MANNING

Spring Integration IN ACTION

MEAP

Mark Fisher
Jonas Partner
Marius Bogoevici
Iwein Fuld

MEAP Edition
Manning Early Access Program
Spring Integration in Action MEAP version 6

Copyright 2011 Manning Publications

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table of Contents

Part 1 - Background

1: Introduction to Spring Integration
2: Enterprise Integration Fundamentals

Part 2 - Messaging

3: Messages and Channels
4: Message Endpoints
5: Getting Down to Business
6: Go beyond sequential processing: routing and filtering

Part 3 - Integrating Systems

7: Splitting and Aggregating messages
8: Handling Messages with XML Payloads
9: Spring Integration and the Java Message Service
10: Sending and Receiving Mail
11: File System Integration
12: Web Services

Part 4 - Advanced Topics

13: Monitoring and Management
14: Managing scheduling and concurrency
15: Spring Integration and the rich web
16: Messaging and batch operations
17: Scaling Messaging applications with OSGItm and AMQP
18: Putting your pipes to the test

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

1 Introduction to Spring Integration

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, technical "progress" seems to drive a steady increase to the number and types of events we are expected to handle. In today's world of hyper-connectivity, it's a wonder that we are ever able to focus and actually get any real work done.

What saves us from such event-driven paralysis is the fact that we can respond to most events and messages at a time that is convenient for us. For example, we can let an incoming call go straight to voice mail. Often we'll do so after first checking the "Caller ID" to gauge the relative urgency of the call. Sometimes we might feel bad ignoring calls, but imagine what life would be like if we were required to respond to every call, email, or message as soon as it was received. We probably would fail to accomplish any significant task due to the constant disruptions. Luckily, the events usually carry enough information for us to make that judgment quickly.

Now, let's turn our attention to software. When we design and build a software application, we strive to provide a foundation that is an accurate model of 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

does not 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 login and review the details. In other words, even though it may be described as a "service", it's likely implemented as an event-driven system. There are simply too many different "services" 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 is often useful to consider the events and messages that occur within the domain at hand. That may lead to a more natural way to think about the problem than being excessively "service-oriented".

As a result of all this, it is increasingly common that we, as enterprise developers, have to 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 are replacements for scheduled back office processes. Sure there are still those nightly batch processing systems that grab a file and process it shortly after midnight, but it is more and more common that we encounter requirements to refactor those systems to be more timely. Perhaps there is a new Service-Level Agreement (SLA) establishing that files must be processed within an hour of their arrival, or maybe the nightly batch option is now insufficient due to 24x7 availability and globalized clientele. These are after all the motivating factors behind such hyped, if oxymoronic, phrases as "near real time". What that phrase usually suggests is 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. Oh, and by the way, those "legacy" systems won't be completely phased out for approximately another 10 years, so we need to support all of the above. That means we also need to be sure that the refactoring process can be done in an incremental fashion.

At this point, you're probably thinking to yourself, "Well, this book sure is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

getting off on a negative note", but that is most definitely not our intention. In fact, we believe that these are very exciting times for developers, and that with the right tools and frameworks, developing these solutions can actually be a lot of fun. It seems that most developers have some degree of fascination with building message-driven applications that take advantage of the high degree of connectivity available today. Yet due to the complexity of building such applications, they can quickly lead to a love-hate relationship with the technologies involved. The projects may be a lot of fun in the prototype stage, and if successful when they go to production, these are the types of applications that can quite literally change the way a company does business. The problem is that in that middle phase, you know the one where 90% of the effort goes, there are tedious tasks and complex challenges.

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, non-invasive, and declarative model for constructing message-driven applications. On top of this, it includes a toolbox of commonly required integration components and adapters.

Spring Integration itself 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 that is powerful because of its simplicity. Second is the book entitled Enterprise Integration Patterns (Hohpe and Woolf, 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 groundbreaking 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 will see that there is a natural synergy 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. So let's begin by zooming out to see what solutions Spring Integration supports at a very high level.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

1.1 What is Spring Integration?

From the 30,000 foot view, Spring Integration consists of two parts. At its core, it is a messaging framework that supports lightweight, event-driven interactions within an application. Then, 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 Figure 1.1 Two areas of focus for Spring Integration: lightweight intra-application messaging and flexible inter-application integration.

Everything depicted within the "Core Messaging" area above would exist within the scope of a single Spring application context. Those components would be exchanging messages in a very lightweight manner since they are running within the same instance of a Java virtual machine (JVM). There is no need to worry about serialization, and unless necessary for a particular component, the message content does not need to be represented in XML. Instead, most messages will contain simple POJO instances as their payloads.

The "Application Integration" area is a bit 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 that is implemented depends on the particular adapter, but Spring Integration provides a consistent model that is very easy to extend. The Spring Integration 2.0 distribution includes support for the following relatively mainstream adapters:

- File system
- UDP
- TCP
- HTTP
- Web Services
- Mail (POP3 or IMAP for receiving, SMTP for sending)
- Java Message Service (JMS)
- Java Database Connectivity (JDBC)
- Remote Method Invocation (RMI)

There is also a dedicated project within the Spring Extensions for adapters that are not considered sufficiently mainstream to be contained within the core and for

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

adapters whose development is community-led. At the time of this writing, the Spring Extensions for Spring Integration Adapters project includes FTP [list others here].

While the diagram above obviously lacks detail, it captures the core architecture of Spring Integration surprisingly well. There are 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].

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

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

There you see that it is 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. While those of us using the Unix pipes and filters model on a day-to-day basis may take it for granted, it actually 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 are not 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.

It just so happens that these same characteristics also describe the foundation of a well-designed messaging architecture. In fact, the Enterprise Integration Patterns book 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 EIP. In other words, if it were representing an actual Spring Integration application, the boxes in the diagram above could be labeled with the names of those patterns to depict the actual roles being performed. All of this makes sense of course 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.

1.2 Enterprise Integration Patterns

Enterprise Integration Patterns [EIP], the book, describes the patterns that are used in the exchange of messages and the patterns that provide the glue between applications. Like our diagram above, it is about Messaging and Integration in the broadest sense, and like the diagram above, 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.

Messages, Channels, and Endpoints

From the most general perspective, there are only three base patterns that make up EIP: Message, Message Channel, and Message Endpoint. 1.2 shows how these components interact with each other in a typical integration application

Figure 1.2 Figure 1.2 A Message is passed through a Channel from one Endpoint to another Endpoint

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 trigger another endpoint to do another bit of work. Messages can contain information in any format that is 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.

The Message Channel is the connection between multiple endpoints. The channel implementation manages the details of how and where a message is delivered but should not need to interact with the payload of a message. While the most important characteristic of any channel is that it *logically* decouples producers from consumers, there are a number of *practical* implementation

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 based on 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 above, 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.

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 will discuss each of those later in this section.

From these three base patterns there are two main ways to differentiate. First there are more specific subtypes of each of these patterns and second there are composite patterns. In this section we will focus on the subtypes so that you have a clear understanding of the building blocks. Composite patterns will be introduced as needed throughout the book.

Messages

Figure 1.3 Figure 1.3 Message

Each Message consists of a header and a payload. The header contains data that is relevant to the messaging system, like the Return Address or Correlation ID. The payload contains the actual data that is 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.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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.

Message Channels

Two endpoints can only exchange messages if they are connected through a channel. The actual details of the delivery process depend on the type of channel being used. We will 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 in between them. Additional components may be needed to connect services that are completely unaware of messaging to the channels. We will 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 EIP, so we will describe it here.

Figure 1.4 Figure 1.4: A Point-to-Point Channel

In point-to-point messaging, each single message that is sent by a producer is received by exactly one consumer. This is conceptually equivalent to a post card 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 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 is configured to fallback to a mobile when nobody is home to answer it.

As these cases imply, which single exact consumer receives the message is not 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, however, since the main

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

characteristic of that pattern is that it offers a consumer-driven alternative to load balancing. When a consumer cannot handle any more load, it simply won't be competing for another message.

Figure 1.5 Figure 1.5 A Publish/Subscribe Channel

Unlike point-to-point messaging, a Publish Subscribe Channel 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 is not informed about message delivery or failure to the same extent as in point-to-point configurations. In publish subscribe scenarios there is often a need for failure handling patterns like Idempotent Receiver or Compensating Transactions.

1.2.1 Message Endpoints

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

CHANNEL ADAPTER

Figure 1.6 Figure 1.6 Channel Adapter

A Channel Adapter connects an application to the messaging system. In Spring Integration we have chosen 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 will go into the details of these different types in the appropriate chapters on different transports later. At this time it is sufficient to remember that a channel adapter is needed at the beginning and the end of a unidirectional message flow.

MESSAGING GATEWAY

Figure 1.7 Figure 1.7 Messaging Gateway

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

In Spring Integration, a Messaging Gateway is a connection that is 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 response channel. For example outbound gateways can be used for invoking Web Services and for synchronous request-reply interactions over JMS.

A gateway can also be used mid-stream in a unidirectional message flow. As with the Channel Adapter we have constrained the definition of Messaging Gateway a bit in comparison to the EIP book.

Figure 1.8 Figure 1.8 Messaging Gateway and Channel Adapters

SERVICE ACTIVATOR

Figure 1.9 Figure 1.9 Service Activator

A Service Activator 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 is being invoked is defined on an object that is referenced within the same Spring ApplicationContext.

ROUTER

Figure 1.10 Figure 1.10 Router

A router will determine 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 does not change anything in the message and is aware of channels. Because of that it is the endpoint that is typically closest to the infrastructure and furthest removed from the business concerns.

SPLITTER

Figure 1.11 Figure 1.11 Splitter

A splitter will receive one message and split it into multiple messages that are sent

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

Figure 1.12 Figure 1.12 Aggregator

An aggregator waits for a group of messages that are correlated, and it will merge them together when the group is complete. The correlation of the messages will typically be 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 gateway.

You will see many more patterns through the course of the book, but what we have covered above should be sufficient for this general introduction. If you were paying close attention while reading the first paragraph in this section, you may have noticed that we said Spring Integration *supports* the enterprise integration patterns rather than saying that it *implements* the patterns. That is a subtle but important distinction. In general, software patterns describe proven solutions to common problems. They should not be treated as recipes. In reality, patterns rarely have a one-to-one mapping to a single implementation, and there are often context-dependent factors that lead to particular implementation details.

As far as the enterprise integration patterns are concerned, some actually are more or less "implemented", such as Message and Message Channel. Others are only partially implemented because they require the addition of some domain-specific logic, such as a Content Based Router where the content is dependent on the domain model or a Service Activator where the service to be activated is likewise part of a specific domain. Yet other patterns describe individual parts of a larger process, such as the "Correlation ID" we mentioned when describing Splitter and Aggregator or the "Return Address" that we will discuss later. Finally, there are patterns that simply describe a general style as we saw earlier with "Pipes and Filters". 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

1.3 EIP meets IoC

Now that we have 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) will be central to this investigation since it is a significant part of the Spring philosophy[3]. For the purpose of this discussion, we will be considering Inversion of Control in very broad terms.

The main idea behind Inversion of Control, 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 cannot be wholly ignored, but ideally they should not 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 is not easy, but it can be much easier when a framework handles much of the grunt work. With that as our definition, we'll discuss two key techniques that invert control when using Spring: dependency injection and method invocation. We will also see, albeit briefly, how each of these plays a role in the Spring Integration framework.

Dependency Injection

Dependency Injection is the first thing most people think of when they hear "Inversion of Control", and that's understandable since it is probably the most common application of the principle and the core functionality of Inversion of Control frameworks like Spring. There are entire books written on this one subject [4], but here we will provide just a quick overview to highlight the benefits of this technique and to see how it applies to Spring Integration.

Object Oriented software is all about modularity. When we design applications, we 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 only make assumptions about such an interface rather than 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. However, it all breaks down as soon as we do something as seemingly harmless as the following:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

Now the code is tightly coupled directly to an implementation type. Sure that implementation is being assigned to an interface, and hopefully the caller is never required to downcast. However, no matter how you slice it, the code is tied to an implementation, and that is a result of a very simple fact: 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 calling a constructor.

There is another option, and it often follows as a seemingly logical conclusion to the problem above. 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 actually the first step down the road of Inversion of Control. The factory is handling a responsibility that was previously handled directly in the caller's code. The control has been inverted in favor of a Factory. The caller gladly relinquishes that control in return for having to make fewer assumptions about the implementation it is using. This seems to solve the problems at first, but to some degree it's just pushing the problem one step further away. It is the programmatic equivalent of sweeping dirt under the rug.

A better solution would remove all custom infrastructure code, like the factory above. That final step to the full inversion of control is to simply define a constructor or setter method. In effect, that declares the dependency without any assumptions about who is 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 is trivial to provide a stub or mock implementation of the dependency directly. Then, in a fully integrated system where there may be many components sharing dependencies as well as

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 above, there is now a very 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 have been defined so that you can declare a namespace[6] 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 is an example of a simple Message Splitter.

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

Another common case for dependency injection is when a particular implementation of a strategy interface[7] needs to be wired into the component that delegates to that strategy. For example, here is a Message Aggregator with a custom strategy for determining when the processed items received qualify as a complete order.

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

Don't worry about understanding the details of the above examples yet. There will be much more coverage of these components throughout the book. The only point we are trying to make so far is that dependency injection plays a role in connecting the collaborating components while avoiding hard-coded references.

Method Invocation

Inversion of Control is often described as following the Hollywood Principle which states "Don't call us, we'll call you"[8]. From the description of dependency

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

injection above, you can see how well this does apply. Rather than writing code that calls a constructor or even a factory method, we can rely on the framework to provide that dependency by calling our constructor or setter methods for us. 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 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 within 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 just a simple object that is itself managed by Spring. There is even a dedicated XML namespace.

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

The container will manage the actual subscription and the background processes that are ultimately responsible for receiving the Messages. There are a number of configuration options for controlling the number of concurrent consumers, managing transactions, and more. We're not going to cover those details here since we just want to provide the background context for Spring Integration's similar features.

In the above example, the listener could be any implementation of the JMS MessageListener interface:

```
public interface MessageListener {

    void onMessage(Message message);

}
```

Where it gets even more interesting and more relevant for our lead up to Spring

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Integration is when we look at Spring's support for invoking methods on any Spring-managed object. Sure, the MessageListener interface shown above 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 that we have achieved by relying on the inversion of control principle. Second, and more severe, it has a void return type. That means that we cannot send a reply Message from the listener method's implementation. Both of these limitations are eliminated if we instead reference a POJO instance that does not implement MessageListener and add the 'method' attribute to the configuration. For example, let's assume we 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="currentQuot
</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 will use 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 Inversion of Control since the listener-container is handling the background processes. It's also a good example of the Hollywood Principle since 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 based on a configured interval. Java itself 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 Quartz[9] to support scheduling based on cron expressions, persistence of job data, and more.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Interacting with any of these schedulers normally requires code that is 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 scheduled it in Java like this:

Listing 1.1 Scheduling a Task programmatically Runnable task = new Runnable() { public void run() { filePoller.poll(); if (file != null) { fileProcessor.process(file); } } }; 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 do not 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:

Listing 1.2 Scheduling Tasks declaratively with Spring <task:scheduled-tasks scheduler="mySchedule <task:scheduled ref="filePoller" method="poll" fixed-rate="60000"/> </task:scheduled-tasks> <task:sc id="mySchedule" pool-size="5"/>

As you can see, this provides a quite literal example of the Hollywood Principle. The framework is now calling the code. This provides a few benefits beyond the obvious one of 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 are much less likely to end up with hard-coded 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 above. The configuration of a Polling Consumer's trigger can be provided through declarative configuration. Spring Integration takes the previous example a step further, however, by actually providing a File polling Channel Adapter:

```
<file:inbound-channel-adapter directory="/tmp/example" channel="files">
    <poller max-messages-per-poll="10">
        <interval-trigger interval="60" time-unit="SECONDS"/>
    </poller>
</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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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-trigger expression="0 * 9-17 * MON-FRI "/>
    </poller>
</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 inversion of control principle and the Spring programming model can be applied to enterprise integration patterns. The best way to reinforce that is by diving into a simple, but complete hands-on example.

1.4 Say Hello to Spring Integration

Now that you have seen the basic enterprise integration patterns and an overview of how inversion of control can be applied to those patterns, it's time to jump in and meet the Spring Integration framework face-to-face. So 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. However, many of those components act as stereotypes or templates. They play a role that is 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 only domain-specific behavior is the following:

```
public class MyHelloService implements HelloService {

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

}
```

There we have it, a classic Hello World example. In fact, this one is flexible enough to say Hello to anyone. Since this is a book about Spring Integration, and we've already established that it is a framework for building messaging applications based on the fundamental enterprise integration patterns, you may be asking: "Where's the Message, Channel, and Endpoint?" The answer is that you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

typically do not have to think about those components when defining the behavior. What we implemented above is a very straightforward POJO with no awareness of the Spring Integration API whatsoever. This is consistent with the general Spring emphasis on non-invasiveness and separation of concerns. That said, let's now tackle those other concerns, but separately, in the configuration:

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

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

<beans:bean id="helloService" class=""siia.ch1.MyHelloService"/>
```

This 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 is 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 does not require any awareness or make any assumptions about that sending component.

All of the configured elements are going to contribute components to a Spring ApplicationContext. In this simple case, we can bootstrap that context programmatically by instantiating the Spring context directly. Then, we can retrieve the Mesaage Channel from that context and send it a Message. We'll use Spring Integration's MessageBuilder to construct the actual Message. Don't worry about the details, you will learn much more about Message construction in Chapter 3.

Listing 1.3 Listing 1.3 Hello World with Spring Integration

```
public class HelloWorldExample {

    public static void main(String args[]) {
        ApplicationContext context = new ClassPathXmlApplicationContext("ssia/ch1/co
        MessageChannel channel = context.getBean("names", MessageChannel.class);
        Message<String> message = MessageBuilder.withPayload("World").build();
        channel.send(message);
    }

}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Running that code will 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 so that it returns a value rather than simply printing out the result itself.

```
public class MyHelloService implements HelloService {

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

}
```

Spring Integration will handle the return value in a way that is very similar to the Spring JMS support described earlier. We will add one other component to the configuration, a gateway proxy, to simplify the caller's interaction. Here's the revised configuration:

```
<gateway id="helloGateway"
         service-interface="siia.ch1.HelloService"
         default-request-channel="names"/>

<channel id="names"/>

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

<beans:bean id="helloService" class=""siia.ch1.MyHelloService"/>
```

Notice that the "gateway" element refers to a service interface. This is very similar to the way that 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 message construction and delivery in this case. You may have noticed that the MyHelloService class above does implement an interface. Here's what the HelloService interface looks like:

```
public interface HelloService {

    String sayHello(String name);

}
```

Now the caller only needs to know about the interface. It can do whatever it

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

wants with the return value. Here, we will 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 1.4 Listing 1.4 Hello World revised to use a Gateway proxy

```
public class HelloWorldExample {

    public static void main(String args[]) {
        ApplicationContext context = new ClassPathXmlApplicationContext("ssia/ch1/co
        HelloService helloService = context.getBean("helloGateway", HelloService.cla
        System.out.println(helloService.sayHello("World"));
    }

}
```

As with any Hello World example, this is only scratching the surface. Later you will learn how the result value can be sent to another downstream consumer, and you will learn about more sophisticated request-reply interactions. The main goal for now is to provide a basic foundation for applying what you've learned above. 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 inversion of control, separation of concerns, and an emphasis on non-invasiveness of the API.

Summary

We have 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 will learn in much greater detail how Spring Integration supports the various enterprise integration patterns. You will also see the many ways in which the framework builds upon the declarative Spring programming model. So far you have only seen 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 (i.e. that control is inverted so that the framework

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 EIP. Figure 1.21 shows at a high level how Spring Integration provides this foundation so that your services and domain objects can participate in messaging scenarios that take advantage of all of these patterns.

Figure 1.13 Figure 1.13 Spring Integration's support for Enterprise Integration Patterns

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

[1] In this context it's probably better to think of filter" as actually meaning processor.

[2] "a great fuss or disturbance about something very insignificant." Random House via Dictionary.com

[3] We are assuming some level of Spring knowledge primarily because we cannot possibly cover everything from the basics of Spring to the full spectrum of the Spring Integration framework. If you are new to Spring itself, you may want to check out some other books, such as Spring in Action, 2nd edition by Craig Walls (Manning).

[4] Dependency Injection in Action (Manning)

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

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

[7] See the "Strategy Method" pattern in "Gang of Four" (Design Patterns, Addison Wesley 1995)

[8] In Hollywood, that probably means "Don't bother us with your calls. In the very slim chance that you get the part, we'll call you. But, we probably won't so get

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

over it". In software we rely on things being a bit more definite.

[9] http://www.opensymphony.com/quartz

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

2 Enterprise Integration Fundamentals

Commercial applications are, most of the time, solutions to problems posed by the business units for which they are 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, in order 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.

In the Introduction, you have seen the most important enterprise integration patterns that will be used throughout the book. They are driven by a few fundamental notions and principles, on which this chapter will focus next, and which are:

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

Each of these concepts plays an important role in designing integrated applications. Loose cpumaking sure that the integrated components do not impose

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

needless restrictions on each other, and that the integrated system is responsive and can process concurrent requests efficiently. Applying them in practice results in several integration styles, each with its own advantages and disadvantages, that need to be taken into account when implementing a solution.

2.1 Introducing loose coupling and Event-Driven Architecture

Since we already discussed about building applications that consist of multiple parts (or are, for that matter, orchestrating functionalities exposed by a few stand-alone applications), it is important to understand the implications on our design. One of the most important consequences of decomposing applications into multiple components is that the latter can be expected to evolve independently. The design and implementation of these parts (or independent applications) can make this process very easy, or, on the contrary, very hard. Our criterion for deciding whether a particular design strategy fosters independent evolution or, to put it plainly, just stands in the way, is coupling.

In what follows next we will explain what coupling is, discussing the differences between loose and tight coupling. As well, we will distinguish between different forms of coupling (type-level, system-level), and will introduce solutions such as dependency injection and event-driven architectures.

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 maintainance. Achieving an appropriate degree of loose coupling allows us to spend more time adding new features and delivering business value, rather than spending time on maintenance and changing existing systems. It's hard to justify why small changes to one system will result in a three man month project, because it is necessary to update a large number of interacting systems. It's even harder when the effort could have been reduced considerably by considering the implications of a particular design.

It is not so much that loosely coupled systems will give us guarantees of 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 close to impossible to truly unit test their constituent parts. If particular units cannot be instantiated in isolation, . due to the complex relationship between components,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

ling has
de level
s a class
d highly
pes may
xample,
e for it.

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

Using referenced types as a measure of how interconnected classes within a system are will give you an idea whether the system is high or low coupled. This is a method that 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 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 does not create a need to update all service clients. In many SOA scenarios, service consumers may not be under the control of the service implementor, and in this case it is loose coupling that insulates us from creating lots of 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, otherwise said, how many assumptions the components of the system make about each other. For example, systems that directly reference RMI for communication in many places are highly coupled to RMI. Or, a system which directly references a third party system in a large number of places can be said to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

be highly coupled to the third party system in cause. The connections we create in our systems and between our systems introduce complexity and act as conduits for change and change generally means cost and effort to address those changes.

SIDEBAR **Coupling**

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

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

2.1.2 Type-level Coupling

Coupling between types is probably the most well understood. Listing 2.25 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 before invoking a meal preference web service.

```
public class BookingService {

    private final BookingDao bookingDao;

    private final WebServiceOperations mealPreferenceWebServiceInvoker;


    public BookingService(){
        this.bookingDao = new OracleBookingDao();
        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();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
        mealPreferenceWebServiceInvoker.sendSourceAndReceiveToResult(
                mealUpdateSource, result);
    }

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

    }
}
```

Listing 1.26 shows the BookingReportService used to generate management reports with the business methods omitted.

```
public class BookingReportingService {

    private final BookingDao bookingDao;

    public BookingReportingService(){
       this.bookingDao = new OracleBookingDao();
    }
... /** Methods omitted */
}
```

Because both classes directly reference the OracleBookingDao and the constructor declared by that class changes to that type may effect both the BookingService and the BookingReportingService. This type of coupling is known as unambiguous type coupling since it is 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 since 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, and which is our next topic.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

2.1.3 Loosening type level coupling with Dependency Injection

Standard dependency injection allows us 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 our services as well as reducing coupling also simplifies the code. This makes maintenance easier, simplifies testing by allowing tests to inject in test only implementations and increases the chances of reuse. Listing X shows the new version of the BookingService which now exposes a constructor that accepts instances of its collaborators.

```
public class BookingService {

    private final BookingDao bookingDao;

    private final WebServiceOperations mealPreferenceWebServiceInvoker;

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

   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>");
    }
}
```

Listing 1.28 shows the Spring configuration to instantiate the BookingService in production.

```
<beans …>

    <bean id="bookingDao" class="OracleBookingDao" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

    <bean id="bookingService" class="BookingService">
        <constructor-arg ref="bookingDao" />
        <constructor-arg ref="mealPreferenceWebServiceTemplate" />
    </bean>

    <bean id="bookingReportingService" class="BookingReportingService" />

</beans>
```

So while we have reduced coupling in our Java code we have introduced a new configuration file that is coupled to our concrete implementations. Taking a broader view of the system we now have that coupling in one place rather than potentially many calls to the constructors and therefore the system as a whole will be less highly coupled since changes to our OracleBookingDao will impact only one place.

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

2.1.4 System-level coupling

While it possible that collaborators might need to change to address new requirements it is almost inevitable that where a large number of systems talk to each other, those systems will evolve at different rates. Limiting one systems 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. Frankly, things could have been worse: the data format could have 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 (for example in a cluster or whenever passivation comes into play), or when data is exchanged between components which are expected to be highly

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

connected to each other (like a client-server application). Exchanging serialized data between independent applications couples them needlessly, as it introduces the assumption that both of them have access to the bytecode of the serialized class (which introduces type-level assumptions), that they use the same version of the class (which assumes that new versions of client classes are incorporated into the clients as soon as they are available), and, simply, that both applications are using the same platform (Java in our case). By using a portable data format like XML, we've just avoided a serious coupling pitfall. Of course, the applications are still coupled by the XML format, but sometimes this is as far as you can go: in the end, both applications have to have a basic agreement what is a correct message and what isn't, it's just that we believe in being more liberal with respect to that.

However, we've still got a problem that the need 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 booking service will fail if the meal preference service is unavailable at the point. Whether that is the desired behavior or not 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. While this does eliminate some coupling 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 booking service with the meal preference service we 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.

Listing 1.29 shows the simplified booking service which now simply enriches the meal preference passed in with the flight reference and the new transformer which converts the message to an xml request.

```
public class BookingService {

    private final BookingDao bookingDao;

    private final WebServiceOperations mealPreferenceWebServiceInvoker;

    public BookingService(BookingDao bookingDao,
            WebServiceOperations mealPreferenceWebServiceInvoker){
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
        this.bookingDao = bookingDao;
        this.mealPreferenceWebServiceInvoker =
                mealPreferenceWebServiceInvoker;
    }
    public MealPreference getFlightRefForBooking(
            MealPreference mealPreference){

        Booking booking = bookingDao.getBookingById(
                mealPreference.getBookingReference());
        mealPreference.setFlightReference(booking.getFlightRef());
        return mealPreference;
    }
}

public class MealPreferenceUpdateRequestTrasformer {
     public Source buildMealPreferenceUpdateRequest(
             MealPreference mealPreference) {

        return new StringSource("<updateMealPreference><flightRef>" +
                mealPreference.getFlightReference() +
                "</flightRef><mealPreference>" + mealPreference
                + "</mealPreference></updateMealPreference>");
    }
}
```

Listing 1.30 shows the Spring Integration configuration we can then use to invoke our simple components in order and the use of a gateway to then invoke the web service.

```
<beans>

    <si:channel id="mealPreferenceUpdatesChannel" />

    <bean id="bookingService" class="BookingService" />

    <si:service-activator input-channel="mealPreferenceUpdatesChannel"
                          output-channel="bookingEnrichedMealUpdates"
                          ref="bookingService"
                          method="getFlightRefForBooking" />

    <si:channel id="bookingEnrichedMealUpdates" />

       <bean id="updateRequestTransformer"
             class="MealPreferenceUpdateRequestTransformer" />

    <si:service-activator input-channel="bookingEnrichedMealUpdates"
                          output-channel="xmlMealUpdates"
                          ref="updateRequestTransformer"
                          method="buildMealPreferenceUpdateRequest"/>

    <si:channel id="xmlMealUpdates" />

    <ws:outbound-gateway uri=http://example.com/mealupdates
                         request-channel="xmlMealUpdates" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
</beans>
```

Focusing on message-based 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 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 is a tendency to simplify the implementation of the component by eliminating the concern of how to communicate with other components. Where events are communicated via channels which can act as buffers in periods of high throughput such a system can be described as having a Staged Event Driven Architecture. SEDA based systems generally respond better to significant spikes in load than a standard multi threaded application. SEDA based systems are also tunable at runtime allowing optimization for the current load being experienced, for example by modifying the number of consumers processing a certain event type.

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 will be dependent 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 and likewise event sinks and consumers are also not distinguished. The one place in which the term event is used in Spring Integration is the implementation of the Event Driven Consumer pattern but here the consumer is consuming messages rather than events it is 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 have just ended a section that shows how this can be done by elliminating assumptions about the concrete types using in the application, as well as replacing method and web service invocations with message passing in order to reduce coupling. With respect to the latter, we mentioned synchronous communication as a way to increase the coupling of two systems. In what follows next, we will address this in more detail.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

2.2 Synchronous and Asynchronous Communication

You have seen already what is the importance of coupling with respect to the quality of a software system. Implementing a well-designed application, turns, very often, in an exercise of reducing coupling by reducing the number of assumptions that components make about each other. One of the most fundamental assumptions is that two components that interact are available at the same time.

2.2.1 What's the difference?

In synchronous communication, one component will wait until the other provides an answer to its request, and will proceed only after a response has been provided.

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

When communicating asynchronously, the component that issues the request proceeds further, without waiting for an answer. The requests aren't actually

Figure 2.3 Asynchronous invocation: the requester does not block and executes in parallel with the provider

Looking at how different these two alternatives are, the decision on when to use what should be based on their strengths and weaknesses.

Of the two, synchronous communication is more straightforward: the recipient

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

of the call is known in advance, the result is received immediately. 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 is a remote call). This allows to propagate a wealth of contextual information, the most important being the transactional context (which helps to make the call part of an atomic operation). In the general case, the infrastructure required to set it up is simpler: a method call or a remote procedure call. Its main weaknesses are that is scales up pretty badly and it's less resilient to failure.

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

- trying to accommodate all requests as they arrive, which will crash the system
- throttle some of the requests for bringing the load to a bearable level[1]

Footnote 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

Either way, given a sufficiently large number of simultaneous requests, the application will eventually fail, and there's little you can do about it.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

This 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 this 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 asynchroInous model the load of the system is controlled by the service provider's

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

side. Furthermore, asynchronous communication has better opportunities for retrying a failed request, which improves the overall resilience to failure of the system.

You've seen that asynchronous communication has obvious advantages in what concerns scalability and robustness, but there is a price to be paid for that: *complexity*. The messaging middleware that performs the message storage and forwarding function is an additional mechanism that needs to be integrated with the application – which means that instead a simple method call, one needs to deal with supplemental APIs and protocols. Besides the fact that the code becomes more complex, there's also a performance overhead that gets introduced.

Also, in a request-reply scenario there's a very simple way to retrieve the result of an operation if following the synchronous approach – 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 (like 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 not, 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). While a powerful tool for increasing the application performance, concurrency is adding complexity to your application too. You have to consider concerns like thread safety, proper resource management, deadlock and starvation avoidance. Also, at runtime, it is harder to trace or debug a concurrent application. Finally, the two concurrent operations may typically end up executing in distinct transactions (if transactional behavior is a concern for your application to begin with).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table .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 does not wait to receive a response. Requests are buffered, and will be processed by the provider whenever ready.
Advantages	• simple interaction model • immediate response • allows the propagation of invocation context (transaction, security)	• good scalability under load • resilience to failure (requester and provider do not have to be available at the same time)
Disadvantages	• lack of resilience to failure • doesn't scale properly under load	• complex interaction model due to increased concurrency • hard to debug

The advantages and disadvantages of the two paradigms should be carefully taken in consideration when choosing one over the other, but, 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 multi-core architectures become more pervasive, there is an opportunity to apply the patterns that are specific to integrating multiple applications to different components of an individual

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

application. The modules of the application may not interact 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 enter into the picture?

While synchronous communication can take place using either procedure calls or messages, by its very nature asynchronous communication is message-driven. We've already 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 is a very abstract structure. For understanding the actions that will be taken in response to various messages, what transformations will they suffer, where to send a particular message in order to get certain results one doesn't need to know whether the communication will be synchronous or asynchronous.

Let's take into consideration 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"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

In what follows, we're anticipating things for a bit, but we'll provide explanations that will explain the parts that are new to you. 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 are supported), they are routed to the appropriate services (which are really just POJOs, completely unaware of the existence of the messaging system). The results of processing the requests are sent to another channel where they will be picked up by another component and so on.

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

That doesn't seem to tell what kind of interaction takes place (is it synchronous or is it asynchronous?), and, from a logical standpoint, it shouldn't. The interaction model is determined by the type of channel that is 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).

To the logical design of the system (how many channels, what endpoints, how are they connected together) this has less importance. However, 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.

NOTE

Staged Event-Driven Architecture

The technical name for this is Staged Event-Driven Architecture or SEDA, and you have seen it in a previous section.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

Both types of channels are configured using the same <channel/> element. By default, its behavior is synchronous, so that the handling of a single message sent to the input-channel, including routing, invocation of the appropriate service and what comes after that will be done in the context of the thread that sends the message. If you want buffer the messages instead of passing them directly through a channel, and 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>
```

So, now the router will poll to see if there any messages available on the channel and will deliver them to the POJO that does the actual processing. All this is taken care of transparently by the framework, who is going to configure the appropriate component. You don't need to change anything in your configuration.

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

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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, etc), and the overall design is message-driven, so that the coupling between components is kept to a very 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 on-line 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: separation of 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 the discussion about coupling. The final part of this chapter is dedicated to comparing the various enterprise integration styles.

2.3 Comparing Enterprise Integration Styles

As the adoption of computer systems in enterprise environments grew, it soon became apparent that there was a need to enable interaction between applications within the enterprise. This allowed to both share data and to make use of functionality provided by other systems. While 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. Each one of these styles is based on a different mechanism for interaction between systems.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

2.3.1 Integrating applications by transfering 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 certainly simple and is generally interoperable, since all that is required is for interacting applications to have the capability to read and write files. Since the basic requirements for using file based integration are simple, it is a fairly common solution, but some of the limitations of file systems mean that additional complexity may be created in the applications having to deal with files.

One of the limitations is that file systems are not transactional and do not provide metadata about the current state and validity of a file. As a consequence it is hard to tell, for example, if another process is currently updating the file. In general, for working properly, this type of integration will require some strategies to ensure that the receiver does not read inconsistent data, like, for example, a half written file. Also, it will be required to set up a process by which corrupt files are move out of the way rather than repeatedly attempting to process them. Another significant drawback is that applications will generally need to poll specific locations in order to discover if more files are ready to be processed, thus introducing additional application complexity and a potential for unnecessary delay.

Considering the above-mentioned drawbacks, before deciding to use file based integration it is always good to see if any other integration style would work better. Nevertheless, in certain situations it may be the only integration option, and it is not uncommon to come across applications that use it. The most important thing to do in those cases is to be diligent and to address the limitations through appropriate strategies, like the ones we mentioned above, and Spring Integration addresses some of them through its support for file based integration, as you will read in chapter 18.

2.3.2 Interacting through a shared database

Databases are better data storage mechanisms than files and provide relief to some of the limitations of the file systems. 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

with the file system style, this has the advantage that meta-data, 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 file system style, 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:

- one-size-fits-all is hardly true in software development, and the compromises that are necessary to implement a domain model (and subsequent database model) that is 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, as changing it will require all the other applications to change as well (at least in the parts that deal with the shared model directly)
- concurrent systems writing frequently the same set of data might face performance problems, since 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 become an issue, since it can become stale without the applications to become aware of it
- 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 for e

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 cannot safely be hidden or ignored. Assuming that it can 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 that they can be translated into a intermediate format that can be

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

transmitted over the wire. From case to case, this can be achieved in different ways, for example through Java Serialization, or XML marshalling through mechanisms such as JAXB. This not only restricts the types that can be sent, it also requires that the application code deals with serialization or marshalling errors.

We're not the first to tell you probably, but it is worth mentioning again that the network is not 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 is much less predictable. The trickiest part however is that the service that is invoked can return successfully, while the invoker doesn't get the response. Assuming that 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 is 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, which we will discuss next, and the one which is the general focus of Spring Integration.

2.3.4 Exchanging messages

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

Message exchanging is the core paradigm of Spring integration, and we will discuss in detail the concepts behind the integration style and how are they applied in the framework in chapters 4 and 5.

2.4 Summary

This chapter has shown the fundamental concepts which stand behind enterprise application integration. We have enounced them, and provided applicable samples from the framework itself, but in the rest of the book you will see them at work,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

First, we have discussed coupling, as a way to measure how many assumptions two independent systems make on each other. You have seen how important is to minimize it, in order 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 dealing the former, whereas the latter is to be addressed by using a message passing aproach instead of direct calls (either local or remote).

Introducing message passing led to the next topic of the chapter, which was a detailed overview of the differences between synchronous and asynchronous communication, and its implications in what concerns coupling, systems' complexity and performance. You have also seen how Spring Integration helps separating the logical design of the system from addressing the dynamic behavior at runtime (whether interaction should be synchronous or asynchrounous).

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

In order to wrap the introductory section of the book, we need to take a look at the example application that we will use throughout the book to illustrate the various features of the framework, and this happens in the next chapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

3 Messages and Channels

Now that you've got an idea of the high level concepts of Messaging and Spring Integration, and you have looked at an example, it's time to dive in a little deeper. As mentioned in the first chapter, the three fundamental concepts in Spring Integration are Messages, Channels and Endpoints. Endpoints will have a chapter of their own, but if you follow the order of the book religiously you get to read about Messages and Channels first.

By introducing Messages, we'll show how the information that is exchanged between the different parts of the application is organized. When describing Channels, we'll show what are the conduits through which Messages travel, and how the different components are connected. And there is more than just explaining what they are: because of the variety of options that Spring Integration provides when it comes to Channels, it's also important to know how to choose the one that's right for the job, and this is what we will focus on next. Finally, there are a couple of more advanced components that you can use to enhance the functionality of the Channels - and hereby we're referring to dispatchers and interceptors, and this is what the chapter is going to end up with.

We promised that at the end of every chapter, you will find a section called "Under the hood", where we discuss the API of the framework, provide details about the framework internals. In this chapter we will break that promise by starting with the introduction of API-related concepts. Why is that? Because Messages and Channels are fundamental concepts, and understanding how Spring Integration handles them is a very good foundation for building your knowledge about the framework. And we believe that the best way to get to trust something is to understand how it works.

So, let's lift the hood and let's take a look at what's there!

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

3.1 What's in a Message?

To begin our journey into the message exchange capabilities of Spring Integration, let's revisit shortly the concept of a message. A message, as described by EIP, is a discrete piece of information that is sent through from one component of the software system to another. The *"piece of information"* bit above means that any data that can be represented in the system (Objects, byte arrays, etc.) can be passed along as part of a message. The *"discrete"* bit means that messages are, in most typical scenarios, independent units, and that a message will contain all the information that is 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 information which is not of any interest to the application *per se*, but it is critical for allowing the infrastructure to handle the information correctly.

The obvious analogy here is a letter, which contains an individual message that is destined to the recipient, is wrapped in an envelope that contains extra information that is useful to the mailing system, but not to the recipient itself: a delivery address, the stamp that indicated that the fees for mailing have been paid, and various stamps applied by the mailing office for routing the messages more efficiently.

Based on the role that 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 reactA

After this overview of the message concept, let's talk about Spring Integration's approach to it, from an API and implementation standpoint, and see how we can actually create a message - granted most of the time you won't need to do that by yourself, but since we're dealing with internals here, it is important to see how this does actually happen.

3.1.1 How is it done in Spring Integration

The nature of a message in Spring Integration cannot be better expressed than the `Message` interface:

```
public interface Message<T> {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
  MessageHeaders getHeaders();

  T getPayload();

}
```

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

The framework uses an interface as the general contract, because of their generic nature. Having an interface as the top-level abstraction allows the framework to be extensible, by allowing to create your own Message implementations, besides the ones included with the framework. In this context, it is important to notice that creating your own Message implementations is hardly necessary, and most of the time you won't even need to deal with the Message objects directly, as you will see in the following chapters.

The content wrapped by the Message can be any Java `Object`, and the parametrized nature of the Java interface allows to retrieve the payload in a type-safe fashion, avoiding an undesirable down-cast.

Besides the payload, a Message contains a number of *headers*, which are metadata associated with this message, like an identifier that helps distinguishing between the different message instances, or some information that correlates various individual messages. Also, the headers may contain information about the message itself. For example, if the message has been created from data coming from an external source (like a JMS Queue, or the file system), the message will contain information that is specific to that file source (like the JMS headers, or the path to the file in the file system).

The MessageHeaders are essentially an implementation of the Map interface. Their role is to store the header values, which can be any Java `Object`, and are referenced by a header name.

```
public final class MessageHeaders implements Map<String, Object>, Serializable {
   ...
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

BUT WHERE ARE THE SETTERS? (AND WHO DOES CREATE THOSE MESSAGES FOR ME?)

As you may have noticed from definition of the Message interface, there are no ways to *set* something on a `Message`. In fact, 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 very good reason: messages are immutable. In publish-subscribe scenarios, the same instance of a message may be shared among different consumers, and immutability is a simple, but effective way to ensure thread-safety - imagine what would happen if multiple concurrent consumers would modify the same Message instance

tations for
for internal
or creating
to create a

or another
ee steps in

Figure 3.1 Creating a message with the MessageBuilder

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 need to be set up. It should be noted that the MessageBuilder itself will initialize a number of headers which are needed by

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

default by the framework, and this is one of the significant advantages of using the MessageBuilder over instantiating Messages directly, as you don't have to worry about setting up those headers yourself.

So, for example, creating a new message with a String payload can take place as follows:

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

On the other hand, creating a message which is a copy of the original message can be done like this:

```
Message<String> helloMessage =
   MessageBuilder.withPayload("Hello, world!")
                 .setHeader("my.custom.header", "HeaderValue")
                 .build();
```

Using the option for setting header values if absent can be applied as follows:

```
Message<String> newHelloMessage =
    MessageBuilder.fromMessage(helloMessage).build();
```

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

```
Message<String> helloMessage =
   MessageBuilder.withPayload("Hello, world!")
                 .setHeader("my.custom.header", "HeaderValue")
                 .build();
```

For the cases when you want to send a header value, if it is not already set you can do this:

```
Message<String> helloMessage =
   MessageBuilder.fromMessage(helloMessage)
                 .setHeaderIfAbsent("my.custom.header", "AnotherHeaderValue")
                 .build();
```

What you have just seen is what are the messages of Spring Integration and how to create one. Next, we're going to discuss how messages are sent and received through *channels*.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

3.2 It has Channels, and it's not Mars

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

Let's appeal again to the same letter analogy that we've used before when we talked about messages. The sender creates the letter, and hands it off to the mailing system by depositing it into a well known location, which is the post box. 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 another reply - but, otherwise, who routes the message, who the physical recipient of the letter is (if, for example, the letter is addressed to an organization rather than the person) are things which are not known to it. From a logical standpoint, the channel is very much like a post box: a place where certain components, called Producers, deposit messages which are later on processed by other components, called 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 the emphasis that the channels play in defining the enterprise integration strategy. They're not just plain information transfer components, but they play an active role in defining the overall application behavior. The business processing takes place in the endpoints, but you just have to alter the channel configuration, to get completely change the runtime characteristics of the application.

So, you'll see channels presented from a logical perspective, but you'll also get an overview of the various channel implementations provided by the framework, what is 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 within the application we use a channel. All channels within Spring Integration implement the `MessageChannel` interface shown below, which defines standard methods for the sending of messages. Note however, that it provides no methods to allow the receipt of messages.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
public interface MessageChannel {

    String getName();

    boolean send(Message<?> message);

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

}
```

The reason for this is that the message channels provided by Spring Integration fall into two distinct categories which primarily differ with regard to the mechanism for handing the message over to the next endpoint.

3.2.2 Do you have any message for me?

`PollableChannels` as defined by the below interface require the receiver or the framework acting on behalf of the receiver to carry out a periodic check to see if messages are available on the channel. This approach has the advantage that the consumer can process messages at a time of their choosing. The approach can also have downsides creating a trade of between longer poll periods which create latency in the processing of a message and computation overhead from more frequent polls that find no messages.

```
public interface PollableChannel extends message channel {

    Message<?> receive();

    Message<?> receive(long timeout);

    List<Message<?>> clear();

    List<Message<?>> purge(message selector selector);
}
```

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

The alternative offered by Spring Integration is the `SubscribableChannel` interface which is implemented by channels that take responsibility for notifying subscribers when a message is available. In most cases however the JOJO consumer code will remain unaware of the capabilities of the channel from which messages are received since the framework will take responsibility for subscribing or polling as appropriate.

```
public interface SubscribableChannel extends message channel {

    boolean subscribe(message handler handler);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
    boolean unsubscribe(message handler handler);

}
```

While it is important to understand the implications of whether a channel pushes out messages or is periodically polled in most cases the framework will take on the concern of connecting a consumer to a channel, alleviating the complications of defining the appropriate consumer types. To put it plainly, your job is to select the appropriate channel type, and the framework will select the appropriate consumer type (polling or event-driven).

3.2.4 The right channel for the job

Spring Integration offers a number of different channel implementations and since `MessageChannel` is an interface you are also free to provide your own implementations. The type of channel selected will have significant implications for your application including transactional boundaries, latency and overall throughput. This section will walk through the factors to consider and through a practical scenario the selection of appropriate channels. In the configuration we will use the namespace but we will 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 in the mind of many businesses is the need to get paid so making sure that we can charge the provided credit card is a high priority since we don't want unpaid for bookings. We will also want to ensure that an update to indicate one less seat is available flight occurs in the case of a booking to ensure we don't over book the flight. The system will also be required to send a confirmation email with details of the booking and additional information on the check in process. In addition to a web site our Internet booking application exposes a rest interface to allow third party integration for flight comparison sites and resellers. Since most of the airlines premium customers come through the airlines own website any design should allow us to prioritise bookings originating from there over third party integration requests in order to ensure that the airlines direct customers experience a responsive website even in times of high load.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

3.2.5 Is choosing a channel all that difficult?

The selection of channels will be based on both functional and non functional requirements, and there are several decision factors that can help us making the right choice. This section provides a brief overview of the technical criteria and the best practices that you should consider when selecting the correct channels.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 3.1 How do we decide what channel we use

Decision factor	What do we need to consider
Sharing context	• Do we 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, e.g. for example transaction context. • Relying on the thread context is in itself a subtle form of coupling and will have an impact when considering the adoption of a highly asynchronous SEDA model. It may prevent splitting the processing in concurrently executing steps, prevent partial failures or introduce security risks like leaking permissions to the processing of different messages.
Atomic boundaries	• Do we have "all or nothing" scenarios? • Classic example: bank transaction where credit and debit should either both succeed or fail. • Typically used to decide transaction boundaries, and because of that it is a specific case of context sharing. Has an influence on the threading model and therefore limits the available options when choosing a channel type.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Buffering messages	• Do we 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 will require more hardware resources for accommodating the peak load situations. Those resources will stay unused when the load decreases, so this could be expensive and inefficient. Moreover, some of the steps may be very slow, so resources may be blocked for a long amount of time. • Consider what requires an immediate response and what can be delayed and 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 will acknowledge the receipt of the request and do some mandatory steps immediately (credit card processing, order number generation) and respond to the client, but will do the actual handling of the request asynchronously in the background (assembling the items, shipping, etc).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Blocking and non-blocking operations	• How many messages can we actually buffer? What should we do when we can't cope with demand? • If our application is unable to cope with the number of messages being received and no limits have been but in place we may simply exhaust our capacity for storing the message backlog, or breach quality of service guarantees in terms of response turnaround. • Recognizing that the system cannot 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 to the system this can be achieved by blocking the acceptance of new messages where the system is approaching its maximum capacity. This can commonly be 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 set a timeout on the blocking operation to avoid indefinite blocking of the requester.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Consumption model	• How many components are interested in receiving a particular message? • There are two major messaging paradigms: point-to-point or 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 of them. • If the processing requirements are that the same message should be handled by multiple consumers, they can do this concurrently and a publish-subscribe channel can take care of that. An example is a mash-up 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 or not. • Conversely, if the request should be always handled by a single component (for example processing a payment), a point-to-point strategy is what we are looking for.

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

3.2.6 A channel selection example

Using the default channel throughout we have three channels one accepting requests and the other two connecting our 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" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

In Spring Integration, the default channels are `SubscribableChannel`s, and the message transmission is synchronous.The effect of this is simple, we have one thread responsible for invoking the three services sequentially, as shown in the figure.

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

Because all operations are executing in a single thread, there is a single transaction encompassing those invocations. That assumes that the transaction configuration does not require new transactions to be created for any of the services then the three service invocations will occur within the scope of transaction.

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

This is what you get when you configure an application configured by 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 certain applications where all three operations must be executed atomically, but takes its toll on the scalability and robustness of the application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

BUT EMAIL IS SLOW AND OUR SERVERS ARE UNRELIABLE

The basic configuration is all well and good in the sunny day case where the email server is always up, responsive and the network is 100% reliable. Reality is different. Our application needs to work in a real world where the email server is sometimes overloaded and the network sometimes fails. Analysing our actions in terms of what we need to do now and what we can afford to do later is a good way of deciding on what service calls we should block on. Billing the credit card and updating the seat availability are clearly things we need to do now in order to be able to respond with confidence that the booking has been made. Sending the confirmation email is not time critical and we 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 us to do just that, charge the card, update availability and send the email confirmation when we can.

Introducing a queue on our emailConfirmationRequests channel will allow 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" />
```

Let's recap how the threading model changes by introducing the QueueChannel:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

We have replaced one of the default channels with a buffered QueueChannel, and have set up an asynchronous communication model. What we got was some confidence that long-running operations will not block the application because some component is down, or just takes a long time to respond. But now we have another challenge: what if we need to connect one producer with not only one, but two (or more) consumers?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

TELLING EVERYONE WHO NEEDS TO KNOW THAT A BOOKING OCCURRED

Up until this point we have been looking 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 only needs to be consumed once however it is common that more than one consumer may be interested in receiving certain messages. In our current version of the channel configuration bookings that have been successfully billed and which 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 our application and systems within the enterprise such as Customer Relationship Management systems which track customer purchases to better target promotions as well as finance systems monitoring the financial health of the enterprise as a whole.

In order to allow the delivery of the same message to more than one consumer we will introduce a publish subscribe channel after the availability check. The publish subscribe channel provides one to many semantics rather than one to one semantics provided by most channel implementations. This can be particularly useful where we want the flexibility to easily add additional consumers to the configuration since if the name of the publish subscribe channel is known that is all that is required for the configuration of additional consumers with no changes to the core application configuration.

The publish subscribe channel does not in itself support queueing although it does support asynchronous operation. This can be done by providing a task executor which is then used to deliver messages to each of the subscribers in separate threads. This may however 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 where the underlying thread pool is exhausted.

In order to ensure that a backlog in sending email confirmations does not block either the sender thread or the entire thread pool for the task executor we can connect the new pub sub channel to the existing email confirmation queue by means of a bridge. The bridge is an EIP pattern that supports the connection of two channels so this will allow the pub sub channel to deliver to the queue and then have the thread return immediately.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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 we have made 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 THEN OTHERS

In order to ensure that the airlines' direct customers have the best possible user experience we have stated we want to prioritise the processing of their requests to allow us to render the response as quickly as possible. Using a comparator that prioritises direct customers over indirect we can modify the first channel to be a priority queue. This will cause the framework to instantiate an instance of `PriorityChannel` which creates a queue which is able to prioritise the order in which the messages are received. In this case we providing an instance of 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" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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 as an example of applying different types of channels for solving the different requirements with which the application has challenged us. We started with the defaults, and as we have worked through the example, we replaced several channel definitions with the ones that are the 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 particular use case may not be something you want to use somewhere else. We illustrated the decision process with the criteria that we find the most relevant in every case.

This wraps up the overview of the Spring Integration channels and makes way to 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 to go beyond simply sending and receiving messages. For this purpose, Spring Integration provides additional components, that we call channel collaborators.

In what follows next, we will look at the `MessageDispatcher` which controls how receiving 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 revceive operations.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

3.3.1 MessageDispatcher

In most of the examples that we have seen so far, a channel has a single message handler connected to it. While having a single component that takes care of processing the messages that arrive on a channel is a pretty common occurence, multiple handlers processing messages arriving on the same channel is a 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 which are capable of handling the message, even if only one of them will actually be selected for doing 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 `MessageDispatcher` interface shown below.

```
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 below, which determines which single`MessageHandler`of potentially many should receive any given message. The only provided implementation of this is the `RoundRobinLoadBalancing`which works through the list of handlers passing one message to each in turn.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
public interface LoadBalancingStrategy {

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

}
```

Providing your own implementations of `MessageDispatcher` is not common and should only be resorted to where other options do not provide the level of control desired over the process of delivering messages to handlers. An example where a custom `MessageDispatcher` could be appropriate would be where different handlers were being used to deal with varying service levels. In this scenario a custom dispatcher could be used to prioritise messages according to the defined service level agreements. In this case the dispatcher implementation could inspect the message and only attempt to dispatch to the full set of handlers 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 have seen how to control the way in which messages from a channel are distributed to the message handlers that are listening on that particular channel.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

In what follow next, we will see how to meddle in the process of sending and receiving messages.

3.3.2 ChannelInterceptor

Another important requirement for an integration system is the ability of being notified as messages are traveling through the system. This can be used for several purposes, ranging from monitoring of messages as they pass through the system to the vetoing of 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. The `ChannelInterceptor` interface allows implementing classes to hook into the sending and receving of message by the 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 looking at the names of the methods it is pretty easy to get an idea when these methods are invoked, but there's more to this component than just simple notification. Let's review them one by one:

- preSend - is invoked before a message is sent and returns the message that will be actually sent to the channel when the method returns. If the method returns null, nothing is sent. This allows the implementor to control what does actually get sent to the channel, effectively filtering the messages.
- postSend - is invoked after an attempt to send the message has been made, and indicates whether the attempt has been successful or not through the boolean flag that is passes as an argument. This allows the implementor to monitor the message flow and learn what has been sent or if sending a message has failed;
- preReceive - applies only if the channel is pollable, and is invoked before a component calls receive() on the channel. It allows to decide whether the channel can return a message or not to the caller.
- postReceive - similar to preReceive, applies only to pollable channels, and is invoked after a message has been read from a channel, but before it is actually returned to the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

component that called receive. If it returns null, then no message is actually received. This allows the implementor to control what does actually get received by the poller (if anything)

Creating a new type of interceptor is typically done by implementing the ChannelInterceptor interface[1] :

Footnote 1 As a matter of fact, Spring Integration provides a no-op implementation that framework users can extend, named ChannelInterceptorAdapter, that allows to implement only the needed functionalities.

```
public class ChannelAuditor implements ChannelInterceptor {
    ....
    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, that logs the charged bookings that have been created by the application, before an e-mail request is sent out.

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 shown below:

```
<beans:bean id="auditInterceptor"
           class="com.manning.siia.bookings.logger.ChannelAuditor">
   <beans:property name="auditService" ref="auditService"/>
</beans:bean>


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

Some of the best examples for understanding the usage of the ChannelInterceptor come from Spring Integration itself, which has support for two different types of interception: monitoring messages and allowing only some of them to be sent on a channel.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 [reference]. As you have seen before, it is 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>
```

So, for all the bookings we charge, we send a copy of the message on the monitoringChannel, where they can be analized by a monitoring handler, independent of the main application flow.

On the other hand, 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 or not 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 as to whether a message is acceptable, taking in consideration a given set of criteria, as shown below:

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

One of the implementations of a MessageSelector that is provided by the framework is the PayloadTypeSelector, that accepts only the payload types that it is configured with, so combining it with a MessageSelecting interceptor 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="o.s.i.selector.PayloadTypeSelector">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
  <beans:constructor-arg value="com.manning.siia.booking.domain.ChargedBooking" />
</beans:bean>

<beans:bean" id="typeSelectingInterceptor"
[CA]class="o.s.i.c.interceptor.MessageSelectingInterceptor">
  <beans:constructor-arg ref="typeSelector"/>
</beans:bean>

<channel id="chargedBookings">


   ...
   <interceptors>
      <interceptor ref="typeSelectingInterceptor"/>
   </interceptors>
</channel>
```

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

NOTE	**MessageSelector is multivalent** As you will progress further, you will encounter the MessageSelector being used by another component, the MessageFilter, which has some pretty similar functionality, but which is a message handler, not a channel interceptor. The difference is rather subtle, as the role of filters is mostly to prevent messages to reach other endpoints (especially in a chain setup), while selectors prevent messages to be sent on channels, so they do pretty much the same thing, but in different types of 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 contorlling threading models, thread utilisation and latency not seen in any other mainstream Java EAI framework. In choosing the correct channel it is vital to understand the behavior of the provided implementations, since choosing incorrectly can have serious performance implications or invalidate the correctness of the application by altering the transactional boundaries.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

4 Message Endpoints

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 while 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 is central to Spring Integration is as simple as it can be:

```
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 JMS MessageListener would. However, those components can be connected to any type of channel, and as reiterated above, 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 based on the type of channel

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

and the role of the handler component being connected to that channel. If the channel is subscribable, the handler will be invoked within the thread of the sender or by the channel's `TaskExecutor` if one has been 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 the aptly named `QueueChannel`, polling is necessary.

In addition to the subscription vs. polling distinction, it is also necessary to consider whether a reply Message is expected, and if so, 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, it is necessary to 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 it's determined that an additional transformation step is necessary prior to sending a reply.

In this chapter we will explore the different endpoints that are available in Spring Integration and see how they can be used to decouple message producing and consuming components from the channel type and response strategy for which they have been configured. First we will describe the difference between Polling and Event Driven components in detail. Then we will look at those differences in the context of inbound and outbound endpoints. Finally we will discuss the different response strategies to clarify the distinction between Channel Adapters and Gateways.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

NOTE **Is Spring Integration an *implementation* of the Enterprise Integration Patterns?**

Spring Integration stays true to EIP naming wherever sensible. However not all patterns have their direct counterpart in the API. In other words, not every *pattern* maps directly to an *implementation*. Some patterns are more conceptual and others describe a broad category within which actual implementations may be classified. The Message Endpoint is an example of such a pattern. The "endpoint" is too generic to be covered by a single concrete implementation. Instead we have opted to have 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, etc.). 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 while the Return Address is supported by a message header that is recognized by any reply-producing message handler that does not have an explicit output channel configured.

Before we can talk about endpoints and decide which one is best applied to a certain problem, it is important to understand the different properties of endpoints a little better. The next section will go 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 very broad term. We need to be very precise in our understanding of the properties of a certain type of endpoint before we can have a meaningful discussion about it.

One of the greatest assets of Spring Integration is the consistent naming in the namespace support. In this section we explore what makes endpoints different enough to to give them different names. Several characteristics are important:

- Polling or Event Driven
- Inbound or Outbound
- Unidirectional or Bidirectional

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

- Internal or External

Taking in account all the possible ways in which 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 certain protocol, which greatly reduces the number of options. Also Spring Integration can implicitly take care of the polling concerns. There are still plenty of options to consider, so let's look at the most important examples in table 4.1.

Table 4.1 This table describes the differentiating characteristics of Endpoints. An endpoint can be either polling or event driven, either inbound or outbound (from the perspective of the Spring Integration application), either uni- or bi-directional (one-way or request/reply respectively), internal or external (with respect to the ApplicationContext).

Endpoint name	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

As the astute reader might have guessed from the table, there are some naming conventions for endpoints. Wherever possible the names have been chosen to stay as close to a pattern as possible, and in addition, the properties above are signified by certain consistently applied tokens. For example a "gateway" is always capable of bidirectional communication. This doesn't change between http or jms, inbound or outbound. In contrast, a "channel-adapter" is always unidirectional: it's either the beginning (inbound) or the end (outbound) of a message flow. Now that you know the choices you can make before selecting an endpoint, it is time to look into the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

implications of those choices in detail. First we will ask what may very well be the most important question when designing a messaging system.

4.1.1 To poll or not to poll?

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

When a user of our application is entering a lot of information we would like to save that 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 is not an option. Even if client side storage is an option technically, in many cases it is simply not an option functionally. Imagine a web application that allows users to collaborate, like 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. While it is important to get the changes from one user to another quickly, its much more important to get the changes from a single user to his 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 would be trailing a sentence behind the current users 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 are 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 the changes of one user to another user's client without making the first user wait for confirmation. As we discussed in the previous chapter we will 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. Since 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

delegated to a `WorkManager` when running in an Application Server environment that requires threads to be managed by the container. We will 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 of messages that have been handed off asynchronously. Even more importantly, we need to understand how to easily 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, or may be files in a directory, messages in a JMS Destination, and so on. It will need 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. [code sample here?] While 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 will never write different code depending on whether you want your component to be active or not. Instead that will be inferred from the configuration and handled by the framework. However configuring the components that will take the responsibility for the polling concerns is still up to you.

EVENT DRIVEN ENDPOINTS

In the majority of cases asynchronous hand-off is not required. In those cases an Event Driven model should be used. This can be as simple as wrapping a POJO. Exposing a web service is also a good example. The essential thing about passive, or Event Driven components, is that they will not take responsibility for thread management. They are still responsible for proper thread safety, but this could be as simple as not maintaining any mutable state.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

4.1.2 Inbound Endpoints

Figure 4.1 Sketch of the behavior of an inbound endpoint. First input is taken from an external source, then the input is converted into a message which is sent onto 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. There are some technical constraints like the fact that are no file system events in Java. But in most cases it depends simply on the system that you are integrating with. If the external integration point is passive the inbound endpoint becomes responsible to actively poll for updates.

In Spring Integration several polling endpoints are provided. There are inbound channel adapters for file, jms, email and a generic method invoking variant. The latter can be used to create custom channel adapters, for example querying twitter, or an RSS feed.

EVENT DRIVEN INBOUND ENDPOINTS

In many cases an external system will actively invoke a service on the messaging system. In those cases the responsibility of thread management can be left out of the messaging solution and left to the invoker (in case of a local Java method call) or to the container (in case of a web service invocation). However, as soon as `QueueChannel`s 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

include: Web Service (through Spring WS), RMI, and JMS (where Spring's listener container takes care of the polling). Again there is also a generic way to create a custom event driven endpoint, using the `@Gateway` annotation.

4.1.3 Outbound Endpoints

At some point in a message flow, you will likely need to invoke a service that is external to the messaging system. This may be a local method call, a web service invocation, or a message sent on another messaging system like JMS or XMPP. It is the responsibility of an endpoint to make this call. A sketch of the general

Figure 4.2 Sketch of the 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, see Gateway.

The algorithm shown in this figure will usually need to implement details such as transactions and security, which we will cover later. The details of conversion and the actual invocation of external services are also assumed to be implementation details of the specific endpoint. However, the reception of the message and whether or not there will be a result to deal with are concepts that are part of the base API.

POLLING OUTBOUND ENDPOINTS

If an outbound endpoint is connected to a `PollableChannel` it will be necessary to invoke the receive method on its input channel to receive messages. That requires the scheduling of an active process to do that polling periodically. See the (reference Triggers, Polling, TaskScheduler). In other words, step one in the algorithm will be triggered by the Endpoint itself. Spring Integration will automatically wrap any passive components in a polling endpoint if they are configured to be on the receiving end of a PollableChannel.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

EVENT DRIVEN OUTBOUND ENDPOINTS

When an outbound endpoint is connected to a `SubscribableChannel` it can be passive. The channel will ensure that there is a thread that will invoke the endpoint when a message comes in. In many cases this will simply be the thread that invoked the send method on the channel, but it could also be handled a thread pool managed at the level of the channel. The advantage of using a thread pool is that the sender does not have to wait, but the disadvantage is that a transactional boundary would be broken since the transaction context is associated with the thread. This is discussed in detail within section 4.2.

4.1.4 Unidirectional and Bidirectional Endpoints

The examples used above all assume a unidirectional communication. Endpoints using this style of communication are called Channel Adapters. While unidirectional communication is in many cases enough to establish a working solution, it is very common to have bidirectional requirements. For these scenarios you can use Gateways. The EIP definition of Gateway is not as clear as we would have liked, so we have done a bit of interpretation of it. Mainly the lack of clear distinction between synchronous invocation and asynchronous hand-off makes the Gateway concept too broad and widely overlapping with other concepts such as Channel Adapter. In Spring Integration a Gateway will be 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, either through polling or not, we have seen that 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 we would use a Channel Adapter as shown in the previous sections. The second case becomes a bit more complicated, but luckily Spring Integration has Gateway to help you support it. The complexity lies in that we need to do something with the response.

Usually we will either want to send it further along its way to the next endpoint, or when all processing for this message is done we might want to send a confirmation back to the original sender. For the first option we would set an

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

output channel, in the second we would 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 pretty good understanding of the different high-level categories of endpoints. We've explored the distinctions between polling and event-driven consumers, and we've also discussed both uni-directional and bi-directional behavior. In the next section, we'll add one of the most important pieces to the puzzle: transactions. Unfortunately, while its importance is undeniable, many developers admit to having a less than clear understanding of the topic. We'll do our best to remedy that situation, at least within the context of Spring Integration applications.

4.2 Transaction boundaries and other thread bound contexts around endpoints

A common mistake in moving from thread per request to asynchronous hand-off is to make incorrect assumptions about transaction boundaries. Another common mistake is to incorrectly assume that the security context has been set in the thread that is being handed off to. Both of these incorrect assumptions are caused by the fact that 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 should not be shared lightly, and transactions should be kept short. More on this in the first subsection.

This section goes into the details of transaction management around Spring Integration endpoints. You have already been introduced to transaction management in (reference?), so we will mainly focus on endpoint related concerns here. Before we can go into detail on the technicalities of transaction management and draw parallels to security, we need to establish the rationale of thread bound contexts.

4.2.1 Why sharing is not always a good thing

Have you ever been working on a shared file and found out that someone else has just fixed the same problem that you've worked on all morning? I bet you have. The first reaction to a situation like that is usually to request immediate notification of any changes that anybody is planning to make. Or it is suggested to do better planning or use a new tool.

Have you ever tried to work on something and been constantly distracted by team members that were trying to work on the same thing and tried to align with

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

you? Have you ever been stuck in planning meetings all day? I bet you have.

These are two sides of the same problem. You can't have your cake and eat it, so you need to carefully consider the consequences of living with either the overhead of sharing, or the overhead of merging changes. This is exactly the choice that needs to be made 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 give him the smallest possible chance of running into a conflict with another users' changes.

Best 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 is servicing a single http request. It has been suggested many times to store transaction contexts in a message instead, but that would potentially increase the scope and duration of the transaction considerably, so this is not natively supported by Spring Integration.

Now we've 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 do not become stale over time, or open the door to merge conflicts. In fact, once properly authorized *and kept in sight at all times* a user doesn't need to enter a password again. The problem is however, 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 very careful to ensure that the security details are only visible from the context of the user. If multiple users are using the application concurrently the least error prone isolation mechanism is to only allow access from the thread currently processing the request.

TIP	**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 isn't needed anymore, just a username will do.

Classic web applications do not usually need to worry about asynchronous hand-off to work. Spring Integration comes in when this becomes relevant, and at that point we need to jump some hoops. Of course the first thing a crafty engineer will ask before jumping hoops is if he can get out of it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

Go to the most senior DBA in your organization and ask him this question, come on it will be fun, I promise! After you've listened to his lecture carefully, remember at least one thing: there is no excuse for being clueless about transactions. That said, there is no excuse for being a zealot either. Transactions are not the answer to all your problems, in fact overusing them can be detrimental to performance and scalability.

ACID as we all know stands for Atomic, Consistent, Isolated, Durable. A transaction is supposed to give you all of this. Don't be fooled though, 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

Bill and Jack are checking in for their afternoon flight to NY. 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 16:20 and they're expecting a cab to the airport at 15:00 it's now 14:49, so they're in a bit of a hurry.

Bill hits the seat selection application first. He tags that he wants to travel with a companion and fills in Jack's details. The application renders a page to Bill that shows Jack's position on the plane. Jack is in an isle seat at row 15, Bill stops to check the make of the plain and the seat arrangement to decide if he wants the isle seat opposite to Bill, or if he 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 Bill's window seat on row 8. It won't be as comfortable as the other seats, but at least they will sit together. Jacks transaction completes fine and he runs for his cab. Bill is done as well hits submit, sees that his seat number has changed, so he continues to synchronise his mail before he goes offline.

It doesn't take a lot of mental effort to predict that Jack and Bill 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 is starting seat selection. But how long are you prepared to wait for that passenger to check the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

seating advise, his mood and any other things that might be distracting him at that particular moment before you're going to allow other passengers to get on? Having a globally consistent view of the world is very commonly too impractical to work.

So are use cases like the story of Bill and Jack impossible to cater for? Surely not! Anything is possible, we just need to find a way to deal with conflicting changes better. We will explore several ways to do this next.

Fix the problem when things go wrong

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

The scenario just described is a form of *Eventual Consistency* achieved by *Compensating Transactions*. After the transaction has been committed we invite the stakeholders to make compensating changes based on the new reality.

Express intent in the message, leave implementation to the mediator

What if Bill and Jack would not pick their seats, but ask to be seated together? Then the scenario would play out in one go. Jack asks to be seated next to Bill first and is moved. Then Bill asks to be seated next to Jack and the server knows that no move is needed.

The concession we do here is that Bill and Jack no longer control their seats themselves. Allowing them to do so would require a much more complex application, and most importantly would make the application more difficult to use. This is not always desired, but it can be very powerful.

If you take a trivially simple implementation of this system, you can still make it work guaranteed 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 executes this directive without any transaction blindly. In this case conflicting changes like sketched in the example might happen, but if you keep sending the message until you see the right result as a client, the system also implements eventual consistency.

In this case eventual consistency is guaranteed through a redelivery policy

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

combined with an *Idempotent Receiver*

The two flavors of eventual consistency are both ultimately based on the fact that they don't need 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 is no excuse for being clueless about transactions remember? Let's look at the details of transactions around endpoints.

WHERE IS MY TRANSACTION

In the previous chapter 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 will 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 have 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 simply by using the right channel in between.

A transaction is started by a poller just before it pulls a message from a `MessageSource`. To be precise the poller will only start a transaction 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 will participate in the same transaction as the downstream endpoints, as long as the boundary is not broken by an asynchronous hand-off.

Similarly, a poller equipped with a transaction manager will start a transaction before pulling a message from a`QueueChannel`. The business transaction in the downstream endpoint will be 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 that you need to keep in mind when designing a system.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

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

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

Although this chapter has been relatively technical in nature, much of the discussion thus far has been theoretical. Even when discussing concerns that are specific to Spring Integration, the examples have been mostly at the level of the namespace-based configuration. In the next section, we'll once again take a look "under the hood" so that you can learn a bit more about the implementation details of Message Endpoints, Polling Consumers, and Event-Driven Consumers. In other words, you'll have a better understanding of the types of components that are actually 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

In section 1 of this chapter you have read that Spring Integration can take care of polling concerns for you. We haven't explained in great detail how this works, and in daily life you don't need to worry about this. If ever you find yourself debugging a Spring Integration application, you might benefit from knowledge on how this works under the hood. Things *will* get technical now, and if you feel that you've seen enough at any point in the following section, feel free to step back into the drivers seat and continue with the last section and move on to Chapter 6. If you're not afraid to look inside, this section is for you.

The base class for all endpoints is `AbstractEndpoint`, and several

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

subtypes exist to take care of the differences between Polling and Event Driven behavior. `PollingConsumer` will wrap a `MessageHandler` and decorate it with a poller so that it can be connected to any `PollableChannel`. `EventDrivenConsumer` will simply wrap a `MessageHandler` and connect it to any `SubscribableChannel`. As you can imagine the latter is trivial. The `AbstractPollingEndpoint` is a bit more complicated, as it has to find a poller and handle exceptions asynchronously. Albeit a tad 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 as this is the place where the decisions between polling and event driven 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 that of 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 are working with than say a simple Spring "bean" element would be. Of course, you can always drop down to a lower level and configure everything as simple beans as long as you are familiar with the API. However, considering most Spring Integration users will rely to a large extent on the schemas defined in the various namespaces, we will include the role of parsers in our discussion here. Let's begin with the following simple 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 will lead to the creation

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

of an instance that implements the `SubscribableChannel` interface. That means that any endpoint referencing that as its input channel would be invoked directly when a sender sends a Message to that channel. The "channel2" element, on the other hand, will lead to the creation of an instance that implements the `PollableChannel` interface, and internally that channel will buffer 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. Now, each element supported by a Spring namespace handler is mapped to the implementation of Spring's `BeanDefinitionParser` that is responsible for working with that element. We'll 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`. That is the common base class for several endpoint parsers in Spring Integration. Its primary role is to handle the connection of `MessageHandler` objects to the correct input channels as defined in the configuration.

At parsing time, however, a limited amount of information is known. For example, when parsing "endpoint1" above, 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 will need to be injected with a reference to that channel. The endpoint parser does not know yet whether that is a pollable or subscribable channel. As a result the parser cannot know whether it needs to create a Polling or Event Driven Consumer. To handle that limitation, Spring Integration's parsers actually rely on another common Spring feature: the `FactoryBean` interface. Any object defined within a Spring context that implements that interface is treated differently than other objects. Instead of just instantiating the object and adding it to the context, when Spring encounters an object that implements the `FactoryBean` interface, it will create that factory and then invoke 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 at the time that getObject() is called when compared with the information available at parsing time. This is how Spring Integration's endpoint parsers avoid the limitation mentioned above.

The implementation in this case is called

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

`ConsumerEndpointFactoryBean`. As its name suggests, it is a generic creator of consumer endpoints. It is actually capable of producing either Polling or Event Driven Consumers. Since the actual 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. Therefore, if the referenced channel is pollable, `ConsumerEndpointFactoryBean` will create a `PollingConsumer` instance, but if the channel is subscribable, it will create 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. As described above, the distinction is based on the type of input channel: PollableChannel or SubscribableChannel, respectively. This becomes quite clear when we investigate the constructors for each of these objects. Both take a MessageHandler that will handle the Messages which a producer sends to the input channel, but the actual interface of the expected channel is different. The PollingConsumer expects a PollableChannel:

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

   ...
}
```

On the other hand, an EventDrivenConsumer expects a SubscribableChannel:

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

   ...
}
```

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

definition for an EventDrivenConsumer:

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

Several times now we have mentioned that the primary responsibility of these consumer endpoints is to "connect" the MessageHandler to an input channel, but what exactly does that mean? It actually has to do with the Lifecycle management of these components. As you can imagine, that is also a different issue for PollingConsumers and EventDrivenConsumers since the former requires an active poller while the latter is passive. We'll end our "under the hood" exploration with a quick investigation of exactly what "Lifecycle management" means in the context of these two consumer types.

First, we should point out that the foundation for managing the lifecycle of any components in Spring Integration is the Spring Framework itself. The core framework defines a `Lifecycle` interface:

```
public interface Lifecycle {

    void start();

    void stop();

    boolean isRunning();

}
```

As you can see, there are methods for starting and stopping a component. For an EventDrivenConsumer, the start operation simply 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 a bit more complicated. The start operation of a PollingConsumer schedules the poller task with the Trigger that has been 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 simply cancels the poller task so that it stops running.

You may be asking "who is responsible for calling these lifecycle methods?" Well, stopping is a bit more straightforward than starting, because when a Spring ApplicationContext is closing, any Lifecycle component whose 'isRunning' method

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

returns 'true' will be stopped automatically. When it comes to starting, Spring 3.0 added an extension to the simple Lifecycle interface called SmartLifecycle. That 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 will all return a value of 'true' from the 'isAutoStartup' method. Also, the phase values are such that any EventDrivenConsumer will be started (subscribed to its input channel) prior to any PollingConsumers being activated. That's important since PollingConsumers often produce reply Messages that may flow downstream to EventDrivenConsumers.

4.4 Summary

This chapter dove pretty deep 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. However, the information you learned here should definitely provide a strong foundation for understanding how those higher level components really 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. You also know that the lifecycle of an endpoint consists of managing the connection between the MessageHandler and the MessageChannel where Messages will be received. In fact, 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 about endpoints in this chapter and about messages and channels in the previous chapter, we're ready to move to a higher level. In the chapters that follow, you will learn about several components that map to the Enterprise Integration Patterns. These include Routers, Splitters, Aggregators, and a wide variety of Channel Adapters. We'll begin this journey by considering the role of Spring Integration within the larger context of a real-world application.

It has already been mentioned several times that Spring Integration promotes a clean separation between "integration" and "business" concerns. Nevertheless, those business concerns are usually the most critical part of an application. Indeed, it is the recognition of that fact that motivates Spring Integration and the Spring

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Framework itself. It is the goal of these frameworks to allow developers to focus on implementing the business functionality within their application's domain rather than spending excessive time on infrastructure. The next chapter will provide several examples of that within the flight management domain of our sample application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

5 Getting Down to Business

In the preceding chapters, you have been introduced to the big picture of Spring Integration. You learned that this framework was originally conceived as a convergence of the Enterprise Integration Patterns with the Spring programming model. You then reviewed some core principles relevant to enterprise integration including the power of event-driven, asynchronous invocation and the importance of loose coupling. Finally, you walked through the top level of the Spring Integration API, including the three key components of the framework: Message, Message Channel, and Message Endpoint. In the process, we should have given you a greater appreciation of how those three components promote loose coupling and therefore contribute to successful implementations of enterprise integration applications. Now it's time to turn to another key ingredient for achieving success: separation of concerns.

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 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 that their applications are portable across different environments without requiring code changes.

Spring Integration follows this same trend by supporting a clean separation of integration concerns from the particular business domain of an application. It provides a non-invasive framework for supporting message driven interactions on top of a business service layer. Up until this point in the book, we have basically

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

focused on the integration concerns. We have explored the construction of messages, different types of message channels, and the behavior of both polling and event-driven consumers. You have seen a few glimpses of how these channels are connected to services, but now we are 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. You will also learn how to connect the business services of an application to the messaging components provided by the Spring Integration framework.

5.1 Domain Driven Transformation

A common requirement for enterprise integration is to transform inbound data into the format that is expected by the consumer of that data. Syntactic transformations like converting a String representation of XML into a DOM Document object can be handled by a generic transformer that builds on existing libraries. Many structural transformations can even be handled by generic transformers such as those that rely on XSLT or XPath expressions. However, things are different when we are dealing with *semantic* transformation. At that point, we must actually focus on content within the business domain of an application. Since the consumer of such content is a business service, it's quite likely that the expected format is a domain object as opposed to something generic like XML or a byte array.

Let's begin with a very 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 CSV or XML. Figure 5.1 depicts schematically how different transformations can morph the `Flight` domain object into different interoperable formats.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 5.1 Three different representations of the flight object. Separate transformations (T, and), transform a flight domain object into CSV, JSON and XML representations respectively

Any of those formats are portable across any number of environments or programming languages. That is a good thing since 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. That means exchanging this data will not 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 will elaborate on both.

5.1.1 Marshalling Flight information

All of this sounds good so far, but 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:

```
public class Flight {
    private String number;
    private Date scheduledDeparture;
    private Airport origin;
    private Airport destination;
    private Equipment aircraft;
    private Crew crew;
    private Map<Seat, Passenger> passengers;

    // getters, setters omitted
}
```

This means that not only do we need the ability to transform domain objects into intermediate formats, we also need to transform in the other direction. Typically transformations as the ones shown in figure 5.1 will implement both

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

marshalling and unmarshalling functionality.

The definition of a Flight class above provides the various properties that one would expect. Those include the flight's number, the origin and destination, the scheduled departure and arrival times, the passenger seat assignments, and even a reference to the crew. Together, all of these objects comprise the *domain* of flight management. Everything you see above is related directly to the business needs of the application. There are no integration concerns mixed in with the domain model or vice-versa. After all, as emphasized in the introduction to this chapter, our 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 detail 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 above flight class:

```
public class Flight {
    private String number;
    private Date scheduledDeparture;
    private Airport origin;
    private Airport destination;
    private Equipment aircraft;
    private Crew crew;
    private Map<Seat, Passenger> passengers;

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

That one minor change may seem well justified since, after all, XML is the language of interoperability. There may be several different users of the flight domain that could benefit from the convenience of this addition. Now they can easily create an XML representation of the flight data in order to send the data 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 the fact that a *fromXml* method is an immediately obvious next step and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

then a pair of *toJson* and *fromJson* methods would be justified based on the XML precedent. The next thing you know, your domain model has been polluted with so many integration details that it is harder 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 need the XML representation while others require JSON. 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

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

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 of the other's different representation or interpretation of the domain. This is yet another excellent example of the power of loose coupling.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The XML example captures what is probably the most common approach for representing data in an interoperable way. However, sometimes even the XML representation would be overkill for a particular use case. The external system that is sending flight data into our application might not itself be concerned with or aware of much of that data at all, and in such cases, it might be possible to use a far simpler representation of the data involved. After all, 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 that 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 one hour and 30 minutes. On the receiving end, however, the FlightStatusService might be expecting a Flight object and an actual Date representation for the new estimated departure. The interface definition and the class it depends on might look like this:

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

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

One may be tempted to add another method to the interface that accepts this simple format from external systems. However, as described above with regard to the XML transformation, that is the type of approach that 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 these types of 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

systems may change frequently and new systems may be added that use different formats. What we need is a dedicated component that effectively "normalizes" the data into the canonical format expected by the business service layer.

Listing 5.1

```
@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.MINUTES, minutes);
        return new FlightDelayEvent(flight, cal.getTime());
    }
}
```

Figure 5.3 shows how the content enricher is collaborating with a repository in order to convert a simple string to a domain object.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Clearly this transformer goes beyond syntactical and structural changes. It depends on a FlightScheduler service itself to actually lookup the Flight domain object for the given flight number. Assuming we 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. That provides a couple benefits. First, it makes the transformer code easier to test since we 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 that will exist in the business service layer. We will see each of these benefits as we continue to develop the example.

WIRING THE COMPONENTS TOGETHER

Now that we have an understanding of the code, let's turn to the configuration. You may have noticed that the constructor has been 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 our application will only have a single instance that implements the `FlightScheduler` interface at any one time, that annotation will be sufficient for providing that dependency to the transformer.

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

Now you may be asking what the `@Component` annotation has to do with this `@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 actually a simple technique. Basically, by adding the `@Component` annotation to another annotation (hence the "meta" designation), components containing that second annotation will be recognized by Spring in the same way they would have been if annotated with `@Component` directly. Rather than just annotating all such classes with the generic `@Component` annotation, a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

more specific one can be used. Even within the Spring Framework itself there are more specific stereotype annotations, such as `@Repository` and `@Service`. The specific annotation, like @MessageEndpoint, can better describe the *role* of the annotated component, and that is why these are referred to as 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`. We don't need to worry about the actual production version of the `FlightScheduler`, since we just want to make sure that the transformer is behaving as expected. The following code depicts a sufficient stub implementation.

```
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 we'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 go ahead and create the configuration. We need to define a bean for the stub implementation above as well as two channels. We also need to enable the component-scanning so that the `@MessageEndpoint` annotation will be recognized on the transformer class.

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

<context:component-scan
    base-package="siia.integration.transformers"/>
<beans:bean class="siia.testing.StubFlightScheduler"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

TESTING THE TRANSFORMER

Now we 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 if that event has an estimatedDeparture property that correctly reflects the delay. Listing #.# provides the full test class example.

Listing 5.2

```
@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 will confirm 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, since the exact line of the test failure will indicate what the actual problem was. You really 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Endpoint that involves sending and receiving Messages across Message Channels. In case you are not familiar with Spring's integration testing support, we'll provide a very quick overview by describing the above test class. If you would like to delve into more detail, refer to the Spring Framework reference manual's testing chapter.

The `@Test` annotation is actually a JUnit 4 annotation and indicates that a particular method provides a test case. This is a nice improvement over the JUnit 3 style where methods had to begin with "test". Another nice change in JUnit 4 is that it is 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 is still possible to extend the testing framework, however, by providing a customized implementation of the test runner itself. This is how Spring provides its integration testing support for a JUnit 4 environment. In the example above, you can see that Spring's test runner class is provided within JUnit's `@RunWith` annotation.

The other class-level annotation, `@ContextConfiguration`, is actually defined by Spring's integration test framework. That 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 above would be named `FlightDelayTransformerIntegrationTests-context.xml`. That file 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 since a resolution based on type alone would lead to ambiguities in this case.

The previous sections have shown you how to marshal and unmarshal objects to and from different formats. You have 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

5.1.2 Content Enricher

Another common requirement is to add content to an existing domain object by invoking a business service. For example, with our flight booking example, we might need to add the email address for a given passenger. Imagine that we are able to uniquely identify the passenger based on a frequent flyer program number and also lookup the email address from the frequent flyer information service. An implementation of this logic within a transformer could be considered a "content enricher". The email address is the enriched content being added to the payload of a Message that did not yet include that information.

In fact, if a user is enrolled in the frequent flyer program, we 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. Of course, the information would probably include mileage credits and status.

The main point is that a `Passenger` instance may or may not contain all of this data. It certainly is not a requirement to be enrolled in the frequent flyer program in order to fly on the airline. So, a minimal `Passenger` instance may only contain the person's name and some form of official identification, such as a passport number, driver's license number, or tax id. Our content enricher will add the extra information if available, but pass on unaffected `Passenger` instances otherwise.

Listing 5.3

```
@MessageEndpoint
public class PassengerProfileEnricher {

    private final FrequentFlyerService frequentFlyerService;

    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;
    }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
}
```

The code in listing #.# is fairly straightforward. As with the previous examples, it would 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 would not 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 will be very clear where the corresponding code change needs to occur. Likewise, if a new service for profile information is released in the future, it would 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.

5.1.3 Header Enricher

Business services are dealing with the payload of a Message, and often that payload can be considered a "document". On the other hand, Channel Adapters and Messaging Gateways that are used to connect external systems might be interested in information that is carried in a header of a Message.

As an example, consider an adapter that sends the flight delay notifications to any passenger who has signed up for such notifications. Continuing with the example from the previous section, let's assume that the email address is available on the payload, which is an instance of the Passenger domain object, because that had been added by the Content Enricher shown above. Basically, if a passenger is enrolled in the frequent flyer program, they 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 any detail on the actual adapter that sends the emails, because we will be covering that (and continuing this example) in another chapter later in the book. For now, it is sufficient to know that the email-sending adapter is expecting the target email address to be provided in a header. The email adapter could advertise that the header name should be "si_mail_to". It is then the responsibility of an upstream component to populate that header with the correct value.

Of course we could implement this header-enriching transformer in code. All we need to do is receive the payload, which is a `Passenger` instance in this case, then grab the email address if available so that it may be added to the map of

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

headers. The code might look like this:

Listing 5.4

```
public class EmailHeaderEnricher {

    public Passenger populateEmailHeader(Passenger passenger,
                        @Headers Map<String, Object> headers) {
        Profile profile = passenger.getProfile();
        if (profile != null) {
            String emailAddress = profile.getEmailAddress();
            if (emailAddress != null) {
                  headers.put(MailHeaders.TO, emailAddress);
            }
        }
        return passenger;
    }

}
```

Again, don't worry about the details for the downstream mail sender. We will cover that later in the mail integration chapter, but the intention of the code above should be clear nonetheless. If the passenger represented by the current Message's payload does have an email address on file, that will be added to the headers. That way, some generic email sending adapter that will eventually receive this message does not need to know how to extract the email address from every possible type of Message payload that it may receive. In other words, the adapter specifies a contract: "I will send a mail IF the `MailHeaders.TO` header has been populated with a valid email address". If the fulfillment of the obligation of that contract is the responsibility of a separate component, the mail adapter is highly reusable. For example, the exact same adapter could be used for sending email to banking customers in an application designed for that domain.

Now, you might be thinking that this seems like a lot of code to write merely for the benefit of a reusable mail sending adapter. That is a valid point, and it also highlights one of the key tradeoffs involved in enterprise integration and messaging application design. We've discussed the benefits of loose-coupling at length, but what we're seeing here is the initial indication that there are some negative consequences. As with most principles, if taken to extremes, the costs may outweigh the benefits. In other words, 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. However, testing and maintaining such a system would be prohibitively difficult. Indeed, it would probably be much

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 #.# 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 provides a great environment for exactly that. The decision of which language to use for a particular task should be based on the nature of that task. It just so happens that many of these integration tasks are a good fit for dynamic languages.

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

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

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

Since expressions are stored in strings, and in this case inside the XML configuration, it makes it all too easy to "program" in XML. Logic stored in this manner is much harder to test and refactor. For these reasons, you should 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 will be used to type safety and relying on good tools and the compiler to warn you of mistakes. These warnings will not be raised when you use the expression language.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Now that you can consider yourself armed (and warned) with knowledge of various types of transformers, including the expression language support, it is time to really get down to business. While 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 will dive into the details of how to accomplish that with Spring Integration.

5.2 Message Driven Services

In the previous section, we converted data from a simple text representation into a FlightDelayEvent object that is part of the domain model. We also briefly reviewed the FlightStatusService interface that would expect such an object. Now we can look at the implementation of that service and see how it would be invoked from the integration layer. The component responsible for that invocation is a Service Activator.

Recall that we 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="com.manning.siia.SimpleFlightStatusService"/>
```

If you have ever worked with the Spring Framework's JMS support, and specifically the "message-driven POJO" feature, the configuration above should look quite familiar. In fact, 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. However, 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 has been connected to that channel.

You may also have noticed that the service does define a return value and so there is a corresponding "output-channel" declared on the Service Activator

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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.

In fact, it is very 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.

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.

As we'll 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.

5.2.1 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 does not expect a return value. In such cases, there would be an ultimate receiver at the other end of the chain. Either the final service to be invoked does not return a value or the chain ends with a Channel Adapter such as a File writer. However, there are other use-cases where 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 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();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Before going any further with this example, we should point out that you will probably never write code like that featured above, since Spring Integration provides support for the reply channel configuration behind the scenes. You will see an example of this in the upcoming section on the Messaging Gateway support. For now, we simply want to consider the implications of the reply channel header for a Service Activator. If the "requestChannel" above 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, we would 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="com.manning.siia.SimpleFlightStatusService"/>
```

The configuration is almost identical to that in the previous version, and even more importantly, there are no changes required to the service code. Switching between a "chained services" and request/reply interaction model is as simple as including or omitting the output channel from the configuration, respectively. Keep in mind, however, that if a service does return a value, and neither the output channel has been configured nor the reply channel header has been provided, an Exception would be thrown at runtime, since the Service Activator component would not know where to send the return value.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

5.3 Message Publishing Interceptors

Thus far in this chapter, we have been exploring the components that consume messages, or more commonly just the content of message payloads. Within the context of a business service, these consumers may also return values that are then published as response messages. In the previous section, we covered the details of how these response messages can be sent to either the endpoint's output-channel or a "return address" header that has been provided with the incoming message for that purpose. However, integrating a messaging system with an application's business layer should not be limited to components that react to messages. It is also quite useful to have business services within the application publish messages based on certain events that occur *within* the business services themselves. A typical example would be 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's aspect-oriented programming technique. A proxy can be generated to intercept method invocations on services that have no direct awareness of the messaging system themselves. 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 our system needs to perform some auditing every time a flight's status is updated. Perhaps we're responsible for gathering these statistics in order to provide data about the percentage of flights that actually depart on schedule. Of course we could write some code that publishes the message directly and then add that code to our FlightStatusService.

```
public class SimpleFlightStatusService interface 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);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
    }

}
```

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

For that reason, Spring Integration provides a cleaner way to handle such requirements. As with any application of aspect oriented programming, the goal is to modularize that functionality so that we are not mixing cross-cutting concerns, such as messaging, with the core business logic. The key is to apply the non-invasive technique of interception. Also, as with many of the other Spring solutions that rely on aspect-oriented programming, such as transactions, this behavior can be enabled by the use of a simple annotation.

```
public class SimpleFlightStatusService interface FlightStatusService {

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

}
```

As you can see, this is much better since the code is focused only on the business logic. There is no need to worry about the cross-cutting messaging concerns when testing or maintaining the code. Likewise, the message publishing code can itself 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 non-invasive 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. However interception is only really useful for reactive use-cases where the publication of a message is considered a *by-product* of some other primary action. There are other equally valid use-cases where publishing a message is itself the primary action, and for those cases, a more proactive technique fits well.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

```
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 do we claim that the above example is less than ideal? There are a number of reasons. First, consider what we would need to do if we encounter a new requirement to add another destination for these events, such as the local file system. We would need to refactor the code to handle that. The revised implementation might look something like this:

```
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
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
        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
    }
}
```

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

We left out the full implementation details of the "writeFile" method but purposefully provided three separate comments to describe what such an implementation would entail. As you will see in the chapter covering Files, each of those steps can even be addressed by separate components or strategies when using Spring Integration's File adapters. As for the mail sending and JMS publishing, the examples above assume usage of the corresponding support in Spring. In both cases though, there is still a dependency on the 'mailSender' and 'jmsTemplate' objects within the code. As you will learn later in the book, Spring Integration builds upon 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 will 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="notifications"
          interface-name="com.manning.siia.FlightStatusNotificationPublisher"/>

<mail:outbound-channel-adapter channel="flightStatusNotifications"
                                mail-sender="mailSender"/>

<jms:outbound-channel-adapter channel="flightStatusNotifications"
                              destination="flightStatusQueue"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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, the configuration above 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 does not 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 we need to add or remove subscribers, it's simply a matter of inserting or deleting the corresponding element. As far as transformation is concerned, it is also treated as a separate concern. As you can see above, the "file-to-string-transformer" element precedes the outbound file adapter. If you need to transform to a byte array instead, you would simply 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 will learn more about each of these adapters and their associated transformers in upcoming chapters.

The one element that 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 have ever worked with Spring's support for remoting via RMI or Spring's own HTTP invoker mechanism, this concept will be familiar. The element is actually triggering the registration of a Spring FactoryBean that produces a dynamic implementation of the declared interface. In this case, the proxy will be 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 "non-ideal" 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 the amount of "in-house" code to be tested considerably. In fact, the only code from the above example that is not part of the framework is the interface. This means that all responsibility for testing the internal functionality of the subscription side actually belongs to the Spring Integration development team. You would likely want to have some level of integration testing to verify the that Mail sender, JMS

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

queue, and File system directory have been configured properly. However, in terms of *unit* testing, the focus can shift to the publishing side. Since that is now reduced to the invocation of a method on an interface, it should be quite easy to build a battery of tests using traditional mock or stub techniques.

5.5 Managing Business Transactions

In the previous chapters you've learned the precise definitions of a channel and an endpoint in Spring Integration. Also it has been mentioned a few times that there is a risk of breaking transactional boundaries and losing user context when processing messages concurrently. In this section we will discuss the details of this as it pertains to a transaction context.

TODO: continue...

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

5.6 Chaining Endpoints

In many cases messages flow linearly through the system from endpoint to endpoint, separated by direct channels. In these cases it would be bad design to

Figure 5.5 Endpoints can be chained together with channels that are implicitly created using a chain

As discussed before synchronous channels are used by default to preserve transactional boundaries. In a chain the synchronous channels need not be specified, but there is 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 it is needed.

We have seen several examples in this chapter so far where a single component is 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:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

<transformer input-channel="passengers"
                output-channel="passengersWithProfile"
    ref="frequentFlyerService" method="configureProfile"/>

<channel id="passengersWithProfile"/>

<header-enricher input-channel="passengersWithProfile"
                    output-channel="flightDelays">
   <header name="#{MailHeaders.TO}" value="#{payload.emailAddress}"/>
</header-enricher>

<channel="flightDelays"/>

<mail:outbound-channel-adapter channel="flightDelays" mail-sender="mailSender"/>
```

That configuration has a lot of unnecessary noise since there are many channels that are only being used once, and many endpoints configured to connect to those channels. Even if we rely on Spring Integration's creation of default channels for any single component's "input-channel" attribute, the configuration is still a bit noisy:

```
<transformer input-channel="passengers"
                            output-channel="passengersWithProfile"
    ref="frequentFlyerService" method="configureProfile"/>

<header-enricher input-channel="passengersWithProfile"
                    output-channel="flightDelays">
   <header name="#{MailHeaders.TO}" value="#{payload.emailAddress}"/>
</header-enricher>

<mail:outbound-channel-adapter channel="flightDelays" mail-sender="mailSender"/>
```

That allowed us to remove the 'channel' elements, but this actually might be even harder to follow when reading the configuration directly. Now, we have to look at the various "input-channel" values that match with "output-channels". If the endpoints were listed out of order, it would be rather confusing. The "chain" element offers a cleaner solution to this. 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 that the chain element provides.

```
<chain input-channel="passengers">
    <transformer ref="frequentFlyerService" method="configureProfile"/>
    <header-enricher>
        <header name="#{MailHeaders.TO}" value="#{payload.emailAddress}"/>
    </header-enricher>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
    <mail:outbound-channel-adapter mail-sender="mailSender"/>
</chain>
```

Figure 5.6 The frequentFlierService 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 is not 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 could not 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 clearly would not fit in any other position within a chain, because all endpoints before the last one essentially have a fixed output channel.

5.7 Summary

In this chapter, we have explored a number of ways that 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 while 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 *non-invasive* programming model, since the methods to be adapted on the POJOs do not need to operate with Messages. Instead, those method's 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 very little effort. As a result, Spring Integration is easy to adopt incrementally. Even more importantly, whether

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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, while 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 have all been relatively simple. Even when multiple steps have been necessary, such as transforming content prior to invoking a service, they have been in the form of linear "pipelines". You even learned how Spring Integration provides support for such a pipeline with the XML configuration of a handler *chain*. However, sometimes the interaction with the business layer is not 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 has been added to the message. In the next chapter, we will explore Spring Integration's support for addressing such requirements.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

6 Go beyond sequential processing: routing and filtering

Until now, you have seen how to use Spring Integration for creating an application from a group of processing units. For that, you need to decide on 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 is 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 very well even for simple, sequential, processing models. There is, though, another form of control that you are going to read about in this chapter, which goes beyond the conveyor-belt model of sequential processing: how to selectively process messages and define alternate routes within the system.

First, you will 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, and we will discuss the benefits of selective processing. When your application provides alternative (or complementary) processing options for your messages, you can make use of routers to let the system decide what should be the actual destination of a message, thus freeing message producers from the decision process. This allows more flexibility in the system, as your components become more reusable.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

6.1 Do you want to get this Message?

Figure 6.1 A Message Filter evaluates incoming messages. Only A's are published to the next channel, while B's and C's are discarded.

In order 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 TBD, 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.

Of course, the decision whether a certain component can process certain types of messages can be taken by the component itself. For example, the POJO implementation of Service Activator can decide what messages are valid are or not. Embedding this decision into the component works well if it is made based on a very general principle, that is applicable everywhere that component is used. Filters are useful wherever the filtering condition is based on an application-specific requirement, that can change more often than the Java implementation of the business service.

6.1.1 Filtering out messages

The application can receive cancellation requests through a gateway, and they will be 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, as the conditions may be different (refund policies, advance notice requirements, etc).

Our cancellations processor knows only how to process 'Gold' cancellations, so we can discard everything else, as in the following sample:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<gateway id="cancellationsGateway"
         service-interface="c.m.siia.f.i.CancellationsGateway"
         default-request-channel="inboundCancellations"/>

<filter id="cancellationsFilter" input-channel="inboundCancellations"
        ref="cancellationsFilterBean" method="accept"
        output-channel="validatedCancellations"/>

<bean id="cancelationsFilterBean" class="c.m.siia.f.i.CancellationRequestFilter>
   <property name="pattern" value="GOLD[A-Z0-9]{6}+"/>
</bean>

<service-activator id="goldCancellationsProcessor"
        input-channel="inboundCancellations" ref="cancellationsService"
        method="cancel" output-channel="confirmedCancellations"/>
```

Please take note of the `<i:filter/>` element, that we use for inserting the filter between the gateway and the service activator that processes the cancellation requests. From a syntax perspective, its definition is not very different from the other message handlers we encountered so far - like a transformer or 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. It is, though, semantically different. Messages do not suffer any transformation when they pass through this component - the very 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 our 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 (there are a lot of things which can be wrong with the request itself and dealing with that is the responsibility of the cancellation service). But, at a minimum, we know that reservation codes have to conform to a certain standard pattern, and we can do a quick test for that. So, here's how the implementation of the filtering class looks like:

```
public class CancellationRequestFilter {
  private Pattern pattern;

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

As well as with the service activator, or the transformer, you can implement the actual logic in a POJO (and we strongly recommend you to do so). The filtering logic consists of a method that takes as argument the payload of a message (as you have seen in the sample above), one or more message headers (by using the `@Header/@Headers` annotation as you have seen before) or even a full-fledged message, and returns a boolean which indicates whether the message will pass through.

WHERE DO REJECTED MESSAGES GO?

In the simplest case, which is also the default, they are just discarded (or for a Unix-based analogy, `/dev/null`). If that is not what you want, you have two other options that you can use:

- 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 - it should be noted that, from the Spoint 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, in the case when both options are active, the message will be sent on the discarded messages channel before the exception is actually thrown (a detail which is important when a synchronous channel strategy is in place).

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

```
<filter id="cancellationsFilter" input-channel="inboundCancellations"
        discard-channel="rejectedCancellations"
        ref="cancellationsFilterBean" method="accept"
        output-channel="validatedCancellations"/>

<bean id="cancelationsFilterBean"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
      class="com.manning.siia.flight.integration.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 .../>
```

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

You have just seen how easy it is to implement the filtering logic as a POJO. In a lot of common cases, however, 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 is computed against the payload or the header values, you can use the Spring 3.0 expression language directly. So, instead of defining a distinct `CancelationRequestFilter`, you can get the exact 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"/>
```

And thus there is no need for a distinct bean to implement the decision logic.

Having two options on hand, when should one create a distinct implementation, and when should one use a inline expression? An inline expression is simple enough, and could be externalized easily (using for example a `PropertyPlaceholderConfigurer`). But it is not very flexible or reusable across the application. So, 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 it requires to consult other collaborating components, you may want to create a standalone implementation and take advantage of Spring's dependency injection capabilities.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

6.1.2 Using filters for selective processing

satisfy the criteria
be simpler if their
at they won't be

ubscribe channels,
to a single source
interested in. Such
essage at the same
ocess or not.

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

In the figure TBD, we have such an example. Three different components, with different interests subscribe to the same channel. It is 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 or not to a next channel. If you decide to configure them with a channel for discarding messages, a Message Filter will act as a switch, choosing one channel or another depending whether the message is accepted or discarded. In that case, they are a simplified form of a more general type of component, that can decide where a message should go next, from a number of multiple destinations.This component is the Message Router, which is the focus of the second part of the chapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

6.2 Whose message is this, anyway?

In Spring Integration, channels play a crucial role not only for propagating messages across the system, but also because they are, by design, the only components of the framework which 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) of the channel on which it should send a message, while the actual destination of the message and its processing sequence are determined by the context configuration. In other words, for having a message processed correctly, it should be sent on an appropriate channel.

As systems become more complex, determining which is the appropriate channel for a message to go next becomes significantly more complicated. But this knowledge is application-specific, so it would be 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. Also, the message-publishing components do not need to know anything about what follows downstream - the only information that they require is which channel should all of them use for publishing messages.

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

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

The aphorism we quoted above has a second part, which is very often ommitted: "but that usually will create another problem". When using a router, its configuration must be aware of all the possible destination channels, so that it can make correct decisions. This may create another problem, as the saying goes, since

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

the configuration of the router must be updated every time the routing logic changes. In practice, though, centralizing the configuration in a single place is, in general, a better choice than spreading the knowledge across the system. If a more dynamic approach is necessary, a routing alternative using publish-subscribe channels and filters is available as well.

6.2.1 Configuring routers

In our SIVA application, customers must, at some point, pay for their trips. But there are so many payment methods that 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 available to certain, trusted customers. Each of these payment methods requires some specific information (an account number, a billing address, etc.) and performs a different actual operation.

So, 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 the latter case the payment notification is not sent by the web application, but by an external process (like a nightly batch, processing transactions overnight).

The individual processing strategies are mutually exclusive alternatives, and it makes sense to provide an individual channel for each of them, thus ensuring that each payment notification is forwarded to the appropriate payment processor. So, now it is 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.

But in this case, rather than giving each component access to the complete list of channels and have them decide where should they post the next message, it makes much more sense to introduce a router. Now all payments are sent to a single channel, from where they will be processed by the router, which will forward them to the appropriate target destination.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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

The figure above corresponds to the following configuration:

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

<si:router  method="routePaymentSettlement" input-channel="payments">
   <bean class="com.manning.siia.integration.routing.PaymentSettlementRouter"/>
</si:router>
```

The router receives messages from the 'payments' channel and invokes the 'routePaymentSettlement' method for deciding what is the next destination for the message. In our application, The `PaymentSettlementRouter` can, for example, be implemented as follows:

```
public abstract class PaymentSettlement {/*...*/}
public class CreditCardPayment extends PaymentSettlement {/*...*/}
public class Invoice extends PaymentSettlement {/*...*/}
public class Invoice extends PaypalPayment {/*...*/}
public class PaymentSettlementRouter {
  public String routePaymentSettlement(PaymentSettlement paymentSettlement){
    String destinationChannel = null;
    if (paymentSettlement instanceof CreditCardPayment)
      destinationChannel  = "credit-card-payments";
    else if (paymentSettlement instanceof Invoice)
      destinationChannel = "invoices";
    else if (paymentSettlement instanceof PaypalPayment)
      destinationChannel = "paypal-payments";

    return destinationChannel;
  }
}
```

The method will return the name of the channel on which the message will be forwarded next, and returning a null means that the message will not be forwarded

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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, and looks like this:

```
public class PaymentManager {
  public void processPayment(Invoice invoice) { //... }
  public void processPayment(CreditCardPayment creditCardPayment) { //... }
  public void processPayment(PaypalPayment payment) { //... }
}
```

Using this class, we can configure a Service Activator to process the messages:

```
<si:channel id="payments"/>
<si:service-activator input-channel="payments" method="processPayment">
  <bean class="com.manning.siia.domain.payment.PaymentManager"/>
</si:service-activator>
```

This strategy does not require a router, because every time an object is sent to the Service Activator, the handler can determine which is the appropriate method it should call, based on the type of the payload and the parameter types of the overloaded methods.

How does this compare with introducing a router? For answering this question, we must remember our primary goals: low coupling and easiness to extend the application. The PaymentManager is a passable solution, and it works well only if the number of options is fixed and known in advance.It's easy to imagine what happens if we need to add another way of paying the bills: presumably, we will introduce a new payment type, and a new method to handle it. Let's assume that we want to handle DirectDebitPayment as well. In this case, the PaymentManager will have to change by adding a new method:

```
public class PaymentManager {
  // all methods previously shown
  public void processPayment{DirectDebitPayment payment) { //... }
}
```

In comparison, the router-based method will allow you to expand the application without modifying anything which exists already (besides, of course, of the application configuration). What you need to do is to add a channel:

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

And, if you are using the most basic router shown above you have to modify the routing logic accordingly:

```
public class PaymentSettlementRouter {
  public String routePaymentSettlement(PaymentSettlement paymentSettlement){
    String destinationChannel = null;
    /* ommitted code shown previously */
    if (paymentSettlement instanceof PaypalPayment)
       destinationChannel = "direct-debit-payments";
    return destinationChannel;
    /* ommitted coded shown previously */
    }
}
```

You may be wondering what do we gain here, if all we do is to trade a class change for another. For one thing, the routing logic is just a thin layer that has no dependency nor direct interaction with the business logic, contrasting with the the PaymentManager which could easily result in developing a 'God class' that does too much and has too many dependency.

But one of the advantages of using routing is that you can get away without implementing a new router class, like we did in our first and most basic example that you have seen in this section. Spring Integration comes with a wide variety of routers provided out of the box, and this is what you will read about in the next section.

6.2.2 Routers provided by the framework

In our example, you have seen a router at work. In that particular case, the decision was made based on the type of the payment, which is also the type of the payload of payment message. Using the terminology of the Enterprise Applications Integration book, they all are *Content-Based Routers*. This means that the decision on where to send the message next is solely based on the contents of the message.

DEFAULT ROUTERS

The PaymentSettlement router you have seen above is a an example of using a default router. The simplest router is created by defining a <router/> element. You have just seen an example for it. When you define a router, you should bear in mind that:

- it must describe how should the routing decision is made - in other words, indicate who is responsible for evaluating the message and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

determining the next channel; and

- 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. Our previous router example has shown how to implement a router using a POJO, so let's discuss

In order to be usable for the routing logic, a method definition must comply, in what concerns its arguments, to the same general requirements as for 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 MessageChannel, or a String representing a message channel name, or a Collection of Strings or MessageChannels, or an array of Strings or MessageChannels. The latter are supported because routers can return multiple values, and in the next section you will see an example of how that works.

If the method returns channel names, they will need to be converted into channel instances. For that, you need to provide a ChannelResolver that does this job. If you don't want to provide one, the framework will provide one for you by default, that looks up channels by their ids in the ApplicationContext. Using a ChannelResolver or not largely depends on how keen you are on using actual channel names inside your routing logic.

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

PAYLOAD TYPE ROUTERS

The router that we implemented for `PaymentSettlement` instances takes into account the actual type of the object for deciding the next target channel. In our router implementation, we have checked that ourselves, but instead of doing that, we could have just used an implemenation provided by the framework. So, we could have just written:

```
<si:channel id="payments"/>
<si:payload-type-router>
  <si:mapping type="com.manning.siia.domain.payment.CreditCardPayment"
            channel="credit-card-payments"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
  <si:mapping type="com.manning.siia.domain.payment.Invoice"
            channel="invoices"/>
  <si:mapping type="com.manning.siia.domain.payment.PaypalPayment"
            channel="paypal-payments"/>
</si:payload-type-router>
<si:channel id="invoices"/>
<si:channel id="paypal-payments"/>
<si:channel id="credit-card-payments"/>
```

As you can see in this example, whenever the next channel can be decided based on 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.

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

Here, it is the PAYMENT_PROCESSING_DESTINATION header that 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 it is added to the message by the framework. A Header Value Router pairs very well with a Header Enricher to populate the relevant header of the message.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

ROUTING BY SPEL EXPRESSIONS

If the routing decision can be made through a simple evaluation of the message, instead of creating a separate POJO implementation and delegate 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 we have seen previously, all credit card processing requests are routed to the same channel, 'credit-card-payments'. But different credit cards may be processed differently, as they will be handled by the different issuing organizations. We can extend the routing logic by adding another router, to deal with credit card payments specifically, as follows:

```
<si:channel id="credit-card-payments"/>
<si:router input-channel="credit-card-payments"
            expression="payload.creditCardType"/>
<si:channel id="visa"/>
<si:channel id="american-express"/>
<si:channel id="mastercard"/>
```

In this example, we don't use a separate class to implement the routing logic. Instead, we use a SpEL expression evaluated against the payload. In this particular case, we evaluate the `creditCardType` property of the payload (which is of the type `CreditCardPayment`). The outcome of the evaluation may be one of *visa, american-express,* or *mastercard*, which are also the names of the potential channels.

In all the examples that we have seen above, the routing logic returns the name of the actual 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, allowing to map the String values returned by the routing logic to actual 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 actual channel names, but 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 thechannel for VISA payments, whichever name it has").

So, 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

actual channel instance.

```
<si:channel id="visa-payments"/>
<si:channel id="amex-payments"/>
<si:channel id="mastercard-payments"/>
<si:router input-channel="credit-card-payments"
           expression="payload.creditCardType"
           channel-resolver="credit-card-channel-resolver"/>

<bean id="credit-card-channel-resolver"
      class="o.s.i.channel.MapBasedChannelResolver">
  <property name="channelMap">
     <map>
       <entry key="VISA" value-ref="visa-payments"/>
       <entry key="MASTERCARD" value-ref="mastercard-payments"/>
       <entry key="AMERICAN_EXPRESS" value-ref="amex-payments"/>
     </map>
  </property>
</bean>
```

In our 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.

All our examples so far have been based on the assumption that the routing scenario implies a single next destination channel. However, routing can also mean that a message can be resent to a number of other channels, as we will see next.

6.2.3 Routers with multiple destinations

Our payment example has shown how to use a router when you have one possible destination out of a few. This is the most common situation that you will encounter, but it is also perfectly possible to have more than one next destination channel for a message.

Let's consider a notification system. There may be different ways of notifying a customer, and different notification types. A simple weather update may be sent through e-mail, but more urgent notifications, such as a cancellation notice, may require sending an SMS, and an e-mail, and placing an automated phone call at the same time, in order to make sure that the customer is reached by all the means possible.

Since there is a single source of notifications, the notifications will be sent on a single channel. From there, a router will distribute them to the channels which

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

correspond to the individual notification strategies: SMS, email, voice. The difference from the payment scenario is that a notification can be sent by multiple channels simultaneously.

MULTIPLE RETURN VALUES

In our previous POJO-based example, the `routePaymentSettlement` method of the `PaymentSettlementRouter` has returned a single value. But this does not have to be always the case. Let's consider the following configuration for notifications:

```
<si:channel id="notifications"/>
<si:router input-channel="notifications" ref="notificationsRouter"
            method="routeNotifications">
</si:router>
<si:channel id="sms"/>
<si:channel id="email"/>
<si:channel id="phone"/>
```

Our intent here is to contact the client in as many ways the business logic requires. This means that routing logic has to take into account both the notification settings of the user, and how urgent the notification is. Receiving an automated phone call if there has been a cancellation is pretty sensible, but being called every time there's a sale going on doesn't fall into the same category. Both are notifications, and will be delivered initially on the 'notifications' channel, where they will be distribute by the router.

```
public class NotificationsRouter {
  public String[] routeNotification(FlightNotification notification) {
    List<String> notificationTypes = new ArrayList<String>();
    if (notification.priority == Priority.HIGH &&
         notification.recipient.getSettings().getPhoneNumber() != null) {
      notificationTypes.add("phone");
    }
    if (notification.priority == Priority.MEDIUM &&
         notification.recipient.getSettings().isSmsEnabled()) {
       notificationTypes.add("sms");
    }
    if (notification.priority == Priority.LOW &&
        notification.recipient.getSettings().getEmail() != null) {
      notificationTypes.add("email");
    }
    return notificationTypes.toArray(new String[0]);
  }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

In this case, the router will send 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 a factor to be taken into consideration).

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 what you have seen so far. Until now, we have seen routers making decisions about the next destination channel. In this case, though, the Recipient List Router is configured with a group of destination channels and will forward incoming messages to all of them.

You may ask at this point: what is that good for? After all, publishing to multiple destinations is the job of a publish-subscribe channel. This may be true when you are designing a system from the scratch, but the Recipient List Router works very well when the destination channels are already defined, and the target components are already listening to them. Another possible use of it is for broadcasting a message to multiple channel adapters.

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

With such a configuration, any message sent to the notifications channel will be automatically forwarded to all the channels defined on the recipient list of the Recipient List Router.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

6.3 Under the hood

The most common way of using filters and routers is through the namespace configuration. This section will dive a bit deeper into details and explain what happens underneath. It is optional reading, because knowing the internals of the framework is not a prerequisite for using it efficiently. As a matter of fact, a lot of effort of developing it goes into the direction of enriching the framework user's experience by using the namespace.

But there are two valid reasons to investigate what happens inside the framework and what are the main actors at play. For one thing, it helps with debugging. As well, the framework will provide a few out-of-the-box components which treat the most common use cases, and allows you to create problem-specific implementations (such as the ones that you will see in the chapter dedicated to XML handling). 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 are going to focus next.

6.3.1 The Message Filter API

The extension point of the Message Filter API is the MessageSelector. It is used in several places (including in the MessageSelectingInterceptor that you have seen in Chapter 4. It is 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 which is injected into the MessageFilter.

Figure 6.4 Sequence diagram of the MessageFilter: a MessageSelector's accept() method evaluates the message to decide if it is 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
public abstract class AbstractMessageRouter extends AbstractMessageHandler {
  public void setDefaultOutputChannel(MessageChannel defaultOutputChannel) {
```

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

The class provides two further configuration options:

to set up a default output channel, which will be used by the router to forward the messages if `determineTargetChannels` returns with no next destination channels

a flag which force sthe router to throw an exception if no next destination channel can be determined (otherwise said, if a resolution is required)

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

6.4 Summary

In this chapter we have moved past the simple sequential model of chaining message handlers, and we have started to deal with more complex configuration problems. You have seen 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 what is valid and not is not a domain-inherent characteristic, but can vary from application to application). 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 will receive only the messages that they are supposed to handle.

In order to handle a message correctly, a producer must send it to the appropriate channel. That is, to a channel that has the intended recipient as a consumer. It is 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 from the processing components, and set up through the application configuration. In this case, a router cam

You have seen 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 that all the routing configuration is in a single place (the router). We cannot 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 a better option. Otherwise, the centralized configuration provided by the router allows for much 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 and 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 undertake the task of dividing a message into its constituent parts, to be processed independently.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

7 Splitting and Aggregating Messages

In the previous chapters we have seen the details of how a single message is processed as a unit. We've seen channels, endpoints like service activators and transformers and routing. All the components have one thing in common: they don't break the unit of the message. If one message goes in either this message or a derivative comes out the other end or it is gone for ever. In this chapter we are going to look at situations where this doesn't holds. We are going to see situations where one message goes in and several messages come out (splitter) and situations where several messages can go in before messages start coming out (aggregator, resequencer).

The most important difference between the endpoints discussed in this chapter and the other endpoints in the book is that these endpoints have to be obviously stateful. An exception is the splitter, but this was added anyway so we can discuss its symmetry with the aggregator. I'm saying obviously stateful because some endpoints have some non obvious state in them for performance reasons. For example the file adapters hold a queue of files in memory to prevent costly file listings (see chapter 12).

As we've discussed before in chapter 1, correlating messages is sometimes essential to implement a certain business requirement. In this chapter we will see the details of the different options for this. Whether it is about reassembling a previously split up array of related data (Splitter - Aggregator) or making sure ordering constraints are enforced on messages (Resequencer) or if it's about splitting up work between specialized services (Scatter - Gather). In all these cases the correlation of the messages is key. Because of that it makes sense to discuss how Spring Integration stores and identifies the correlation of messages and how you can extend this. We will look at this in detail in the Under the hood section, for

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

now it is enough to understand the functional concept of correlating messages in a group.

7.1 Correlated messages

The aggregator in figure 7.1 will wait for messages in a certain group to arrive and then send out the aggregate.

Figure 7.1 An aggregator is an endpoint that combines several related messages into one message. In this diagram you can see an aggregator that waits for order items to come in. As soon as all order items for an order are received it sends an order to it's output channel.

The part we're interested in first is how it determines what messages belong together. Details will follow in later sections. In Spring Integration we use the concept of a *message group* that 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 something determined by looking at the message and it may differ between endpoints.

At this time it is good to look at an analogy from our daily lives, so we can anchor the important terms used in this chapter.

Let’s say you’re going to have guests for dinner. Maybe you’re not the cooking kind, but you are most likely familiar with the concept of home cooking and that’s more than enough. We shall look at the whole process of preparing and serving the meal and see what would be involved in terms of messaging to automate this. Just for the sake of the argument we will ignore the possibility of ordering food - which would surely decrease the complexity of the setup enormously. In the figure below you we've sketched the scenario that is described in the rest of the section.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 7.2 The flow of messages in the home cooking example. A Recipe is split into ingredients that are aggregated onto a shopping list. This shopping list is converted into bags filled with products in the supermarket. These bags are then split into products which are aggregated into a mis 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 will be the host, the kitchen, the guests and the shops. The host will orchestrate the whole thing, the guests will consume the end product and turn it into entertaining small talks and other things irrelevant to our story. The kitchen is the framework that the host uses to transform the ingredients he will get from the shop into the actual dinner.

The sequence of events after a date has been set is the part we're interested in.

It starts with the host selecting a menu and gathering the relevant recipes from his cook books or the internet. The ingredients needed for the recipes are then to be bought at the shops, but to buy them one at a time, with a roundtrip to the shop for each product is unreasonably inefficient, so the host needs to process them in a smarter way. The trick is then for the host to create a shopping list for each shop in advance. He does this by picking the ingredients from each recipe one by one and putting them on the appropriate shopping list. He then does a trip to each shop and delivers the ingredients back to his kitchen.

With the ingredients in the house each shopping bag needs to be unpacked and ingredients for the right dish put together. This is what professional cooks call the mis en place. Our host will do the same to impress his guests of course. Then each dish can be put together, this usually involves putting the ingredients together in some way in a large container. When the dish is done it is divided over plates and it can be served.

But what does this have to do with messaging? Well, it might be more than you think.

Let's say the recipe is a message. Then it would be fair to say this message is split into ingredients, which can be sent as messages by themselves. These ingredients (messages) then reach an aggregator that aggregates them onto the appropriate shopping list. 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 mis an 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

them at the table. What we've seen here is a lot of breaking apart and putting together of payloads. The recipes are split and the ingredients aggregated into 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 are helpful to think back on later in the chapter. Splitting is a relatively easy job. It is however important to keep track of which ingredient belongs to which recipe. This is called the correlation id. We're aggregating in a different configuration than we were originally splitting for. Most easy examples of splitting and aggregating will use a splitter that takes a thing apart and an aggregator that turns all the split parts into a thing again. In fact 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 will very often not have a symmetric splitter.

It now becomes important to think on how we will know when the aggregate is done. Or to stay with the example, when is the shopping list done? The answer here is that it is only done when all the recipes have been split and all ingredients are on a 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 also becomes harder to figure out which messages belong together. Or, in this example, which ingredients go on which list. Usually it is not some key on the message itself natively, but a key that needs to be generated by a business rule. For example it could be that all ingredients that are a vegetable will go on the greenery list and all ingredients that are meat will go on the list for the butcher. This assumes that you're not just taking everything from the supermarket, but even if you do you can probably imagine that it makes sense to organize your shopping list roughly by type to reduce your walking distance in the supermarket.

In the next sections we will see how the components I introduced are actually used. For those interested in the home cooking example there will some code available with this on the Spring Integration in Action repository (now under http://github.com/hierynomus/kitchen/).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

7.2 Splitting, Aggregating and Resequencing

It is very common in domain models to contain high level aggregates that consist of many smaller parts. An order for example consist of different order items, an itinerary consists of multiple legs. When in a messaging solution one service is dealing with the smaller parts and another is dealing with the larger parts, it is 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). In the next few sections we will show typical examples and the related configuration of splitters, aggregators and resequencers.

7.2.1 Splitting messages with a 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 based on a collection that was in the original payload, but this is not 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, we would like to turn them into notifications about trips and send them to impacted users. In order to be do this, we will enrich the header of a Flight notification with a list of impacted trips. At the end of the chain we can then use a Splitter that creates a `TripNotification` for each of the trips related to the flight.

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

Interesting to note here is that the splitting is done based on a list in a header, instead of chopping up the payload. 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.

In our home cooking example there is also a splitter. This splitter chops up a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

recipe into its ingredients as shown in the sample below.

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

As you can see here we are using a chain around the splitter to pop the original recipe on a header, so we can use it as a correlation key later on when we're aggregating products. The splitter is a simple expression that just gets the ingredients (a list) from the recipe payload.

These splitters are relatively simple to configure. You can use a POJO or a simple SpEL expression to retrieve the desired information from the message in the form of a`List`.

7.2.2 Gluing messages together with an Aggregator

We have looked at splitting messages using a splitter. Before we can think about doing the reverse we need to think backwards through the splitting process. The output of the splitter are sets of messages that have been caused by a single message going in per set. It is the original message that correlates them (see section 7.2).

In Spring Integration messages that are correlated through a correlation key, can be grouped together in certain types of endpoints. These endpoints keep the notion of a MessageGroup, more details on this are to be found in the Under The Hood section at the end of this chapter.

Messages can belong to the same group for many reasons. They could be originating from the same splitter or publish-subscribe-channel, but there could also be business concerns that correlate them. For example in our flight notification application we could have a feature that allows users to have the system batch up the notifications they receive per mail based on certain timing constraints. In this case there is no concept of an original message; you will often see examples where aggregating is not used in relation with any splitting logic.

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

```
<i:aggregator
 id="kitchen"
 input-channel="products"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
  output-channel="meals"
  ref="cook"
  method="prepareMeal"
  correlation-strategy="cook"
  correlation-strategy-method="correlatingRecipeFor"
  release-strategy="cook"/>
```

This 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 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 are released. More details on this will follow in the Under the Hood section at the end of this chapter. For now we will 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 based on their recipe.

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

The release strategy delegates to the recipe itself to determine if all the ingredients have been met by products.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

7.2.3 Ordering Messages with a Resequencer

When different messages belonging to the same group are processed by different workers, it is possible that they will arrive at the end of a message flow in the wrong order. As we've seen in chapter 4 there is a priority channel that can order messages internally, but this channel does not consider the whole group. Instead it just passes along the message first in line, not caring about gaps in the sequence caused by messages that haven't arrived at the channel yet.

The solution for this problem is a Resequencer. The resequencer can guarantee that all messages in the group arrive exactly in the right order on the resequencer's output channel. This pattern is more like Aggregator than you would realise on a first glance. The resequencer, just as the aggregator 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.

SIDEBAR **Why ordering should be avoided if possible**

Before looking at some examples of where a resequencer might be useful, we first need to 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 reason for this is that the resequencer is a stateful component and in order 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 actually 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 in case of sequence dependencies. If you can design the system in such a way that messages that are older than the last message processed are simply dropped this is a fundamentally more robust solution. That said, in some cases resequencing is actually very convenient, so you should understand the concept anyway.

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

When groups of messages are processed concurrently, say when we start

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

splitting recipes with multiple people, or when we let multiple people shop at the same time, it is obvious that we introduce race conditions. As long as no checking is done later and no ordering requirements are presented we will not see any negative side effects. It will be good to remember the rule of thumb: adding concurrency increases the reordering of messages.

On the shopping list we will get the ingredients in semi random order determined by the ordering in which the recipes are split. A shop usually has an internal order and it pays off to put the items on your list in the most efficient picking order. This is a traveling salesman problem that we will simplify now by assuming the shop has only one possible walking route and we only have to avoid backtracking.

It is important to make sure that the shopping list is reordered when it is completed (not before). This is done by the <resequencer/> which is much alike the aggregator, but instead of releasing a single message will release all messages ordered according to their sequence number, or a custom comparator. In our case the sequence number will not be used, but instead the ingredients will be compared on shopping order.

In this case (of a custom comparator) it is not possible to release partial sequences because we cannot know in advance if there will be another ingredient that needs to be inserted in the middle of the list.

When doing the mis en place it is a good practice to put the ingredients in the order they appear in the ingredient list. The mis en place is essentially the resequencing of the correlated ingredients according to the order they appear in the recipe. This happens just before they are aggregated into the pan (or bowl or what have you).

In this case with our resequencing we can depend on the sequence number and size set by the recipe splitter. Because of this we can also releases partial sequences. If we have an ingredient list that starts with onions and has garlic as a second item, we can be sure that if we pull the onions and the garlic out of the shopping bag first we can already prepare them before pulling the next items out.

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

After reading this section you should have a good general idea on what splitting, aggregating and resequencing are and how you can use them in an

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

architecture. In the next section we will elaborate a bit more on some common but non standard configurations before we dive into the code and see how things are done under the hood.

7.3 Other common Correlations

So far we have seen mostly obvious correlations between messages. All our cases showed a well defined group of payloads that were released as a group. This is not the only possible use however. This section elaborates on two other use cases for correlation. This is important because aggregation is much more diverse than you would think at first. Not only the way the different payloads and headers are aggregated is important, but also what messages belong together and how strong this correlation is from a business point of view.

Let's for example assume that an aggregator assembles a hamburger. In this scenario it is important that all the ingredients of the hamburger are put together in the right order, but it is completely irrelevant which instance of a particular type of ingredient ends up in the burger. The customer doesn't care which slice of cheese he gets, as long as it is the right kind.

This is a far fetched example, although it might be of use to a hamburger assembly robot developer, but along the same lines we can think of the more actual example of an order fulfillment system. Let's say broker A places a bestens long order for 1000 shares for Glorp corporation (GLC). Then customer B places a short order for 600 shares GLC, customer C places a short for 400 shares of GLC. These three orders can be fulfilled against each other, but if there would be a customer D that placed a bestens long order of 500 shares GLC this could also be fulfilled against customer B's short order. In a trading system there should be race conditions between these two resolutions, simply because being completely fair is unpractical, if not impossible given the performance requirements.

In the next two sub sections we will focus on two common patterns in aggregation that do not immediately fall under the straight forward 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 not only based on the group of messages, but also on external factors, like time. We will now have a look into such a scenario and see how it is supported by Spring Integration.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Refining the shopping list aggregation

Let's quickly think back on the dinner example. When the host is aggregating ingredients on the shopping lists, he can of course wait until all recipes have been split, but that might make for a very long list. It might also take a lot of time, for example say just 10 minutes longer than it takes for his spouse to leave for a particular shop. If that is the case he could also hand his unfinished but long list over before it is complete in its full sense and start a new one. This early completion strategy is very 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 efficiency.

So in terms of Spring Integration's aggregator support, what is happening here? First of all we see a time based constraint. At a particular time the aggregated list is sent regardless of its being complete or not. After this of course the newly arriving messages should still be aggregated, so we will send multiple aggregates instead of just one. In figure 7.3 you can see how this might work in practice.

Figure 7.3 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, that it will have to send out without the rest of the aggregate.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

SIDEBAR **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 time out is important to the end user it is still exposed as a flag on the <aggregator/> element. The message store will give the aggregator a callback when it's time to time out and when you set the 'send-partial-result-on-expiry' flag this will cause the incomplete group to be sent.

When an incomplete group was 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 is a business rule that can tell you if you had all the messages. In our shopping list example we could check if all recipes were split already and given that we use a direct channel for the sending of ingredients this would allow us to complete the last group without checking the size.

As you've seen now aggregation can be based on more than just business keys, and even on the same key over and over again. The next thing we will do is look at a situation where 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 is not always the case however. In this section we will look at a common use case that doesn't follow this pattern.

Isn't Scatter-Gather the same as Map-Reduce?

In the next few paragraphs are about the definition of Scatter-Gather and how it is different from Map-Reduce. I'm assuming that you already know what Map-Reduce is, but even if you don't 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

difference with Map-Reduce is that the different nodes might have a different functions. There are many other resources where you can learn about Map-Reduce and because Spring Integration is not a Map-Reduce framework we will not go into detail on that here. It is important to note that Scatter-Gather and Map-Reduce are by no means mutually exclusive, they are instead complementary and a good architect should be able to weigh the applicability of both or either one of them against the complexity that they inevitably add to the system.

Enterprise Integration Patterns (http://www.enterpriseintegrationpatterns.com/) defines Scatter-Gather as: 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 very broad definition, and it could even be said that Map-Reduce is a subtype of Scatter-Gather. We will look at an example where the different nodes have in fact different functions so that we are forced to stay clear of Map-Reduce concerns.

The home cooking example that we are using throughout this chapter contains a very good candidate for Scatter-Gather. When we split the ingredients over multiple shopping lists we might find that certain shops offer the same products. There are several behaviors we could implement to account for this.

For example a certain product might often be out of stock in shops. If it is then no problem to have too much of it (because it is conservable), we can just try to buy it at all the shops. If we serialize the shopping, or allow communication between the shoppers, we could decrease the risk of overbuying.

If we are looking at an expensive product we can allow shoppers to compare prices with each other when they are shopping in parallel. This happens at the cost of synchronization overhead of course. The amount of synchronization we need to do here is wholly dependent on how bad it would be if we bought too much or not at the cheapest price.

In our example we will assume that no synchronization is done and we'll try to buy the ingredient at all shops. The one that ends up on the mis en place is closest to the best-before date. In figure 7.4 you can see a schematic overview of a this scatter-gather scenario.

Figure 7.4 The needed ingredient is scattered over all shopping lists and sent to each shop (A and B). The gatherer decides based on the best-before date, or any other criterion what is the best product to use and sends that to the mise en place (not in this picture).

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

To scatter an ingredient after splitting we need to route it to multiple nodes. We can do this by configuring a router that does it's best to route to a single shopping list, but when 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 that we are sure cannot be found at the shop. Yet another option is to let the filtering occur naturally by asking each shop for each product and take all that are available. It depends on the situation which will be more efficient and we will not 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 not use things that are not needed. The other option is to compare the duplicate products best-before date when all products for a mis en place are complete and store the one that can be conserved longer.

In this section you have seen two examples of usages of the aggregator that differ from the standard usage of reassembling some collection. There is only one thing left to do and that is to open up 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 is the class that is wired by the AggregatorParser with collaborators that make it into an aggregator, and wired by the ResequencerParser with collaborators that make it into a resequencer. In this section we will look at the steps the CorrelatingMessageHandler performs to group, store and process messages, then we will look at two examples of wiring a CorrelatingMessageHandler as an aggregator and as a resequencer.

7.4.1 Extension points of the CorrelatingMessageBarrier

Roughly the way a group of messages that is processed by this component can be split in two: message in and message out. When the message comes in it is correlated and stored. When a message group might go out it is released, processed, and finally marked as completed. So Correlate, Store, Release, Process, Compete. We will look into the details of each of those steps and introduce collaborators as we go along.

When a message hits the CorrelatingMessageHandler the first thing it needs to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

do is figure out what MessageGroup this message belongs to. This 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 just 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 we use a MessageGroupStore, which defaults to an in-memory implementation. Storage used by the CorrelatingMessageBarrier 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 is important to consider memory consumption and performance in case of large groups or large numbers of incomplete aggregates.

After the message is stored the group of that message is considered for release. For example a completed aggregation will be released, but also a partially completed group that may contain the first few elements of a sequence. The release strategy says nothing about the completeness of the group. Its only responsibility is to decide wether or not the message processor may process this particular group or not.

Once a group is released it is handed to the MessageProcessor of the CorrelatingMessageBarrier. 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 is also responsible for marking the messages that 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 we will see the implementation of aggregator and resequencer as examples of the mentioned strategies. In both aggregator and resequencer correlation and storage is the same (and trivial), so we'll go into the details of release and processing only.

7.4.2 How do Resequencer and Aggregator do it?

The aggregator, as discussed before, takes a group as a whole and forges a new message out of it. As such it should release the group only when it is completed (or timed out). We will look at the release and processor strategies in detail in the next few paragraphs.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The release strategy of an aggregator should only release the group when it is complete. The SequenceSizeReleaseStrategy implementation does just this. In fact the for this common case the MessageGroup has an isComplete() method which has a default implementation that compares the sequence size header to the size of the group. This is convenient if you are implementing a custom release strategy, but you 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 will automatically unwrap the elements in the list if they are not a message. The return value will be wrapped in a message if needed and sent on 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 were coming in.

By default the release strategy used by the resequencer is also the SequenceSizeReleaseStrategy. In 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 more smooth 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) that uses several strategies to delegate to. To find the correlation id of the message group it uses the CorrelationStrategy. To store the message group it uses the MessageGroupStore. To decide when to release the group for processing a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 have learned to deal with splitters, aggregators and resequencers. We have also looked at examples of some non trivial aggregator use cases and finally looked at the design that is at the core of Spring Integration. Let's list the things we've learned about splitters first.

- One message goes in, many messages come out
- The output can be based on the payload, but also other things like headers
- The splitter will set correlation id, sequence size and sequence number for each message

On the other hand we had endpoints that that grouped messages together before sending reply messages along. At the start of the chapter we looked at correlation and in the Under The Hood section we looked at it in detail. Correlation is the base for both aggregating and resequencing. The following extension points are relevant to remember:

- CorrelationStrategy finds the correlation key based on a message. This doesn't have to be the correlation id (e.g. 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 strategy and release strategy are complementary to the splitter.

The Resequencer processes messages by reordering them. The correlation strategy and release strategy are similar to the aggregator with the exception of releasing partial sequences. When partial sequences are released, multiple releases of the same group may happen.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Now that you've read this chapter you should have a much better idea how Spring Integration can help you when you need to split up some work or aggregate the results of some operations back together. In particular Aggregation is a very complex use case that often differs subtly from the examples that you can find in this book or online. In some cases it pays off 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.

In the next chapter you will work with XML, and there you will see concepts from this chapter reused in for example the Xpath splitter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

8 Handling Messages with XML Payloads

In chapter 7 we've discussed Routing. You have seen there that the flow of a Message through the system, or the chain of events caused by the arrival of a message is usually very dependent on it's payload. As we have discussed in Chapter 6, transforming payloads to and from an intermediate format, like XML, is essential to integrate different systems with each other.

There are many exchange formats in the wild, but none is as widely spread as XML. Virtually all programming languages have some sort of support for XML. From Javascript to PL/SQL, from Smalltalk to Scala, all these languages share knowledge of this common vocabulary.

Often the integration components do not need detailed knowledge of the domain, but they do need to react on the contents of a message. If both input and output are XML messages, marshaling the message as we discussed in Chapter 6 will cause unnecessary complexity and overhead. It is better to work on the XML directly, using for example XPath or XSLT.

Because XML is the most commonly used interoperable format, it only makes sense to equip an integration framework with support for this language. In the following chapter we will 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 in order to make working with XML simple.

This chapter introduces the XML capabilities of the Spring Integration framework by taking an example message flow and looking at how we can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

implement that using the provided XML support. The chapter then finishes with a detailed explanation of the approach taken in the implementation of these endpoints and details on customization points for advanced usage.

8.1 Generating quotes for trip legs using XML based messaging

In our SIVA travel example application a quotation request for a trip may contain multiple legs. Having split the trip into legs the message flow then needs to generate quotation requests for the different elements of a trip such as the hotel accommodation and car rental. In order to do this we will need to separate out the constituent parts of the trip relating to hotel, flight and car rental before transforming our internal canonical representation to one that is understood by the third party systems. One option would be 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 can become unwieldy as the number of systems with which we integrate grows. It also likely that the request format of the third party system will be some form of XML. Therefore we have opted to implement the process of generating a quote for each leg of the trip as an XML based message flow.

The trigger for the leg quote generation flow is the receipt of a message containing a `LegQuoteCommand`. This 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 flow firstly converts the message into the canonical XML form of a trip leg. An example of such an XML document in this format is shown below. An Extensible Stylesheet Language Transformation is then used in order to add the leg information to each criteria section prior to splitting. Having duplicated the leg information to each criteria section we are then able to split the document into separate requests for processing in parallel. To achieve this we will use a provided XPath splitter implementation to split each of the children of the *legQuote* element. The leg quote flow is then able to process each request in parallel and route to the appropriate third party system, again this will utilize an XPath expression which will routes the message according to the route element name. Finally before making a request to the third party system we validate that each of the requests conform 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<?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>
```

8.2 Marshalling our LegQuoteCommand into the canonical XML form

In order to convert our message between XML and Objects we will use Object/XML mapping. This is commonly referred to as marshalling and unmarshalling . Spring provides a generic technology agnostic abstraction of the process of mapping between Java and XML. The OXM package as it is known provides a common interface across a number of leading Java OXM solutions. The OXM module has been moved to the core Spring project from Spring Web Services since Spring 3.0. By providing a common abstraction the OXM module decouples code from the implementation details and is able to provide 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 XML `Result` and the `Unmarshaller` interface which maps from a `Source` to the `Object` representation. Both these interfaces are shown below.
1

Footnote 1 As of Spring 3.0 the OXM framework is now part of the main Spring framework project.

```
public interface Marshaller {

    boolean supports(Class<?> clazz);

    void marshal(Object graph, Result result)
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
                  throws IOException, XmlMappingException;
}
```

```
public interface Unmarshaller {

    boolean supports(Class<?> clazz);

    Object unmarshal(Source source)
                   throws IOException, XmlMappingException;
}
```

Figure 8.1

8.2.1 Annotating our domain classes for use with JAXB

In the case of the leg quote flow we have chosen to use the JAXB 2 OXM technology in conjunction with Spring OXM. Since we already had a set of existing rich domain classes we have opted to annotate our domain classes with JAXB annotations rather than generate our classes from an XML schema. Below is our annotated `LegQuoteCommand` which maps to the root of our canonical XML leg quote document. To indicate that this maps to the root of an XML document we have annotated it with the `@XmlRootElement` annotation. In addition we 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.

```
@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;
    }

    ...
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

By default JAXB will output all fields of the annotated type although it is only capable of converting standard Java types out of the box. This can be extended by providing `XmlAdapter` implementations capable of mapping types unsupported by JAXB to types supported by JAXB. An example of this is our Leg class which contains start and end dates and locations. This class makes use of the Joda Time project to represent dates. The code below 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`.

```
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` as below. This adapter simply converts between the date types using UTC time.

```
public class JodaDateTimeAdapter
                     extends XmlAdapter<Calendar, DateTime> {

    @Override
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
    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;
    }
}
```

8.2.2 Configuring the Spring Integration marshaller endpoint

The Spring Integration marshalling and unmarshalling support is a fairly thin adapter layer on top of Spring OXM therefore the first thing we need to do is to configure an instance of marshaller or unmarshaller as appropriate. In fact the out of the box implementations all implement both the `Marshaller` and `Unmarshaller` interfaces. Below we configure an instance of `Jaxb2Marshaller` providing a list of the annotated types, in this case just the `LegQuoteCommand`.

```
<bean id="legMarshaller"
    class="org.springframework.oxm.jaxb.Jaxb2Marshaller">

    <property name="classesToBeBound">
        <value>
            com.manning.siia.domain.trip.LegQuoteCommand
        </value>
    </property>
</bean>
```

On top of this we can configure a marshalling endpoint to convert from our `LegQuoteCommand` to our canonical XML form using the Spring Integration XML namespace as below.

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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
  http://www.springframework.org/schema/
                integration/xml/spring-integration-xml.xsd">

<si:channel id="javaLegQuoteCommands"/>

<si-xml:marshalling-transformer input-channel="javaLegQuoteCommands"
    output-channel="xmlLegQuotes"
    marshaller="legMarshaller" result-transformer="resultToDocumentTransformer"/>

<bean id="resultToDocumentTransformer"
    class="org.springframework.integration.xml.transformer
                          .ResultToDocumentTransformer"/>
</beans>
```

The Spring Integration marshalling transformer requires an instance of `OXM Marshaller`. In addition where it is being used standalone, that is outside the scope of a chain it also require an input-channel and output-channel. Looking at the standard `Marshaller` interface you will notice that the marshaller will convert the `Object` graph and into an instance of `Result`. Since the Spring Integration marshallers primary purpose is to be a thin adapter layer on top of the OXM support this is the same behavior you will get with Spring Integration by default. So if we were to configure our marshaller with nothing else but an instance of `Marshaller` and references to the input and output channel 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` is not ideal. In fact without casting the Result the only thing we can do is get or set an optional system id which may be used for error messages or to relate to an file on the local file system. 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 will attempt some form of type negotiation whereby the payload output type will be determined by the input. For example if we pass in a `String` most provided XML endpoints will pass back a String. Note that the supported payload types and strategies for determining output type will be discussed later in this chapter in the under the hood section. The marshaller is however different since we are passing in an `Object` and asking for the Object as XML the marshalling endpoint has no clues as to how we would like the XML. Do we want a `String` an `org.w3c.dom.Document` or something else? To obey the principle of least surprise therefore it passes back a `Result` to be consistent with the underlying OXM marshaller. In order to make this payload more useful in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

this case we provide an optional collaborator of type `org.springframework.integration.xml.transformer.ResultTrans` to convert the Result to something more useful. To do this we set the optional *result-transformer* attribute of the marshalling-transformer element with a reference to our `ResultTransformer`. Out of the box two implementations of `ResultTransformer` are provided, one to convert to a `Document` the other to convert to a `String`.

8.3 Enriching the leg quote using XSLT

Now we have our leg quote 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 which will provide 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 our message flow is to use an Extensible Stylesheet Language Transformation to add the leg information to the criteria elements and to rename our criteria elements. To achieve this we 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>
```

Using the Spring Integration XML namespace we will proceed to configure an XSLT endpoint as below. When configuring the XSLT endpoint it is possible to provide an instance of the Spring

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

`org.springframework.core.io.Resource`. The endpoint will then use 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.

```
<si:channel id="xmlLegQuotes"/>

<si-xml:xslt-transformer input-channel="xmlLegQuotes"
    output-channel="xslTransformedLegQuote"
    xsl-resource="classpath:/xsl/enrichCriteriaWithLeg.xsl"/>
```

Passing our example message through this transformation produces the below XML which is now suitable for splitting into the three separate parts of the leg in order to generate quotes each. Passing our example message through this transformation produces the below XML.

```
<?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>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

8.4 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 simple 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" />
```

Since commonly XML namespaces will be used to avoid element name clashes generally XPath expression will need to incorporate a namespace. This can be achieved in one of two ways. In the simple case where there is only one namespace it 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>
```

8.5 Splitting hotel, car rental and flight quotes

Now we have a quote in a form which allows us to split it into three self contained requests. To achieve this we are going to use an XPath expression which defines how we 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. In order to split our leg quote an XPath expression which evaluates to give the direct children of the *legQuote* element which is simply */legQuote/**. Below is the configuration for the XPath splitter that will separate out the different types of quote request.

Figure 8.2

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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 containing each 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. By importing the node the new document is isolated from the source document since the node is copied and becomes the root element of the new document. In our case this means that the *carQuote, flightQuote and hotelQuote* elements in the post split messages are no longer children of the *legQuote* element. This is also has means that changes made to any of the documents created by the XPath splitter will not be seen in the original document.

8.6 Routing messages based on their XML payloads

Now we have our three separate requests the next thing we need to do is route the requests to the different third party systems for processing. Since each of our request types has a well known root element we can use the provided XPath router to route according to the root element name.

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 configuration below we are also providing a channel resolver. If no channel resolver is provided the default strategy is to resolve the value returned by the XPath expression as a bean id of a channel defined within the application context. While it may be possible to use this strategy it is more usual to provide an explicit mapping between the evaluation result and the channel to route to since this allows the use of meaningful channel names rather than having channel names which reflect the structure of the XML messages being processed. Spring Integration provides a channel resolver implementation which can be configured with a map for this purpose.

```
<si-xml:xpath-router id="quoteRequestRouter" input-channel="splitQuotes"
           channel-resolver="quoteChannelMap">
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
    <si-xml:xpath-expression expression="local-name(/*)"/>

</si-xml:xpath-router>

<bean id="quoteChannelMap"
    class="org.springframework.integration.channel.MapBasedChannelResolver">

    <property name="channelMap">
        <map>
            <entry key="carQuote" value-ref="carQuoteChannel"/>
            <entry key="flightQuote" value-ref="flightQuoteChannel"/>
            <entry key="hotelQuote" value-ref="hotelQuoteChannel"/>
        </map>
    </property>
</bean>
```

8.7 Validating XML messages

It is important to ensure that we are only sending valid requests to third party systems. So as a final step before we transmit the xml we will validate the quote requests produced by the leg quote message flow. To carry out XML validation we will use the provided validating router implementation which is capable or validating XML payloads against a schema definition provided in either RELAX NG or XML Schema format. The validation support in the XML module takes on the form of a router implementation which routes to either a valid or invalid channel according to the outcome of the schema validation. Below 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"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
            </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:schema>
```

Plugging that into the pipeline is then very simple as show in the configuration below. The schema itself is loaded as a resource and references to the input channel, 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" />
```

8.8 Under the hood of the Spring Integration XML module

The Spring Integration module in common with other modules and other Spring projects tries to simplify the work of the application developer. Specifically it tries to address concerns such as the creation of instances of required classes required for XML processing such as `org.w3c.dom.Document`, `javax.xml.transform.Result` and conversion between different representations of XML such as `String`, `Document`, `Result` and `Source`. This generally makes the life of the developer easier. However knowing what the framework is doing and how to change the default strategies being employed can be very useful. For example passing around XML as a String can be quite appealing since Strings are easy to create and easy to log and debug. However in a high performance application which is making extensive use of XPath evaluation understanding that this involves a considerable amount of on the fly conversion between a String and a DOM representation is important.

When dealing with large xml payloads tree based parsers (DOM parsers) tend to induce very poor performance characteristics in an application. The first thing people do to improve this, is to move to a streaming parser (SAX, StAX). An unfortunate fact to take into account there is that streaming and messaging don't mix well. A stream is in fact a leaky abstraction under messaging and it will break

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 do so 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 to give you the detail you need to understand the implications of using different payload types and also how to customize the framework behavior where it is required.

8.8.1 Supported payload types and return type matching

One of the main challenges when starting to consider the XML module for Spring Integration was deciding which of the many APIs for working with XML in Java to support. It is possible to interact with data expressed as XML from Java in many different ways such as using SAX events, parsing into a tree representation such as org.w3c.dom.Document or using one of the additional projects which aim to make working with XML simpler and more intuitive from Java, for example Dom4j or JDom. 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`. There were a number of reasons for focusing primarily on String representations that could easily be converted into DOM on the fly and DOM itself. Using JAXP and `org.w3c.dom` packaged classes minimizes the need for third party libraries and also kept the implementation relatively simple. In addition many of the patterns supproted such as split and route rely on XPath evaluation and since the standard implementations of XPath require a fully built document to evaluate against this was another reason to initially focus on DOM rather then event based processing of XML payloads.

The downside of focusing on DOM and String representations exclusively is that processing very large XML documents is not currently well supported since they result in large in memory representations. This has clear implications for concurrency and the maximum size of document we can support. In the future support for streaming through STAX and/or SAX is something we hope to be able to add to Spring Integration.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

8.9 Conclusion

This chapter has introduced the capabilities that the Spring Integration XML module provides for working with XML payloads in a style that compliments the POJO and Spring namespace approach of the framework as a whole. While using XML payloads may not be the default for most projects it is still a valuable tool for projects where transforming received XML messages into Java representations is an unnecessary overhead.

In the next chapter we will see that now we have our XML messages we can use the built in Web Service support to pass XML messages to external systems.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

9 Spring Integration and the Java Message Service

For many Java developers, the first thing that comes to mind when they hear "messaging" is the Java Message Service (JMS). That is understandable considering it is 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. There are also a number of open-source JMS implementations. One of the open source options is ActiveMQ, a pure Java implementation of the JMS API. We'll be using ActiveMQ in some of the examples within this chapter since it is easy to configure as an embedded broker. We will not be going into any specific ActiveMQ details, however. If you would like to learn more about it, please refer to 'ActiveMQ in Action' (Manning, 2010).

Hopefully, by this point in the book, you realize that messaging and event-driven architectures do not necessarily require the use of such systems. Given that we have been discussing messaging in several chapters without having to dive into the details of JMS thus far 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 direct invocation of methods with arguments. In other words, messaging is really a paradigm, and the same underlying principles apply whether that 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 intra-process messaging and the inter-process messaging

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

that JMS enables across many MOM providers. In this chapter, you will learn how to map between Spring Integration Messages and JMS Messages. You will 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 very straightforward. However, 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 will 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 intra-process and inter-process messaging. The primary role of Channel Adapters and Messaging Gateways is to connect a local channel to some external system without impacting the actual producer or consumer components' code. Another benefit that 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 File system, or mapping to another messaging API. Of course, 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 intra-process and inter-process components as belonging to two layers but with a consistent model.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 9.1 The top configuration shows inter-process integration using JMS. The bottom configuration shows intra-process integration using Spring Integration. Which type of integration is appropriate depends on the architecture of the application.

Even though we may focus on "external system integration" when discussing the benefits of JMS, there are benefits to using JMS even within an application. As we will describe later in this chapter, JMS may be useful between producers and consumers running in the same process since 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 still looks like any other channel as far as the message producing and consuming components are concerned, so it can be used as an alternative at any point within a Message flow.

Figure 9.2 This figure shows the 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.

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> sub-element within a <channel> without referencing any MessageStore. By default, such a channel does not *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. In fact, they will even be lost if the process is shutdown intentionally, but before those Messages are drained from the queue by a consumer. In certain cases, when dealing with real-time event data that does not require persistence, the loss of those Event Messages upon process termination might not be a problem. Indeed, it may be well worth the tradeoff for asynchronous delivery that allows the Producer and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Consumer to operate within their own threads, on their own schedules. With these Message Channels backed by a JMS Destination, however, 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 are only being used by Producers and Consumers in the same process space.

The main point we're trying to make here is that even though we often refer to JMS as an option for messaging between a number of individual processes, that is not the *only* time to consider JMS, or other inter-process broker-based messaging solutions such as AMQP, as an option. Of course when multiple processes are involved, the other advantages become quite evident. First among these benefits is the natural load-balancing that occurs when multiple consuming processes are pulling Messages from a shared Destination. Unlike producer-side load-balancing, the consumers can naturally distribute the load based on 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 and in turn producing 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 only exists within acceptable limits upon 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. Essentially, this is nothing more than the benefit of loose-coupling inherent in any messaging system but 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 are not merely talking about unforeseen system outages. It is increasingly common for modern applications to have "zero

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 that of transactions. We will revisit this in greater detail near the end of this chapter, but one quick point is relevant to the discussion at hand. In the scenario described above, 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 has been configured. In JMS, there are a variety of options corresponding to different points on the spectrum of guaranteed delivery. One configuration option is to require an explicit "acknowledgement" from the consumer. It might be that a consumer acknowledges each Message after it is able to store it successfully on disk. Another more robust option is to enable transactions. The consumer would only commit the transaction upon successful processing of the Message, and it would rollback the transaction in case of a known failure. When adding this functionality to the example above, 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

9.1.1 Mapping between JMS and Spring Integration Messages

When considering inter-process 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 does not have any idea that there is an external system involved. That means the Channel Adapter not only handles the actual communication via the particular transport and protocol being used, but it also must provide a Messaging Mapper (http://www.enterpriseintegrationpatterns.com/MessagingMapper.html) 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, while 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 while the payload is usually the "business" data which has some meaning within the domain of the application. Thinking of a message as fulfilling the "Document Message" pattern from EIP, the payload represents the document while headers contain additional metadata, such as a timestamp or some information about the originating system.

It just so happens that the construction of a Message according to the JMS specification is very similar to that of a Spring Integration Message. This should not come as a surprise 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 will go into the details in a later section, but for now it is sufficient to point out that there are merely some differences in naming. In JMS, the Message has a "body", while in Spring Integration, we refer to that as the "payload". Likewise, a JMS Message's "properties" correspond to a Spring Integration Message's "headers".

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 9.3 This figure shows a Spring Integration and JMS Message in a side-by-side comparison. The terminology is different, but the structure is the same.

9.1.2 Comparing JMS Destinations and Spring Integration Message Channels

By now you are quite familiar with the various Message Channel types available in Spring Integration. One of the most important distinctions we have covered is that between Point-to-Point channels and Publish-Subscribe channels. You have seen that, when it comes to configuration, the default type for a "channel" element in XML is Point-to-Point while the Publish-Subscribe channel is clearly labeled as such. The JMS specification uses "Destination" instead of "Message Channel", but it does make a similar distinction. The two types of JMS Destination are Queues and Topics. A JMS Queue provides Point-to-Point semantics while a Topic supports Publish-Subscribe interaction. When using a Queue, each Message will be received by a single consumer, but when using a Topic, the same Message can be received by multiple consumers.

Table 9.1 Comparing EIP to JMS

EIP	JMS
Channel	Destination
Point to point channel	Queue
Publish subscribe channel	Topic

Now that we've discussed the relationship between Spring Integration and JMS at a high level, we're almost prepared to jump into the details of Spring Integration's JMS adapters. First, however, 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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.

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 like the JdbcTemplate and the TransactionTemplate. These components are all realizations of the "Template" pattern as described in the "Gang of Four" Design Patterns book ('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'll look at code that does not use the JmsTemplate but instead performs all actions directly with the JMS API. You will notice that even a simple operation such as sending a text-based Message will involve 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:
            Connection connection = connectionFactory.createConnection();
            connection.start();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLED
            Destination queue = new ActiveMQQueue("siia.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 (JMSException e) {
            throw new RuntimeException("problem occurred in JMS code", e);
        }
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
    }

}
```

The code above is about as simple as it can get when using the JMS API directly. We are using ActiveMQ, and it enables running an embedded broker (as you can see from the "vm://localhost" URL we provided to the ConnectionFactory). Many JMS providers would be configured within the JNDI registry, and that would require additional code to lookup 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) {
        JmsTemplate jmsTemplate = new JmsTemplate(new ActiveMQConnectionFactory("vm:
        jmsTemplate.setDefaultDestination(new ActiveMQQueue("siia.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 going to be caught and converted into RuntimeExceptions within Spring's JmsException hierarchy. The JMS resources, such as Connection and Session, are also going to be acquired and released as appropriate. In fact, if a transaction is active at the time that this send operation is invoked, and some upstream process had already acquired a JMS Session, this send operation would be executed within 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 within the transaction throws an uncaught RuntimeException, all operations that occurred within that same transactional context would be rolled back. If, on the other hand, all operations were successful, the transaction would be committed.

You probably also noticed that the template method we're invoking is called "convertAndSend" and that its argument was just an instance of java.lang.String. There are also send() methods that accept a JMS Message that you have created, but by using the convertAndSend versions, you are able to rely on the JmsTemplate to construct the Messages. The conversion itself is actually a pluggable strategy. The JmsTemplate delegates to an instance of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

MessageConverter interface, and the default implementation (SimpleMessageConverter) will automatically perform the following conversions:

Table 9.2 Default type conversions supported by MessageConverter

Type passed to MessageConverter	JMS Message Type
java.lang.String	TextMessage
byte[]	BytesMessage
java.util.Map	MapMessage
java.io.Serializable	ObjectMessage

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 above (e.g. TextMessage to java.lang.String).

Sometimes the default conversion options are not a good fit for a particular application. That is why the MessageConverter is a strategy interface that can be configured 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 TextMessages over a JMS Queue, but the application's developers prefer to hide the XML marshalling and unmarshalling logic within 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 JAXB.

Of course, it's also possible to provide a custom implementation of the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Marshaller and Unmarshaller interfaces or even a custom implementation of the MessageConverter itself. For example, you could implement the MessageConverter interface in order 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 TextMessages where the actual text content is formatted using JSON.

9.3 Asynchronous Message Reception with Spring

Earlier chapters probably made it clear that receiving Messages is usually more complicated than sending. Even though the receiving part of the JmsTemplate example above looks just 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 will 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 that receiveTimeout value.

When relying on the blocking receive operations, such JmsTemplate usage is essentially an example of the Polling Consumer pattern from EIP. In an application where an extremely low volume of messages is expected, polling within a dedicated "background" thread might be okay. However, as we mentioned earlier in the book, most applications using messaging would prefer to have Event Driven Consumers.

Of course 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 rather 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 based on 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 is a common concern for many components within Spring Integration. To understand the details of how the framework handles those requirements, be sure to read the chapters that cover Message Endpoints and Concurrency.

As far as the JMS adapters are concerned, the crux of the problem is that the actual invocation of the JMS receive operation must itself be performed within the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

context of the transaction. Then if the subsequent handling of the Message causes a failure, that is 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 upon might be down at the moment but will be available again shortly. If a JMS consumer rolls back a transaction, then the Message will not be removed from the Destination; that is what enables redelivery. However, 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 have 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 actual Message content.

At this point, you are probably convinced that implementing an asynchronous message-driven solution is not trivial. It's the type of generic, foundational code that should be provided by a framework so that developers do not 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. Here, we are using JmsTemplate for the sending side only. The message is received by a MessageListener asynchronously, and the DefaultMessageListenerContainer is handling all of the low-level concerns.

```
public class MessageListenerContainerDemo {

    public static void main(String[] args) {

        // establish common resources
        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("vm://lo
        Destination queue = new ActiveMQQueue("siia.queue");

        // setup and start listener container
        DefaultMessageListenerContainer container = new DefaultMessageListenerContai
        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).getTex
                }
                catch (JMSException e) {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
                    throw new RuntimeException(e);
                }
            }
        });
        container.afterPropertiesSet();
        container.start();

        // send Message
        JmsTemplate jmsTemplate = new JmsTemplate(connectionFactory);
        jmsTemplate.setDefaultDestination(queue);
        jmsTemplate.convertAndSend("Hello World");
    }

}
```

The code above 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 that we've just added quite a bit more code to the example, and we're back to dealing with some JMS API code directly. For example, we have provided an implementation of the JMS MessageListener interface, and we are catching the JMSExceptions in order to convert into RuntimeExceptions ourselves. In the next section, we will take things two steps further. First, we will rely on Spring's MessageListenerAdapter to decouple our code from the JMS API completely. Second, we will refactor the example to use declarative configuration and a dedicated Spring XML schema. In other words, we will demonstrate a Message-driven POJO.

9.4 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 actual 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 actually quite straightforward. The adapter does implement 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 code above 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 just a few lines of XML.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<listener-container>
    <listener destination="someQueue" ref="somePojo" method="someMethod"/>
</listener-container>
```

There are many configuration options on both the 'listener-container' and 'listener' elements, but the example above provides a glimpse of the simplest possible working case. The XML schema itself is well-documented if you would like to explore the other options. Our goal here is to provide a necessary level of background information so that 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 will now turn our focus back to Spring Integration to see how it offers an even higher level approach.

9.5 Sending JMS Messages from a Spring Integration Application

Now that you have seen the similarities between Spring Integration Messages and JMS Messages and learned about the core Spring support for JMS, we are well prepared to look at the process of sending JMS Messages from a Spring Integration application. As with most adapters in Spring Integration, there is a unidirectional Channel Adapter as well as a request/reply Gateway. Since it's considerably simpler, we will begin the discussion with the unidirectional Channel Adapter.

Spring Integration's outbound JMS Channel Adapter is essentially a JMS Message publisher that is encapsulated within 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 would be 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 is just a configuration detail.

The main class involved in the one-way outbound JMS adapter is called JmsSendingMessageHandler. However, if you were to take a look at its implementation, you would discover that it actually 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. A majority of the code in Spring Integration's adapter is simply handling the various configuration options, of which there are many. As far as most users are concerned, even those configuration options are simply 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'll walk

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

through several of these in just a moment, but first let's take a 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 are able to rely on the defaults, then that's all you would need to configured. It is quite 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 quite common that the Queues and Topics are administered objects that developers should always access via JNDI lookups. Fortunately, we can rely on the Spring Framework's ability to handle that. Instead of using the "destination-name" attribute, you could simply provide a "destination" attribute whose name is actually 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. While it's perfectly acceptable to define that FactoryBean instance as a low-level "bean" element, there is XML namespace support for a much more concise configuration options as shown below.

```
<jms:outbound-channel-adapter channel="toJMS" destination="fromSIQueue" />

  <jee:jndi-lookup id="fromSIQueue" jndi-name="jms/queue.fromSI"/>
```

The functionality would be exactly the same. The difference is limited to the configuration. Often it may be that access by name is sufficient in the development and testing environments while JNDI lookups are required for the production system. In those cases, you can manage the configuration excerpts appropriately by using "import" elements within the configuration or other similar techniques. The important factor is that you do not need to modify any code to handle those different approaches for resolving JMS Destinations.

Fortunately, the configuration of the ConnectionFactory and Destinations can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 will change the focus to the receiving side. We'll begin with the inbound Channel Adapter that serves as a Polling Consumer.

9.6 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 will lookup 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. Of course the inbound-channel-adapter element also requires a poller unless you are relying on a default context-wide poller element. Here's a simple example of an inbound Channel Adapter that will poll for a JMS Message every 3 seconds.

```
<int-jms:inbound-channel-adapter id="pollingJmsInboundAdapter" channel="jmsMessages"
    <int:poller fixed-delay="3000" max-messages-per-poll="1"/>
</int-jms:inbound-channel-adapter>
```

Just like the outbound version, if you are 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 an actual 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. Below is an example of the corresponding inbound configuration.

```
<int-jms:inbound-channel-adapter id="pollingJmsInboundAdapter" channel="jmsMessages"
    <int:poller fixed-delay="3000" max-messages-per-poll="1"/>
</int-jms:inbound-channel-adapter>

<jee:jndi-lookup id="myQueue" jndi-name="jms/someQueeue"/>
```

As mentioned earlier, polling is rarely the best choice when building a JMS-based solution. Considering the underlying JMS support in Spring already provides MessageListeners that can be invoked asynchronously as soon as a JMS Message arrives, that is almost always going to be a better option. The only

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

exceptions might be those cases where you want to configure a poller to run very infrequently or only at certain times of the day. If it is indeed limited to certain times of the day, you would most likely use the "cron" attribute on a poller element. Other than 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 is no longer a need to define a poller.

```
<int-jms:message-driven-channel-adapter id="messageDrivenJmsInboundAdapter" channel=
```

It may seem odd that, unlike most adapters that you've seen, the element does not include "inbound" in its name. However, 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.

9.7 Request Reply Messaging

All of the discussion and examples in this chapter have thus far been focused on unidirectional Channel Adapters. On the sending side, we have not yet discussed the case where we might be expecting a reply, and on the receiving side, we have not yet discussed the case where we might be expected to send a reply. We have seen 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 is going to introduce Spring Integration's bidirectional Gateways for utilizing JMS in a request/reply model of interaction. Much of the functionality will be the same, such as mapping between the JMS and Spring Integration Message representations. The difference is that these request/reply "gateway" adapters are responsible for mapping in both directions. As with the unidirectional Channel Adapter discussions, we will 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 actually a receive as well, assuming that the JMS reply Message does in fact arrive as expected. So, the simplest way to think of the outbound Gateway is a send-and-receive adapter.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

9.7.1 The Outbound Gateway

In the simplest case, the outbound configuration will look very similar to the outbound-channel-adapter configuration that we saw earlier.

```
<int-jms:outbound-gateway request-channel="toJMS" reply-channel="jmsReplies" request
```

The "request-channel" and "reply-channel" attributes refer to Spring Integration MessageChannel instances. Any Spring Integration Message that is sent to the request channel will be converted into a JMS Message and sent to the Gateway's request Destination (within this context, "Destination" will always refer to a JMS component, while "channel" will always be the Spring Integration channel). In the example above, 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". Since the Gateways need to manage sending and receiving separately, many of its attributes are qualified as either affiliated with the "request" or the "reply". The "reply-channel" is where any JMS reply Messages will be sent - after they are converted to Spring Integration Messages.

You may have noticed that we did not provide a "reply-destination-name". That attribute is optional, but it's quite common to leave it out. The Gateway implementation will provide the JMSReplyTo property on each request Message that it sends to JMS. If you do not provide a specific Destination for that, then it will handle creation of a temporary queue for that purpose. This assumes that wherever these JMS Messages are being sent, there is a process in place that is checking for that JMSReplyTo property so that it knows where to return a reply Message. That server side behavior is something we will discuss in the next section when we cover the inbound Gateway. However, for the time being, we will be discussing 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 itself. Therefore, it is commonly supported functionality for server side JMS implementations that accept request Messages from a sender who is also expecting reply Messages. You would need to be sure that you are sending to a Destination that is indeed backed by a listener implementation with that behavior. The inbound Channel Adapter we discussed above would *not* be a good choice, since as we emphasized, it is intended for unidirectional behavior only. On the other hand, the inbound Gateway that we will discuss in the next section would be a valid option for such server side

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

request/reply behavior. In fact, the core Spring support for Message-driven POJOs that we discussed above also supports the JMSReplyTo properties of incoming Messages as long as the POJO method being invoked has a non-void and non-null return value.

9.7.2 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 that we were describing in abstract terms as "the server side" in the previous section where we discussed the outbound Gateway. The outbound Gateway would itself be "the client side" as far as that discussion is concerned. The inbound gateway listens for JMS Messages, maps each one that it receives into a Spring Integration Message, and sends that to a Message Channel. Thus far, that is no different than the role of an inbound Channel Adapter. The difference, however, is that the Message Channel in this case would be the initiating end of some pipeline that is expected to produce a reply at some point downstream. When such a reply Message is eventually returned to the inbound Gateway, it will be 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 will take precedence. If no JMSReplyTo property had been sent, the inbound Gateway will fallback 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" since the JMSReplyTo property on a request message does take precedence). If configuring just 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 is neither a JMSReplyTo property on the request Message or a configured reply Destination, then an Exception will be thrown by the gateway since it would obviously 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, however, that it builds directly on top of the underlying Spring JMS support that we described earlier in this chapter. Now

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

you can probably appreciate why we did go into a considerable amount of detail about that underlying support earlier. 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 into a Spring Integration Message, it's just sending that to a Message Channel. What makes each gateway unique is simply *what* it receives and *how* it maps that into a Spring Integration Message.

Now that you understand the role of the inbound gateway for JMS, let's look at an example. We'll start with the very 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 are 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 there is no 'reply-channel' attribute present. It is actually an optional value for inbound gateways in general. If it's not provided then the gateway will simply create a temporary, anonymous channel and set that 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 there is a JMS ConnectionFactory instance named "connectionFactory" within the Application Context. Simply add that attribute if for some reason the JMS ConnectionFactory you need to reference has a different bean name.

As you might expect, since you now know that this is building on top of Spring's MessageListener container, there are a number of other attributes available. Many of them are passed along to that underlying container. For example, you might want to control the concurrency settings. Here is an example that indicates 5 "core" consumers should always be running, but when load increases beyond the capacity of those 5, the number of consumers can increase up to 25. At that point, each of those "extra" consumers can have up to 3 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<jms:inbound-gateway id="exampleGateway"
                     request-destination-name="someQueue"
                     request-channel="requestChannel"
                     concurrent-consumers="5"
                     max-concurrent-consumers="25"
                     idle-task-execution-limit="3"/>
```

9.8 Messaging between multiple Spring Integration runtimes

We've seen several examples of how the adapters make use of Spring's JMS MessageConverter strategy to convert to and from JMS Messages. So far, the examples we've seen have been mapping between the JMS Message body and the Spring Integration Message payload. Likewise, the JMS properties have been 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.

However, in a particular deployment environment it might be a well-known fact that Spring Integration based applications exist on both sides of the JMS Destination. One Spring Integration application would be playing the role of a producer and the other would be playing the role of a consumer. The interaction may be unidirectional, using the Channel Adapters we saw at the beginning of this chapter, or the interaction might involve request-reply exchanges where one of the applications contains an outbound Gateway and the other contains an inbound Gateway (keep in mind that a Gateway plays both the role of producer and that of consumer). If that is 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. In other words, to send a Spring Integration Message as the actual body of a JMS Message, simply 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 itself as the JMS Message body, it is necessary to have a serialization strategy. The standard Java serialization mechanism is one option, since Spring Integration Messages do indeed implement the Serializable interface. One thing to keep in mind when choosing that option,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

however, is that any non-Serializable values that are stored in the Message headers will not be passed along, since they cannot be serialized with that approach. An even more important factor to keep in mind is that Java serialization requires that the same class is defined on both the producer and the consumer side. 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 essentially a Serializable instance.

At first, it seems rather convenient to simply 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 do 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 (OXM) Marshaller and Unmarshaller implementations (see the XML chapter 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 is not necessary to have the exact 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/from completely different object types.

Regardless of the chosen serialization mechanism, you will need to configure a MessageConverter on the gateway any time that you do not want to rely on the default which uses Java serialization. You might choose the MarshallingConverter that is provided by Spring for Object-to-XML conversion or you might implement

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

MessageConverter yourself. Either way, just 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, since having the mapping behavior on both sides does promote 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 will map to a JMS TextMessage or BytesMessage, respectively, when relying upon 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. If, however, your Spring Integration payload or JMS body is a domain object, then it's definitely important to consider the degree of coupling since the default MessageConverter will rely on Java serialization at that point.

Spring Integration does provide bidirectional XML transformers within its XML module (again, see the XML chapter for more detail), and it provides bidirectional JSON transformers within the core module. Both the XML and JSON transformers can be configured using simple namespace-defined elements within 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 upon 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 are not 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 would not be very difficult to implement in the meantime. Of course, you can provide any

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

other custom logic or delegate to any other serialization library that you choose. If going down that path, you would simply define the chosen MessageConverter implementation as a bean and then reference it from the channel adapter or gateway's 'message-converter' attribute.

9.9 Managing Transactions with JMS Channel Adapters and Gateways

When working with JMS, it's important to understand exactly how and where to apply transactions on the producer side as well as 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 options are beyond the scope of this book, but the JMS provider's documentation would be a good place to start. If you are using ActiveMQ, its options are covered in detail within 'ActiveMQ in Action'. Here, we'll cover just the basics of JMS transactions and specifically how they can be configured on Spring Integration's JMS Channel Adapters and Gateways.

9.9.1 JMS Transaction Basics

Let's step back for a minute and 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 themselves as well as the Producer and Consumer instances. When creating a Session from a Connection, one must decide whether that Session should be transactional or not. Here is a simple example using the JMS API direction (without Spring):

```
Connection connection = ConnectionFactory.createConnection();
Session session = connection.createSession(true, Session.AUTO_ACKNOWLEDGE);
```

That boolean value (true in the example above) indicates that the Session should be transactional. You'll also notice that the second argument indicates the "acknowledge" mode for the Session. The two values are actually mutually exclusive. In other words, if you do provide a value of true to indicate that you want a transactional Session, the acknowledge mode value will actually be ignored. On the other hand, if you provide false to indicate that you do not want a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

transactional Session, then the value will play a role. We will briefly return to the acknowledge modes in a moment, but first let's discuss the publishing side where things are quite straightforward.

When publishing JMS Messages, that transactional boolean value passed when creating the Session plays an important role. The Session is itself used as a factory for creating any MessageProducer instances, and the transactional characteristics of those producers would reflect that boolean flag. If the Session is not 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, in the case that 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 itself.

When using a transactional Session, similar behavior is available on the receiving side. A MessageConsumer created from a transactional Session will not 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 essentially 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 above, when the transactional boolean value is "false", the acknowledge mode will play 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 will be the default. You can set the value to CLIENT_ACKNOWLEDGE instead. That setting indicates that you expect the client consuming the JMS Messages to explicitly invoke the acknowledge() method on those Messages. Actually an acknowledge() call on a single Message will acknowledge *all* previously unacknowledged Messages that have been consumed on that same Session. There is a third option, called DUPS_OK_ACKNOWLEDGE, which is similar to AUTO_ACKNOWLEDGE in that no explicit calls are required of the consuming client. It differs in that the acknowledgements may be sent lazily thus allowing for the chance that duplicate Messages might be received in the case of certain failures

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

within the window of vulnerability. The details of acknowledge modes are covered in the JMS specification itself. 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 essentially rolls these options into a single "acknowledge" property. It can take a value that corresponds to any of the three acknowledge modes as 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 will 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 very important final note: if no value is provided, the default will be "auto" acknowledge mode and hence Sessions will not be transactional. Transactions do add overhead, so if you are simply sending non-essential event data where

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

occasional message loss is not a problem, the default might be fine. However, if your messages are carrying document data to be processed, you will want to be sure to consider this setting carefully.

9.9.2 A Note about Distributed Transactions

The Spring Framework provides an abstraction for transaction management that makes it very easy to switch between different strategies for handling transactions within an application. In most cases, the only changes that are necessary are in the configuration. Most likely that change is limited to replacing a single 'bean' since the code depends on the abstraction. Furthermore, since 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 would expect for managing transactions regardless of the details of the underlying system.

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition) throws Transa

    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. We refer to those as "local" and "global" transaction management. What we mean by local is that transactions can be managed directly against the transactional resource itself without any need for a mediator. In the case of JMS the "local" transactional resource would be the JMS Session instance. As you saw above, 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 is also a "local" transactional resource since it provides commit and rollback methods. While 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. In other words, explicit commit (or rollback) is then required, so the Connection is then *transactional* in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

that multiple operations can be handled within the scope of a single unit-of-work. So, 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 is where Spring's PlatformTransactionManager can step in to provide a consistent view. In fact, there are implementations available out of the box for these two cases: JmsTransactionManager and DataSourceTransactionManager.

Earlier we mentioned a common case where a unit-of-work actually 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 upon XA-capable resources and a JTA implementation that supports 2-phase commit as the transaction manager. However, it's not always necessary to go that far. A "distributed transaction" in the more general sense just means that there is more than one resource involved. Rather than assuming you need to use XA and 2-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 "chaining" multiple local resources together. The tradeoff 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. Of course, in the case that everything goes smoothly, the database insert would be committed, and the JMS message would be 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 afterwards. 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 other words, a record might be ignored if the same data already exists based on a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

where clause condition. If the operation is not inherently idempotent, there might be a bit 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 any more. 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 indeed be simpler and add less overhead than a full XA 2-phase commit option. For much more detail on these types of distributed transaction patterns and the tradeoffs involved, refer to the following article by Dave Syer: [http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html]. Dave is the lead of the Spring Batch project and a committer on several other Spring projects, including Spring Integration.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

10 Going e-postal

If JMS is the first thing that comes to your mind when associating Java and messaging, then there's a big chance that doing the same for messaging and internet will make you think of email. We all know it very 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 indispensible 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 vs bottom-posting : we are interested only in the implications of using email as an enterprise application integration tool. Because, besides its primary role as a means of communication between people, e-mail can also be a useful interaction medium in applications, either with users - by sending notifications or receiving requests, or with other applications as well, due to its firewall-friendly nature and ability of transporting data payloads as attachments.

In this chapter, we will provide you with an overview of the most significant use cases that require email support, and you will learn how you can use Spring Integration for adding it to your application. We will show you how to send and receive e-mail, what are the various options that you have for doing that and how to choose between them. Because sending e-mails is a more typically encountered use case, this is what we will begin with.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 just 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 (which may just be your print of 'Spring Integration in Action'), yet you do not know the exact date and time when you have to be at home to receive it. You may try to find out when you are supposed to be home by repeatedly checking the shipper's site, but this can become quickly a tedious exercise, and very often a futile one, as the information may change frequently. "Hold on", you say, "I'd rather get a notice when you *know* something" and subscribe to receive a notification when the status changes. Now you may carry on with your own business, assured that you will stay informed without having to dedicate too much of your time for that.

You can find many other examples where a notification system is a valuable add-on to an application, and in order for a notification system to be efficient, it must use a mechanism which allows easy access on the recipient's side. This assumption holds very well for e-mail: it is widely available, either as a public service or as part of the corporate communication infrastructure, and since mobile devices become more popular, e-mail is accessible even while on the go.

While e-mail is a widely used notification method, there are other ways of sending short notifications to users, such as SMS and social media - tweet and chat. What is unique about e-mail is that it can also be used as a transport for message exchange between systems. It is not usual to do so for integrating applications that are hosted on the same machine or deployed in the same local network, where specifically designed messaging middleware, accessible via JMS or AMQP is a far better fit when considering any relevant criterion (performance, throughput, transaction capabilities, delivery guarantees), but e-mails can be used for integration over the web, where they can bear data payloads as attachments. They also do not require any special setup for remote access, like exposing ports and addresses, as a message broker may normally require.

The process of sending e-mail from within an application can be divided in two logical steps: preparing an e-mail message by populating its subject, body,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

attachments and recipients list, and the actual operation of sending it. A complete solution involves a chain of Spring Integration endpoints with specific roles, typi

Figure 10.1 Sending e-mail requires a channel adapter and includes optional transformers for preparing outbound messages

Since the key component of this chain (and actually the only one whose presence is mandatory) is the channel adapter, let us begin by introducing it first.

10.1.1 The outbound channel adapter

The actual operation of sending e-mail messages from your application is fairly straightforward and intuitive, considering that it is based on a pattern you are already familiar with: using outbound channel adapters. As a first step, you will need to configure such an adapter, which can be as simple as in the code snippet below, where the username, password attributes describe the login credentials for using the SMTP services provided by the host.

```
<mail:outbound-channel-adapter channel="outboundMail"
    host="${host}" username="${user}" password="${password}"/>
```

What you actually sending a message can be, in the simplest case, equivalent to the following snippet: the outbound channel adapter will compose message with the payload as body, and the more recipients lists (to, cc, bcc) and subject as header values.

```
Message<String> message = MessageBuilder("Hello World!")
                          .setHeader(MailHeaders.TO, "recipient1@samplehost.com")
                          .setHeader(MailHeaders.SUBJECT, "Greeting")
                          .build();
outboundMailChannel.send(message);
```

We said equivalent, since sending mail in a programmatic fashion is not the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

only way, and perhaps not even the most common: e-mail 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 e-mail channel adapter listens, which composes an e-mail message based on payload and header content and sends it.

One aspect to be taken into consideration when composing messages for the e-mail 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 that is accepted by the channel adapter is a byte array, case in which the resulting e-mail message will add the binary content as an attachment. You can use additional headers for describing the attachment - file name, content type, etc. A complete list of the message headers and how are they used for constructing the outgoing message can be found in Table 10.1. The header name refers to String constants defined in the MailHeaders helper class.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 10.1 Spring Integration mail message headers

Header Name	Meaning
MailHeaders.FROM	The sender of the message
MailHeaders.TO	List of recipients
MailHeaders.CC	List of carbon-copy (CC) recipients
MailHeaders.BCC	List of blind carbon-copy (BCC) recipients
MailHeaders.SUBJECT	Email subject
MailHeaders.REPLY_TO	Reply-to address
MailHeaders.MULTIPART_MODE	Indicates whether the message supports alternate texts, attachments and inline content - used only for byte array messages
MailHeaders.CONTENT_TYPE	Can be used for setting an alternative content type for message body (e.g. for sending HTML messagesi instead of plain text)
MailHeaders.ATTACHMENT_FILENAME	File name of the attachment - used only for byte array messages

There are more advanced ways of composing e-mail messages and configuring outbound e-mail channel adapters. In order to understand them, we need to take a closer look to the innards of the mail-sending process.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

10.1.2 Advanced configuration options

Under the hood, the Spring Integration e-mail outbound channel adapter is based on the Java Mail API and additional utilities provided by the Spring framework itself.

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 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">
[CA]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">smtps</prop>
  <prop key="mail.smtps.auth">true</prop>
  <prop key="mail.smtp.starttls.enable">true</prop>
</util:properties>
```

For applications that need to send e-mail (especially the ones that do not use Spring Integration), using the Java Mail API directly can become quickly a cumbersome exercise, which involves operating directly with the low-level components of the API and direct manipulation of resources. In order to simplify the general process of sending and receiving e-mail, the Spring framework provides a higher-level API for sending e-mails, the JavaMailSender, which simplifies the task of sending an e-mail.

In a nutshell, sending or receiving e-mail, requires interacting with instances of the type javax.mail.internet.MimeMessage, which is the Java Mail representation of an e-mail message. When receiving messages, dealing directly with such objects is fairly simple, since they provide access to all the properties of the e-mail (as you will see 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 is simply impossible to separate concerns in a good manner and create a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

se ture-independent
e- ed for unit testing
su r abstraction, an
ir ilMessage, as an
ir ages, as shown in
fi e-mail section of
'S

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 on the right side

The usage of 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 infrastructure configuration for sending emails (rather than, for example, providing this via attributes on the channel adapter element). On the other hand, the outbound channel adapter accepts a two other types of object as payload - the raw JavaMail MimeMessage and the Spring-specific MailMessage, which allows developers to use compose messages using the Spring framework utilities, and send them through the channel adapter.

Sending e-mail is usually part of a larger chain of events, and in order to illustrate that, let us consider a business example. You have seen in chapter 6 that, as soon as the application is notified of a flight schedule change, it must inform all the customers that are booked on that flight. The upstream service adapter will take care of searching for accounts and producing one or more Notification objects.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

10.1.3 Transforming outbound messages

So, what can you do with messages whose payloads are not Strings, byte arrays or MailMessage/MimeMessage instances? It may seem, after all, that this channel adapter will only fit a bunch of corner cases. One of the strengths of Spring Integration is decoupling between components, so it may not seem too far from reality that a service adapter will not want to produce dedicated MailMessage instances, but domain-specific Notifications, as we described above, leaving to the downstream components to translate them into "e-mail ready" instances. You may need this also when supporting more than one kind of notification type by the application - besides e-mail, we 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 e-mail sending component being just one of the subscribers.

The pipes-and-filters architecture of Spring Integration has a simple solution for this: all that needs to be done is to is to include a message transformer that converts the message with a POJO payload into one with of the four compatible payloads, as shown in figure 10.1.

Generally speaking, however, 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 SpEL expressions evaluated against the message to be enriched. In general, the best practice is to insert the header enricher, since the transformer may remove information which is 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 snippet below.

```
<service-activator input-channel="notificationRequest"
    output-channel="notifications"/>

<header-enricher input-channel="notifications"
    output-channel="enrichedNotifications">
  <to expression="payload.email"/>
  <from value="${settings.email.from}"/>
  <subject value="Notification"/>
</header-enricher>

<transformer input-channel="enrichedNotifications"
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
  output-channel="outboundMail"
  ref="templateMessageGenerator"/>
```

This concludes the section dedicated to sending e-mail. It is time now to move to the next section, which will discuss how to handle inbound email messages.

10.2 Receiving Email

While sending notification e-mails is the most common use case, a significant number of applications also support an e-mail driven application flow. Sending documents as e-mail 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 web site. Also, e-mails carry the identity of the sender, which works pretty well in request-reply scenarios, since the identity of the sender and the reply-to information is communicated as part of the e-mail. And, as we mentioned in the previous section, e-mail can serve as transport for carrying data over the web, where using a actual message broker would create too many logistical and infra

For example, your application may offer the opportunity of performing flight status checks through e-mail. The company provides an e-mail address to which customers will send an e-mail with the the keyword "STATUS" and the flight number in the subject, and the application will reply with the flight information for today - this may be useful if the

Our application will act as an e-mail client, reading the contents of the mailbox and processing every incoming message as a request. The general structure of an e-mail receiving application is shown in figure 10.3. The inbound channel adapter will create messages with a Java Mail MimeMessage payload which 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.

Figure 10.3 Receiving e-mails requires a channel adapter and includes optional transformers for extracting message content

The first step for creating such an application is deciding how to receive e-mails

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

- or, otherwise said, what kind of a channel adapter you would like to include. You can either poll the mailbox or be notified when new messages have arrived, so let us examine these two options first, before we discuss how to process inbound messages.

10.2.1 Polling for e-mails

Polling is the most basic e-mail receiving strategy and it is supported by all application servers. A client will connect to the mailbox and download 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>
```

While the example above describes an IMAP-based channel adapter, Spring Integration supports both of the two major e-mail access protocols: 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 the above example indicates that secure IMAP, 'imaps' is in use, and the set of Java Mail properties set on the adapter. The actual protocol that should be used in a particular case depends on the setup of the mail server where the incoming mailbox is located, and it is not typically left to you as an option.

While considering the differences between protocols may not help you choose one over the other, it may still play an important role in understanding how the work and what consequences do 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 e-mail client. This means that the server is somewhat aware that a message has been read, but only for the duration of a session. In Spring Integration, an e-mail client session is started when the application starts and is closed on shutdown, therefore any messages which are not deleted from the server before shutdown may be received multiple times. So, in to solve the issue of duplicate messages, an application can choose to delete messages

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 - since there is no such concept as a transaction, a failure that occurs between the moment when the e-mail has been read and the moment when it has actually been processed runs the risk of 'losing' the message. Moreover, using the mailbox in a multiple-client, publish-subscribe scenario would be problematic if one of the subscribers starts deleting e-mails. Since there is no single best choice on this, Spring Integration forces you to state explicitely which one of the two parameters you would like to set, essentially deciding between providing a contingency on loss or duplication.

On the other hand, IMAP is a protocol designed for storing messages remotely. Mailboxes are the permanent storage for emails, and e-mail clients will, by default, just provide a view of the remote mailbox state, rather than retrieving content locally. Since the remote mailbox does actually have a state, deleting messages is not really necessary in order 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, since POP3 does not provide an updatable persistent mailbox state.

IMAP provides a more robust alternative to POP3 in polling scenarions, but it's true advantage is that, in certain circumstances, allows replacing polling with event-driven reception.

10.2.2 Event-driven e-mail reception

While polling is an option that is available for any e-mail server, for particular server configurations it is also possible to set up an event-driven inbound channel adapter for receiving e-mails. This functionality is available when connecting to IMAP servers that support IDLE, a feature that allows clients to request for real-time notifications on mailbox changes. As a result, clients do not have to poll the server for new messages: they will be notified when new messages are added to the mailbox.

This approach is a bit lighter on resource consumption and traffic, as the application does not have to actively check the server for new messages. It also improves responsiveness, as applications will be notified as soon as a message arrives, without the lag introduced by a polling cycle. However, it is important to note that the 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 to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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"/>
```

Either if it is polling or event-driven, the inbound e-mail channel adapter will create a message with a payload of the JavaMail's Message type. For handling these messages in the application, they need to be transformed to a more neutral format - like a String or a POJO.

10.2.3 Transforming inbound messages

Once we have configured our inbound channel adapter, the next step would be to process the received messages downstream. For handling the lower-level JavaMail Message objects withing the domain-aware services, it is typical to include a transformer before the objects enter the application flow.

The simplest possible solution is by using a SpEL-based transformer that is evaluated against the JavaMail Message payload of the inbound message. For example, the transformer in the following snippet will extract the message body and will create a new Spring Integration message using it as a payload. Of course, you can write more complex expressions including other properties of the message like the subject, sender or date.

Listing 10.1

```
<transformer input-channel="emails" expression="payload.content"
[CA] output-channel="transformed-emails"/>

<service-activator input="transformed-emails"
  output-channel="handled-emails" ref="emailHandlingService"/>
```

Spring Integration provides an out-of-the box transformer that you can use for extracting the e-mail body or the binary content into a String payload. One of the benefits of using it is that it also extracts the other e-mail properties as message headers, mirroring what happens when you send a message - check Table 10.1 for a reminder on the mail property/message headers mapping. An example can be found below.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Listing 10.2

```
<mail:mail-to-string-transformer input-channel="emails"
  output-channel="transformed-emails"/>
```

If you want to customize the transformation rather than extracting the payload as a String, you can replace the `<mail-to-string-transformer>` above with a regular transformer that accepts an inbound javax.mail.Message payload. You do this easily by extending the AbtsractMailMessageTransformer class provided by the framework and implementing the logic for extracting the message content as in the snippet below.

Listing 10.3

```
public class MyMailMessageTransfomer
[CA] extends AbstractMailMessageTransformer<InputData> {
    @Override
    protected MessageBuilder<InputData>
[CA] doTransform(javax.mail.Message mailMessage) throws Exception {
        InputData inputData = new InputData();
        // populate inputData from message
        return MessageBuilder.withPayload(inputData);
    }
}
```

After which you can add your implementation to the chain using a regular custom transformer setup.

```
<transformer input-channel="emails"
[CA] output-channel="transformed-emails">
  <bean class="example.MyMailMessageTransformer"/>
</transformer>
```

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 provide automatically access to entries such as the originator of the message or the reply address, which may be useful anyway when handling the message, but are especially important if you want to implement an ad-hoc "gateway", that sends back an e-mail response to an inbound e-mail request. For implementing such a round-trip solution, all you need to do is to create a sequence of endpoints that starts with an inbound channel adapter and ends with an outbound one.

And by getting here, we have completed our overview of how Spring

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Integration supports receiving e-mail, with which our coverage of e-mail integration is now complete. Let us recapitulate quickly this chapter's takeaways.

10.3 Summary

E-mail is a widely-used modern communication technology that can be used either for user interaction, allowing applications to send notifications or receive requests via e-mail from users, but also as a transport mechanism especially between applications that are not located in the same local network. On criteria such as performance and reliability, it cannot compete with dedicated messaging middleware, but it has some qualities that make it attractive in internet communication: it is widely available, accessible to users, can transfer information over the internet without requiring firewall access, and supports small-to-mid sized attachments.

Spring Integration provides support for sending and receiving e-mails through dedicated channel adapters, which cover all the major connectivity options and e-mail transfer protocols (POP3, IMAP, SMTP). It provide out-of-the box support for handling the most typical use cases -text messages (including HTML) and file attachments, whereas special situations such as complex e-mail structures or conversion to POJOs can be easily addressed by including customized transformers in the processing chain.

This wraps up our introduction to e-mail support in Spring Integration, and it is time to make room for our next topic: file support.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

11 File system integration

EIP defines 4 basic integration styles: file transfer, shared database, RPC and messaging. This chapter deals with 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 (e.g. are both systems streaming to the same large file, or are there multiple files moved around) are not part of the definition. After reading this chapter you will 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 JMS and email. In many ways messaging based integration is desirable over file based integration, but 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. Very often one process writing a file in a directory and another reading from that directory is 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 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 will pose the need to interact with the file system. In this chapter you will read how to do this with Spring Integration, but also some more generic pointers around designing interaction with the file system. At first, in section two we will go into writing files; the details around reading files

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

will be discussed in the next section. Fourth section will describe different ways of dealing with files. Finally we will dive into the details of how the different components work under the hood, and discuss several advanced problems that you might encounter.

Before going into how we will interact with the file system, let us first carefully examine *why* it would make sense to do so in the first place. An example is in order too.

11.1 Can we be friends with the file system?

Most Java developers dread dealing with the file system like they dread dealing with concurrency. In fact 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've 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 by far the best friend you are going to have in your programming career. In many more cases than you'd expect things like the file system can be safely ignored as long as you keep searching for that one holy grail: the simple solution that just works.

Of course something common like the file system 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 like the jvm dealing with the file system requires an extra abstraction and is not the simplest solution. Let's list the advantages and disadvantages of using files before we move on.

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
There is much more room on the disk than in memory, so running out of room is less likely	Access to the file system is much slower than memory access
Files are easier to integrate in other applications than more advanced data stores	There are no ACID or REST semantics, and more complicated scalability problems

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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* like that look at the file system seriously before you jump into even more complex solutions. We're going to look at an example that can best be implemented using file sharing. The rest of the chapter will teach 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 file system integration is an editor. In this chapter we will 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 go a little bit 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. It will not surprise you that for this example, we will make it a point to worry about those things. In our flight booking application you can connect with friends and keep a diary of the trip online. We expect heavy usage, we don't want to bombard the database with updates and we do not want to keep everything in memory either. Putting it this way the file system seems like a decent middle road. To keep the undo history we will create small files with changes and store them alongside the base file. Whenever a change is done in the editor, a file is created which contains that change.

Now when the editor is closed we can reload the file and the full undo history at startup, but the organisation we've chosen also allows us to do something even better. We could 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 will pick up the file and update their screen.

The examples discussed here could however easily be extended to a rich text editor, a spreadsheet or something else. The only restriction really 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 our application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 it's user back to the working directory.

Let's explore the responsibilities of the components in figure 11.1 in more detail. First a single client that creates a new document to work on. This results in a directory for the document, an empty file as a starting point. Then after each change a file with the change is created.

SIDEBAR

Change

A change is a consistent set of manipulations to the document. It is recorded between two cursor positions.

We might want to more fine grained modifications later on, perhaps we'd like to update other clients on a key by key basis, but for the sake of the example this will work.

Now if another client joins in things get interesting. If both clients watch the directory with the base file and changes they will 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. With an appropriate scan rate and change size this can lead to a seamless collaboration between different users.

We've chosen this example because it requires the right interactions with the file system to be suitable as an illustration of Spring Integrations file support. The first thing that you should remember about file support in Spring Integration is the namespace. It is not required 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 it becomes important to consider the underlying plumbing in detail.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

SIDEBAR

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:xsi="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/s➡
pring-integration.xsd
http://www.springframework.org/schema/integration/file
http://www.s...org/schema/integration/file/spring-in➡
tegration-file.xsd">
```

Notice in particular the addition of `xmlns:file="http://www.springframework.org/schema/integration/file"`. This is where we add the file namespace.

The file namespace will allow you to use the shorthands 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.

In the next section we will use the namespace to configure a the file writing leg of our application. After that we will have need to read files from the working directory too. After that we will look a bit more into interpreting file content. Finally we will explore some more advanced use cases. But first we focus on writing.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

11.2 Writing Files

Writing a file to the local file system 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, an array of bytes or a string to write to the file and a file name to write this data into. If you would try to do this using the raw File API you would have to write some less than elegant boilerplate code that you have probably reprocuded 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 as will be detailed in the next few sections. The only thing left to do is wire them up in your application context.

Figure 11.2 Schematic design of the Spring Integration File Outbound Channel Adapter. The outbound adapter is a passive component that reacts on incoming messages by writing their payloads to a file.

More interestingly there are some concerns related to the intended reader of the file that need to be taken into account. What are we going to do about encoding? If writing a file takes a long time, how do we prevent readers from picking up incomplete files? In this section we focus on the standard solution to this problem, at the end of the chapter we will go into more complex scenarios.

The easiest way out of a concurrent read write bug is to write to a file until it is done and only then move it into a place where the reader can pick it up. This is exactly what the outbound adapter described in the next section will do by default.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

11.2.1 Configuring the file writing endpoint

To setup a an outbound channel adapter you need to use the `<file:outbound-channel-adapter/>`. This will create a file writing component that will write the payloads of messages to a directory of your choosing. There are several things you can configure for this element. The following table lists the attributes you can use to do this configuration.

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 will be received. If no channel is specified a channel will be created automatically like with any other channel adapter.
output-directory	The directory where the files should be written to. 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 will be deleted.

Of these attributes only id and output-directory are required. In keeping with good practices in Spring Integration components there are reasonable defaults for the other properties.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

WHAT DOES THIS DO?

When a file is received from the input channel the file channel adapter will open it up and look at the payload. Several payloads are supported like`byte[]`,`File`, `String`. If the payload can be written to a new file a *temporary* file will be opened. This file will have a suffix so that readers can prevent opening files that are not ready yet. Then the adapter will write the contents to the file and if all went well it will move 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 it's needed to do something with an input file after it has been written to a directory successfully. See alsodealing with files. To do this you can configure a`<file:outbound-gateway/>`. This component behaves in a way very similar to the channel adapter, but it allows you to send the created file to another endpoint immediately. This can be an excellent option when it is required to notify another service when the file is written.

WHAT COULD GO WRONG?

There are several things you should consider when writing files. Of course you could encounter an`IOException`, for example because you suddenly lost the ability to write to the disk. In this case the a `MessageDeliveryException` will be thrown containing the original `IOException`. There is nothing special about this failure, only that Spring Integration allows you to let the exception bubble as a `RuntimeException`.

It is also possible that you overwrite an existing file unintentionally. Be careful to implement a custom `FileNameGenerator` in such a way that it doesn't generate the same file name for files that should be named differently. The default name generator will ensure that within the same application context the names are unique, but you will probably need more control over the filenames. It's also important to avoid overwrites between multiple runs of the application, be it in parallel or in sequence.

Now we have the tools, so it is about time we try out. With the outbound adapter we can implement the saving part of our collaborative editor.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

11.2.2 Writing increments from our collaborative editor

In our example application introduced earlier we've 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 when we discuss how to deal with files you will learn how we can 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 that allows an endpoint processing it to apply it to a string. The details of applying the changes are quite tricky, but we won't go into detail on those now. It's not the responsibility of the channel adapter to ensure the content of the files is in an appropriate format for the reader.

Figure 11.3 The writing leg of our 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.

At this point it's good to inspect the file writing related code.

```
&lt;file:outbound-channel-adapter
        channel="outgoingChanges" directory="#{config.diary.store}"
        auto-create-directory="true" filename-generator="nameGenerator"/&gt;

&lt;bean id="nameGenerator"
      class="com.manning.siia.trip.diary.ChangeFileNameGenerator"&gt;
    &lt;constructor-arg value="#{config.processId}"/&gt;
&lt;/bean&gt;
```

Now we have one leg done, we can write files to our work directory. But before we can actually work with them in our collaborative editor we need the other leg too. So let's quickly move on to the reading of files.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

11.3 Reading Files

Reading files is a little bit more complex than writing. Actually the process of reading itself isn't complicated at all. Once it is clear which file should be read it's just 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();
}
```

There is some boiler plate in the previous snippet, and judging from countless examples where people have messed this up, it is quite error prone.

NOTE	**Automatic Resource Management in Java 7** Closing resources properly might become easier in the future with Automatic Resource Management. This is a suggested improvement to the Java language by Josh Bloch, which likely will be in Java 7.

Although it's interesting to discuss the low level correctness of file system access it is the primary goal of this chapter to deal with the functional complexities of dealing with files. Back to the example.

The Trip Diary module needs to read files from a directory to update the displayed diary with changes from other editors.

The complexity lies in determining what 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. On first sight this seems simple, but this is actually quite tricky. Imagine that you are looking at a directory that someone else is dropping files in. How can you know what files to pick up and what files you've already processed? How do you ensure that there is nobody still writing to that file?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 11.4 Design of the Spring Integration Inbound Channel Adapter

By default Spring Integration's file inbound channel adapter will read all files in a certain directory, but once read a file will not be picked up again. As a user you can customize the behavior by specifying which files should be read, you can do this through `FileListFilter`s. More on this later. First we'll make sure to understand Java's file API a little better.

11.3.1 A File in Java is not a file on your disk

Central to the file system interaction in Java is the File class. A File is immutable. This means that once created it cannot change anymore. This is a good thing, because as mentioned earlier, immutable payloads make your life much easier once you start 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 sleep too soon. We all know that files change all the time, so there must be something more to it.

WHAT IS IN A FILE?

Looking at the File source code will reveil to the astute where problem is hidden. The File's path is indeed effectively immutable. Most operations on File however do not access immutable fields, instead*they access the file system*. You can understand that getting the last modification date requires accessing the filesystem, to check if the file was changed lately. However, this means you have to be very careful to distinguish between operations that do 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 file system access. Good lazy programmers shun responsibility.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

KEEPING IT SIMPLE

A common tactic in dealing with this problem is to make sure that while writing files you make sure reading processes don't see them yet. For example by writing in a directory that isn't visible to the reading process, or, like we do automatically in Spring Integration, write 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. Just exclude in progress files using a `FileListFilter` and let Spring Integration deal with the visible files with the default behavior.

But how exactly will we make Spring Integration do all that? The next section will show you the details of the configuration.

11.3.2 Configuring the file reading endpoint

The `<file:inbound-channel-adapter/>` element will create a file reading adapter. The attributes for this element are listed in the following table.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 11.3 Attributes of the <file:inbound-channel-adapter/> element

name	description
id	The id of the endpoint or the implicit channel leading to it (see channel)
channel	The channel on which messages for the files in the input directory should be placed. If no channel is specified a channel will be created automatically like with any other channel adapter.
directory	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 in what type of `ApplicationContext` this bean is loaded.
filter	Specify a custom filter to determine which files are to be picked up. This overrides the default filters.
scanner	Specify 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 is injected into it. Look at `RecursiveLeafOnlyDirectoryScanner` for an example implementation.
comparator	Custom Comparator that is used to determine the order in which files will be received.
filename-pattern	Ant style pattern that will be used in a filter that is applied to the files in the directory. This filter is *added* to the default `AcceptOnceFileListFilter`, so that still there will be no duplicates picked up. This option should not be used together with the filter attribute.
regex-pattern	Similar to filename-pattern, in this case 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, like with a <service-activator/>.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

WHAT DOES THIS DO

You have quite a few options here, but since 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 would already pick up files written to the work directory and send them onto a channel called fileGuzzler wrapped in a message. It would also make sure they are picked up only once during the life time of the application.

When `filter` nor `filename-pattern` are specified the `FileReadingMessageSource` that is created will use an `AcceptOnceFileListFilter` by default. This filter will remember all files it has seen before and not let them pass again. The default filter works quite well in combination with for example a `SimplePatternFileListFilter` that excludes unfinished files as agreed with the writing process as described in section11.8. However, if the writing is not under your control you might have no option but to write a more elaborate (and more stateful) `FileListFilter`. To do this you can either extend`AbstractFileListFilter`, or implement `FileListFilter` directly.

When the poller invokes `receive()` on the internal `FileReadingMessageSource` it will list the files in the directory, add them to an internal queue and let the poller iterate over the files until it has satisfied its `maxMessagesPerPoll`. The internal queue is prioritized according to the `Comparator`, if provided. Otherwise the files will come out in natural order.

In the next section we will again apply the tools to our application. The tricks we've just learned will help us implement the reading leg of our application.

11.3.3 From the example: picking up incremental updates

Now that we've cleared up how to read files we can 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 11.5 Inbound components of the collaborative editor: a file is picked up from the working dir and sent on the incomingFiles channel wrapped in a message. It is then transformed into a String and passed along to the editor.

If we distinguish each editor uniquely by a key generated on startup, restarting A will result in a running editor C, where C != A. This is convenient, because this way a newly started editor can read all the changes to the file, even a restart will read all the files written by the previous editor run by the same user. As mentioned before we could use a combination of a timestamp and process id to be quite sure there is no duplication.

```
&lt;file:inbound-channel-adapter
        channel="incomingChanges" directory="#{config.diary.store}"
        filter="onlyNewChangesFilter"/&gt;

&lt;bean id="onlyNewChangesFilter"
    class="org.springframework.integration.file.CompositeFileListFilter"&gt;
    &lt;constructor-arg&gt;
        &lt;list&gt;
            &lt;bean class="org.sfw.int.file.AcceptOnceFileListFilter"/&gt;
            &lt;bean class="...RefuseWrittenByThisProcess"&gt;
                &lt;constructor-arg value="#{config.processId}"/&gt;
            &lt;/bean&gt;
        &lt;/list&gt;
    &lt;/constructor-arg&gt;
&lt;/bean&gt;
```

You've learned in this section how to configure a file reading component and we've discussed the more elaborate details of the possible configuration options. In the examples you can clearly see that the typical boilerplate code is replaced by somewhat more readable declarative xml. Our collaborative editor now has both legs to stand on.

Great, now we can read and write files. But what are we going to do with them inside our application?

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

11.4 Handling file based messages

When a file is read and wrapped into a message, we need to do something with it that makes sense within the context of our application. Only very rarely is a `File` something that is meaningful within the application domain. If it isn't it needs to be converted into something that does make sense. Typically this will be an object from the domain. The process consists of two parts, first the contents of the file need to be read into memory, upon which the raw data can be converted into domain objects. The last process is called unmarshalling, and there are plenty of frameworks that can do it, for example Spring OXM which is discussed in more detail in chapter 8. Spring Integration file support does not care how you do your unmarshalling. So even though it is an important concern, it 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 will discuss it from several angles in this section.

11.4.1 Transforming files into objects

There are many different types of content that could reside in a file. However from a high level view it's all just bytes. The built in transformers of Spring Integration can transform `File` objects into either a `byte[]` or a `String` for you. To do this you can use a `FileToByteArrayTransformer` or a `FileToStringTransformer` respectively. Transforming a File means your system is going to deal with it. At this point you might choose to delete the file. If the system fails to deal with the file's content successfully however having deleted the file might cause data to drop off the board.

Imagine you are 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 needs to be transformed into a String, and after that an unmarshalling step will prepare the objects that will ultimately be persisted through a repository. If the transformer deletes the file it will do so after it has finished the transformation, but it is not unthinkable that subsequent requests are separated by asynchronous hand-off. In this case it is obvious that the file will be deleted before the objects are persisted, but even if there is no asynchronous hand-off problems can happen because the transaction boundaries are broken. If the process crashes before the file is deleted, but after the inserts have been committed, restarting the process might cause duplication.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

In the next section we will outline some common file handling scenarios, discussing the pros and cons of each approach.

11.4.2 Common scenarios when dealing with files

There are several ways of managing files on disk from several applications. There is a lot of passionate discussion on how this should work and what what is the proper way of doing it. In the end, as usual, it depends. However, there is no reason we can't have a look at a few different strategies. Three strategies will be discussed here: 1. just let the file sit in the input directory, 2. delete the file as soon as it is consumed by a transformer, 3. simulate a transaction that attempts to make delivering the message and deleting the file an atomic operation.

JUST LEAVE IT

If there is no harm in picking up the file again when the application is restarted, or if there is a clear way to filter out old messages, it is often best to leave the files in the input directory. If dealing with a file is a non destructive operation it is 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 if a file was processed based on its name or location. Of course if a new file is created that can be correlated to the input, or if the application itself can be queried to determine if a file was processed this problem can be worked around.

Usually if the files are left on the file system, it is quite easy to come up with 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 very hard to write. It is important however 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 obviously.

DELETE ON CONSUMPTION

Sometimes incoming files are not very valuable and dropping a file or two is not a big deal. For example if a weather service gets forecast updates every minute, dropping one when it crashes will not be noticed when the service restarts. In this case deleting the file from the transformer makes perfect sense. However 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. Never go this way if you cannot afford to lose some data.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

SIMULATE TRANSACTION

There are cases where the files are to big to just leave them, but you can also not afford to lose a single one. In this case you cannot delete the file from the transformer, as at that point you could lose the data if the jvm is killed. You will have to delete the file in process however. 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 has been fully processed. Usually the process should write a file to a 'processed' directory after it is done with it. This could be done with a <file:outbound-channel-adapter ... delete-source-files="true" />. This is not a real transaction however. If the file system is full, or the process crashes in the middle, input files could still sit in the input directory even though they are already processed. To mitigate this risk you could 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. These are the the `<file:file-to-bytes-transformer/>` and the `<file:file-to-string-transformer/>`. Each of them supports the same attributes, so the only difference is the type of the payload of the outgoing message. The following table shows a short description for the allowed attributes:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 11.4 Attributes of the <file:file-to-string-transformer/> element

name	description
id	The id of the endpoint.
input-channel	The channel from which the file messages that should be transformed will be received. Can only be omitted if the transformer is part of a chain.
output-channel	The channel to which output of the transformation is sent. When left out the the result will be sent to the reply channel set as a message header.
delete-files	Boolean flag that determines wether the file payloads of incoming messages will be deleted by this component. `false` by default.

The configuration of this element is almost trivial. Only input-channel and id are really needed, in a chain the element can even be used without any attributes. Setting the delete-files attribute to true is equivalent with choosing the "Delete on consumption" strategy. There is a good reason that this attribute is set to false by default, as you probably would not be very happy if your application had deleted a file even though the content wasn't processed correctly. In some use cases it's perfectly reasonable to not have strict guarantees around 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 finally see the application work, so we'll put the ends together in the next section.

11.4.4 Applying incoming changes to the collaborative editor

In our collaborative editor the incoming changes contain snippets of text and a position at which they should be inserted in the overall text. We're going to avoid diving to deep into the tricky domain at this point, if you're interested in the details you will find it interesting to check out the sources and play around with them for a while. As far as Spring Integration support is concerned it is just about reading the files and passing the resulting strings along to the editor. Whatever tricks the editor needs to do with the supplied strings, the transformer is blissfully unaware of.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 quite simple, and if the file interaction you're planning from your application is as simple as this, there is no more to it than configuring inbound and outbound adapters and a transformer. However, as mentioned in the introduction of this chapter the devil is in the details. If the details become important the next section will help you find your footing when dealing with issues around for example locking, files being picked before they're ready or files being picked up multiple times.

11.5 Under the hood

This section deals with the nitty gritty details of file system integration. It is interesting for people that 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 file system integration support. If you need to implement more elaborate file related use cases, painful debugging can be avoided when the underlying concepts and rules are better understood. Reading this section in advance is highly recommended in such cases. Here we will 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 will therefore naturally focus on the `FileReadingMessageSource`.

11.5.1 `FileReadingMessageSource`

The FileReadingMessageSource is the core of the <file:inbound-channel-adapter>. It is a completely passive component that should be polled on its `receive()` method. The endpoint that the component is wrapped in by Spring Integration will take care of the polling transparently.

When a poll occurs for the first time it will follow the steps depicted in figure11.6.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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, 4. return the head of the queue wrapped in a message.

On subsequent `receive()` invocations the internal list will only be synchronized with the filesystem if it is empty, or if the `scanEachPoll` flag is set to`true`. This can have implications on the internal ordering which will become clear in the next section.

ORDERING FILES

In some cases the order in which files are read is important. Our collaborative editor for example is highly dependant 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 is ordered. However the order is not well defined on different operating systems and under different configurations. To keep a long story short: it is not a good idea to rely on the ordering of this array. The `FileReadingMessageSource` doesn't. Since ordering is required in some cases as mentioned, it does keep the files in a `PriorityBlockingQueue` internally. This will 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 are not written in order it can occur that the Queue needs to be reordered just after a file was wrapped in a Message and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

sent on a channel by a poller. If it turns out that the file just sent should have come after the file added to the queue just after the ordering is broken. There is nothing the `FileReadingMessageSource` can do about this. We wouldn't be telling you this if there was nothing you could do about this however.

If you write the files into the directory in the order you want them to be received in *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 the files might be received in the right order, but if you tweak the concurrency settings this might change. It is never a good idea to depend on message order implicitly. It is better to avoid depending on order, and if you can't you should use a Resequencer. This makes the system more robust and easier to maintain. Another problem that we've already touched upon in the introduction of this chapter is the risk of reading unfinished files. It is good to consider it in some more detail before moving on.

READING UNFINISHED FILES

As mentioned in section 11.3 it is usually important to pick up files only once the writing process is done writing to them. For example: the built in transformers (see section 4) 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 is 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.

In the next few paragraphs you will learn about typical concerns and solutions regarding the problem of reading unfinished files.

The best way of avoiding premature reading of a File is to have the writing process decide when the file may 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 of its own. One of them is that the move operation is not 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 something that is typically configured from the outside in a properties file, so what type of directories should be used now needs to be part of the documentation of the application.

If the writing process out of the control of the reader it might still be possible to determine if 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 do not have the last end tag.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 will need to open the file and read the last part of it. This obviously performs a lot worse than the move when ready strategy mentioned before.

Another tricky problem arises when the writing process is writing in multiple parts of the file concurrently. Now the last part could be done while the middle part is still under construction. BitTorrent uses this strategy, but in other fields such as image processing this is used too. In cases where a file is not moved when it's done and looking for terminators is either too inefficient, or simply not possible, there is yet another option to consider. This is the option of file locking.

File locking works differently on different operating systems, but there is a Java abstraction over that in the java.nio libraries. Spring Integrations file support was expanded to include support for filtering based on java.nio in version 2.0. Simply inject a NioFileLocker and make sure the writing process is respecting locks too. It is also possible to 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 in fact it is much more complex than you might think at first. 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 have seen how to work with files using Spring Integration. We've 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 you are freed from the responsibility of opening `FileOutputStream`s or `FileWriter`s and closing them again. In the writing section we've also discussed the outbound gateway that is useful if you need to postprocess the file that is written.

You have also seen how to read files from the file system. How to prevent duplication and deal with pattern matching filters. We've discussed the `FileListFilter` extension point that allows you to filter incoming files on any

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

criterion of your choosing. Or even to go 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 ones have been implemented in the Spring Integration file transformers. It goes without saying that you can just as easily implement a file transformer that suits the needs of your business case better.

Finally we've taken a look under the hood of the discussed components. The ordering and locking within the `FileReadingMessageSource` were discussed. We've urged you to avoid relying on these complexer options as much as you can. In the last section we've explored reading unfinished files in more detail. You will have more of an idea of the advanced problems you will come to deal with 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 file system. With the adapters described here it is rather simple to convert between messages and files. As a final note it is important to remember that the file support in Spring Integration, while convenient, is not a competitor for frameworks that focus really on the interpretation and processing of files such as Spring Batch or other ETL tools. It is a very healthy idea to use Spring Integration to create the events that drive the specialized process that deals with the files.

This chapter concludes section three of the book. You have been presented with the most typical integration options, other options are quite easily implemented as extensions of the Spring Integration adapter module, or integrated as generic Service Activators, Gateways or Channel Adapters. In the next section we will discuss advanced topics such as concurrency and performance, OSGi and Rich Clients. In these chapters we will build on the knowledge you've built up so far.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

12 Spring Integration and Web Services

In the previous chapter we saw Spring Integrations support for integration through the exchange of files. While file based exchanges between systems are still very common, enabling two systems to interact through the file system can be painful for a number of reasons. Firstly most systems will not share a file system since they will likely be on separate hardware, potentially in different parts of the world. This means that file transfers over the network will be required 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 file system. Because

Figure 12.1 Integration over HTTP is simple since the network exists and HTTP passes relatively easily in and out of enterprise networks

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 WS, Arjen Poutsma followed the good practice to decouple transport and message. Spring Integration follows in this tradition. In fact, before Spring Integration was released there was support for web services over JMS, Email and XMPP built on top of Spring Web Services. However, in this book we are writing separate chapters about separate transport mechanisms.

Most people using the term web service think of communication using XML messages sent with the HTTP protocol. Even though the definition is much broader, it is this form of web service that we will focus on in this chapter. Other chapters deal with the other transport protocols.

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 however its use is by no means universal so this definition does not reflect the reality of web services even when only considering those using HTTP. These attempts to standardise web services gained great popularity in the nineties with most large software vendors happily selling a broad range of tools and generating an increasing number of add on specs known as the WS-* specifications covering 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 to resort to plain XML, commonly referred to as Plain Old XML or just POX. These POX payloads would be exchanged over HTTP typically. Another increasingly popular is to use simpler formats which can be customized for the task at hand. Simpler forms of web services can make more use of the HTTP protocol than just as a transport. The biggest growth in this form of web service has come through the popularity of RESTful web services

Spring Integration provides support for exchanging XML based web service

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

messages over HTTP by building on the Spring Web Services project for both SOAP and POX based web services. We will 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 REST-ful services. We will 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 Web Services

Spring Web Services is a Spring portfolio project which focuses on contract first web services with XML payloads. As mentioned before, not all XML web services use SOAP. Spring Web Services provides support for POX based Web Services among others. One of the strongest points of Spring web services is the way in which 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 Web Services the creation of even a simple SOAP message as below 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 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 actually get to the message we want to pass. It is 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 which arguably would be better

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

suited to the use of the HTTP GET method. This is because SOAP simply uses HTTP as a mechanism for passing data and makes no real use of the HTTP protocol. We will see that this contrasts with the more HTTP centric alternatives to SOAP later in this chapter.

The Spring Web Services project provides a simple programming model which 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 Web Services. It adds the power and simplicity of messaging combined with the established Enterprise Integration Patterns implementations.

Where a web service implementation is more complex using EIP patterns and messaging will generally produce 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 of differing formats, invoke a number of different quote providers using different technologies and then subsequently process a number of responses into one single web service response. While this can be achieved by using Spring Web Services alone it is much less work to implement by a combination of the two and that is what the web service support in Spring Integration is intended to do.

To get started we need to make sure the application can send and receive external messages through Spring Webservices. In the next section we will look into wiring the beans needed to expose an inbound gateway to do this.

12.1.1 Exposing a Spring Web Services 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 which is 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 like when using Spring Web Services directly the front controller is an instance of `org.springframework.ws.transport.http.MessageDispatcherServ` 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...
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
                      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 Web Services strategy interface org.springframework.ws.server.EndpointMapping to be provided by the application context configured via the contextConfigLocation param name. This strategy interface provides support for mapping a given request received by the servlet to a particular endpoint. Spring Web Services 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 is 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" />
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

By default the inbound gateway will publish 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 below.

```
<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 example below 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 will then be 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 simply configuring the inbound or outbound gateway with a marshaller and/or unmarshaller allowing automatic conversion between Java and XML for both the inbound and outbound messages.

12.2 Simple HTTP Endpoints

SOAP and POX based endpoints are not the only form of Web Service supported by Spring Integration. It has become increasingly popular to provide web services which do not 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 REST-ful approach HTTP itself is used as the service protocol. The term REST originates from the doctoral dissertation of Roy Fielding. Roy Fielding is one of the authors of the HTTP specification. Hence the close relationship between REST-ful services and HTTP.

An example of a REST request would be for example on receipt of a HTTP request using the DELETE HTTP method and a URI of *http://www.example.org/book/123* the request would be interpreted as a request to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

delete book number 123.

```
DELETE /books/123 HTTP/1.1
Host: www.example.org
```

However receiving a GET request as below where only in the HTTP method is different would be result in simply retrieving a representation of *book 123*.

```
GET /books/123 HTTP/1.1
Host: www.example.org
```

In HTTP based communication for example it is common not to make use of all the HTTP methods and instead use GET or POST for a multitude of purposes including requests which are really deletion 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 REST-ful Web Services as an extension to the existing MVC framework for Web Applications. This support focused on making the mapping of controller methods to requests using both the URI and the HTTP method while also making it easy to map parts of the URI to controller method parameters. The below 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 makes use of the Spring 3.0 REST support through a set of HTTP adapters which 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 while it is possible to use Spring Integration to expose services in a REST-ful fashion the HTTP can also be used to expose services which would not be considered REST-ful. Support for all HTTP methods is something that may simply not exist in some technologies such as AJAX which use almost exclusively GET and POST HTTP methods and will often not use URI schemes that reference resources.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

12.2.1 Processing HTTP Inbound requests

The HTTP inbound gateway and channel adapter endpoints allow for the consumption of a 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. In order to start processing inbound HTTP requests we first need a servlet to act as the entry point for requests. We configure this using a HttpRequestHandlerServlet mapped to the appropriate URL via the standard web.xml. We 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 below 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 however this is not required and when not provided a temporary channel will be used to receive responses.

```
    <si-http:inbound-gateway id="httpTripInboundGateway"
                             request-channel="tripQuoteRequestsChannel"
                             reply-channel="tripQuoteResponseChannel"/>
```

The inbound gateway will configure an instance of either `HttpRequestHandlingMessagingGateway` or `HttpRequestHandlingController`. Which one is instantiated at runtime

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

will depend 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 will be created otherwise the gateway implementation with its own response rendering logic will be used. When using the controller-based gateway, the HTTP response can be rendered by one of the large number of view technologies supported by Spring MVC including JPS, 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 12.1 Additional configuration attributes for HTTP inbound gateway namespace

view-name	Name of the view to use to render responses. If this is provided, then 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. The default is 'true'. Supported by gateway and controller.
reply-key	The key to use for the response message when adding it to the model map that will be passed to the view. controller only
reply-timeout	Timeout for waiting on a response. controller and gateway
message-converters	List of HTTPMessageConverters which will be used in order to convert the HTTP messsage.
supported-methods	The HTTP methods that are allowed for this gateway. controller and gateway
convert-exceptions	Should the provided message converters be used to convert any exception thrown to a HTTP message. Defaults to false. Gateway only

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

request-payload-type	Used to indicate to the message converters the desired Java type to convert the request to. Controller and gateway
error-code	Code to be used to indicate failure in any errors object passed to the view. Controller only
errors-key	The key to use for an Errors instance containing binding errors in the case that an exception is thrown during the processing of the request. Controller only
header-mapper	An optional reference to a bean implementing HeaderMapper<HttpHeaders>. This strategy converts the HTTP headers to and from Message headers. If not specified the default strategy will be used. This maps all headers prefixing the HTTP header name with 'X-'. Controller and gateway
name	e.g. "/getQuote". When used with a Spring DispatcherServlet and the BeanNameUrlHandlerMapping, this provides the path prefix for requests that this endpoint will handle. *Controller and gateway*

The ability to provide a view gives a great deal of flexibility with regard to generating a response.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

12.2.2 Inbound-only messages using inbound-channel-adapter

For processing HTTP requests which only require confirmation that the request was successfully accepted, Spring Integration provides an inbound channel adapter. The implementation for the channel adapter is exactly the same classes, `HttpRequestHandlingMessagingGateway` and `HttpRequestHandlingController`, but this case, 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. Here, we've used specified the view-name attribute, so this will be handled by the Controller.

```
<int-http:inbound-channel-adapter id="httpSmsSubscriptionInboundChannelAdapter"
channel="smsSubscriptionsChannel"
supported-methods="GET,POST"
name="/subscribe"
view-name="confirmReceived" />
```

Note that in this example we have assigned a path as the name of the bean. This allows the bean to be used with the default behaviour of Spring's DispatcherServlet as configured below.

```
    <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 of course that is only one side of the story. Next up is the outbound part of it.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 Spring Integration message. Below 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. The RestTemplate is used to make a 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 this is used to then 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 will be 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 past two sections we have talked about synchronous messaging. This is natural because HTTP is an inherently synchronous protocol. However, there is no reason why we cannot use a synchronous protocol for asynchronous communication architectures. The next section will go into the outbound channel adapters that can be used to do this.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

12.2.4 Outbound channel adapters

An outbound channel adapter is much like an outbound gateway, except that, it is not sending a response on a reply channel. Behind the scenes the HTTP outbound channel adapter is using the RestTemplate from the Spring framework. Because of this the determination of what is an error is done by an implementation of `ResponseErrorHandler`. The error handler can be provided by setting an attribute on the the outbound channel adapter element. If no custom error handler is provided then an instance of `DefaultResponseErrorHandler` will be used. This error handler may not give you the expected behaviour. 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 will not 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. Below is an example configuration which 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 Conclusion

Integration over the internet is becoming increasingly important allowing 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 requirements as to format. While standards such as SOAP are still favored within large corporations many people are moving to more flexible approaches to web services. The de facto standard on the web is REST, not in the last place because it fits so well with the HTTP paradigm. To integrate systems that take different approaches to web services it is needed to have flexibility within your integration solution. Spring Integration provides this flexibility.

In this chapter we have seen 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 web services or consume web services. Combining Spring Integration with Spring

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Web Services and the REST support provided by Spring MVC helps the developer to use EIP patterns when exposing or using web services.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

13 Chatting and Tweeting

We have covered a lot of Spring Integration adapters so far. We've seen how to invoke both SOAP and REST based Web Services over HTTP. We've seen how to send and receive Messages using the Java Message Service API (JMS). Those adapters, and most of the others that have been covered up this point, have been 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 enterprise edition specifications. In this chapter, however, we are shifting that focus to two different technologies: XMPP and Twitter. Both are well on their way to reaching "mainstream" status, if not already there. However, while many enterprise Java developers probably use both technologies daily, they have relative newcomer status in the realm of enterprise Java applications. That's a trend that is very likely to shift, and in this chapter, you will learn just how easy it can be to introduce these technologies into your applications via Spring Integration's corresponding adapters.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

13.1 XMPP

XMPP stands for "eXtensible Messaging and Presence Protocol". It defines a fairly generic model for packets containing content and metadata that can be passed along with that content to describe it. As indicated by the acronym, 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 services. Even if you have never heard of XMPP, you might use it on a daily basis. Several popular IM services are built upon 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 just 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. The following example demonstrates a chat session where both parties are busy typing a message at the same time rather than taking turns. You probably experience this on a daily basis.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 13.1 XMPP supports full-duplex messaging, meaning that communication occurs bidirectionally, even simultaneously.

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 is a distinctly powerful feature of the protocol. Consider the "roster" features of a typical Instant Messenger client. When using an

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

XMPP-based IM client, and you see the members of your friends list that are online, that is driven by presence notifications. Likewise when someone updates their status, your client will receive that update in your roster. Those presence notifications are separate from but parallel to the actual message content. Let's take a look at the adapters, beginning with the sending of simple message content.

13.1.1 Sending XMPP Messages

We are going to start with what is 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 instant messaging scenarios between friends. Two machines can exchange messages as a way to pass data or notify each other about important events. Likewise, 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 will explore those adapters later.

The first step to using any of the XMPP adapters is to setup 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. Below 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 is generally recommended to externalize the connection settings in a simple text file as key value pairs. Taking the same example as above, we could accomplish that by creating a properties file, such as "xmpp.properties" shown below.

```
xmpp.username=johndoe
xmpp.password=bogus
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
xmpp.host=somehost.org
xmpp.service=mychat
xmpp.port=5222
```

Then, we could associate placeholder variables with the keys from that properties file. To load the properties, we would just include a single property-placeholder element within the configuration file. All of the relevant configuration is shown below.

```
<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 has been defined, you can setup the adapter. Since we are focusing on the outbound adapter that publishes XMPP messages, the element to use is the 'outbound-channel-adapter'. Since we already provided all of the connection details with out XMPP connection, this configuration is trivial. In fact, the only thing that is required is the name of the Spring Integration channel where the application will send messages that are intended for the XMPP chat.

```
<int-xmpp:outbound-channel-adapter channel="chatChannel"/>
```

If for some reason you have named the XMPP connection something other than "xmppConnection", then you would need to provide its name explicitly via the "xmpp-connection" attribute on the outbound-channel-adapter element.

```
<int-xmpp:outbound-channel-adapter channel="chatChannel" xmpp-connection="myXmppConn
```

Hopefully that configuration seems straightforward. Now we can turn to the usage of these adapters from application code. 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 actual 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.CHAT_TO. The following code provides an example of sending a message along with that header.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
Message<String> message = MessageBuilder("Hello XMPP!").setHeader(XmppHeaders.CHAT_T
chatChannel.send(message);
```

As you know by now, it's not necessary to work directly at the API level. There are less invasive alternatives that promote a separation of concerns. One of those alternatives is to use the AOP-based technique that is driven by the @Publisher annotation. Let's look at an example where we send 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);
}
```

The example above would work perfectly if you intend to send the chat messages as a by-product of the service method invocation. In other words, that method is already being called as part of your application logic. You are simply 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 that is defined in the core Spring Integration namespace.

```
<annotation-config/>
```

You can even add the default channel to use for any @Publisher annotation that does not include one explicitly. That works well if you have a simple application and can rely upon 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 upon a proxy, it's actually best suited for a different type of use-case. We already saw that the intercepted application service method treats messaging as a by-product while the primary responsibility of the code is still handled by the application service itself. With the Gateway proxy, there is 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 would simply point to the fully-qualified name of that interface.

```
<gateway id="chatGateway" service-interface="example.ChatGateway" default-request-ch
```

In this case, Spring Integration will create 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 a bit more control, a MessagingTemplate instance. The advantage of using the gateway proxy is that it is non-invasive. It's basically 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 non-invasive techniques based on coding to interfaces rather than implementations.

You may have noticed that we are missing one important detail in the recent examples. While we are passing the chat message payload text, we have left out the recipient's username. Obviously that information needs to be provided as well so that our messages can actually be sent to someone. Therefore, we need to provide the CHAT_TO header. Again, we would like to keep this non-invasive.

Probably the simplest option, if it will work from the perspective of the calling code, is to rely upon a method argument. Here is a slightly modified version of the ChatGateway interface to demonstrate such an argument:

```
public interface ChatGateway {
    void chat(@Header(XmppHeaders.CHAT_TO) String username, String text);
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
}
```

The non-annotated argument will be used as the payload of a newly created Spring Integration Message, and the argument annotated with @Header will be added to that Message with the annotation's value as the header name. In the above example, we are relying on the CHAT_TO constant, since that is what the downstream XMPP adapter will be expecting.

If you are not willing or able to include the username in the method invocation, then another option that is even less invasive, is to add a header-enricher to the message flow. Assuming our 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="myXmppConnecti
```

One obvious limitation of that particular configuration example is the fact 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 as shown above. Here we're showing just the header-enricher's chat-to sub-element 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 is defined within the application context. For example, we might have the Account object, but it does not provide the user's instant messenger username. At the same time, imagine that we do 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)"/
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 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. Nevertheless, in Spring Integration each direction of communication is handled by a distinct channel adapter. The outbound channel adapter that we have already discussed represents the role of sending, since a chat message is sent *out* from the perspective of the Spring Integration application. Now, we will turn to the corresponding inbound channel adapter. Of course, its responsibility is to receive messages that are coming *in* to the Spring Integration application.

The inbound adapter has much in common with the outbound adapter. Not only do they rely upon the same protocol and the same underlying API for negotiating that protocol, but they both rely on the same connection configuration in order to use that API. That means that any of the 'xmpp-connection' element examples displayed within 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"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

13.1.3 Sending and Receiving Presence Status Updates

You have now seen 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 instant messenger application, you may realize that you also depend heavily on the ability to know which of your friends are available. In fact, most instant messenger applications support "status" messages that provide more detail than whether someone is simply online or offline. Your friends are listed on the roster or"buddy list, and some may have status messages that indicate "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 are known as presence notifications. So, 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 with the chat message support, there is a pair of inbound and outbound adapters for presence.

Here is an example of a simple inbound adapter for receiving presence notifications.

```
<int-xmpp:presence-inbound-channel-adapter channel="presenceChannel"
                                           xmpp-connection="xmppConnection"/>
```

First you will notice that the presence adapter can be defined with an element from the same 'xmpp' namepsace. 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 itself. As with the chat adapters, the "xmpp-connection" attribute is optional. As long as the name of the actual XmppConnection bean is "xmppConnection", then the attribute is not necessary.

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"/>
```

Of course 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

implement a chat service, you would 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.

13.2 Twitter

The first thing that most people think of in relation to Twitter is the timeline. When someone "tweets", they are 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 user's you follow, as well as your own tweets. However, Twitter does provide more features through its API. You can perform a search for all tweets that include some interesting text, regardless if those tweets were posted by users you follow. Similarly, you can easily find all tweets that mention your own username, and those results also may include tweets from users whom you do not yet follow. Finally, Twitter even provides support for sending and receiving "direct" messages, which are essentially private alternatives to the more common broadcasting that is visible to anyone. Since Twitter provides so many different options, Spring Integration provides a wide variety of different Twitter adapters. The following table 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 will focus on the adapters listed in the above table. While it has been rather common in this book for us to begin with the sending side, here we are going to begin with the simplest case: Twitter search. The reason that search is simpler than most of the other adapter types, is that the underlying API enables search capabilities without requiring authentication. So, let's begin this tour of Twitter adapters with the ability to convert search results into inbound Spring Integration Messages.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

13.2.1 Receiving Messages from a Twitter Search

One of the most common uses of Twitter within a Spring Integration application is to perform a search periodically and send the results downstream within Messages. The element to define within XML-based configuration is the 'search-inbound-channel-adapter'. It also happens to be quite simple in its configuration since a Twitter search does not require authentication. We will explore other Twitter adapters that do require authentication since they are 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="#springintegrati
    <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 sub-element dictates that the Twitter search should be performed every minute. The results will be 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 Integration. That class defines a number or properties, listed below. This same Tweet type will be common to all of the adapters you see within 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 fromUserId;
private String languageCode;
private String source;
```

Knowing those properties, you can control the actual content of downstream messages by using the Spring Expression Language. Consider our logging example above. 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="#springintegrati
    <int:poller fixed-rate="60" time-unit="SECONDS"/>
</int-twitter:search-inbound-channel-adapter>

<int:logging-channel-adapter id="tweets" expression="payload.fromUser + ': ' + paylo
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Of course, you might be doing other things downstream, rather than logging. A more general approach would be to define a transformer element.

```
<int-twitter:search-inbound-channel-adapter channel="searchResults" query="#springin
    <int:poller fixed-rate="60" time-unit="SECONDS"/>
</int-twitter:search-inbound-channel-adapter>

<int:transformer input-channel="searchResults" expression="payload.fromUser + ': ' +

<int:service-activator input-channel="tweets" ref="searchResultAnalyzer" method="add
```

At this point, you know everything you need to know in order 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 a bit more configuration since 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 will cover the timeline reading adapter shortly, but first we need to discuss how you should provide the configuration that is necessary for creating a Twitter template instance that is capable of passing OAuth credentials.

13.2.2 OAuth Configuration for the Twitter Template

The Twitter template is an implementation of the TwitterOperations interface, and that interface defines the methods that all of these Spring Integration Twitter adapters depend upon 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 search adapter examples above, you did not see the Twitter template instance, because it is not a required reference attribute on the "search-inbound-channel-adapter" element. If not provided explicitly, it is created behind the scenes. However, if you were to look into the search adapter's code, you would see that it does indeed delegate to the TwitterTemplate. The reason it did not need to be created explicitly is that the search adapter does not require authentication. To perform any Twitter operation that does require authentication, a TwitterTemplate must be configured within the Spring configuration so that the authentication credentials may be provided. The authentication mechanism used by

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

this template is OAuth, and its credentials are comprised of a consumer key and an access token as well as their associated "secrets". To learn more about OAuth itself, you may want to visit http://oauth.net/. If you are 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.

In order to create a TwitterTemplate with OAuth capabilities, you must first register your application at http://dev.twitter.com. Upon registration, you will be given 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 in order to receive the access token and its associated secret. You will want to be very careful to store all of the keys and secrets in a safe place.

In the examples throughout this chapter, we will be storing the OAuth configuration properties in a file. Such a file could then be protected such that the application has read access but others don't. The properties file might be named 'oauth.properties' 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 Twitter template can be configured as a bean definition using property placeholders.

```
<bean id="twitterTemplate" class="org.springframework.integration.twitter.core.Twitt
 <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>
```

Now that you have seen how to configure a Twitter template instance that uses OAuth configuration properties, we can move on to discuss some of the adapters that require authentication. We will begin with the inbound Twitter adapter for reading your timeline.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

13.2.3 Receiving Messages from your Twitter Timeline

Along with the search functionality that you saw earlier, reading the timeline is the other major use-case for an inbound adapter accessing Twitter. However, unlike search, timeline reading does require an authenticated connection. Basically, you have to be logged in since 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 have created a "twitterTemplate" that has the requires OAuth properties as shown in the excerpt above, then you can provide a reference to that when creating a timeline-reading Twitter adapter. Here's what the adapter itself would look 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 has been configured to poll every 10 seconds. With each poll, it will attempt to read up to 25 Tweets from the timeline. That will be the timeline of the user whose OAuth credentials have been provided to the template. One thing you might notice is that the element defined in the namespace for this adapter is simply 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 will explore what is probably the most popular use of Twitter for *outbound* adapters: updating the 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 had not already known, you learned that the Twitter timeline is essentially a collection of status updates - typically referred to as "tweets" - from all of the Twitter users whom you follow. In this section, we will explore the other side of that relationship. When you update your status by posting a "tweet", all of your followers should soon see that in their timelines.

As with reading the timeline, it's necessary to be authenticated with valid Twitter account information prior to sending status updates. The configuration for

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

the OAuth credentials is the same for all adapters that require authentication. In fact, each of these adapters is actually just delegating to the underlying TwitterTemplate instance, and it's really just a matter of ensuring that the TwitterTemplate itself 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. Since the TwitterTemplate can be used concurrently, it is most likely that you would 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 within an application, as long as those adapters are intended to represent that account.

Once the TwitterTemplate has been defined, the status updating adapter is rather trivial to define. Since updating your status is generally synonymous with "tweeting", the adapter on the outbound, or sending, side that is used for updating status has the unqualified element name: 'outbound-channel-adapter'. It is basically the mirror image of the timeline-reading adapter on the inbound side.

```
<int-twitter:outbound-channel-adapter id="statusUpdatingAdapter"
                                      twitter-template="twitterTemplate"
                                      channel="outboundTweets"/>
```

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 the fact that someone who does not 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 instant messenger 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. While you might simply be tweeting about something you find interesting without any intention of starting a discussion, it might grow into a very interesting discussion that pulls in people whom you had never known. On the other hand, you might initiate such a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

discussion more intentionally by posting an interrogative or poll message, such as "which one of these laptops should I buy?". Compared to an instant messenger or a chat room, a dialog via Twitter might be an unintended but welcome consequence and may include people who just happened to stumble upon one of the tweets within that dialog. While there is nothing in Twitter itself that organizes tweets as being part of a threaded discussion, any tweet may have an "in-reply-to" value. In any case, it is less structured than a chat room, but that lack of structure is what differentiates twitter as an open-ended form of messaging amongst a massive community. In the next section, we will walk through the process of monitoring Twitter for any mentions of your username within retweets or replies.

When it comes to messaging styles, you can think of Twitter as a publish/subscribe scenario where the person posting a tweet is the publisher and all of that person's followers are the subscribers. In fact, considering 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 will see the tweet. If they want to provide some commentary of their own or an answer to a question, then they can "reply" instead. There's a subtle difference, 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 simply refer to someone else's username in a Twitter message, such as "I just had dinner with @z_oleg". If you want to find all mentions of yourself, of course you could perform a search upon your username. However, since 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, with just a different element name. Like the timeline-reading adapter, it does require a template instance that has been injected with the user's OAuth configuration properties.

```
<int-twitter:mentions-inbound-channel-adapter id="mentionsReadingAdapter"
        twitter-template="twitterTemplate" channel="mentions">
    <int:poller fixed-rate="10" time-unit="SECONDS" max-messages-per-poll="25" />
</int-twitter:mentions-inbound-channel-adapter>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Before taking this discussion too far, we should also consider the fact that Twitter does indeed provide 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 are sending a . This goes to show that Twitter itself provides both publish/subscribe and point-to-point messaging semantics. In fact, 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 take a look at the latter option.

13.2.6 Sending and Receiving Direct Messages via Twitter

In the previous section, we mentioned that Twitter, in general, supports a very broad publish/subscribe style of messaging. While probably not quite as popular, Twitter does also support a point-to-point style via direct messages. The concept is rather simple. If you want to send a message to someone who follows you, and you only want 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 "micro-blogging", the direct message functionality may be thought of as "micro-mailing". 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 instant messenger "chat" session, the actual 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 the other party is currently available, they may prefer to respond when its more convenient. Going back to the phone call analogy, it's the same reason voicemail is so commonly used even on a mobile phone that is 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 really 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The outbound channel adapter for sending direct messages via Twitter looks very similar to the status updating channel adapter. The key difference is that its element name is qualified with the "dm-" prefix. Since sending direct messages does require authentication, you will 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 is one more requirement when sending a direct message via Twitter. Of course, you must somehow specify the user to whom you would like to send the message. As you would likely expect by now, the target user information should be specified in a message header. There is a constant defined to represent that header's name. It is 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 dm-outbound-channel-adapter example above.

```
Message<String> message = MessageBuilder.withPayload("hello")
        .setHeader(TwitterHeaders.DM_TARGET_USER_ID, "someuser")
        .build();
dmOutChannel.send(message);
```

The receiving side is a bit simpler since the target user ID is a responsibility of the sender. All you need to do on the receiving side is make sure that you have 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 is an example.

```
<twitter:dm-inbound-channel-adapter id="dmOutAdapter"
                                    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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

same across all of them, and there is nothing else required. Now you know how 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 have 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. It's probable that Spring Integration will evolve to support additional chat and instant messaging protocols, such as IRC and SMPP. In fact, some prototypes have already been created within the Spring Integration sandbox repository for both of those. It is even more likely that 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.

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 a future version of Spring Integration will be refactored to rely completely on that library's version of the TwitterTemplate. The goal will be to minimize the noticeable changes for end users. In fact, if you are relying on the namespace support, it's very likely that you will not need to make any changes whatsoever.

More interestingly, 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 upon common OAuth configuration just like you have seen above 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 easily rely upon 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 since even when new channel adapters are added and the corresponding namespace support is provided, the underlying functionality would be the same. In other words, the new Spring Integration adapters will be invoking the same templates that you can go ahead and start using in the meantime.

This chapter concludes section three of the book. You have been presented with the most typical integration options, other options can be implemented as

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

extensions of the Spring Integration, or integrated as generic Service Activators, Gateways or Channel Adapters. In the next section we will discuss advanced topics such as concurrency and performance, OSGi and Monitoring. In these chapters we will build on the knowledge you've built up so far. And give you the tools that will help maintain solid integration solutions.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

15 Managing scheduling and concurrency

In the previous chapters you have learned about the core and specialized components, and how can you put them together to build a successful application. But finding all the parts you need and putting them together into a structure it's not enough. This chapter takes a turn to discuss application-building from a different perspective: dynamic configuration.

This requires shifting focus from dealing with the logic of the application to the expected behaviour at runtime. The latter means to be able to control the schedule of various timed events in the application, such as the frequency with which external message sources are polled, and also the ability to configure various parts of the application to run concurrently, making optimal use of the resources of the system, and increasing the rate at which messages are being processed. This is what you are going to learn about in this chapter.

And a final word before we start: sometimes, you will find that fine tuning the dynamic performance of an application has a bit more to do with art, since no two applications are completely the same. There's is no exact prescription on when to do what, and very often you will need to find what works best in your case. But engineering is an art too, so we will provide you with the options, and guidelines on the best practices, leaving it up to you to decide in the context of your application.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

15.1 Controlling timed events

As the previous chapters have made it abundantly clear, communication is the predominant point of interest of enteprise integration applications. In order to integrate properly, the components have to communicate effectively. Aspects such as the format of the data that is being exchanged, the processing services, ordering play a critical role in achieving this goal. But timing cannot miss from this list.

Quite oftenly, integration applications have to satisfy timeliness requirements, which can be of various types. There can be responsiveness requirements, defining how quickly the system should respond when a new message becomes available, or there can be scheduling requirements, that prescribe how often or at what precise time should the application interact with its counterparts. In order to meet these criteria, they have to be actively addressed, and this section will mainly focus on the framework features that control the timed behavior of the application.

We will start by discussing poller configuration, as this 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 ones that are 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 is, ultimately, related to how aspects such as frequency, timeout or number of messages processed per polling cycle are configured. On the other hand, timed events are not just about polling, and you will also see how to set up a scheduled producer for invoking existing beans and sending out messages at specific moments.

15.1.1 Pollers and their configuration

You have encountered polling in the previous chapters. Bringing to mind an analysis from Chapter 5, the main distinction between event-driven and polling components is the way in which they receive the messages that they have to process. Event-driven components receive messages directly from their invokers (see Figure 15.1), while polling components will have to reach out and consult a message store in order to find out if there's any new data available for them. In Spring Integration, polling is used either 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 15.1 Event-driven components receive messages directly from their invokers

Figure 15.2 Polling components receive messages indirectly. The picture shows a typical scenario for a polling endpoint: the invoker will send the message to a buffered channel (1)

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

and the polling endpoint will receive it by polling the channel (2)

The fact that the application needs to use polling at all is very often a matter of choice - like deliberately introducing 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 our application, but very often, it is the application itself that 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 which is exposed publicly and can be invoked by the interesting parties, as opposed to receiving notifications from a subscriber services. 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 have seen `<poller/>` elements whenever we configured a channel adapter or a polling endpoint. Since polling is a sensitive topic, and its configuration has a strong impact on performance, there is no default 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 actuall polling task should be deferred to a task executor.

This doesn't mean that every polling component definition must include a poller configuration definition: they can also rely on the existence of an application-wide default configuration, which has to be provided by the developer. This makes possible to simplify component definitions in general (as the default is usually a configuration that applies to the majority of the polling components). At the same time, this scheme allows to customize the configuration wherever the default does not fit the needs.

In both cases, a `<poller>` element is used for defining a set of polling configuration parameters. Listing 15.1 is an example of how to configure two different polling endpoints using the default and individual configuration.

Listing 15.1

```
<poller id="defaultPoller"                                #1
  fixed-delay="500" default="true"/>                      #1

<channel id="input1">
  <queue capacity="10"/>
</channel>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<channel id="input2">
  <queue capacity="10"/>
</channel>

<service-activator id="activator1"                      #2
input-channel="input1"/>                                #2

<file:inbound-channel-adapter id="inputDirPoller"       #3
  directory="${java.io.tmpdir}"                         #3
  filter="filter"                                       #3
  comparator="testComparator"                           #3
  auto-startup="false">                                 #3
    <poller fixed-delay="10000"/>                       #3
</file:inbound-channel-adapter>

#1 Default poller configuration
#2 Endpoint with default poller
#3 Endpoint with custom poller
```

The definition of the default poller configuration is shown at #1. The service activator, defined at #2 will use this configuration. The channel adapter defined at #3, on the other hand, defines a different policy, since it needs to consult the message source much less frequently. In general, it is better to create a default configuration to be used for queue channels and polling endpoints - on one hand this simplifies the overall configuration, on the other hand, channel adapters are much more likely too need a customized configuration anyway, so they will rely on the default in rare occasions.

One thing that you may notice on the configuration above are the numeric values configured in the 'fixed-delay' and 'fixed-rate' attributes of the poller elements. They are used for configuring the polling frequency, and we will look at them in the next subsection.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 also on the responsiveness of the system. The more frequently the application will poll a MessageSource or a queue channel, the bigger is the chance that the polling event will happen close to the moment when the message has become 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 is consistent with the actual timeliness requirements of the application, which, very often, define what is an acceptable delay. As a consequence, the polling frequency should be configured so that it balances performance with resource usage.

Considering that an application needs to be highly responsive, one could think of setting the frequency to the highest possible values. But every polling operation keeps the system busy, and consumes computing resources that may be needed elsewhere. This is appropriate, if the effort - in terms of resource usage- of checking that a new message is ready is very little, 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 (like remote accesses), it's a better idea to perform them less frequently, otherwise the system will be busy all the time performing checks which may not produce any result.

Let's take a 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 milliseconds. This means that after each polling operation, the system will wait half a second until it makes another try. On the other hand, the channel adapter uses a fixed rate frequency of 10 seconds, which means that the system will execute the polling operation every 10 seconds, regardless how long the previous

Figure 15.3 captures the differences between how the two strategies. Notice how fixed-interval polling causes the timing of the actual polling event to drift over time, but fixed-rate polling takes place at predictable moments.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 15.3 Fixed-interval polling (1) vs. fixed-rate polling (2): a fixed-rate scenario guarantees that polling will happen at the same time, but the duration beween polls may vary, while a fixed-interval scenario guarantees that the interval between polls is constant, while the polling event may drift over time

The two approaches are very 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 15.1 shows you a few points that you should consider when deciding between the two.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 15.1 How to decide between fixed-rate vs fixed-interval strategies

Scheduling type	What to consider
Fixed-rate	• ensures that messages will be read at well-defined moments; • works best when the duration of polling operation is fixed and short relatively to the polling period, otherwise it may force the system in a busy loop;
Fixed-interval	• 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 will have access to it, too.

15.1.3 Scheduling jobs at precise times

Fixed-rate scheduling produces a predictable sequence of polling events, but they are still not entirely deterministic. Let's consider an e-mail client that synchronizes with the server every 10 minutes. In this case, it is of little importance when exactly the 10-minute period starts, as long as the interval is respected. But there are other operations for which the start time is very important. Batch operations such as payment transactions processing, or bulk reference data changes are usually processed outside the normal application operation hours Iwe will cover batch operations in the next chapter, and we also encourage you to read Spring Batch in Action). They are resource-intensive too, and running while other users are accessing the application will slow down operation.

In this case, "execute this every 24 hrs" is not precise enough. What we want in this case is to say "execute this job every day at midnight". Neither fixed-rate scheduling nor fixed-interval scheduling have the capacity to describe this type of configuration. So, in order to provide this type of configuration, you need to use a different type of trigger: cron-expression based triggers, as in the following example:

```
<poller id="defaultPoller" default="true">
   <cron-trigger expression="0 0 0 * * *"/>
</poller>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The frequency and polling strategies - fixed rate, fixed interval and predefined times are the most frequently used settings for customizing the polling behavior. However, there are other parameters that affect the performance of polling as well, such as timeouts and the number of messages that are 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. To illustrate this, let's consider a case where we need to poll a message source for which a read operation is very time-consuming, regardless whether there is data available or not. which happens quite often when doing invocations across the network. The polling operation will be blocked until the read operation completes, but we can't wait there indefinitely: it prevents the application from performing other useful jobs. So there must be a limit on the time that a poller can spend while 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="
```

With this setting, the timeout is set to two seconds. If a message arrives afterwards, it will be picked up by the next polling operation. Setting a reasonable timeout for polling operations is very important for the performance of your application. Setting up a large timeout value without any basis will not help in any way. 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 is very little chance of reading any data from it. But the timeout value cannot be set too low either, since long-running read operations must be allowed to complete. Generally speaking, timeouts should be set as low as possible, but not lower than then 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, while reading from a queue channel would be capped at a few seconds at most. However, this is just an example and your settings should be based on how your application works.

SIDEBAR	**Timeouts should be set as low as possible, but not lower than then 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".**

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 MessageSource. With what we have configured so far, here what is going to happen: when the first polling cycle begins, the poller reads the first message and starts processing by delivering it to an endpoint. However, the second message will be left unprocessed until the next polling cycle.

You can immediately see how this 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, there is another option that can help in this case: receiving more than one message per polling cycle. This can be done 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. Of course, you can choose any maximum value for the number of messages that you can receive in a cycle, but setting the value too high is not a good idea either, as it may increase the time that the poller spends reading messages beyond an acceptable threshold. Figure 15.4

Figure 15.4 The effect of using max-messages-per-poll: above - the messages are processed one by one, the second message has to wait for a complete polling cycle in order to get processed; below - the messages are processed as part of the same polling cycle

This concludes our overview of poller configuration as a way of controlling the time-related aspects of the application. But there is another mechanism that you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

can use for triggering timed events, the scheduled publisher.

15.1.5 Publishing messages according to a schedule

Besides polling, there is another type of timed event that can take place in your application. Spring Integrations 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 for polling. Let's assume that we want to execute a periodic cleanup task on our application's database. A possible configuration can be seen in Listing 15.4.

```
<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" met
```

```
<inbound-channel-adapter expression="${reservations.findUpcoming()}" channel="notifi
   <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"/
```

15.2 Managing concurrency

Enterprise applications can increase their performance and responsiveness by executing concurrent operations. Concurrency may be an requirement - 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 now, when multi-core and multi-processor 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. On one hand, you will need to find an application design that allows operations to execute in parallel - mainly by splitting multi-step operations into finer grained parts that get executed asynchronously . One important counterpoint is that single-threaded operations can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

propagate context information across endpoints, such as the transactional and security context, and this decision also involves a tradeoff. The first subsection will walk you through the mechanics of breaking down single-threaded processes into multiple asynchronous tasks.

On the other hand, in order for the operations to be actually performed in parallel, you need to allocate resources - i.e. threads - for their execution. Again, this involves a number of tradeoffs between maximizing the number of concurrent operations and making sure that system's resources are used efficiently. So in the second subsection, we will describe the 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, one can start by ignoring whether this series of transformations happens within the scope of a single thread or it 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 very well 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.

This may work well, when the goal of using Spring Integration is to encapsulate a multi-step process that includes invocations across multiple services and systems. In these situations, you don't need to worry about concurrency. Multi-staged processing pipelines are good candidates for parallelization, if it is acceptable for the various stages to execute in separate thread contexts.

In order to illustrate the concept, let's begin with a simple example: a car assembly operation. Building a car goes through two stages: assemblying and painting. Each stage takes 1 second and producing a complete car requires 2 seconds in total. The simplest possible setup can be seen in listing 15.9.

```
<i:channel id="input"/>

 <i:channel id="paint"/>

 <i:channel id="output"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<i:gateway id="supplyInput" service-interface="com.manning.siia.pipeline.SupplyI

<i:service-activator id="assemblyLine" input-channel="input" output-channel="pai
    <bean class="com.manning.siia.pipeline.AssemblyLine"/>
</i:service-activator>

<i:service-activator id="paintShop" input-channel="paint" output-channel="output
    <bean class="com.manning.siia.pipeline.PaintShop"/>
</i:service-activator>

<i:service-activator id="exitCounter" input-channel="output" ref="counter"/>

<bean id="counter" class="com.manning.siia.pipeline.Counter"/>
```

A PieceKit payload is delivered to the AssemblyLine where it is assembled into a Car, which is painted in the PaintShop and sent to the final counter, which will

Figure 15.5 The two car assembly operation consists of assemblying 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, we discover that the process will take 20 seconds.

We can save time if by allowing for creating cars in parallel. This can be done, for example, 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 would like to focus on, however, is how to use the framework for handling parallelization internally. So, instead of executing both stages in the same thread, we will execute them in parallel - after being assembled, the car will be handed over asynchronously to the PaintShop, and the invoking thread will return, making room for another PieceKit to be sent to AssemblyLine. The AssembledCar will be painted in a separate thread. The differences between the two scenarios can be seen in Figure 15.10. Single-threaded processing will produce a car every 2

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

seconds, but breaking the thread boundary and doing part of the processing in a separate thread will produce a car every single second.

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.

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.

After changing the input to a queue channel rather than a direct channel, the actual processing of the message by the endpoint will not execute in message sending thread, which loses control over the message once it is sent to the queue. Instead, the message will be processed in the scope of a new task, initiated by the poller of the endpoint that is connected to the channel. The message sending thread returns to the loop and it 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 will repeatedly remove elements from the queue and process them.

```
<i:channel id="paint">
   <i:queue capacity="10"/>
</i:channel>
```

We can get a similar result by using an executor channel. This is a subscribable channel that is connected to a task executor, which will provide a thread in the context of which the application will process the new message. Anticipating the topic of section 15.2.2 a bit, the task executor is the main backing abstraction for thread allocation in Spring, Instead of creating a new thread directly for executing

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

a concurrent task, the application components will 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.

```
<i:channel id="paint">
   <i:dispatcher task-executor="executor"/>
</i:channel>
```

From a pure concurrency-focused perspective, the queue channel and the executor channel work in a pretty similar way, by interrupting the thread context of the message sending thread and executing the actual processing of the message asynchronously. However, there are diferences between them which make each of them more suitable for different scenarios. You can take a quick look at the differences in Table 15.2

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Table 15.2 Queue channel vs Executor Channel - what you need to consider

Channel Type	Features
Buffered Channel	• wide choice of queing 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	• no polling lag - messages are processed as soon as there is an available thread; • 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 hand-off is a solution for processing multiple messages concurrently, which is a common occurence in scatter-gather or 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, case in which every resulting message will be processed independently.

If you are using a publish-subcribe 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

tasks, let's discuss about how to set up the backing infrastructure for asychronous execution, and review in more detail the setup the task executor.

15.2.2 Configuring the infrastructure

There is a distinction between designing your application so that it has parallel execution capabilities and whether they really execute concurrently. The distinction comes from the fact that, no matter how much do we desire to run things in parallel, the application needs to allocate an appropriate number of virtual machine threads. In the previous section we started talking about the task executors, which is the main components that allows the asynchronous execution of tasks. Now, we will see an example of how to configure such a task executor. As you will see, it is very simple. To understand the motivations behind it, however, we will take a look first at the main concerns regarding thread allocation.

THREAD ALLOCATION CONCERNS

Thread allocation is an infrastructure concern, which is best to be kept separate from the business logic, in order to allow an easy switch between environments - like 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 actually bad, and executing too many tasks at once may backfire. Each thread consumes its own share of memory, CPU and other resources, and, adding up, a very 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 we limit the number of possible concurrent activities in the system, creating new threads is an expensive process. So, destroying a thread after we'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 we need to launch a new concurrent process, the solution is to use a pool of pre-created threads, which can be borrowed as necessary and returned upon completion.

Threads are expensive resources to maintain. There can be only as many concurrent tasks executing at once as there are threads in the pool. One impulse

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

could be to improve the performance by increasing the number of threads in the pool. While setting the size of the pool too low can mean that concurrent activities are forced to execute sequentally, setting it too high could mean that there's a large amount of resources which is reserved by the current application, but not actually in use. Threads in the pool are consuming resources such as memory even when they are not running, and this may affect other processes which are executing on the same machine. Very often, the solution is to allow the number of threads in the pool to fluctuate between a lower and upper bound. This elasticity allows for the concurrency to increase when the system is under heavy load and to decrease when things are coming back to normal.

Managed environments do not allow the direct creation of new treads. 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

These concerns above 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 will send it to the task executor, which will allocate a thread for its execution. We will look at it in more detail later, but, anticipating a bit, 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 it is executor's job to allocate a thread that is free or keep track of the tasks that need to be executed and launch them later, when a thread becomes available . Adding such a component to your application is very easy and 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 MessageSource 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

endpoints, as long as the hand-off is synchronous. This means that a polling operation can take a pretty 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 will be kept busy for a very short time, and it will delegate the actual execution of the polling operation to the task executor. This hasn't actually increased the degree of concurrency in the application, but has introduced 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 is intended to be used by default by a is supported by a Spring namespace. You have previously seen a definition for it, and here is 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 limit of 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

15.3 Under the hood

In the previous sections we have 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, in order to adjust performance and resource usage in the application, as well as ensuring that the application meets its timing requirements. For a typical user, this perspective is enough, since the namespace support and abstraction implementations for task execution and scheduling covers the typical use cases.

This section focuses on the needs of the more special cases, which are not covered by the defaults. It will introduce you to the Spring API for task scheduling and execution, and supply will show you how you can provide your own implementations that cust allocation of threads or triggering the timed events.

15.3.1 The TaskExecutor API

As you have seen 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 here are situations when the J2SE 5.0 Executor cannot be used directly, like for example 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). Such environments are Websphere or 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 JNDI-bound WorkManager. While not configurable directly by the application developer, it serves a similar purpose with 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, the developer can choose between one that is leveraging the J2SE 5.0 or an adapter around the CommonJ WorkManager. You can also implement this interface on

On the other hand, in a managed environment, one can use an environment-specific TaskExecutors:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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 actual behaviour is different, and reflects the fact that you are using different implementation.

The TaskExecutor allows the launch of concurrent activities, and, generally speaking, the assumption is that it will do this as soon as possible (i.e. as soon as a thread is available to execute it). Also, it is assumed that the Runnable encapsulates a one-off activity that will execute and finish within a reasonable timeframe (so that the thread on which it executes is freed up and another Runnable can be picked up for execution). However, an application may need to schedule periodic activities, as well as tasks that execute at specific moments in time, and the TaskExecutor is not enough for that. For covering those cases, the Spring Framework provides a richer abstraction 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, in order to understand the scheduling mechanism, we need to take a look at a more different mechanism: task scheduling. The task executor is relatively simple mechanism: once a task (i.e. Runnable) has been received, it will try to acquire the first available thread for it and, as soon as a thread becomes available, it will execute it. "As soon as possible" is, however, not a solution when tasks need to be executed at precise moments. So we 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 (either at a fixed rate or fixed interval). The role of the ScheduledFuture is to give the invoker a handle for enquiring 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Trigger interface shown below, along with the associated TriggerContext, that allows an implementor 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();c
}
```

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) do not cover the needs of your use case. A rare occurence, but a possible one nevertheless. Let's consider that we 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, therefore we 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 are not applicable. Similar to the WorkManagerTaskExecutor from the previous section, there is a TimerManager-based scheduler that can be used in a managed environment.

This wraps up the overview of the internal task execution and scheduling API. While not a common need in day-to-day usage, providing your own

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

implementations of those abstractions may come in handy in special cases. Also, now you have a better understanding of how the abstractions work, and can make better decisions on what to use when. It is time to wrap up the chapter, so let us take a quick look at what we have covered,

15.4 Summary

You have just learned about the dynamic configuration options of Spring Integration. They are complementary to the component definitions that you have learned about in the 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, however, does not mean 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 have also learned how to trigger operations that need to take place at a given frequency or at specific times, by using publishers.

Applications need to be able to do multiple things at once. But the degree of concurrency in an application, a factor which plays an important role in determining its throughput, is controlled in two separate 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 hand-off in order to allow their execution on multiple threads. On the other hand, you need to allocate the resources - i.e. threads - so that these operations can really run in parallel. Both aspects can be addressed by using the configuration options of the Spring Integration framework. There is a tradeoff too, because important information, such as transactional context, is lost in the process of handing-off messages asynchronously.

By wrapping up the handling of timelines and concurrency, we can move further and look at a particular type of enterprise application integration: massive dataset processing and batch operations.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

16 *Batch applications and enterprise integration*

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 are 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 progresses in user interface and communications technology have 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 will become familiar with the most significant features of batch applications, recognizing the main differences between them and other paradigms, for example online transaction processing. In what concerns this book, batch applications are important for the role that they play in enterprise integration.

Since this is a book that focuses on Java enterprise software development, we strongly recommend one of Spring Integration's siblings, Spring Batch as an implementation solution (and if you take an strong interest in batch job implementation we also recommend you to read *Spring Batch in Action* for learning about it in detail). In this chapter you will get a quick glimpse on how to use it for implementing a typical batch job example. Apart from that, you will see that there's no sibling rivalry here - in fact, the two frameworks complement each

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

other very well to produce sophisticated event-driven and highly scalable applications, and learning about the various integration opportunities is the part that will conclude our foray into batch application development.

So, let us begin by taking a look at what makes batch jobs so special.

16.1 Introducing batch jobs

Before introducing you to Spring Batch, we would like to provide some background on the typical features of batch applications. This section is a must-read especially if they are new or at least not familiar to you, and 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 will be collected in advance and provided upfront in a machine-readable format, such as files or database entries, and the output is, in counterpart, machine-consumable as well. As a result, these applications are not 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 that is necessary. Currently this application model is facing strong competition from online transaction processing, which was made possible by advances in user interface technology and communications infrastructure.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

16.1.1 Online or batch, that is the question

In order to understand the differences between the two approaches, as shown in figure XREF online-vs-batch, consider a payment processing application. The online transaction processing approach would take the payments as soon as they are 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 will bundle the payments together and process them later. We stress the importance of bundling together as opposed to simple deferral. Batch processing is not the same as asynchronous processing, where 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 is 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.

Figure 16.1 Online vs. batch processing - requests are processed individually or as a group, respectively

There are use cases when using a batch processing approach makes little sense, such as airline reservations. We could collect the travel information and process requests overnight, returning an answer with the available options on the the next day. It's an absurd example, but it serves well to explain why highly interactive and online transaction processing systems gained so much traction: in such situations the users want to receive answers immediately and be able to act upon 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

16.1.2 Batch processing, what is it good for?

For one thing, there are business processes which 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 real-time is preferred because of the increased security (when the list of transactions is sent once a day it is easier to monitor and authorize than if interactions take place ad-hoc, as well as simplified 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 it's catalogue of products every month - the store needs to be offline while the catalogue is updated. On the other hand, this may be done for performance reasons - for example, a large import which involves writing a massive amount of data to the database can seriously impair the performance of a running application that uses the very same storage system due to the high resource consumption or frequent table locking that may need to take place during the import.

Extract-transform-and-load processes (see Figure 16.2) are frequently used in enterprise integration for transferring data between applications.

Figure 16.2 A typical ETL job: data is extracted from an external source, transformed and loaded into the target

This is a fairly generic description of a batch job, and at a first glance it looks fairly simple - in fact a lot of the batch jobs are implemented as scripts or simple stored procedures which are executed periodically. A closer look will reveal, however, that they have a pretty complex set of requirements which, in certain cases, require a significantly more sophisticated approach. So let us take an example to guide us through the intricacies of a batch job implementation.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

16.1.3 Batch by example

Let's 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 need to be updated. The transactions are transmitted as a text file, each line containing the payer and payee accounts, and the amount that is transferred between them, as well as the transfer date, as in example 16.1. Our application must record the payments and update the payor and payee accounts accordingly.

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 can be seen in figure 16.3:

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)

INPUT DATA MUST BE STREAMED

Each row of the input file represents a payment, and our application must parse this data and convert it into actual data structures. Due to the data volume, it is 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 hold that data might exceed the physical capacity of the machine and if it doesn't, it will cripple the performance of the application nevertheless.

Instead of doing that, our application will have to process the input data gradually, row by row, or otherwise said, 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 where the application must implement its own strategy of partitioning the content into records, from database access, where the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

records are already identified but the application may still need to maintain an open cursor in order to be able to deal with a large amount of input data (think a SELECT statement returning millions of data records).

WORKING IN CHUNKS

But reading data is just 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 them into one or more output items (the alternative being a simple aggregate calculation). In order to ensure data consistency, operations must be transactional. The simplest approach, which is not necessarily the right one, would be to process each item in its own transaction. But that may be very inefficient, as every transaction start and subsequent commit introduce 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 actual size of a chunk varies from case to case - very large chunks will buffer the items in memory and drain resources, while at the same time risking the loss of data in case of an error, while extremely small chunks limit the benefit of this strategy.

DEALING WITH FAILURE

When 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 decide whether to ignore the item for which the failure has occured, to retry the processing of the faulty item or to 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 non-recoverable but losing this piece of data is non-critical) or completely aborting the execution.

When a batch process stops due to an irrecoverable error, it usually needs to be restarted after the errors have been fixed. But usually this doesn't mean that everything has to be started again from the beginning. Consider an application that is importing data. Restarting the process after half of the data has been imported can mean in the best case that the items previously imported have been overwritten, and in the worst case that they will be imported twice. So, ideally, the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

state of the batch process must persistent, and the 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 is not too complicated, implementing a batch application can become a pretty complex task due to the additional requirements for streaming, chunking and recoverable execution. As it is always the case with enterprise middleware, most of the effort may be spent on the infastructure instead of the business logic itself. 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 implements 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 as well as the extended set of modules that simplify ORM integration, 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 to launch new job instances;
- a state management infrastructure;

We will illustrate how Spring Batch provides these facilities through a simple batch process.

16.2.1 A batch job in 5 minutes

In order 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 needs to be updated with the transactions that have been carried out during the day. The transactions are transmitted as a text file, each line containing the payer and payee accounts and the amount that is transferred between them. Our application must record the payments and update the payer and payee accounts accordingly.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The batch job definition for this example is in example 16.2.

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"
       [CA] commit-interval="5"/>
     </batch:tasklet>
   </batch:step>
 </batch:job>


 <bean id="itemReader" class="o.s.b.i.f.FlatFileItemReader" scope="step">
   <property name="resource"
   [CA] 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"
           [CA] 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 (there can be multiple steps in a job, although this example has just one) which consists of importing the batch file, more sophisticated applications can have multiple steps.

The most common way of implementing a Step is by 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 etc). The actual work performed in a tasklet can be a simple service call or a very complex operation. The most typically used tasklet implementation is the chunk-based takslet (see figure 16.4), which processes a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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) and the ItemWriter aggregates output data and writes it back as soon as a chunk is assembled.

Figure 16.4 Chunk-oriented step: items are read individually by the ItemReader, processed by the ItemProcessor, then written in bulk by the ItemProcessor

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 multi-step job is the right way for doing so. Also, steps may specify various levels of fault tolerance, allowing to retry or skip items for certain types of exceptions rather than simply 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 the item writing strategies. In practice however, 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 individual objects. Your responsibility as a developer 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 example 16.3. The actual location of the file can be passed on 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 {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
        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;
    }
}
```

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.

16.2.2 Getting the job done

What we have created so far is the definition of a batch job. In order to put it to work, we need to start it. This is done through the executor API. In order to get access to it you need to define a distinct type of bean in your context definition: a batch job launcher.

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 like 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 which are specific to this job execution instance (such as the name of the file that needs to be processed), but also 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The returned JobExecution is a handle that allows the invoker to access the state of the executing Job. Spring Batch supports synchronous and asychronous 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've got a hint on what we will discuss next in the previous section. The JobLauncher bean we configured in listing 16.4 is injected with a jobRepository. As we explained already, the state of the batch jobs is persistent, so that jobs can be resumed after a failure or a shutdown. Resuming a job that has stopped in the middle of processing a data stream requires to avoid the potential duplication of data that may occur as a result of processing items twice. So, the job repository will record the ongoing progress of the running batch jobs, including information such as the current step and how far did the application go 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, JobParameters customize a specific job instance

Unless you have some very special needs, the only thing that you need to do for creating a job repository is to include a bean definition in your context. The repository uses a set database tables, which will be created automatically when the repository is instantiated for the first time.

```
<batch:job-repository data-source="dataSource" id="jobRepository"
 [CA] 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 very easily pretty complex batch jobs, to launch them and manage and interrogate their state, all while writing a very limited

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

amount of code. So let us see what are the opportunities of reusing all this in enterprise integration scenarios and, mainly, how does Spring Batch interface with Spring Integration.

16.3 Integrating Spring Batch and Spring Integration

Using Spring Batch will greatly simplify the process of implementing a batch job. Apart from the job implementation however, 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 itself.

Fortunately, this goal can be achieved with a minimal effort by pairing Spring Batch with Spring Integration. The two frameworks complement each other, and combined 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 application. It has access to a set of job definitions and to a job repository and provides support for:

- inspecting jobs and adding new job definitions;
- launching and stopping jobs - this includes 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. As such, it provides a set of utility classes that implement most of the collaboration patterns that will be discussed in this section and can be used out-of the box in your applications.

In what follows next, we will examine the various ways in which the Spring Integration and Spring Batch can collaborate and will illustrate this with examples that use components provided by the spring-batch-integration module of Spring Batch Admin.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

16.3.1 Launching batch jobs through messages

Let us assume that we finished implementing the batch job, as described in the previous section. The next step is to actually launch it. As we have shown already, you can do this from the Spring Batch API, by invoking the JobLauncher.run() with the appropriate parameters. But, how does this method actually 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 we can do this very easily. A file channel adapter will monitor the directory, and will send out a message whenever a new file has been detected like 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 (3,4). Other components may trigger a launch by sending a File payload (5)

Spring Batch Admin provides a JobLauncherMessageHandler that can be used for launching 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 listing 16.5, 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"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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.siia.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>
```

While 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 it needs to be implemented as part of the solution. An example is shown in listing 16.6.

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,
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
                        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 e-mails, JMS messages or FTP-accessible files. As a matter of fact, the trigger does not have to be a channel adapter, after all - 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 of taking on multiple input formats simultaneously. The batch job itself 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 batch processes can 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 is critical to get up-to-date information about its progress, and most importantly, whether it completed successfully or it had to stop because of an irrecoverable 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 which 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 is its current status, if it stopped because of an error, etc, and, in the case of running processes it can be used for stopping them. But polling the JobExecution repeatedly is sub-optimal. An event-driven approach is superior and preferrable.

Spring Batch provides a mechanism for registering listeners such as StepListener, ChunkListener or JobExecutionListener, that get invoked during the execution of a job, on events such as before/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 will send a notification message with a JobExecution payload after a job has stopped running. A router will decide,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

based on the execution status, whether it is necessary to take any further steps (like for example relaunching the job if it failed and the failures are recoverable, perhaps with a delay) - case in which the message will be sent to be processed to an appropriate endpoint or it's simply a matter of sending a notification message (completion was successful or the failures are irrecoverable), case in which it the message will be sent to an outbound email channel adapter.

Figure 16.7 Handling batch events using informational messages

For this, we need to add the listener to the job definition and also to add the rest of the bus configuration, as shown in example 16.7.

Listing 16.7 Registering a job listener and handling batch notifications through the message bus

```
<batch:job id="importPayments">
       <!-- other properties ommitted -->
       <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>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
<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 expand the feature set of the application. Besides that, the two frameworks can collaborate for providing a scalable execution infrastructure.

16.3.3 Externalizing batch process execution

The way we have described it so far, it would seem that the natural way of combining the two frameworks is by wrapping Spring Batch jobs into 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 actual processing of items to the message bus. This especially useful when item processing logic is fairly complex and involves invoking multiple transformations or service invocations, either local or even remote (through the use of channel adapters and gateways). In situations where item processing on the bus takes a relative long time, like for example when the application performs a remote invocation, you can increase performance by introducing asynchronous item processing. This can be done by using two wrapper components provided by Spring Batch Admin for your item reader and item writer: AsyncItemProcessor and the corresponding AsycnItemWriter. This is essentially a fork-join scenario: by using the AsycItemProcessor, the invocations on the gateway will be 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.

Figure 16.8 A Gateway is used as an ItemProcessor

In a more elaborate use case, an entire chunk can be sent externally by an item writer. In this case, the batch job only reads items and groups them, and once a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

chunk has been sent out, it will continue to read items and assembly chunks, without waiting for the result. 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 to read the chunks, handle them individually and return information about their successful completion.

The latter part in necessary because the batch process must update the JobExecution state continuously, in order to be able to acknowledge whether the step execution has been succesful or not. Because of that, the request and response messages contain metadata that helps the batch job executor to carry out this process (did we get responses for all chunks?, were there any errors?). So, the chunk messages are received by another special component, the ChunkProcessorChunkHandler which processes the chunk by delegating to an actual ItemProcessor and ItemWriter.

Figure 16.9 Externalizing chunk processing

By looking at Spring Integration as underlying support for boosting the concurrency and distribution capabilities of Spring Batch we have concluded our overview of the collaboration between the two frameworks. It is time to move on to the next chapter, but before that, let us quickly review the takeaways of this chapter

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

16.4 Summary

This chapter has provided you with an overview of batch processes and their typical features, and with a general understanding 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 and handling the state.

The main strength of Spring Batch is dealing with the complexities of defining and running batch jobs, in all details - such as chunking and transaction management, as well as state management, but does not include facilities for scheduling batch jobs or executing them in an event-driven fashion. It finds an ideal companion in Spring Integration, whose strength is dealing with the challenges of interacting with external systems and creating an event-driven environment which can be used for launching and managing batch jobs.

We have shown how 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.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

17 Scaling Messaging applications with OSGi™ and AMQP

In this chapter we investigate scalability of messaging applications. There are multiple types of scalability. Most dicussed is the scalability of an application at runtime, but the word is also used in other contexts, such as the scalability of a business model and the scalability of a project. This chapter will focus on scalability at runtime and scalability at development time.

First we will introduce OSGi and link it to Messaging to show the complementary nature of the two. OSGi addresses the scalability at development time mostly. After the part on OSGi we will introduce AMQP and show how it complements Spring Integration to address several runtime scalability concerns.

Since 1998 OSGi has been primarily driven as a needed extension of Java. Needed because Java's module system is simplistic. Java will load 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 and so OSGi was introduced to resolve this problem.

OSGi did not start out primarily as a modularity fix for the Java classloader, but first and foremost as an extension that would help to run dynamic applications that wouldn't have all their services active at all times during their life cycle. A very important side effect that was recognized very 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 bigger not only modularity is a concern, but also response times and deadlock prevention. The biggest danger to these two technical requirements are synchronous invocations over the network, to which OSGi doesn't offer a solution in itself. After getting this far in the book the following statement should be obvious: OSGi and Messaging are a match made in heaven. Spring and

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

OSGi have once been pictured by Peter Kriens as a puppy and a kitten lying in the same basket. Spring DM has taken as a long way to make this picture accurate, and from a functional point of view Spring Integration completes the circle. Spring Integration should not be credited too much in this, because as you will see in the coming sections *it just works* because of Spring DM. As a developer you can get a lot of benefit from using them together. As you can see in figure 17.1 Spring Integration can run in any OSGi container using Spring DM.

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.

To make the most of a Spring Integration application on an OSGi platform it is recommended that you read up on Spring DM and OSGi too using for example Spring DM in Action and OSGi in action (Manning).

Before we can show off the combination of messaging and OSGi we will first lay a foundation. We will first explore OSGi's module system quickly After that we will focus on the Service Registry, and only then are we ready to combine messaging (Spring Integration) and dynamic modules (Spring DM+OSGi). Finally we will look at how we can replace service implementations in a running system in detail.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

17.1 The OSGi module system

In this section we will go over the main functionality that is enabled by OSGi, it's module system. We will cover the reasons why the the module system is designed the way it is. Also we'll show how it is possible to dynamically (re)load bundles at runtime.

Let's say you're working on software in a team. The application grows bigger and you need to modularize it. As the requirements keep coming in and the application gets more successful you need different teams to work on parts of the application and after a while you might find that these different teams have different release cycles and depend on each others work. Modularity is becoming a hairy problem and you need a rock solid solution to it.

We're looking for a solution that allows you to do something that is depicted in figure 17.2. Different teams should be able to deliver on their own schedule without causing outage.

Figure 17.2 1. The integration team has set up the server and deployed some basic bundles to it. 2. The 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 source of notifications. 3. The flight notifications team writes a flight notifier bundle that works with the bus, in the mean time the bus and the sms service have been deployed. The obsolete basic bundles are uninstalled as their replacements kick in. 4. The new bundles are working in concert, replacement versions for individual bundles can be deployed as needed without bringing the system down.

Apart from a technical solution, to get to a situation like in figure 17.2 you'd need significant changes in the organisation. Readers should be aware that most of the effort in getting to this place will be in thoroughly defining and changing processes and architecture. Still, OSGi is an option nowadays and it can only work to your advantage if you are aware of it's workings and the options that it brings.

As mentioned OSGi brings modularity to Java. In this section we will gloss over the features of OSGi that are relevant to the suggested application of OSGi and messaging in enterprise architecture. But first we will need to look a little bit deeper into the vides in the Java type system.

In a normal Java runtime your code can be modularized into different packages,

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

within a jar and different jars on the classpath. Classes can hide their internals from other classes by using modifiers giving them encapsulation. Jars however give no such option. Once a classloader has has loaded a class it can never unload it. The only way to work around this is to use a dedicated classloader for the set of classes that you want to reload. It can be done, but it's not easy.

In figure 17.3 the internal representation of types in a classic classloader is shown. Because of this oversimplification, there is hardly any protection against type duplication and versioning issues.

Figure 17.3 Modularity and it's runtime representation in Java.

Not only is the runtime representation of types too simplistic, it is also too rigid, as is explained in the next paragraph.

The second problem you'll run into when you want to dynamically modify your running system is to change the classpath at runtime. If a new module needs to be deployed you might not want to restart all the other modules. Again you will find that it can be done, but it is not easy. The combination of these two problems becomes prohibitive for a single team with a production deadline.

In the late nineties a lot of companies faced the above problems and most of them found their way out of them by using RMI and web services. The downside of this is that going over the network is expensive (either in terms of performance, or in terms of hardware). So if it can be avoided as a requirement that would be good news.

OSGi provides a solution that gives us this good news. It abstracts away the complex but not impossible solutions to type reloading that are provided in standard Java. In return it introduces a powerful vocabulary that makes sense of modularity. First of all in OSGi a jar is turned into a *bundle* which is a jar with special manifest. In the manifest you can tell the OSGi container what the bundle

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

is, how to start and stop it, what it depends on and what it has to offer. By using the OSGi life cycle modules will be started in the right order and services will only be invoked if they can do their job.

17.1.1 The bundle life cycle in an OSGi environment

A typical happy scenario for a bundle life cycle starts with the installation of a bundle. The OSGi framework is now aware of the bundle and it is said to be INSTALLED. Now 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, another bundle needs to expose all the types that that bundle needs. If this is successful the bundle enters the RESOLVED state. After all the code dependencies are resolved a 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 is stopped (back to the RESOLVED state). You can see a state diagram of all these states in the figure 17.4

Figure 17.4 OSGi specifies a very clear life cycle for bundles. In contrast to plain Java, where class loading is out of the developers control, bundles can be installed, uninstalled, started and stopped as needed. The transitions between each of these states as depicted by arrows, are very clearly defined and restricted by the standard.

There are more details of course, for example a bundle might not move to the RESOLVED state because not all its required dependencies are available, and inside the ACTIVE state there are sub states that can be of interest.

During the rest of the chapter some of these details will be explained in more detail, in particular the way Spring DM hooks into bundle activation is important to understand so it is outlined in the next section. However this section serves as a a primer into OSGi, this book is not a book on OSGi, so we will not cover all the intricate details at length. We will merely cover just enough to establish the relationship between OSGi and Spring Integration. If you want to design a solution

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

around OSGi and messaging it is a good idea to refer to OSGi in Action (Hall et al). Now that you've got the concepts of OSGi in your mind you're ready for the next section, where we will 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

In the first section we've covered the basics of OSGi, but we're here to use OSGi in concert with Spring Integration. Before we can add Spring Integration back to the mix we should have a better understanding how OSGi and Spring DM enable dynamic reloading of services. It is these services that we can use from Spring Integration, but we 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 POJO's 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 there is another challenge. The problem that remains is that this registry as a simple and low level API which will require you to write some unwieldy code.

```
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. 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. Since the whole point of IoC is to make the client agnostic about the origin of it's dependency you typically don't need to change any java code to migrate an application to OSGi.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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)

Figure 17.5 Inside the META-INF directory we see the OSGi standard MANIFEST.MF file containing the dependency directives. In addition to this there is a spring folder. Spring DM will read the xml files in this folder and start a special application context (BundleContext) containing the beans describes in these files.

When a bundle is loaded with Spring DM, some special steps will be taken during startup that will instantiate the `ApplicationContext` specific to this bundle. This BundleContext is like a normal Spring application context, with some added features particular to OSGi.

Let's look at an example of a service that is exposed by bundle com.manning.siia.integration.notifications.sms and referenced from bundle com.manning.siia.integration.notifications. The service implements an interface exposed as a type by [TODO]. Now in the sms bundle we can expose the service like this:

```
<osgi:service ref="smsNotifier" interface="c.m.i.n.Notifiable"/>
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Which will create a `ServiceRegistration` for the smsNotifier bean in the sms bundle's context. Subsequently we can reference the service from the notifier bundle like this:

```
<osgi:reference id="smsNotifier" interface="c.m.i.n.Notifiable"/>
```

This will create a proxy for the object in the ServiceRegistry 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've used the osgi namespace here. This will only work if you add it to the header of your spring xml files as we've seen a few times before.

```
<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/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">
```

So we have two bundles, both with their own Spring context, but we've 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. In fact it is possible to 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!*

As an astute reader you might have noticed that while showing you OSGi we've also lost a few of our established advantages of Spring Integration. In fact we have completely avoided the use of Spring Integration upto here. Instead we've created a reference between bundles based on the Notifiable interface, which is fine, but in that process we have also had to introduce a dependency from the notifier bundle on the bean name of the smsNotifier. Suddenly generic notifications become 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

other than the payload. This is exactly what we will 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 is passed between them. This is very helpful in itself as you have seen in the previous chapters, but without a fix for Java's flat classpath the services will still be visible to each other within the same jvm, or you will have to take on the extra complexity of adding a transport layer in between VM's.

17.3.1 Reasons to combine OSGi with Messaging

Let's say you would like to get cash from your account. You can walk into a bank and draw a number, but that will expose you to all kinds of services that you are not interested in. You will have to avoid stepping in gum, you'll have to deal with the attitude of the clerk. Even worse other customers can interact with you too, they could tell you about their day, ask you directions. The bank is a dangerous and confusing place. If you would just interact with the cash machine things would be a lot simpler. The service will get only the information that you want to expose. OSGi is like that, it avoids unwanted coupling between the service and its clients. In figure 17.6 you can see the conceptual difference of using OSGi in a bank.

Figure 17.6 On the left the interaction with the bank analogous to the classic Java interaction with a lot of unwanted potential interaction. On the right the improvement of the exposed cash machine, analogous to a service exposed using OSGi.

In the bank example you might argue that there is social value talking to other human beings, but inside a jvm this is typically not an advantage.

The picture is completed by OSGi that gives you 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 bigger than what a single team can manage on a single release cycle. In a system that is large, but stays within the boundaries of one corporate entity, there is no merit in separating modules with a transport layer, security boundaries and all the performance drag

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

that that implies. Because of this, OSGi combined with messaging is particularly useful in large enterprise architectures.

A system that is modularized with OSGi and decoupled with messaging can evolve much more naturally while multiple teams work on it in their own rhythm. The production system runs on as little nodes as possible to reduce the network overhead, while bundles are installed in different release schedules, even without ever bringing the system down.

This idea might sound less reliable than redeploying everything on a single node and bring that back up, but there is in fact no fundamental difference in terms of deployment architecture, apart from the fact 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. In fact in comparison with a typical production setup, the equivalent OSGi based solution would leave less room for error because of the strict life cycle definition provided by the OSGi standard.

The real downside is to be found in development time complexity. Debugging and testing an OSGi based application requires a different skill set than debugging a classic java program, so like with any technical solution OSGi should only be used when the reduction of maintenance costs will more than balance 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 his toolbox.

In the next few sections we will show some examples of how bundles can be wired together using Spring Integration and OSGi. We will look at publish subscribe communication, point to point messaging, load balancing and also 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 pub-sub configuration the publisher is mostly indifferent as to whether or not someone listens. When a twitterer puts up a tweet, he is hopefully not too concerned with the fact if someone or the other will read his message.

Publishing is in fact a service that is offered to interested subscribers. This is an interesting contrast with point to point messaging, where delivery of the message is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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, have the publisher expose a publish subscribe channel as a service, and then reference it inside the subscriber bundle. In the publisher context:

```
<publish-subsubscribe-channel name="notifications" />
```

This is 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 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 in places where one endpoint specifically wants to tell another endpoint something. Because there is 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 will be a bit disappointed.

Now when we talk about examples of point to point messaging, it is usually the sender using a service of the receiver. This is the exact opposite of publish subscribe messaging, where the service is that you can subscribe to the feed of published messages. If two OSGi bundles are connected through point to point messaging, the startup is actually a bit tricky. There are in fact two ways to connect bundles and it depends on what you would like to do which way you should go.

It is quite obvious that sharing a channel makes sense, but you will still have to decide which bundle gets to own the shared channel. It really depends on the way you want to design each bundle what makes the most sense 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 light weight loadbalancer within the application. This is the Scatterer part of the Scatter-Gather pattern. If the channel sits with the receiving

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

end you can let multiple senders sink their messages into the same bundle. Thus it is natural to call this pattern Gatherer.

Together these patterns can form a Scatter-Gather application. The have the additional benefit that you can scale out at runtime. Just add a bundle that does it's work on a different (new) node in your cloud and you're done. In the next section we will look at the boundaries of the messaging infrastructure and integrate with bundles that are not depending on Spring Integration at all.

17.3.4 Using Gateways and ServiceActivators to avoid Spring Integration dependencies

When you're composing a large application built by several teams it is highly unlikely that you would 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 and having the same dependency from all these modules is in contradiction with this goal.

A large application using Spring Integration will typically have one or several modules using Spring Integration to wire the rest of the modules together.

Figure 17.7 The green bundles here have a Spring Integration dependency, but the white ones do not.

In figure 17.7 you can see that the Spring Integration dependent bundles are using services in the non Spring Integration bundles. This begs 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, we can

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

simple 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. We want to keep the twitter support bundle free of Spring Integration dependencies, so we cannot expose a channel as a service through the OSGi service registry.

Instead of a channel we simply expose a gateway, implementing an interface from the twitter support bundle. So in the notifier bus we 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 Spring Integration for that matter.

TODO: transition to next main section

17.4 Robust and scalable messaging with AMQP

The single most discussed feature in all messaging systems is Guaranteed Delivery. The reason for this is that whenever a sensible person thinks about messaging, the first risk that will pop into his head is the risk to lose a message. We have described ways to mitigate this risk earlier [TODO references, check for consistency]. Still, having a system at your finger tips that provides the desired guarantees is a promise that should at the very least make you sit up and listen. This section is about such systems, in particular AMQP implementations.

AMQP is a standard that was developed to bridge the gap in functionality that lies between JMS and raw networking protocols: a platform independent solution to guaranteed delivery. JMS as described in chapter 10 is a popular solution, but it

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

has a few capital flaws that deserve your attention before you default to it.

First JMS is practically tied to Java. There are some implementations of JMS to make it usable from for example .NET, but these are vendor specific and hardly compliant. Implementing messaging between different platforms with JMS is painful.

Second in retrospect JMS got their abstractions wrong. It shoe-horned publish subscribe messaging and point to point messaging into the same abstraction (Destination) and thereby hid some very important differences that new developers would discover too late at their own expense. In defense of the JMS spec leads, they didn't have books like Enterprise Integration Patterns and experience to draw from. Now, ten years later it is easy to note these things in front of an audience that for a large part has already felt the pain of those mistakes. It is sensible to prevent repeating those mistakes in a new specification. This is what AMQP has done.

17.4.1 What do you need to run an AMQP implementation

In this book we are using RabbitMQ as an AMQP platform. When writing this chapter SpringSource had just acquired Rabbit Technologies. To some it's comforting to know that when combining Spring and RabbitMQ there is a company that is supporting the configuration. More importantly, SpringSource and Rabbit engineers will be encouraged to work together on these platforms.

On the other hand, since AMQP doesn't require any vendor specific extensions (unlike JMS) there is no reason why the things we're showing wouldn't work transparently on another AMQP platform. That means more power to the developer. I like that.

To run RabbitMQ you need to install Erlang and then you're good to install RabbitMQ server. There are plenty of good instructions online, and the whole installation shouldn't take more than a few minutes. For other platforms like 0MQ, Qpid and others installation is similarly straight forward. Only when you need to set up a distributed system you need to jump a few more hoops.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

17.4.2 The basics of the AMQP model

If you come from a JMS background there is a new jargon to be learned for AMQP. Luckily the basics are quite simple and we will reproduce them here. Messaging in AMQP is established by three collaborators: Publishing Application, Server, Consuming Application. The publishing application sends a message to an Exchange on the server the Server holds a binding between the Exchange and the Message Queue. Finally the Consuming Application consumes the message from the Message Queue. In figure 17.8 you can see these actors work together.

Figure 17.8 On the left there are two client applications: consumer and publisher. On the right you can see the server. The server holds the exchange that is responsible for routing the message and the queue that holds the message until the consumer requests them.

17.4.3 Accessing AMQP from Java

Now that you know how things work conceptually, the only thing left is to access the AMQP server from Java. At the time of writing an AMQP API for Spring is being developed as a Spring Extensions project. The code base hasn't matured yet, but as an illustration I will show you how it works and more importantly how to tie it to Spring Integration. From the earlier chapters you should have built the knowledge to tie any java code to Spring Integration, so those details we can glance over quickly.

Most AMQP implementations have their own java client API. So does RabbitMQ. If you have the com.rabbitmq.amqp-client on your classpath you would be able to run the following test without problems:

```
@Test
public void shouldSendAndReceiveWithPlainRabbit() throws IOException {
    Connection connection = connectionFactory.createConnection();
    Channel channel = connection.createChannel();
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
        channel.exchangeDeclare("exchange", "direct");
        channel.queueDeclare("queue");
        channel.queueBind("queue", "exchange", "foo");
        channel
                .basicPublish("exchange", "foo", null, "Hellow".getBytes());
        GetResponse response = channel.basicGet("queue", false);
        assertThat(response, is(notNullValue()));
        channel.close();
        connection.close();
    }
```

If you're used to the niceties of the profusion of templates in the Spring framework, like the JmsTemplate or the JdbcTemplate, you might miss something here.

First of all, when an exception is thrown here, the connection will not be closed. Second, the exception is checked, so you need a catch or a throws clause. Third, creating the connection is expensive, so you need some kind of caching. The list goes on a bit, but you have heard these arguments before I'm sure. At the bottom line this is not a bad API, it is just a little simple. For use in a large project it is sensible to add another layer that takes care of the particulars.

For this book we used a snapshot build of the Spring AMQP Extensions project (which had not seen it's first milestone at the time of writing). Using a nightly build of an API is far from recommendable for production code, but for a book we dared take the chance. Just to be clear on this, the wrapper code that we use here is pre-alpha, but RabbitMQ and it's java client are ready for prime time.

When you have the wrapper code in place it should be possible to do something like this:

```
    @BeforeClass
    public void setupAmqp() {
        RabbitAdminTemplate adminTemplate =
                   new RabbitAdminTemplate(connectionFactory);
        adminTemplate.declareQueue(new Queue("myqueue"));
    }

    @Test
    public void shouldSendAndReceiveWithSpring() {
        RabbitTemplate template = new RabbitTemplate(connectionFactory);
        template.convertAndSend("myqueue", "foo");
        Object result = template.receiveAndConvert("myqueue");
        assertThat(result, is(notNullValue()));
    }
```

This looks a bit more tidy and in the future the integration with a Spring

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Application context is going to be more streamlined using this approach.

Finally sending from a Spring Integration app is as easy as wrapping a `<service-activator>` around a service that sends messages to AMQP. In the future it is likely that a namespace will be developed that will make these things even easier.

Using an AMQP implementation to take care of the heavy lifting of messaging and the Spring Integration semantics to take care of the integration part, you have a powerful combination at your disposal. In contrast to OSGi, AMQP answers the runtime scalability demands rather than focus on the development time scalability. In this way AMQP and OSGi are two very different sides of the same puzzle.

17.5 Summary

In this chapter we've introduced OSGi and showed you how it fits very well with a Messaging application. We've described some architectural patterns that are made possible through the use of OSGi. We have also shown you some of the more intricate details of loading a Spring Integration application that is spread over multiple bundles.

With OSGi we have focussed mainly on development time scalability and we have shown that OSGi and Spring Integration give you a lean alternative to an ESB.

Setting up a runtime that would make you comfortable doing things around OSGi described in this chapter on production is a bit of a job of course, so you shouldn't expect OSGi to magically make complex things happen without giving you some interesting puzzles along the way. As with any technology, be conservative when deciding whether or not to use it, but if you have a use case on your hands that would clearly benefit from it, be brave, it's worth it.

With AMQP we have given you an answer to runtime scalability issues like Guaranteed Delivery and scaling out to multiple nodes. With both these options at your disposal you can attack different types of scalability problems. And with that you are ready for even the toughest challenges of enterprise integration development.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

18 Putting your pipes to the test

One of the great accomplishments of our industry over the last 20 years has been Test Driven Development [TDD]. Where many methodologies have proven only to work in theory, or have simply never proven their need, TDD has flourished. The reason for this is simple: clients only pay willingly for working software and there is 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 only possible proof of correctness: if you don't have a clue how you are going to test the application, you do not have any business building it.

There are of course many ways of testing software. One of the oldest ways is to just do it manually. While this option is still valid and in wide use because of its simplicity, any experienced developer will 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 a huge return on investment. SUnit, invented by some of the bright minds that also started the Agile movement was in the late nineties ported to Java. If you don't know JUnit yet, firmly pull the handbrake towards you and make sure you do before you read any further.

After reading the rest of the book and being exposed to test code there as well, there will have been little news in terms of what you can do. The main thrust of the chapter is about *why* you should pick a certain option.

Because in this chapter we are discussing a topic that cuts across the other topics, the chapter is organized a bit differently. Like the other chapters it uses code samples from the sample application, but it does not have a particular use case it focuses on. It also does not have a dedicated Under the Hood section, first

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

of all because the test framework code is simple enough to embed with it's usage, and also because it the test code itself serves as an example of how to extend JUnit to deal with the messaging domain.

This chapter will build on top of JUnit tests from the sample to show you the intricate details of testing asynchronous and concurrent programs that have been built using the pipes and filters architecture supported by Spring Integration. As you might have experienced testing these types of applications is more convoluted than testing classical applications. When you're using Spring Integration you will find that it is often necessary to write tests that assert things about the payload of a message that is received from a channel, or to write assertions about particular headers on such a message. The boilerplate code that would normally be needed to do this is for a large part taken care of by Spring Integration's testing framework. This test framework builds on top of Hamcrest to offer custom matchers that can be used with `assertThat`. It also boasts some convenience classes related to the asynchronous nature of certain Spring Integration components like `QueueChannel` .

SIDEBAR

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, is best known for it's use in JUnit's `assertThat` method.

```
assertThat("Tango", is(not("Foxtrot"));
```

Which can make both the test code and the thrown exceptions more readable.

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 in a different schedule than the configuration that makes up the message bus. In this chapter you will read about different use cases for the test module, since the authors strongly believe that tests

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

should not hide complexity, there will be no under the hood section. Instead you will find all the details right with the usage examples.

SIDEBAR **Testing behavior with Mocks**

Mocking is used to allow assertions on behavior instead of state. Frameworks like EasyMock, JMock or Mockito help create mocks that allow easy verification of behavior. Mockito is probably the simplest mocking framework around, although certain advanced features are not supported this makes it an excellent candidate to use as illustration. If you are not familiar with mocking, you are encouraged to read a the article "Mocks aren't Stubs" by Martin Fowler.

To be able 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 or not. This jar is packaged separately from the main Spring Integration distribution, as we don't want to force transitive dependencies on Mockito and Hamcrest on all Spring Integration users.

```
Source does NOT exist: /home/aadev/tomcat/webapps/aadev/static/publishedVersions/374
```

So you have added another Spring Integration jar on your classpath, now what? Let's take a look at what's inside that jar.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

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 at least do better than that. In the next section you'll see how we can factor the unwrapping logic out of our testcases.

18.1.1 Unwrapping payloads

With Spring Integration's test module you can make use of the matchers that deal with unwrapping internally. First we'll look at an example, then we'll look at the underlying details. Starting with the simple example above we've already noticed some pain points in the test code. The main problem is in the assertion. The `is` matcher isn't particularly well suited to deal with messages.

Ideally we would 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. The only thing you need to do is to add the following import statement:

```
import static org.s..i..test.matcher.PayloadMatcher.*;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

This will give 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 above.

```
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, while the code that is needed is almost trivial. Let's just have a look at the code of the PayloadMatcher in listing 18.1.

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);
    }
}
```

As you can see here we extended `TypeSafeMatcher` and implemented 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

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Integration test. There are a few more features bundled in Spring Integration's test module. For example you might require matching on headers too, as we'll see in the next section.

Matching messages is particularly useful if the framework is doing work that is 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 all that much sense in an integration test (as most of that would already be covered in a unit test around the service. In some cases for example when using expression language things change however. Let's take another example from the sample application.

As you might remember from chapter 3, 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 up into sub commands for rental cars, hotel rooms and flights. We're going to zoom in on the tests for the splitter. In figure 18.1 you can see the way we've modified the fixture to allow us to control incoming and outgoing messages.

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.

We can now make sure that 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 will remain relatively simple as you can see.

```
Source does NOT exist: /home/aadev/tomcat/webapps/aadev/static/publishedVersions/374/examp
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

The trick here is to plug into the existing system without replacing logic that you want to test. In this case we've overridden the subCommands channel with a queue channel and made sure that no other components are receiving from it. As you can see it's quite simple and useful to write test that make assertions on the payload of a message. More often than not however the headers of messages play at least as big a role in the integration of the system. In the next section we will 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 rather than 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 you integration test again. However, 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 that contains a command that has already been executed. It is then followed by a Service Activator that keeps track of all the successfully executed commands, for example in a table that is also used by the filter.

Now this construction contains enough complexity and business value to make it the target of a test, but testing all these components in isolation does not 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")
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

```
        , 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 quite 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. One remark that needs to be made with the matchers is that it's not trivial to deal with the fact that a message received from a channel does not have the benefits 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 a bit more complicated. You will now have to deal with the particulars of the mocking framework as well. We will 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 is not relevant for your test you can use mocks or stubs to factor their influence out of the test fixture. People often speak of *mocking mocking out* a dependency to refer to such a refactoring. A briefing on mocking goes beyond the scope of this chapter, but it we will 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 that is typically used to test state and is typically created as an inner class in a test case.

In this chapter we will show only mocks using Mockito, this serves our needs 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). However, 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 actually

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

becomes sensible to mock out business code and let a message flow through the system, just to see if it is handled correctly by the infrastructure. This would concern routing, filtering header enrichment, 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 finally 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, therefore it should have a designated unit test. If you're using SpEL (TODO: reference to earlier chapter / appendix), there is 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 you would genericly 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 is something that would only make sense as an integration test for the framework itself, but it serves as a simple example. As the complexity of your configuration increases it will become increasingly useful to verify the flow of the message through the system.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Figure 18.2 First record the behavior of the mock (0). Then send a test message to the input channel (1). After waiting for the message to come out of the other end (2), verify that the appropriate operations on the mock have been invoked (3). Variations on this recipe will work also in more complex cases.

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. Because of this it makes sense to replace the service with a mock, but since this service is wired as a bean in a Spring context, this is not as easy as injecting a component with mocked collaborators as you would do in a normal unit test. There is a trick you can use however.

```
<bean id="service"
      factory-method="mock"
      class="org.mockito.Mockito">
   <constructor-arg
       value="com.you.ServiceToMock"/>
</bean>
```

This overrides the service bean with a bean that is 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're using a mock you can simply verify that it was touched and that's it. This is a very good option for channel adapters too, as it gives you a generic way to deal with them. In section 18.3 we will use this strategy as well to deal with the need to wait for an invocation to happen before we start asserting things about the result.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

This section focused only on Mockito, but similar support for EasyMock, JMock, RMock can be implemented along the same lines. It's not very likely that Spring Integration will natively support all of these frameworks in the near future. After reading this chapter you should have some idea on how to implement the test support of your choosing, and there is a good chance that someone out there has shared some matcher you might reuse.

The next section will take a small dip into the realm of concurrency. We have already secretly used some concurrency features of Spring Integration to our advantage in tests, but now it's about time to walk over the different possible strategies in this regard in detail. Especially the combination of mocking and latching is very powerful, so stay tuned!

18.3 Testing an asynchronous system

One of the trickiest things to solve cleanly in tests are assertions around related actions by multiple threads. A big advantage of 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 decouple cause and effect using a framework, rather than having the complexities of asynchronous hand-off emerge in business code. At runtime however, these subtleties are essential to the proper functioning of the system. Therefore it is essential that they are accounted for in tests. This section focuses on the concerns around testing an asynchronous system. This is a system where multiple threads are involved in performing a bit of work (e.g. processing a message). In the end we will address testing thread safety of business code.

If a part of your system is designed to process messages asynchronously there are certain things that you should keep an eye on. As you've seen in chapter 4, processing messages asynchronously can be done in quite a few different ways. You can use a <queue ... /> element, or a <dispatcher .../>. Also you when using a publish subscribe channel configured with a task executor you are using asynchronous hand-off. Finally there are a few endpoints that can be configured with a `TaskExecutor` that will process a message in a different thread then the thread pushing the message in.

To give you a handle on this you can remember that whenever an endpoint or channel is using storage or a task-executor, it can cause asynchronous hand-off.

Whenever asynchronous hand-off is involved, *there are no chronological guarantees without explicit locking*. Luckily getting explicit locking in place is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

quite 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 fact in most cases that's precisely what you should do. We will look at a few exceptions later, but in the vast majority of cases plugging into the output channel of your context and indeed just waiting for the output message to arrive is sufficient. So how do we go about this?

You might be expecting to see a loop with `Thread.sleep(..)` in there, or if you're a little more familiar with Java's concurrency support you might have expected a `CountDownLatch`. Things are a lot simpler than this in fact.

For a standard test case there is only this simple recipe to follow: 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 we receive `m` from the output channel *before* we assert that the service has been invoked. Also we use a timeout in the receive call to ensure the test will not run indefinitely. The assertions are done in the order that you expect them to succeed.

We're piggybacking on the contract of the receive method here. Since `receive` is a blocking call, we don't have to do any additional waiting to ensure a *happens before* relation ship between the service invocation and the verification. However complex you contexts get, it is almost always possible to find some output that will only arrive after the behavior that you're trying to verify has been executed.

The timeout is also very important. When designing a test case it is important to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

understand that the main function of the test is to fail when the software is not behaving correctly. This main function is best implemented when the failure clearly points out what part of the behavior was incorrect. If there would be no timeout on the receive call, and an exception would be thrown from the service, the test could run indefinitely. The test would not fail in this particular case of malfunctioning in the software under test, therefore it would be flawed. You could put a general timeout on the test itself, and that would prevent the test from running indefinitely. This would fix the flaw, but the failure message would have no relation to the cause of the failure. This would make the test correct, but not very useful.

And finally the assertions should be in the right order. If the service isn't invoked, most likely there will be no output message. In this case mixing the order of the assertions will make the test fail with a "Output didn't arrive" message, or worse, a `NullPointerException`. Thinking carefully about the order of the assertions will avoid this problem.

Before we look at exceptional cases where waiting for output is not an option, or not enough, we'll examine the need for proper testcases 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 very clear that debugging is a skill. It's a skill that any excellent developer has and that any novice developer should strive to learn. It is 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 it's lifetime". Before you dive into hours of debugging, you should ask yourself this simple question: 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 sysadmin for the log file.

The reason we're bringing this up is that in a concurrent scenario debugging is much harder. Spring Integration is inherently a concurrent framework, and if there is a concurrent bug in your code it might very well be exposed by wiring your service in a Spring Integration context. In single threaded scenarios debugging is great. It helps you understand the code much quicker than just reading through it would. In many cases you don't want to fix all the logging in your program, you

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

just quickly want to see what's going on. That's fine, usually. However, if multiple threads are entering the problem area of your code, debugging loses all its powers. Suddenly the debugger changes all the timelines and very often completely hides the bug from your sight. You could say that concurrency is debugger kryptonite.

Luckily logging and test cases stay much more loyal, also when dealing with concurrent access. That's not to say that finding and analysing a concurrency issue is easy. It's merely possible and that is just about good enough. So head 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, we're ready for the next section that will show you how to wait for messages to terminate inside an endpoint using a latches and mocks.

18.3.3 Injecting latches into endpoints

Some times an endpoint has no output. As have read in Chapter 5 these types of endpoints are called channel adapters. A channel adapter will take the payload of a message and feed it to a service. This service may be a bean in your context, but it might also be the database, a webservice, the file system or standard output.

The tricky part is of course to wait for the invocation of this service before you start making assertions about the state of the system. In this section we will 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 goes too far for 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've shown how to mock out services from tests. In that example there was no need for latching in side the mock because we could just wait for the message to come out of the output channel as discussed in the previous section. So let's explore an endpoint that is different in that respect.

When notifications are sent to the user, we send them over an asynchronous communication channel. We specifically don't want to wait for the external system to confirm something was sent synchronously. That would block too many of our threads. So looking at the email outbound channel adapter for instance we need to confirm a message reaches this adapter, but we don't need to test the actual sending of the email in this test. There should of course be another test for this, but that's out of the scope of this book.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

Let's say we're interested in testing the following snippet:

```
Source does NOT exist: /home/aadev/tomcat/webapps/aadev/static/publishedVersions/374
```

We design our 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.

To do this we first need to make sure we replace the `smsNotifierBean` with a mock. This can be done with the same trick mentioned before.

```
Source does NOT exist: /home/aadev/tomcat/webapps/aadev/static/publishedVersions/374
```

Once that job is done, we can focus on the test itself.

The test itself simply 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, as the channel is a `QueueChannel` We have to wait for the message to arrive before we can verify. This can be done using a latch injected into the mock.

```
Source does NOT exist: /home/aadev/tomcat/webapps/aadev/static/publishedVersions/374
```

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 simply becomes:

```
Source does NOT exist: /home/aadev/tomcat/webapps/aadev/static/publishedVersions/374
```

Because the `notify` method returns void, we are using Mockito's `doAnswer` method to record the behavior. We're essentially telling 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 and then we can execute assertions under the safe assumption that they will happen after the message has arrived at the endpoint.

18.3.4 How do I prove my code thread safe?

The short answer: you don't. You can prove the correctness of your code under concurrent access, but it is usually unfeasible to test all possible concurrent scenarios and make sure they meet the specifications. There are however a few things that you can do to make it very likely that your code will be thread safe.

We will not go into great detail here, as concurrency is already discussed in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

detail in Chapter 14. Just repeat to yourself:*pass immutable objects between stateless services*.

Now when testing is concerned, you can do your best to make sure that concurrency bugs have a chance to surface in your test. For one you should use *at least* the same amount of threads as in your production application in some of your integration tests. 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 full-proof. Test written like this might cause intermittent failures, which in many cases means there is a concurrency bug in your code.

This is where we are reaching the boundaries of the Spring Integration framework. Spring Integration is very good at running things concurrently, 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. There are frameworks out there like *ConcuTest* that are helpful in provoking concurrency bugs by injecting yields and waits into your bytecode. If you learn these tools you will have a better chance at resolving concurrency bugs.

18.4 Summary

In this chapter we've formalized our understanding of testing Spring Integration applications. First we've discussed the test support in Spring Integration's own test framework. After that we've detailed the strategies and rationale for mocking out external dependencies and business services form message flow tests. Finally we discussed testing asynchronous applications on a broader level, we've shown how to enforce chronological order in tests with `QueueChannel`s or mocks and latches and we've discussed thread safety.

Within the scope of the test framework, we've seen 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. After discussing the message payload matching we've done the same for header matching. Not only support for unwrapping headers has been discussed, but also some of the extras needed when dealing with the whole map of headers. We have also discussed the Mockito support for matching messages.

After discussing the framework support, we have made a case for mocking business services out of tests. Because Spring Integration configurations contain a

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615
Licensed to Roger Studner <rstudner@gmail.com>

code that is 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've explained how to use a blocking receive call to ensure chronological order in tests. Also we have explained when this doesn't work and you need to use latches within mocks to enforce *happens-before* relationship.

This is in fact the last chapter. This doesn't mean it is least important. A proper understanding of how to test your application is both the end and the beginning of craftsmanship in software engineering. In the remainder of the book we will summarize what you have learned from this book about Spring Integration as a whole.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
http://www.manning-sandbox.com/forum.jspa?forumID=615