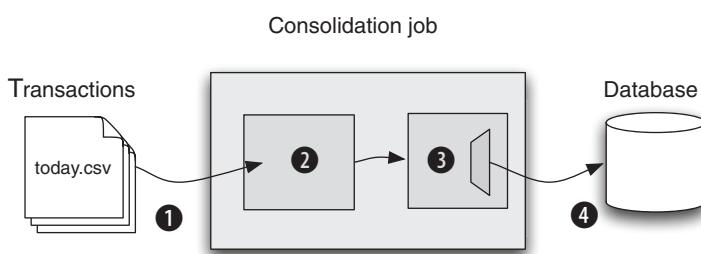


Figure 16.3 A simple batch job definition: The transactions are loaded from the input file ①, parsed ②, processed ③, and written to the database ④

hold that data might exceed the physical capacity of the machine—and if it doesn't, it'll cripple the performance of the application nevertheless.

Instead of doing that, the application must process the input data gradually, row by row—the applications must be capable of streaming the data. Streaming may need to be performed for various types of input sources ranging from input files for which the application must implement its own strategy of partitioning the content into records to database access where the records are already identified but the application may still need to maintain an open cursor so it can deal with a large amount of input data (think a `SELECT` statement returning millions of data records).

WORKING IN CHUNKS

Reading data is only half of the problem. Gradually reading and processing input data implies gradually writing output data as well. More often than not, the goal of processing is to process and transform input items into one or more output items (the alternative being a simple aggregate calculation). To ensure data consistency, operations must be transactional. The simplest approach, which isn't necessarily the right one, would be to process each item in its own transaction. But that may be inefficient because every transaction start and subsequent commit introduces an overhead. The impact of the overhead is inversely proportional to the number of processed items, which is why it can be neglected in an online system, where items are processed as soon as they arrive, but may have a serious impact when a huge number of items is at stake. The solution in this case is to span a transaction over a chunk of multiple items. The size of a chunk varies from case to case: large chunks will buffer the items in memory and drain resources, at the same time risking the loss of data if an error occurs, whereas extremely small chunks limit the benefit of this strategy.

DEALING WITH FAILURE

If the application fails while processing an item, it has a few options on what to do next: it can ignore the current item and pass to the next; it can retry processing the item; or it can decide whether to ignore the failed item, retry processing the faulty item, or abort the process altogether. A robust implementation should provide an automatic mechanism that can decide, based on the nature of the error, whether the application should retry to process an item (if the error is the result of a temporary condition), skip it (if the error is nonrecoverable but losing this piece of data is non-critical), or completely abort the execution.

When a batch process stops because of an irrecoverable error, it usually needs to be restarted after the errors are fixed. But usually this doesn't mean that everything has to be started again from the beginning. Consider an application that's importing data. Restarting the process after half of the data has been imported can mean, in the best case, that the items previously imported will be overwritten, and in the worst case, that they'll be imported twice. Ideally, the state of the batch process must be persistent, and the application must provide the necessary infrastructure for recovering after a crash by resuming from where it left.

As you can see, even if the business logic isn't too complicated, implementing a batch application can become a complex task because of the additional requirements for streaming, chunking, and recoverable execution. As is always the case with enterprise middleware, most of the effort may be spent on the infrastructure instead of the business logic. Fortunately, as sophisticated as they are, the fact that all the batch applications have a virtually identical set of requirements becomes a good opportunity to provide a generic solution: a framework.

16.2 Introducing Spring Batch

Spring Batch is a batch application development framework built on top of Spring. As a batch-oriented framework, it provides the generic constructs and infrastructure that implement the main components of a batch job, allowing you to fill in the details with your own implementation. Being built on top of Spring, it can take advantage of container's features, such as dependency injection, handling of cross-cutting concerns, and the extended set of modules that simplify Object-Relational Mapping (ORM) integration and database and messaging access.

Fundamentally, Spring Batch covers the implementation of a batch process by providing the following essential capabilities:

- A batch job skeleton and a toolkit of components for building the definition of a batch process as a collection of Spring beans
- A job execution API that allows you to launch new job instances
- A state management infrastructure

We illustrate how Spring Batch provides these facilities through a simple batch process.

16.2.1 A batch job in five minutes

To complete a reservation, our travel agency must receive the payment for it. The transactions are carried out through an external payment system. Overnight, the application must be updated with the transactions that took place during the day. The transactions are transmitted as a text file, each line containing the payer and payee accounts and the amount that's transferred between them. The application must record the payments and update the payer and payee accounts accordingly.

The batch job definition for this example is shown in the following listing.

Listing 16.2 Batch process definition in Spring Batch

```
<batch:job id="importPayments">
    <batch:step id="loadPayments">
        <batch:tasklet>
            <batch:chunk reader="itemReader" writer="itemWriter"
                ➔ commit-interval="5" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

```

<bean id="itemReader" class="o.s.b.i.f.FlatFileItemReader"
      scope="step">
    <property name="resource"
              value="file:///#{jobParameters['input.file.name']}"/>
    <property name="lineMapper">
      <bean class="o.s.b.i.f.m.DefaultLineMapper">
        <property name="lineTokenizer">
          <bean class="o.s.b.i.f.t.DelimitedLineTokenizer">
            <property name="names"
                      value="source,destination,amount,date"/>
          </bean>
        </property>
        <property name="fieldSetMapper">
          <bean class="com.manning.siia.batch.PaymentFieldSetMapper"/>
        </property>
      </bean>
    </property>
  </bean>

<bean id="itemWriter" class="com.manning.siia.batch.PaymentWriter">
  <constructor-arg ref="dataSource"/>
</bean>

```

The example uses the dedicated Spring Batch namespace as a domain-specific language to define batch jobs. Every batch job consists of a sequence of steps. More sophisticated applications can have multiple steps, but this example has just one: importing the batch file.

The most common way to implement a step is delegating to a tasklet, which defines the activity that must be performed, leaving the proper step implementation to deal with the boilerplate aspects of the execution (maintaining state, sending events, and so on). The work performed in a tasklet can be a simple service call or a complex operation. The most typically used tasklet implementation is the chunk-based tasklet (see figure 16.4), which processes a stream of items and writes back the results in bulk. The streaming and chunking aspects of batch processing can be easily identified: the `ItemReader` parses the raw input into a stream of items, releasing data as soon as a complete item is read; the `ItemProcessor` handles any data transformations (if necessary);

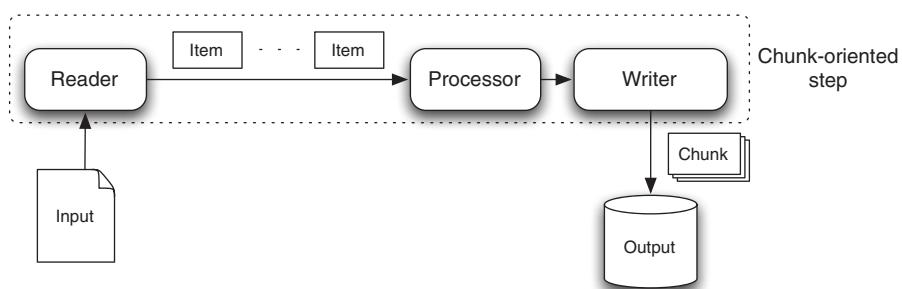


Figure 16.4 **Chunk-oriented step:** Items are read individually by the `ItemReader`, processed by the `ItemProcessor`, then written in bulk by the `ItemProcessor`.

and the `ItemWriter` aggregates output data and writes it back as soon as a chunk is assembled.

The framework allows for constructing batch jobs that consist of multiple steps, so, for example, the job may execute some additional tasks after importing the data, and the multistep job is the right way to do so. Also, steps may specify various levels of fault tolerance, allowing the process to retry or skip items for certain types of exceptions rather than stop a job if an error occurs.

Your job as an application developer is to provide the application-specific details of the job: the item reading, item processing, and item writing strategies. In practice, you don't have to implement all of it. Spring Batch provides its own toolkit of predefined components, which includes support for reading a file and transforming its content into a sequence of individual objects. Your responsibility in this case is to provide the strategy for transforming the individual row content into an object that can be further processed, as illustrated by the `PaymentFieldSetMapper` class from the following listing. The location of the file can be passed as a job property (more about it in the next section) so that you can execute this job for different physical files.

Listing 16.3 A FieldSetMapper converts raw input data read from the file (FieldSet) into a domain object

```
public class PaymentFieldSetMapper implements FieldSetMapper<Payment> {  
    @Override  
    public Payment mapFieldSet(FieldSet fieldSet) throws BindException {  
        Payment payment = new Payment();  
  
        payment.setSourceAccountNo(fieldSet.readString("source"));  
        payment.setDestinationAccountNo(  
            fieldSet.readString("destination"));  
        payment.setAmount(fieldSet.readBigDecimal("amount"));  
        payment.setDate(fieldSet.readDate("date"));  
  
        return payment;  
    }  
}
```

The same goes for writing the item back. Your only responsibility is to provide a strategy for persisting the read data in the form of a `PaymentWriter`.

As you can see, the batch job definition can be created as a Spring bean, using dependency injection for customizing various aspects such as the tasks that must be performed during the job or the read and write strategy. In most cases, you can use one of the out-of-the box reader implementations and customize it for your application needs so that you can eliminate the critical yet tedious task of dealing with file content, database cursors, and so on. The only thing you need to provide is what's specific to your application: a definition of your business entities and a strategy for converting the basic records read from the file to them, as well as the functionality that needs to be performed with these items once they're extracted from the incoming stream.

Next you need to set up the infrastructure so that you can execute the job.

16.2.2 Getting the job done

What you've created so far is the definition of a batch job. To put it to work, you must start it using the executor API. To get access to it, you must define a distinct type of bean in your context definition: a batch job launcher, shown in the following listing.

Listing 16.4 A JobLauncher bean is used for starting job executions

```
<bean id="jobLauncher" class="o.s.ba.c.l.s.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository"/>
</bean>
```

With this definition in place, you can start a new batch job as in the following example (noting that the `Job` instance is the bean defined in the previous section):

```
JobParametersBuilder jobParametersBuilder = new JobParametersBuilder();
jobParametersBuilder.addString("input.file.name", "payment.input");
JobExecution execution =
    launcher.run(job, jobParametersBuilder.toJobParameters());
```

The role of the `JobLauncher` is to create a new batch job instance according to the `Job` definition. The `JobParameters` passed as arguments while launching the job fulfill a dual role: as their name indicates, you can use them to pass particular properties that are specific to this job execution instance (such as the name of the file that needs to be processed), and you can also use them as unique identifiers of the `Job` execution.

There can be no two `JobExecutions` for the same `Job` and `JobParameter` set of values. This is an important point to take into account when executing batch jobs automatically: when launching a new job instance, the list of parameters must contain at least one unique attribute; otherwise, the launcher will complain that the job has already completed.

The returned `JobExecution` is a handle that allows the invoker to access the state of the executing `Job`. Spring Batch supports synchronous and asynchronous execution of batch jobs (which can be configured through the `jobLauncher` bean definition), which means that the call to `JobLauncher.run()` may return while the job is in progress. As such, an application may track the progress of a batch job by repeatedly inquiring on the `JobExecution` instance about the current state.

This leads us to the third and final component of Spring Batch: the job execution state management infrastructure.

You got a hint in the previous section about what we discuss in the next section. The `JobLauncher` bean we configured in listing 16.4 is injected with a `jobRepository`. As we explained earlier, the state of the batch jobs is persistent so that jobs can be resumed after a failure or a shutdown. Resuming a job that stopped in the middle of processing a data stream requires avoiding the potential duplication of data that may occur as a result of processing items twice. The job repository records the ongoing progress of the running batch jobs, including information such as the current step and how far the application got while processing the data stream that corresponds to a particular step.

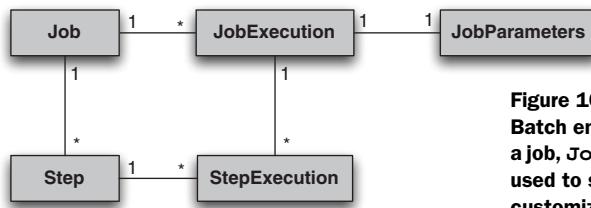


Figure 16.5 The relationship between Spring Batch entities: `Job` and `Step` define the logic of a job, `JobExecution` and `StepExecution` are used to save the state, and `JobParameters` customizes a specific job instance.

Unless you have special needs, all you need to do to create a job repository is include a bean definition in your context. The repository uses a set of database tables, which is created automatically when the repository is instantiated for the first time.

```
<batch:job-repository data-source="dataSource" id="jobRepository"
  transaction-manager="transactionManager" table-prefix="BATCH_"/>
```

This concludes our quick overview of Spring Batch and its capabilities. As you can see, it allows you to create pretty complex batch jobs, launch them, and manage and interrogate their state, all while writing only a limited amount of code. Let's see what opportunities are available for using all these capabilities in enterprise integration scenarios and, mainly, how Spring Batch interfaces with Spring Integration.

16.3 Integrating Spring Batch and Spring Integration

Using Spring Batch greatly simplifies the process of implementing a batch job. Apart from the job implementation, a complete solution must be able to schedule and launch jobs automatically, monitor them, and interact with the environment and other applications. For achieving these goals, implementors must look beyond the framework.

Fortunately, this goal can be achieved with minimal effort by pairing Spring Batch with Spring Integration. The two frameworks complement each other, and together they can provide a complete solution for creating enterprise integration applications that use batch processes for handling large quantities of data.

Spring Batch Admin is an open source project that provides a web-based administration console for Spring Batch applications. It has access to a set of job definitions and a job repository, and it provides support for:

- Inspecting jobs and adding new job definitions
- Launching and stopping jobs, including uploading data files that can be used as input when launching a job
- Inspecting job executions

Of interest to our topic is that most of the interaction between the web application and the batch jobs is done via Spring Integration. Spring Integration provides a set of utility classes that implement most of the collaboration patterns discussed in this section and can be used out-of-the-box in your applications.

We examine the various ways in which Spring Integration and Spring Batch can collaborate, and we present examples that use components provided by the spring-batch-integration module of Spring Batch Admin.

16.3.1 Launching batch jobs through messages

Let's assume that you finished implementing the batch job, as described in the previous section. The next step is to launch it, which you can do from the Spring Batch API by invoking `JobLauncher.run()` with the appropriate parameters. But how does this method get called in an application? You can write a web application that invokes the method from a controller, or you can invoke it from the `main()` method of a launcher class for command-line operation. In the latter case, you can create a shell script or schedule the batch job using a scheduler such as cron.

But you can create even more powerful scenarios, such as running batch jobs in an event-driven fashion, for example, whenever a file is dropped in a target directory. Using Spring Integration, it's easy. A file channel adapter will monitor the directory and send out a message whenever a new file is detected, as shown in figure 16.6.

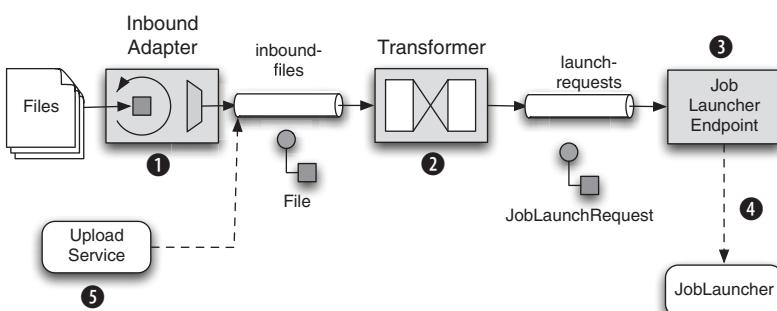


Figure 16.6 Event-driven batch job launch: The file channel adapter monitors a directory ①, a file message is converted to a `JobLaunchRequest` ②, which triggers the batch job launch (③④). Other components may trigger a launch by sending a file payload ⑤.

Spring Batch Admin provides a `JobLaunchingMessageHandler` that can be used to launch batch jobs using information provided by an inbound message with a `JobLaunchRequest` payload, which is a wrapper around the `Job` that needs to be launched and the `JobParameters` for this launch. The configuration for such a message bus is displayed in the following listing, where the launching message is triggered by dropping a file in a directory monitored by a file channel adapter.

Listing 16.5 Triggering jobs with messages sent by a file channel adapter

```

<si:channel id="files"/>
<si:channel id="requests"/>
<si:channel id="statuses">
    <si:queue capacity="10"/>

```

```

</si:channel>

<si-file:inbound-channel-adapter
    directory="classpath:/${incoming.directory}"
    channel="files"/>

<si:transformer input-channel="files" output-channel="requests">
    <bean class="com.manning.siiia.batch.FileMessageToJobRequest">
        <property name="job" ref="importPayments"/>
        <property name="fileParameterName" value="input.file.name"/>
    </bean>
</si:transformer>

<si:service-activator method="launch" input-channel="requests"
    output-channel="statuses">
    <bean id="messageHandler"
        class=
            "org.s..batch.integration.launch.JobLaunchingMessageHandler">
        <constructor-arg ref="jobRunner"/>
    </bean>
</si:service-activator>

```

Whereas most of the components are provided out-of-the-box, the conversion between the file sent by the channel adapter and the JobLaunchRequest is application-specific and therefore must be implemented as part of the solution. An example is shown in the following listing.

Listing 16.6 Transforming a file into a JobLaunchRequest

```

public class FileMessageToJobRequestTransformer {
    private Job job;
    private String fileParameterName;

    public void setFileParameterName(String fileParameterName) {
        this.fileParameterName = fileParameterName;
    }

    public void setJob(Job job) {
        this.job = job;
    }

    @Transformer
    public JobLaunchRequest toRequest(Message<File> message)
    {
        JobParametersBuilder jobParametersBuilder =
            new JobParametersBuilder();
        jobParametersBuilder
            .addString(fileParameterName,
                message.getPayload().getAbsolutePath());
        return new JobLaunchRequest(job,
            jobParametersBuilder.toJobParameters());
    }
}

```

The event-driven approach for launching jobs is extremely flexible, allowing for a large number of variations in implementation. For example, any kind of channel

adapter output can be transformed into a `JobLaunchRequest`, so you can trigger the batch jobs for inbound emails, Java Message Service (JMS) messages, or file Transfer Protocol (FTP)-accessible files. As a matter of fact, the trigger doesn't even have to be a channel adapter: for example, a scheduled publisher may be used for launching batch jobs periodically.

Besides the sheer variety of sources, another benefit of integrating the batch job launch in a pipes-and-filters architecture is the ability to take on multiple input formats simultaneously. The batch job may require a canonical format, but different sources of data may use different formats, using intermediate transformation steps for adjusting the data formats.

But message-based integration can work either way: not only can batch processes be launched using messages, but they can also send notifications while they change state.

16.3.2 Providing feedback with informational messages

A batch job can run for a long time. For operators, it's critical to get up-to-date information about its progress, and most important, whether it completed successfully or had to stop because of an unrecoverable failure. As in any similar situation, this information can be gathered through active polling or in an event-driven manner.

Spring Batch provides a `JobExecution` interface that reflects the current status of the eponymous job execution, so the application that launched the job can use it as a handle for inquiring whether the job is complete or still running, what its current status is, if it stopped because of an error, and so on, and in the case of running processes, it can be used to stop them. But polling the `JobExecution` repeatedly is suboptimal. An event-driven approach is superior and preferable.

Spring Batch provides a mechanism for registering listeners such as `StepListener`, `ChunkListener`, and `JobExecutionListener` that get invoked during the execution of a job on events such as before and after processing an item, on read/write errors, or on the completion of a job execution. Figure 16.7 shows an example of using listeners for message-based integration. In this example, a `JobExecutionListener` registered with the Job sends a notification message with a `JobExecution` payload after a job has stopped running. A router decides, based on the execution status, whether it's necessary to take any further steps (such as relaunching the job if it failed and the failures

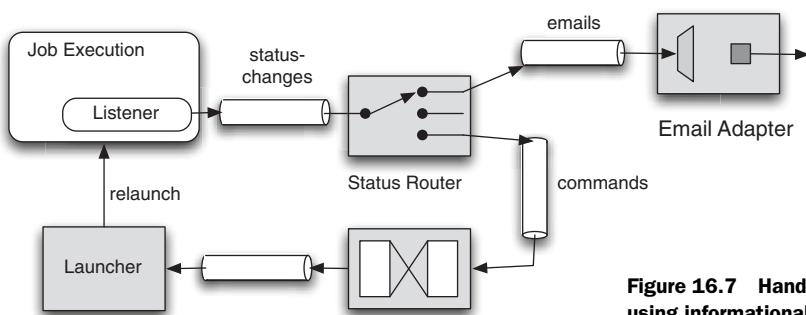


Figure 16.7 Handling batch events using informational messages

are recoverable, perhaps with a delay), in which case the message is sent to an appropriate endpoint to be processed, or it's a matter of sending a notification message (completion was successful or the failures are unrecoverable), in which case the message is sent to an outbound email channel adapter.

For this, you need to add the listener to the job definition and add the rest of the bus configuration, as shown in the following listing.

Listing 16.7 Registering a job listener and handling batch notifications through the message bus

```
<batch:job id="importPayments">
    <!-- other properties omitted -->
    <batch:listeners>
        <batch:listener>
            <bean
                class="com.manning.siia.batch.PaymentJobExecutionListener"/>
        </batch:listener>
    </batch:listeners>
</batch:job>

<si:gateway id="notificationExecutionsListener"
    service-interface=
        "org.springframework.batch.core.JobExecutionListener"
    default-request-channel="jobExecutions" />

<si:router id="executionsRouter"
    input-channel="jobExecutions">
    <bean
        class="com.manning.siia.batch.JobExecutionsRouter"/>
</si:router>

<si:transformer id="mailBodyTransformer"
    input-channel="notifiableExecutions"
    output-channel="mailNotifications">
    <bean
        class="com.manning.siia.batch.ExecutionsToMailTransformer"/>
</si:transformer>

<si:chain input-channel="jobRestarts">
    <si:delayer default-delay="10000"/>
    <si:service-activator >
        <bean class="com.manning.siia.batch.JobRestart"/>
    </si:service-activator>

    <si-mail:outbound-channel-adapter id="notificationsSender"
        channel="mailNotifications" mail-sender="mailSender"/>

```

Message-driven job launch and message-driven notifications are two examples of Spring Batch–Spring Integration collaboration, which expands the feature set of the application and enables the two frameworks to collaborate to provide a scalable execution infrastructure.

16.3.3 Externalizing batch process execution

The way we've described it so far, it would seem that the natural way to combine the two frameworks is by wrapping Spring Batch jobs in a Spring Integration outer shell that can take care of the interactions with the external world. But it isn't necessarily so. Spring Batch can use Spring Integration internally, too, for delegating the processing of an item or even a chunk outside the process.

In a simpler case, the `ItemProcessor` can be a messaging gateway, as shown in figure 16.8, thus deferring the processing of items to the message bus. This is useful when item-processing logic is fairly complex and involves invoking multiple transformations or service invocations, either local or remote (through the use of channel adapters and gateways). When item processing on the bus takes a relatively long time, such as when the application performs a remote invocation, you can increase performance by introducing asynchronous item processing. You use two wrapper components provided by Spring Batch Integration for your item reader and item writer: `AsyncItemProcessor` and the corresponding `AsyncItemWriter`. This is essentially a fork-join scenario: by using the `AsyncItemProcessor`, the invocations on the gateway are performed concurrently rather than sequentially, leaving the `AsyncItemWriter` to gather the results and write back the chunk as soon as all the results are available.

In a more elaborate use case, shown in figure 16.9, an application can use a writer, for example the `ChunkMessageChannelItemWriter` provided by Spring Batch Integration, to send an entire chunk externally to a gateway. In this case, the batch job only reads items and groups them, and once a chunk is sent out, it continues to read items and assemble chunks without waiting for a result. It is the job of the `ChunkMessageChannelItemWriter` to get the results from the gateway and integrate them in the batch process. By introducing asynchronous processing (for example, by using a buffered channel instead of a direct channel), you can increase the concurrency of the system. By using channel adapters, you can completely externalize the processing of a chunk, creating the premise of a distributed architecture. For example, you can use a JMS channel adapter to send chunks on a message queue, letting external components read the chunks, handle them individually, and return information about their successful completion.

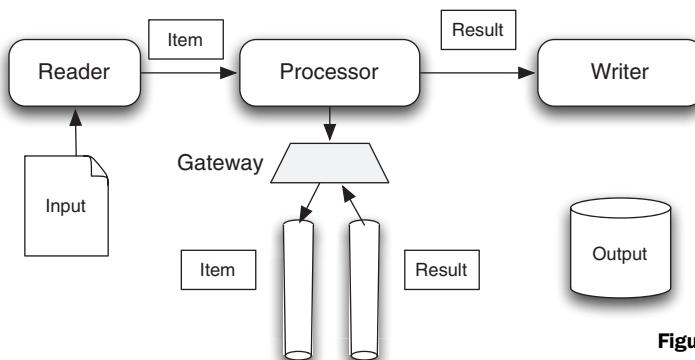


Figure 16.8 A gateway is used as an `ItemProcessor`.

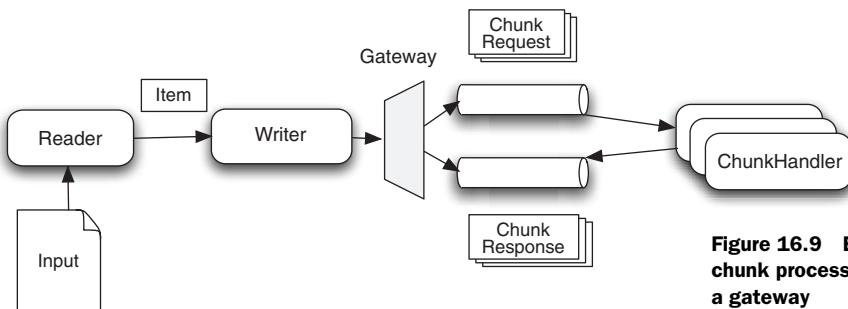


Figure 16.9 Externalizing chunk processing by using a gateway

The latter part is necessary because the batch process must update the `JobExecution` state continuously to acknowledge whether the step execution succeeded. Because of that, the request and response messages contain metadata that helps the batch job executor carry out this process (Did we get responses for all chunks? Were there any errors?). The chunk messages are received by another special component, the `ChunkProcessor`/`ChunkHandler`, which processes the chunk by delegating to an `ItemProcessor` and `ItemWriter`.

By looking at Spring Integration as underlying support for boosting the concurrency and distribution capabilities of Spring Batch, we've concluded our overview of the collaboration between the two frameworks. It's time to move on to the next chapter, but first, let's quickly review the takeaways of this chapter.

16.4 Summary

This chapter provided an overview of batch processes and their typical features, and you gained a general understanding of how they fit in the enterprise integration landscape. In a Java EE environment, you can create powerful batch processing applications by using Spring Batch, which, in a typical Inversion of Control fashion, provides the infrastructure for defining and executing jobs, including facilities for managing state.

The main strength of Spring Batch is in dealing with the complex details of defining and running batch jobs, such as chunking and transaction management as well as state management. Spring Batch doesn't include facilities for scheduling batch jobs or executing them in an event-driven fashion, but it finds an ideal companion in Spring Integration, whose strength is in dealing with the challenges of interacting with external systems and creating an event-driven environment that can be used for launching and managing batch jobs.

The two components of the Spring portfolio are complementary frameworks that can and should be used together for implementing a complete batch job-based enterprise integration solution. The integration between the two goes from simple, event-driven launch of batch jobs from Spring Integration to more sophisticated notification and automated feedback mechanisms using events published by Spring Batch and, even beyond that, to creating parallel and distributed batch jobs using Spring Integration as the underlying infrastructure.

17

Scaling messaging applications with OSGi and AMQP

This chapter covers

- Overview of the OSGi module system
- Reasons to combine messaging with OSGi

In this chapter, we investigate scalability of messaging applications. Of the multiple types of scalability, the most discussed is the scalability of an application at runtime, but *scalability* is also used in other contexts, such as the scalability of a business model and the scalability of a project. This chapter focuses on scalability at runtime and at development time. First we introduce the OSGi module system and then we link it to messaging to show the complementary nature of the two.

Since 1998, OSGi has been primarily driven as a needed extension of Java—needed because Java’s module system is simplistic. Java loads all classes in all jars it sees on the classpath linearly, and thereby exposes all known types on the classpath to all other known types. In large applications, this can be catastrophic, so OSGi was introduced to resolve this problem.

OSGi didn't start out primarily as a modularity fix for the Java classloader but as an extension that would help to run dynamic applications that wouldn't have all their services active at all times during their lifecycle. An important positive side effect that was recognized early was that it would also help structure large applications and fix the simplistic Java classloader mechanism. Now both modularity and dynamic type loading are essential features of OSGi.

As systems grow larger, not only modularity but also response times and deadlock prevention are concerns. The biggest danger to these two technical requirements are synchronous invocations over the network, for which OSGi by itself doesn't offer a solution. At this point in the book, the following statement should be obvious: OSGi and messaging are a match made in heaven. Spring and OSGi were pictured by Peter Kriens as a puppy and a kitten lying in the same basket. Spring Dynamic Modules (Spring DM) goes a long way toward making this picture accurate, and from a functional point of view, Spring Integration completes the circle. Spring Integration shouldn't be credited too much in this because, as you'll see in the coming sections, *it just works* because of Spring DM. As a developer, you can benefit greatly by using them together. As you can see in figure 17.1, Spring Integration can run in any OSGi container using Spring DM.

To make the most of a Spring Integration application on an OSGi platform, you should read up on Spring DM and OSGi. We recommend *Spring Dynamic Modules in Action* by Arnaud Cogoluegues, Thierry Templier, and Andy Piper (Manning, 2011) and *OSGi in Action: Creating Modular Applications in Java* by Richard S. Hall, Karl Pauls, Stuart McCulloch, and David Savage (Manning, 2011).

Before we show off the combination of messaging and OSGi, we must lay a foundation. We briefly explore OSGi's module system and then focus on the Service Registry. Only then are we ready to combine messaging (Spring Integration) and dynamic modules (Spring DM plus OSGi). Finally, we look in detail at how to replace service implementations in a running system.

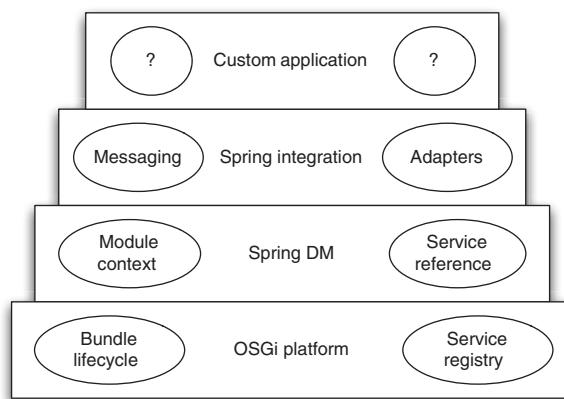


Figure 17.1 On an OSGi platform (bottom), you can run bundles that make use of Spring DM (middle). Spring Integration (top) can be used in combination with Spring DM to combine messaging with dynamic loading characteristics.

17.1 The OSGi module system

In this section, we describe the main functionality enabled by OSGi: its module system. We cover the reasons behind the module system's design, and we show you how to dynamically (re)load bundles at runtime.

Let's say you're working on software in a team. The application grows large, and you need to modularize it. As the requirements keep coming in and the application becomes more successful, you need different teams to work on parts of the application. After a while, you find that the teams have different release cycles and depend on each other's work. Modularity is becoming a hairy problem: different teams should be able to deliver on their own schedule without causing downtime. You need a rock-solid solution—one similar to that shown in figure 17.2.

Apart from a technical solution, to modularize your workflow as in the figure, you need significant changes in the development and delivery processes. Most of the effort involves defining and changing processes and architecture. OSGi is an option that can work to your advantage if you know how it works and what functionality it offers.

As mentioned, OSGi brings modularity to Java. In this section, we review the features that are relevant to the application of OSGi and messaging in enterprise architecture. But first we look a bit deeper into the Java type system.

In a normal Java runtime, your code can be modularized into different packages inside individual jars on the classpath. Classes can hide their internals from other classes by using modifiers, which gives them encapsulation. Jars offer no such option. Once a classloader loads a class, it can never unload it. The only way to work around this restriction is to use a dedicated classloader for the set of classes you want to reload. It can be done, but it's not easy.

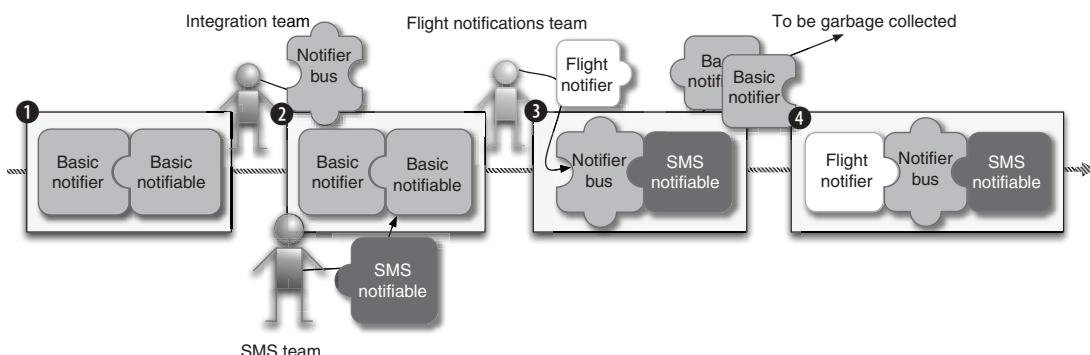


Figure 17.2 ① The integration team sets up the server and deploys some basic bundles to it. ② The Short Message Service (SMS) team develops an SMS bundle to replace the basic receiver for notifications. In parallel, the integration team replaces the basic notifier with a bus so they can connect multiple sources of notifications. ③ The flight notifications team writes a flight notifier bundle that works with the bus, and in the meantime, the bus and the SMS service are deployed. The obsolete basic bundles are uninstalled as their replacements kick in. ④ The new bundles are working in concert, and replacement versions for individual bundles can be deployed as needed without bringing the system down.

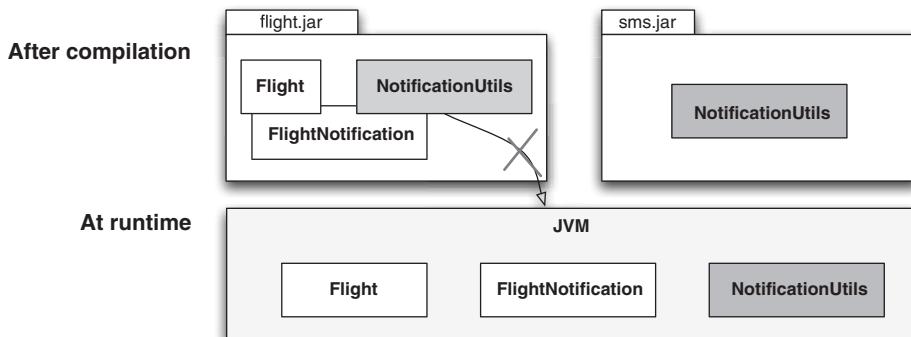


Figure 17.3 Modularity and it's runtime representation in Java.

Figure 17.3 show the internal representation of types in a classic classloader. Because of this oversimplification, hardly any protection is provided against type duplication and versioning issues.

Not only is the runtime representation of types too simplistic, it's also too rigid, as explained in the next paragraph.

The second problem you run into when you want to dynamically modify your running system is changing the classpath at runtime. If a new module must be deployed, you might not want to restart all the other modules. Again, it can be done, but it's not easy. The combination of these two problems becomes prohibitive for a single team with a production deadline.

In the late 1990s, many companies faced these problems, and most of them found their way out of them by using Remote Method Invocation (RMI) and web services. The downside of this solution is that going over the network is expensive (in terms of performance and/or hardware). You should avoid this approach if you can.

OSGi provides a better solution. It abstracts away the complex but not impossible solutions to type reloading that are provided in standard Java and introduces a powerful vocabulary that makes sense of modularity. In OSGi, a jar is turned into a *bundle*, which is a jar with a special manifest. In the manifest, you can tell the OSGi container what the bundle is, how to start and stop it, what it depends on, and what it has to offer. When you use the OSGi, lifecycle modules are started in the right order and services are invoked only if they can do their job.

17.1.1 The bundle lifecycle in an OSGi environment

A typical happy scenario for a bundle lifecycle starts with the installation of a bundle. Once the bundle is installed and the OSGi framework is aware of it, the framework can start looking for the bundle's dependencies. The dependencies of the bundle consist of types and services. Before a bundle can start (while it's still in the INSTALLED state), another bundle must expose all the types that bundle needs. If this is successful, the bundle enters the RESOLVED state. After all the code dependencies are resolved, a

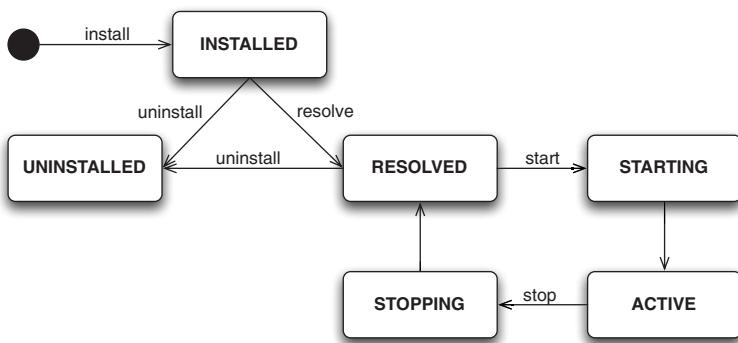


Figure 17.4 OSGi specifies a clear lifecycle for bundles. In contrast to plain Java, where class loading is out of the developer's control, bundles can be installed, uninstalled, started, and stopped as needed. The transitions between states, as depicted by the arrows, are clearly defined and restricted by the standard.

bundle may be started, which takes it through the `STARTED` state to `ACTIVE`. At this point, the bundle is eligible to receive incoming service calls until it's stopped (back to the `RESOLVED` state). A diagram of these states is shown in figure 17.4.

You need to learn more about bundle behavior to use it effectively. For example, a bundle might not move to the `RESOLVED` state because some of its required dependencies are unavailable, and inside the `ACTIVE` state are substates that may interest you.

Because this book is not about OSGi, we don't cover all the intricate details in depth. We cover just enough to establish the relationship between OSGi and Spring Integration. If you want to design a solution around OSGi and messaging, it's a good idea to read *OSGi in Action*. Now that you know a few basics about OSGi, you're ready for the next section, where we look into the Service Registry and what Spring DM brings in terms of integrating OSGi with Spring.

17.2 Accessing the Service Registry through Spring DM

Let's look a little closer at how OSGi and Spring DM enable dynamic reloading of services. You can use these services from Spring Integration, but you need a way to access services in other bundles, which is addressed by the OSGi Service Registry.

OSGi has a Service Registry that can be used to register plain old Java objects (POJOs) as OSGi services so that an object from one bundle can invoke a method on an object in another bundle. That solves one problem, but another challenge is that this registry is a simple, low-level API that requires you to write some unwieldy code. The following code sample is adapted from an article by Neil Bartlett, “A Comparison of Eclipse Extensions and OSGi Services” (<http://mng.bz/d7CO>).

```

ServiceReference ref = context.getServiceReference(
    ICustomerLookup.class.getName());
if(ref != null) {
    ICustomerLookup lookup = (ICustomerLookup)
        
```

```

        context.getService(ref);
    if(lookup != null) {
        Customer cust = lookup.findById(id);
        context.ungetService(ref);
        return cust.getName();
    }
}

```

This is where Spring DM comes in. As you already know, Spring is good at decoupling dependencies through Inversion of Control (IoC). In a normal Spring application, this means services are instantiated by the framework and no longer by their clients directly. Spring DM extends this paradigm by registering and referencing services in the OSGi Service Registry. This means the main benefits of Spring stay the same; the only difference is that the instance of the dependency is taken from a different place. Because the whole point of IoC is to make the client agnostic about the origin of its dependency, you typically don't need to change any Java code to migrate an application to OSGi.

The best way to understand how Spring DM and OSGi work together is to look at the typical layout of a bundle using Spring DM (see figure 17.5).

When a bundle is loaded with Spring DM, some special steps are taken during startup to instantiate the ApplicationContext specific to this bundle. This Bundle-Context is like a normal Spring application context but with some added features particular to OSGi.

Let's look at an example of a service that's exposed by bundle com.manning.siiia.integration.notifications.sms and referenced from bundle com.manning.siiia.integration.notifications. The service implements an interface exposed as a type by another bundle. Now in the SMS bundle, you can expose the service like this:

```
<osgi:service ref="smsNotifier" interface="c.m.i.n.Notifiable"/>
```

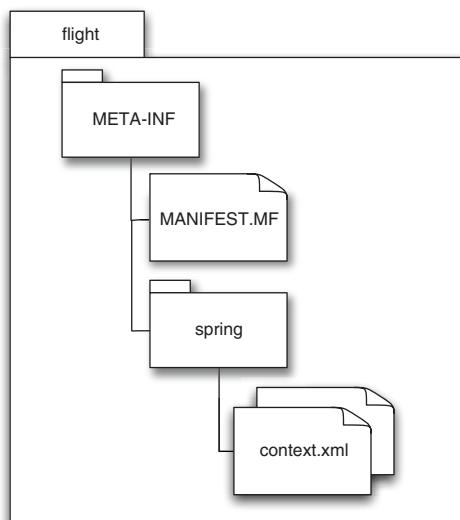


Figure 17.5 Inside the **META-INF** directory, we see the OSGi standard **MANIFEST.MF** file containing the dependency directives. We also see a **Spring** folder. Spring DM reads the .xml files in this folder and starts a special application context (**BundleContext**) containing the beans described in these files.

This creates a ServiceRegistration for the smsNotifier bean in the SMS bundle's context. Subsequently, you can reference the service from the notifier bundle like this:

```
<osgi:reference id="smsNotifier" interface="c.m.i.n.Notifiable"/>
```

This creates a proxy for the object in the Service Registry that implements the given interface and can be injected into another bean in the notifier context without changing a line of code in the client.

To make this look good, we used the osgi namespace here. This will work only if you add it to the header of your Spring .xml files, as you saw earlier.

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

Now you have two bundles, each with its own Spring context, but you created a bridge between them using the OSGi Service Registry. From here, things get seriously interesting. Because Spring DM has placed a proxy in the middle that can deal with the details of OSGi, it can hold method calls while the target service is being replaced. You can even start one version of the SMS bundle, run the application on it sending notifications, and then *replace the service in a running system without any outage!*

You might have noticed that in our focus on OSGi, we haven't put Spring Integration's advantages to use at all. Instead we created a reference between bundles based on the Notifiable interface, which is fine, but in that process, we also introduced a dependency from the notifier bundle on the bean name of the smsNotifier, which means generic notifications are now coupled to the SMS implementation.

OSGi offers a couple of tricks to compensate for this, but a channel seems like the perfect shared concept to bridge the gap here. If we were to share a channel between bundles, neither sender nor receiver would have to know about anything other than the payload. This is exactly what we explore in the next section.

17.3 Messaging between bundles

Spring Integration is all about modularization. It allows two services to collaborate without having any dependency on each other apart from a shared comprehension of the payload that's passed between them. This is helpful, as you learned in previous chapters, but without a fix for Java's flat classpath, the services will still be visible to each other within the same Java Virtual Machine (JVM) unless you take on the extra complexity of adding a transport layer in between virtual machines.

17.3.1 Reasons to combine OSGi with messaging

Let's say you want to get cash from your account. You could walk into a bank and wait in line, but that exposes you to all kinds of services that you aren't interested in. You'll have to avoid tripping over children, you'll have to be social to the teller, and worse, you'll have to interact with other customers who might want to chitchat about their day or ask you directions. The bank is a dangerous and confusing place. If you interact with the automatic teller machine instead, your cash withdrawal would be much simpler. The service will get only the information you want to expose. There is no need to talk about the weather and distracting interactions. OSGi is like that: it avoids unwanted coupling between the service and its clients. In figure 17.6 you can see the conceptual differences between classic interactions and using OSGi in a bank analogy. In the bank example, you might argue for the social value of talking to other human beings, but inside a JVM, such interaction is typically not an advantage.

OSGi provides clean modules that hide their internals from each other. With OSGi and Spring Integration, two services can collaborate without any visibility between them other than their contract. This type of decoupling becomes more valuable when a system grows larger than what a single team can manage on a single release cycle. In a system that's large but stays within the boundaries of one corporate entity, there's no gain in separating modules with a transport layer, security boundaries, and the associated performance drag. Because of this, OSGi combined with messaging is particularly useful in large enterprise architectures.

A system modularized with OSGi and decoupled with messaging can evolve much more naturally as multiple teams work on it at their own pace. The production system runs on as few nodes as possible to reduce the network overhead, and bundles are installed in different release schedules without ever bringing the system down.

This idea might sound less reliable than redeploying everything on a single node and bringing that node back up, but there's no fundamental difference in terms of

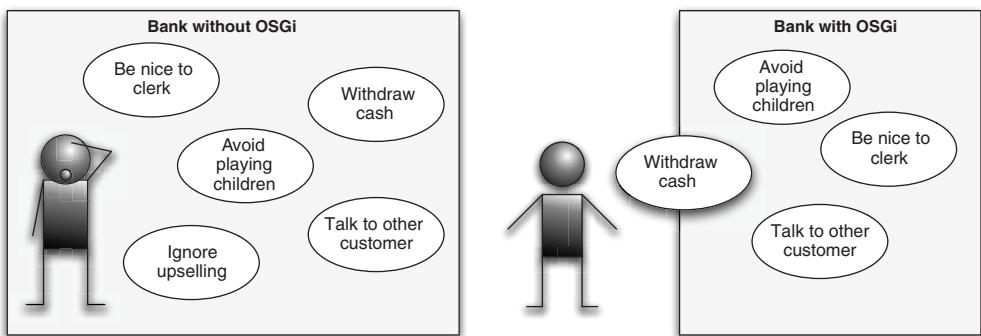


Figure 17.6 On the left is the interaction with a bank, analogous to the classic Java approach involving a lot of unwanted potential interaction. On the right is the improved interaction with an exposed automatic teller machine, analogous to a service exposed using OSGi.

deployment architecture except that a bundle can be brought up and down much faster and a new version can be installed before the old version is taken down without having to commission more hardware. Compared with a typical production setup, the equivalent OSGi-based solution leaves less room for error because of the strict lifecycle definition provided by the OSGi standard.

The real downside is in development time complexity. Debugging and testing an OSGi-based application requires a different skill set than debugging a classic Java program, so as with any technical solution, OSGi should be used only when the reduction of maintenance costs more than balances the development costs. Maintenance costs are typically undervalued during development, so even the added initial complexity might be outstripped surprisingly quickly once a system goes into production. In any case, to a Java developer, OSGi is just one more tool that belongs in the toolbox.

In the next few sections, we show some examples of how bundles can be wired together using Spring Integration and OSGi. We discuss publish-subscribe communication, point-to-point messaging, and load balancing, and we look into exposing gateways as a way to avoid unneeded Spring Integration dependencies.

17.3.2 Publish-subscribe messaging between bundles

Publish-subscribe messaging is the easiest form of messaging to implement with OSGi. In a publish-subscribe configuration, the publisher is mostly indifferent to whether someone listens. When someone posts a tweet, he or she is probably not concerned with whether others will read the tweet.

Publishing is a service that's offered to interested subscribers. This is an interesting contrast with point-to-point messaging in which delivery of the message is a service offered to the sender. The service interface in this case is `SubscribableChannel`, or an interface of your own choosing in accordance with the Observer pattern.

To connect to bundles as a publisher and subscriber, you have the publisher expose a publish-subscribe channel as a service, and then reference it inside the subscriber bundle. In the publisher context:

```
<si:publish-subscribe-channel name="notifications" />
  <osgi:service ref="notifications"
    interface="o.s.i...SubscribableChannel"/>
```

In the subscriber context an OSGi reference allows you to refer to the channel as you normally would:

```
<osgi:reference id="notifications"
  interface="o.s.i...SubscribableChannel"/>
```

That's all there is to it for publish-subscribe messaging, but in many cases, you have more stringent requirements to the relation between sender and receiver. One of such situations is when a single message may only be processed by a single consumer. This requirement is met by point-to-point messaging in the next subsection.

17.3.3 Point-to-point messaging and sharing the load

Point-to-point messaging is used when one endpoint specifically wants to tell another endpoint something. Because you have one receiver and one sender, the expectations between the two are stronger. Usually, the sender changes state and, in many cases, even expects the receiver to trigger another state change. For example, when you order a taxi, you start expecting a taxi to come pick you up. When it doesn't come, you're a bit disappointed.

When we talk about examples of point-to-point messaging, it's usually the sender using a service of the receiver. This is the exact opposite of publish-subscribe messaging in which you can subscribe to the feed of published messages. If two OSGi bundles are connected through point-to-point messaging, the startup is a bit tricky. You can connect bundles in two ways, depending on your purpose.

Sharing a channel makes sense, but you still have to decide which bundle gets to own the shared channel. This decision depends on how you want to design each bundle. Two main patterns are possible. Either put the channel with the sending bundle or put it on the receiving bundle.

Putting the channel on the sending bundle allows loading multiple receivers at runtime to listen to the same sender. Depending on the dispatcher of the channel, this serves as a lightweight load balancer in the application. This is the Scatter part of the Scatter-Gather pattern. If the channel sits with the receiving end, you can let multiple senders sink their messages into the same bundle. Thus, it's natural to call this pattern Gatherer.

Together these patterns can form a Scatter-Gather application. They have the additional benefit that you can scale out at runtime. Just add a bundle that does its work on a different (new) node in your cloud, and you're done. In the next section, we look at the boundaries of the messaging infrastructure and integrate with bundles that don't depend on Spring Integration at all.

17.3.4 Using Gateways and Service Activators to avoid Spring Integration dependencies

When you're composing a large application built by several teams, you're unlikely to want all these decoupled bundles to depend on a messaging framework, even one as awesome as Spring Integration. The whole point of modularizing an application is to reduce the complexity, so having the same dependency from all these modules contradicts this goal.

A large application using Spring Integration typically has one or several modules using Spring Integration to wire the rest of the modules together. In figure 17.7, you can see that the Spring Integration-dependent bundles use services in the non-Spring Integration bundles. This raises the question: How are these integrated in an OSGi architecture?

The integration depends on the direction of the dependency. If a message needs to result in a service activation in a non-Spring Integration enabled bundle, you can

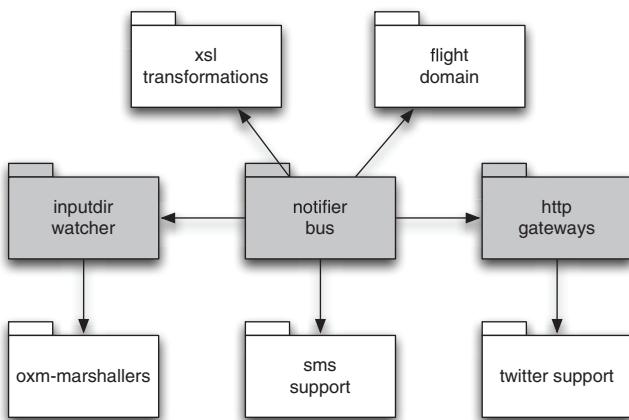


Figure 17.7 The green bundles here have a Spring Integration dependency, but the white ones do not.

put a Service Activator in the bundle that knows about Spring Integration and reference the service in the other bundle from it.

In the opposite direction, you can use a similarly elegant trick. Let's say the Twitter support bundle receives a direct message on Twitter from a user and the resulting notification object needs to be sent on a channel in the notification bus. You want to keep the Twitter support bundle free of Spring Integration dependencies, so you can't expose a channel as a service through the OSGi Service Registry.

Instead of a channel, you expose a gateway, implementing an interface from the Twitter support bundle. In the notifier bus, you now have the following:

```

<int:gateway id="twitterNotifier"
              default-request-channel="notifications"
              service-interface="twitter.TwitterNotifiable"/>
<osgi:service ref="twitterNotifier"
               interface="twitter.TwitterNotifiable"/>
  
```

This is complemented by a reference in the Twitter support bundle:

```

<osgi:reference id="twitterNotifier"
                 interface="twitter.TwitterNotifiable"/>
  
```

With this simple trick, you can use the `twitterNotifier` gateway in the notifier bus without any dependencies on classes in that bundle or in Spring Integration for that matter.

Now that you've understood how OSGi can enable teams to better modularize their software and in doing so scale their development organization, it's time to look at the issues of scalability at runtime. In the next section you'll be introduced to AMQP, which can be used to scale the messaging system itself.

17.4 Summary

This chapter introduced OSGi and showed you how it fits with a messaging application. We described some architectural patterns made possible through the use of

OSGi. We also showed you some of the more intricate details of loading a Spring Integration application that's spread over multiple bundles.

With OSGi, we focused mainly on development time scalability, and we showed you that OSGi and Spring Integration give you a lean alternative to an ESB.

Setting up a runtime that would make you comfortable working with OSGi is a bit of a job. You can't expect OSGi to magically make complex things happen without giving you some interesting puzzles to solve along the way. As with any technology, be conservative when deciding whether to use it, but if you have a use case on your hands that would benefit from OSGi, be brave: it's worth it.

18

Testing

This chapter covers

- Test-driven development in the context of messaging
- Hamcrest and Mockito matchers for messages
- Debugging asynchronous applications

One of the great accomplishments of our industry over the last 20 years is *test-driven development (TDD)*. Where many methodologies have proven only to work in theory or have never proven their need, TDD has flourished. The reason for this is simple: clients only pay willingly for working software, and there's only one way to prove that software works: test it. In essence, TDD makes the developer responsible for proving that the software works. A green test is the ultimate proof of correctness. If you don't have a clue how you're going to test the application, you don't have any business building it.

There are many ways to test software. One of the oldest ways is to test it manually. Manual testing is still valid and in wide use because of its simplicity, but experienced developers dread the tedious work of so-called monkey testing. A lot of this work can be automated. Even better, if you can isolate a part of the program in such a way that it doesn't require every test fixture to simulate human interaction, writing tests becomes a simple development task with excellent return on investment. SUnit, invented by some of the bright minds that also started the Agile

movement, was ported to Java in the late 1990s. If you don't know JUnit yet, firmly pull the handbrake toward you and make sure you learn JUnit before you read any further.

After reading the rest of the book and being exposed to test code there as well, you'll find little new here in terms of *what* you can do. The main thrust of this chapter is about *why* you should pick a certain option.

Because the topic of this chapter cuts across the other topics, this chapter is organized differently. Like the other chapters, it uses code samples from the sample application, but it doesn't focus on a particular use case. It also doesn't have a dedicated "Under the hood" section, first because the test framework code is simple enough to embed with its usage and also because the test code serves as an example of how to extend JUnit to deal with the messaging domain.

This chapter builds on top of JUnit tests from the sample to show you the intricate details of testing asynchronous and concurrent programs that were built using the pipes-and-filters architecture supported by Spring Integration. As you might've experienced, testing these types of applications is more convoluted than testing classical applications. When you're using Spring Integration, you'll find that it's often necessary to write tests that assert things about the payload of a message that's received from a channel or to write assertions about particular headers on such a message. The boilerplate code normally needed to do this is in large part taken care of by Spring Integration's own test framework. This test framework builds on top of Hamcrest to offer custom matchers that can be used with `assertThat`. It also has some convenience classes related to the asynchronous nature of certain Spring Integration components, such as `QueueChannel`.

In addition to Hamcrest matchers, Spring Integration tests can make use of Mockito extensions. Mocking services out of tests relating to the message flow through the system becomes more important as a system becomes more complex or when service implementations are developed on a different schedule than the configuration that makes up the message bus. This chapter discusses different use cases for the test module. The authors strongly believe that tests shouldn't hide complexity, so, as mentioned, we include no "Under the hood" section. Instead, you'll find all the details right with the usage examples.

Assertions with Hamcrest matchers

Since version 4.4, JUnit added Hamcrest support and in later versions also repackaged the Hamcrest framework. Hamcrest is a matching framework that allows a user to compare an object against a predefined matcher with a readable API. Hamcrest has a more generic use than just JUnit testing, but it's best known for its use in JUnit's `assertThat` method:

```
assertThat("Tango", is(not("Foxtrot")));
```

This makes both the test code and the thrown exceptions more readable.

Testing behavior with mocks

Mocking is used to allow assertions on behavior instead of state. Frameworks like EasyMock, JMock, and Mockito help create mocks that allow verification of behavior. Mockito is probably the simplest mocking framework around. Certain advanced features are not supported in Mockito, but that makes it an excellent candidate to use as illustration. If you're unfamiliar with mocking, you're encouraged to read "Mocks Aren't Stubs" by Martin Fowler (available at <http://mng.bz/mq95>).

To make use of all the test goodness, you should depend on `spring-integration-test` or `org.springframework.integration.test` depending on whether you're using OSGi. This jar is packaged separately from the main Spring Integration distribution, because we don't want to force transitive dependencies on Mockito and Hamcrest on all Spring Integration users.

```
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-test</artifactId>
    <version>${spring.integration.version}</version>
    <scope>test</scope> </dependency>
```

You've just added another Spring Integration jar on your classpath—now what? Let's look at what's inside that jar.

18.1 Matching Messages with the Spring Integration testing framework

What goes in must come out. When a message moves into the system, it must come out in some form or another, either through being consumed by an outbound channel adapter, as another message being sent on a subsequent channel, or as an `ErrorMessage` being sent to the `errorChannel`. In a test fixture, you're usually interested in the properties of the outgoing messages, but these properties might be hard to reach:

```
@Test
public void outputShouldContainDelayedFlight() {
    inputChannel.send(testMessage());
    Message output = outputChannel.receive();
    assertThat(((FlightDelayedEvent) output
        .getPayload()).getDelay(),
        is(expectedDelay));
}
```

As you can see here, getting to the delay requires a cast and two method invocations. Let's see if we can do better than that. In the next section, you'll see how to factor the unwrapping logic out of your test cases.

18.1.1 Unwrapping payloads

With Spring Integration's test module, you can use the matchers that deal with unwrapping internally. First we look at an example, then we look at the underlying details. Starting with the previous example, you probably already noticed some pain points in the test code. The `is` matcher isn't particularly well-suited to deal with messages.

Ideally, you'd have a matcher that can be used like this:

```
assertThat(outputChannel.receive(), hasPayload(expectedDelay));
```

It's no coincidence that with the `PayloadMatcher` you can do exactly this. All you need to do is add the following import statement:

```
import static org.s...i..test.matcher.PayloadMatcher.*;
```

This gives you two methods related to payloads: `hasPayload(T payload)` and its overloaded cousin accepting `Matcher<T>`. This way, you can also use other variants of the theme.

```
assertThat(outputChannel.receive(), hasPayload(expectedDelay));
assertThat(outputChannel.receive(), hasPayload(same(expectedDelay)));
assertThat(outputChannel.receive(), hasPayload(is(FlightDelay.class)));
```

This makes your life as a Spring Integration user a lot easier, and the code you need is almost trivial. Let's look at the code of the `PayloadMatcher` in the following listing.

Listing 18.1 The PayloadMatcher

```
public class PayloadMatcher extends TypeSafeMatcher<Message<?>> {
    private final Matcher<?> matcher;
    PayloadMatcher(Matcher<?> matcher) {
        super(); this.matcher = matcher;
    }
    public boolean matchesSafely(Message<?> message) {
        return matcher.matches(message.getPayload());
    }
    public void describeTo(Description description) {
        description.appendText("a Message with payload: ")
            .appendDescriptionOf(matcher);
    }
    @Factory
    public static <T> Matcher<Message<?>> hasPayload(T payload) {
        return new PayloadMatcher(IsEqual.equalTo(payload));
    }
    @Factory
    public static <T> Matcher<Message<?>> hasPayload(Matcher<T>
        payloadMatcher) {
        return new PayloadMatcher(payloadMatcher);
    }
}
```

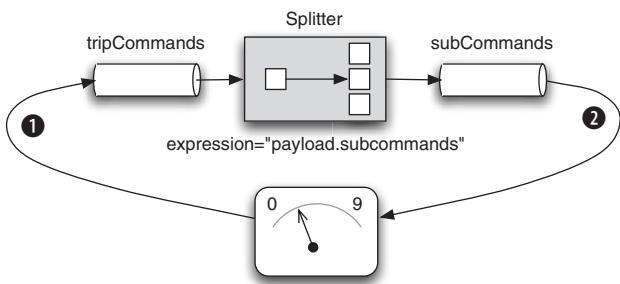


Figure 18.1 The test sends a message on the `tripCommands` channel and receives the subcommands that were sent by the splitter. Now a test can verify that the splitter is configured correctly by asserting that the payload of the messages match the contents of the original `TripCommand`.

As you can see, this listing extends `TypeSafeMatcher` and implements two factories for the matcher. If your only concern is to match payloads, you might even opt to add this class to your project and avoid the extra dependency on Spring Integration test. A few more features are bundled in Spring Integration's test module. For example, you might require matching on headers too, as we see in the next section.

Matching messages is particularly useful if the framework is doing work that's important for business concerns. In many cases, the message payload is determined by business logic in Java code, and asserting things about the payload doesn't make much sense in an integration test (because most of that would already be covered in a unit test around the service). In some cases, though, such as when you use expression language, things change. Let's take another example from the sample application.

In the sample application, a user can fill out a form creating a new trip, and from that a `CreateTripCommand` is sent into the system wrapped in a message. The message goes through a splitter that chops the command into subcommands for rental cars, hotel rooms, and flights. Let's zoom in on the tests for the splitter. In figure 18.1 you can see how we modified the fixture to allow us to control incoming and outgoing messages.

We can now make sure the expression is correct in a controlled test case. All we need to do is receive three messages from the `subCommands` channel and assert that their payloads meet our expectations.

Our test case remains relatively simple as you can see in the following code:

```
@Autowired
MessageChannel tripCommands;

@Autowired
PollableChannel javaLegQuoteCommands;

@Test
public void splitterShouldSplitIntoSubcommands() {
    CreateTripCommand tripCommand = mock(CreateTripCommand.class);
    Message<CreateTripCommand> tripCommandMessage =
        MessageBuilder.withPayload(tripCommand).build();
    final Command carCommand = mock(Command.class);
    final Command flightCommand = mock(Command.class);
    final Command hotelCommand = mock(Command.class);
    given(tripCommand.getSubCommands()).willReturn(
        Arrays.asList(carCommand, flightCommand, hotelCommand));
```

```

tripCommands.send(tripCommandMessage);
List<Message<? extends Object>> received =
    Arrays.asList(javaLegQuoteCommands.receive(100),
                  javaLegQuoteCommands.receive(100),
                  javaLegQuoteCommands.receive(100));
assertThat(received.size(), is(3));
}

```

The trick here is to plug into the existing system without replacing logic that you want to test. In this case, the `subCommands` channel is overridden by a queue channel, and no other components are receiving from it. As you can see, it's simple and useful to write tests that make assertions on the payload of a message. More often than not, though, the headers of messages play at least as big a role in the integration of the system. In the next section, we go into the details of header matching.

18.1.2 Expectations on headers

In many cases, when you're testing the integrated application, it's more important to make assertions about the infrastructural effects on messages than on the business services effects on messages. Typically, the effects of business services are already covered by a unit test, so you don't need to cover all the corner cases in your integration test again. But headers on messages are typically set by components that are decoupled from services and can only do their work in an integrated context. For headers set in this manner, you need to test all the corner cases in an integration test.

Let's look at the booking of a flight again. From the UI, a command describing the desired booking comes in. This command is consumed by the booking service, which puts an event on the bus that signals the result of the booking (success or failure). To guarantee idempotence, the service activator for the booking service is preceded by a header-enricher that stores a reference to the original command in the headers and a filter that drops any message containing a command that has already been executed. It's followed by a service activator that keeps track of all the successfully executed commands, for example, in a table that's also used by the filter.

This construction contains enough complexity and business value to make it the target of a test, but testing all these components in isolation doesn't assert anything about what Spring Integration will do with the header values. You need to make assertions about headers, which you could do manually.

```

@Test
public void outputHasOriginalCommandHeader() {
    //when
    inputChannel.send(testMessage());
    Message output = outputChannel.receive();
    //verify
    assertThat(
        ((BookFlightCommand) output.getHeaders()).get("command")
        , is(expectedCommand)
    );
}

```

But similar to matching payloads, matching headers manually causes smelly code. Again, there are matching facilities in Spring Integration's test module that can help you. The implementation is similar to the `PayloadMatcher`, so we only need to look at the usage here.

This section established a solid foundation in terms of matching messages based on payloads and headers. Regarding the matchers, it isn't trivial to deal with the fact that a message received from a channel doesn't have the benefit of generics in many cases. If you choose to implement your own matchers, you should expect to invest some of your time in fine-tuning parametrization.

Matching the message state is only part of the equation. You should also verify that service activators, transformers, and channel adapters invoke services correctly. For this, you can use mocks. When you're using a mocking framework, things get more complicated because you must deal with the particulars of the mocking framework as well. We outline the support for Mockito in the next section.

18.2 Mocking services out of integration tests

When your test subject has a dependency that isn't relevant for your test, you can use mocks or stubs to factor their influence out of the test fixture. People often refer to such refactoring as *mocking out*. A briefing on mocking is beyond the scope of this chapter, but we keep a strict definition of a mock as something that can be used to test behavior (and is usually created by a mocking framework), as opposed to a stub, which is typically used to test state and is created as an inner class in a test case.

In this chapter, we show only mocks using Mockito, which serves our need for concise and readable code samples. Other mocking frameworks or stubs can be used in the same manner; the particulars of Mockito are irrelevant to the point being made.

Most unit tests require a test harness that simulates the external dependencies (for example, through mocking or stubbing). But if configuration becomes a major part of the behavior of your application, as with Spring Integration, it becomes important to test the configuration itself. This means that it becomes sensible to mock out business code and let a message flow through the system just to see if it's handled correctly by the infrastructure. This would concern routing, filtering, header enrichment, and interaction with other systems. For example, you might want to pick up a file from a certain directory, set its name as a header value, then unzip the file and unmarshall it to domain objects. All this can be considered infrastructure—*customized infrastructure* if you will.

Customized infrastructure is usually important to the business without being tightly related to a particular business use case. For example, properly setting a header is essential for your system to perform its tasks as designed, but setting this header is only a small part of the story. The particular header-enricher has a place as a unit in the system, so it should have a designated unit test. If you're using SpEL, you have only XML configuration to test.

Let's say you have a header-enricher that sets the original command as a header so it can be used later in the chain when the payload is already referencing the response.

```
<header-enricher>
    <header name="originatingCommand" expression="payload" />
</header-enricher>
```

Even though the expression is trivially simple, this needs to be tested thoroughly. For example, a change that postpones the unmarshalling to a `BookingCommand` could cause a regression where the `originatingCommand` header suddenly references a `Document` instead.

In figure 18.2 you can see how to generically set up a test that verifies the framework behavior. The example chosen is a service activator that invokes a method on a mock. Arguably, this setup would make sense only as an integration test for the framework, but it serves as a simple example. As the complexity of your configuration increases, it becomes increasingly useful to verify the flow of the message through the system.

In a test like this, you're not interested in the behavior and effects of the service that receives the objects as message payloads. In fact, a failure in that service might distract you from the purpose of the test. It therefore makes sense to replace the service with a mock, but because this service is wired as a bean in a Spring context, it isn't as easy as injecting a component with mocked collaborators, as you would do in a normal unit test. But there's a trick you can use:

```
<bean id="service"
    factory-method="mock"
    class="org.mockito.Mockito">
    <constructor-arg
        value="com.you.ServiceToMock" />
</bean>
```

This code overrides the service bean with a bean that's a mock created by Mockito. This bean, `@Autowired` into your test case, can be used like any other mock with the only difference being that its lifecycle will be managed by Spring instead of JUnit directly.

This strategy is particularly useful to avoid calling services that operate on external systems. Invoking an external system is more problematic to clean up, but it's also more complicated to verify the invocation happened correctly. If you use a mock, you can simply verify that it was touched, and that's it. This is a good option for channel

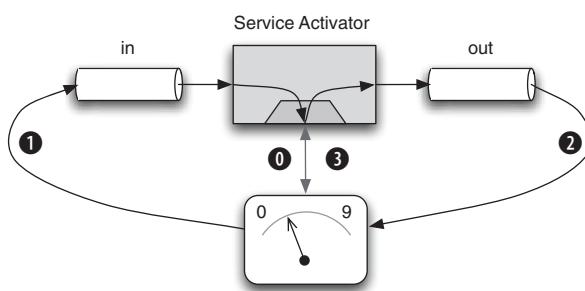


Figure 18.2 First record the behavior of the mock ①. Then send a test message to the input channel ②. After waiting for the message to come out of the other end ③, verify that the appropriate operations on the mock have been invoked ④. Variations on this recipe work also in more complex cases.

adapters too, because it gives you a generic way to deal with them. In section 18.3, we use this strategy as well to deal with the need to wait for an invocation to happen before we start asserting the result.

This section focused only on Mockito, but similar support for EasyMock, JMock, and RMock can be implemented along the same lines. It's unlikely that Spring Integration will natively support all of these frameworks in the near future. After reading this chapter, you should have some idea how to implement the test support of your choosing, and chances are good that someone out there has shared some matcher you might reuse.

The next section dips into the realm of concurrency. We already secretly used some concurrency features of Spring Integration to our advantage in tests, but now it's time to explore the different concurrency strategies. The combination of mocking and latching is especially powerful, so stay tuned!

18.3 Testing an asynchronous system

One of the trickiest things to solve cleanly in tests is assertions around related actions performed by multiple threads. A big advantage of staged event-driven architecture (SEDA; see chapter 1) is that components become passive and react to events rather than actively changing the world around them. This opens the door to decoupling cause and effect using a framework rather than having the complexities of asynchronous handoff emerge in business code. At runtime, though, these subtleties are essential to the proper functioning of the system. Therefore, they must be accounted for in tests. This section focuses on the concerns around testing an asynchronous system. An asynchronous system is a system in which multiple threads are involved in performing a bit of work (such as processing a message).

If a part of your system is designed to process messages asynchronously, you should keep an eye on certain things. As you saw in chapter 3, processing messages asynchronously can be done in several different ways. You can use a `<queue ... />` element or a `<dispatcher ... />` element. Also, when you use a publish-subscribe channel configured with a task executor, you're using asynchronous handoff. Finally, there are a few endpoints that can be configured with a `TaskExecutor` that will process a message in a different thread than the thread pushing the message in.

To give you a handle on this, remember that whenever an endpoint or channel is using storage or a task executor, it can cause asynchronous handoff.

Whenever asynchronous handoff is involved, *there are no chronological guarantees without explicit locking*. Luckily, getting explicit locking in place is simple in Spring Integration, but if you're not familiar with what happens under the hood, you can be sucked into hours of fruitless debugging.

18.3.1 Can't we wait for the message to come out the other end?

You sure can! In most cases, that's precisely what you should do. We look at a few exceptions later, but in the vast majority of cases, plugging into the output channel of

your context and just waiting for the output message to arrive is sufficient. How do you go about it?

You might be expecting to see a loop with `Thread.sleep(..)` in there, or if you're more familiar with Java's concurrency support, you might've expected a `CountDownLatch`. Things are even simpler than that.

For a standard test case, you just follow this simple recipe: make sure your output goes to a `QueueChannel` and receive from this channel before doing any assertions.

```
@Test
public void shouldInvokeService() {
    given(service.invoke(payload))
        .willReturn(newPayload);

    in.send(testMessage);
    Message m = out.receive(100);

    verify(service).invoke(payload);
    assertNotNull("Output didn't arrive!", m);
    assertThat(m, hasPayload(newPayload));
}
```

As you can see, you receive `m` from the output channel *before* you assert that the service has been invoked. Also, you use a timeout in the `receive` call to ensure the test doesn't run indefinitely. The assertions are done in the order that you expect them to succeed.

You're piggybacking on the contract of the `receive` method here. Because `receive` is a blocking call, you don't have to do any additional waiting to ensure a *happens-before* relationship between the service invocation and the verification. However complex your contexts get, it's almost always possible to find some output that will arrive only after the behavior you're trying to verify has been executed.

The timeout is also important. When designing a test case, you must understand that the main function of the test is to fail when the software doesn't behave correctly. This main function is best implemented when the failure clearly points out what part of the behavior was incorrect. If the `receive` call had no timeout, and an exception were thrown from the service, the test could run indefinitely. The test wouldn't fail in this case of malfunctioning in the software under test, so the test would be flawed. You could put a general timeout on the test to prevent it from running indefinitely. The timeout would fix the flaw, but the failure message would have no relation to the cause of the failure. The test would be correct but not very useful.

Finally, the assertions should be in the right order. If the service isn't invoked, you'll most likely get no output message. In this case, mixing the order of the assertions will make the test fail with an "Output didn't arrive" message, or worse, a `NullPointerException`. Thinking carefully about the order of the assertions can prevent this problem.

Before we look at exceptional cases in which waiting for output isn't an option or isn't sufficient to prove that the system functions correctly, we examine the need for proper test cases in asynchronous scenarios a bit further.

18.3.2 Avoiding the wicked ways of debugging

Before you get the wrong idea, let's make it clear that debugging is a skill that all excellent developers have and all novice developers should strive to learn. It's also the mother of all time wasters.

To paraphrase an old saying: "Debug a program and you fix it for a day; improve the tests and the logging of a program, and fix it for its lifetime." Before you dive into hours of debugging, you should ask yourself, What is this test not telling me that I need to know? The answer is often right in front of you, hard to reach with a debugger, easy to log. Once you have a test suite and some decent logging around the problem, another developer can continue where you left off. Better yet, in the unthinkable scenario that the problem happens in production, you can ask the system administrator for the log file.

The reason we bring this up is because debugging is much harder in a concurrent scenario than in a single-threaded scenario. Spring Integration is inherently a concurrent framework, and if you have a concurrent bug in your code, it might be exposed by wiring your service in a Spring Integration context. In single-threaded scenarios, debugging is great. It helps you understand the code more quickly than just reading through it would. In many cases, you don't want to fix all the logging in your program; you just want to see what's going on. That's fine, usually. But if multiple threads are entering the problem area of your code, debugging loses all its power. Suddenly, the debugger changes all the timelines and often completely hides the bug from your sight. You could say that concurrency is debugger kryptonite.

Luckily, logging and test cases are much more reliable even when dealing with concurrent access. That's not to say that finding and analyzing a concurrency issue is easy. It's merely possible, and that's just about good enough. So heed this advice: especially when facing a concurrency bug, try to avoid the debugger and fix the problem with tests and logging. Now that you've learned to prefer test and logging over the debugger in concurrent scenarios, you're ready to learn how to wait for messages to terminate inside an endpoint using latches and mocks.

18.3.3 Injecting latches into endpoints

Sometimes an endpoint has no output. As you read in chapter 4, these types of endpoints are called *channel adapters*. A channel adapter takes the payload of a message and feeds it to a service. This service may be a bean in your context, but it might also be the database, a web service, the filesystem, or standard output.

The tricky part is to wait for the invocation of this service before you start making assertions about the state of the system. In this section, we show you how you can inject latches into endpoints that have been mocked with Mockito. Similar strategies exist for other mocking frameworks, and the problem can also be solved with channel interceptors or AOP. Going over all these options is beyond the scope of this book, but this section should be enough to spark your imagination.

It's time to look back to our example. In figure 18.2, we showed how to mock out services from tests. That example required no latching inside the mock because you could just wait for the message to come out of the output channel, as discussed in the previous section. Let's explore an endpoint that's different in that respect.

When notifications are sent to the user, they're sent over an asynchronous communication channel. You specifically don't want to wait for the external system to confirm something was sent synchronously. That would block too many threads. Looking at the email outbound channel adapter, for instance, you need to confirm a message reaches this adapter, but you don't need to test the sending of the email in this test. There should be another test for sending, but that's beyond the scope of this book.

Let's say you want to test the following snippet:

```
<i:publish-subscribe-channel id="tripNotifications"
    datatype="com.m..siai.i..notifications.TripNotification"
    task-executor="taskScheduler"/>

<i:outbound-channel-adapter id="smsNotifier"
    channel="tripNotifications" ref="smsNotifierBean" method="notify"/>
```

You design your test to verify that a notification is passed into the `smsNotifierBean`'s `notify` method whenever a message containing the same notifier is sent to the `trip-notifications` channel.

First you must make sure you replace the `smsNotifierBean` with a mock. You can use the same trick mentioned earlier:

```
<bean id="smsNotifierBean" class="org.mockito.Mockito"
    factory-method="mock">
    <constructor-arg
        value="com.m..siai.i..notifications.SmsNotifiable"/>
</bean>
```

Once that job is done, you can focus on the test itself.

The test sends a test message containing the test notification to the channel. It then verifies that the method was invoked, but a simple verify call doesn't work here because the channel is a `QueueChannel`. You have to wait for the message to arrive before you can verify. This can be done using a latch injected into the mock.

```
private Answer countsDownLatch(final CountDownLatch notifierInvoked) {
    return new Answer() {
        @Override
        public Object answer(InvocationOnMock invocationOnMock)
            throws Throwable {
            notifierInvoked.countDown();
            return null;
        }
    };
}
```

With the answer returned by this method, you can tell Mockito to count down the latch passed in whenever a certain method is invoked. Let's go over the usage.

Then the JUnit test becomes:

```
@Autowired
MessageChannel tripNotifications;

@Autowired
SmsNotifiable smsNotifier;

@Test
public void notificationShouldArriveAtSmsAdapter() throws Exception {
    TripNotification notification = mock(TripNotification.class);
    Message tripNotificationMessage =
        MessageBuilder.withPayload(notification)
                      .build();
    CountDownLatch notifierInvoked = new CountDownLatch(1);
    doAnswer(countsDownLatch(notifierInvoked))
        .when(smsNotifier).notify(notification);
    tripNotifications.send(tripNotificationMessage);
    notifierInvoked.await(100, MILLISECONDS);
    verify(smsNotifier).notify(notification);
}
```

Because the `notify` method returns void, you use Mockito's `doAnswer` method to record the behavior. You're essentially tell Mockito: "When the `notify` method is invoked on `smsNotifier`, react by counting down the `notifierInvoked` latch." Then it's a matter of awaiting the latch so you can execute assertions under the safe assumption that they'll happen after the message arrives at the endpoint.

Before we round up, we should give you some guidelines for making your applications easier to test. This isn't an easy thing, but it's a skill worth honing.

18.3.4 Structuring the configuration to facilitate testing

We can't overemphasize that changing the application to improve testability is a good thing. In Spring Integration applications, you usually see good decoupled code that's easy to test. But what about the configuration? With all that XML containing all those little SpEL expressions and intricate dependencies, you could easily get lost.

It's said that programming in XML is a bad thing (which it is). It's possible to do programming in XML with Spring. It's also possible to write really bad code that way. This section offers some pointers to help you spot problems in this area and combat them with your test goggles on.

AVOID LOGIC IN XML

You can do complex things with Spring and Spring Integration, particularly using SpEL and complex routing. Don't! It might seem powerful, even simple at first, but testing logic that's embedded in XML is tough to the point of headache.

Instead, design your flows in linear steps as much as you can. If you want to use SpEL, delegate to Java code for the real decisions. It's fine to invoke methods on other objects from Java directly: not everything needs to be a service activator.

SPLIT THE MAIN FLOW INTO SUBFLOWS

As your application gets larger, the configuration files grow too. At some point, it becomes hard to find that part of the configuration that you need to change. Take out the detailed flows and integrate them using import statements.

If you're used to Spring, you might've put configuration related to data access in a separate file, or you might've created several servlet contexts using the same root context. With a messaging application, splitting in layers isn't a good fit. It's better to divide the flow into different phases and give each phase its own context.

One way to make the subcomponents more testable is to define input and output channels in each subcontext and use bridges to glue them together in the main context. This way, the test around a subcontext can easily use the same concept to wire the input and output to test specific channels.

In the next section, we take a brief glimpse into the realm of threading.

18.3.5 How do I prove my code thread-safe?

The short answer is that you don't. You can prove the correctness of your code under concurrent access, but it's usually unfeasible to test all possible concurrent scenarios and make sure they meet the specifications. But there are a few things you can do to help ensure your code is thread safe.

We don't go into great detail here, because concurrency is already discussed in detail in chapter 15. Just repeat to yourself: *Pass immutable objects between stateless services*.

Where testing is concerned, you can do your best to make sure concurrency bugs have a chance to surface in your test. For one, you should use *at least* the same number of threads in some of your integration tests as is used in your production application. This ensures that the code is at least run concurrently in your continuous integration build. Some failures will still be unlikely to occur in a test, so this is by no means foolproof. Tests written this way might cause intermittent failures, which in many cases means you have a concurrency bug in your code.

This is where we are reaching the boundaries of the Spring Integration framework. Spring Integration is good at running things concurrently, but it doesn't boast a full concurrency test suite.

Concurrency bugs are best tackled by logging and testing, but they can be a huge pain to reproduce. Some frameworks, such as *ConcuTest*, are helpful in provoking concurrency bugs by injecting yields and waits into your bytecode. If you learn these tools, you'll have a better chance of resolving concurrency bugs.

18.4 Summary

In this chapter, we formalized our understanding of testing Spring Integration applications. First we discussed the test support in Spring Integration's own test framework. Then we detailed the strategies and rationale for mocking out external dependencies and business services from message flow tests. Finally, we discussed testing asynchronous applications on a broader level and showed you how to enforce chronological order in tests with QueueChannels or mocks and latches. We also discussed thread safety.

Within the scope of the test framework, you saw different ways of matching messages, either by their headers or by their payloads. Matching payloads is helpful when you want to avoid unwrapping messages and casting their payloads to a type of your choosing. We discussed support for unwrapping headers and gave you pointers for dealing with the whole map of headers. We also discussed the Mockito support for matching messages.

We made a case for mocking business services out of tests. Because Spring Integration configurations contain a code that's dynamically used at runtime, it becomes important to test this configuration in relative isolation too. Mocking away external dependencies is an excellent way to achieve this goal.

Finally, we went into the details of testing the full asynchronous message flow. We explained how to use a blocking receive call to ensure chronological order in tests. Also we explained that when this doesn't work, you need to use latches within mocks to enforce happens-before relationships.

This is the last chapter, but that doesn't mean it's least important. A proper understanding of how to test your application is both the end and the beginning of craftsmanship in software engineering.