



MANNING

# Spring Integration

IN ACTION

Mark Fisher  
Jonas Partner  
Marius Bogoevici  
Iwein Fuld





**MEAP Edition  
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to

[www.manning.com](http://www.manning.com)

# Table of Contents

## **Part 1: Background**

- Chapter 1: Introduction to Spring Integration
- Chapter 2: Enterprise Integration Fundamentals
- Chapter 3: Introduction to the Sample Application

## **Part 2: Messaging**

- Chapter 4: Messages and Channels
- Chapter 5: Message Endpoints
- Chapter 6: Getting Down to Business
- Chapter 7: Message Routing

## **Part 3: Integrating Systems**

- Chapter 8: Working with XML
- Chapter 9: JMS Integration
- Chapter 10: Sending and Receiving Mail
- Chapter 11: File System Integration
- Chapter 12: Web Services

## **Part 4: Advanced Topics**

- Chapter 13: Monitoring and Management
- Chapter 14: Concurrency and Performance
- Chapter 15: Spring Integration and the Rich Web
- Chapter 16: Messaging and Batch Operations
- Chapter 17: OSGi(tm) and dm Server
- Chapter 18: Testing

# *Introduction to Spring Integration*

Events happen. No, that's not a bumper sticker for enterprise integration developers with a deranged sense of humor, it's just a fact of modern life. Throughout the 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. 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.

Now, let's be honest, we all occasionally let an incoming call go straight to voice mail. The pessimists among us may even believe this is the real reason "Caller ID" was created. So maybe you feel bad about that or maybe you don't, but imagine what it would be like if you were required to respond to every call, email, or message as soon as it was received. You probably would fail to accomplish any significant task due to the disruptions. What saves us from such event-driven paralysis is the fact that we can respond to most of those events and messages at a time that is convenient for us. Others are urgent and must be addressed as soon as possible. Luckily, the events usually carry enough information for us to make that judgment quickly (ok, so maybe the pessimists are right about Caller ID).

Not to get overly metaphysical, but apparently we design software in our own image. It is increasingly common that we, as enterprise developers, have to build solutions that respond to a wide variety of events. 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 outdated 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. 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.

These are exciting times for developers. Many of us have some degree of fascination with building these message-driven solutions that take advantage of the high degree of connectivity available today. Yet due to the complexity of building such applications, it 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.

## 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.

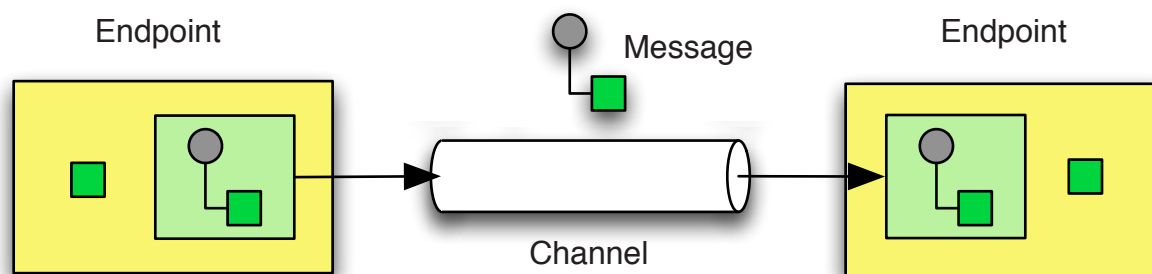


Figure .1. Two areas of focus for Spring Integration: lightweight intra-application messaging and flexible inter-application integration.

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.

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
- 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 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.

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. 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. 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.

### Message, Channel, Endpoint

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

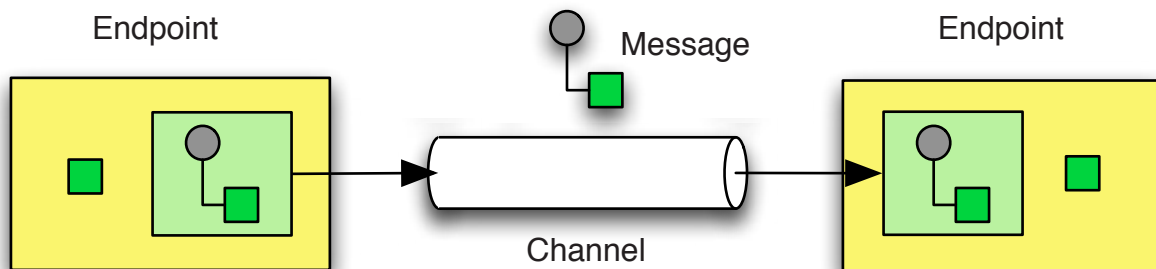


Figure .2. Figure 1.2 A Message is passed through a Channel from one Endpoint to another Endpoint

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 or simply a reference to a record in a database.

The Message Channel is the connection between multiple endpoints. The Channel can be used to implement the details of how and where a message is delivered, but should not need to interact with the payload of a message. Messages can be delivered in the same thread, or they can be buffered in a queue shared by separate sender and receiver threads. Messages can be delivered to a single endpoint (point-to-point) or to any endpoint that is listening to the channel (publish-subscribe). Apart from these details, the main goal of the channel is to decouple the endpoints on both sides from any transport concerns and from each other.

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, 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

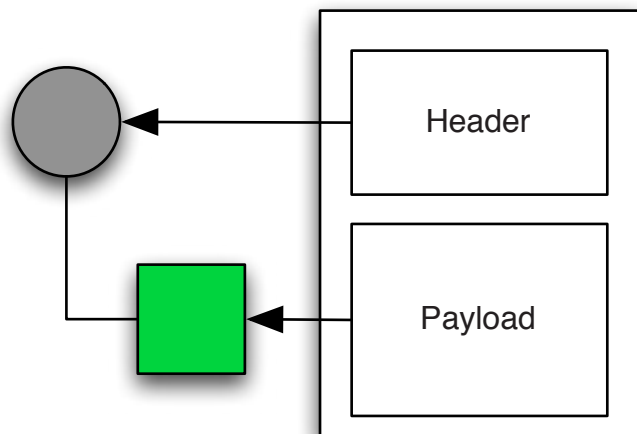


Figure 3.

### 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 Object over the channel, 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.

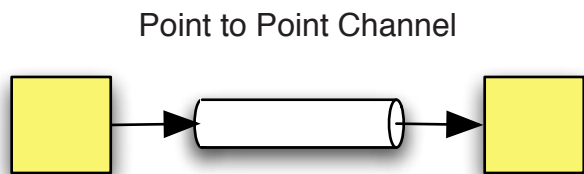


Figure 4.

### 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 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.

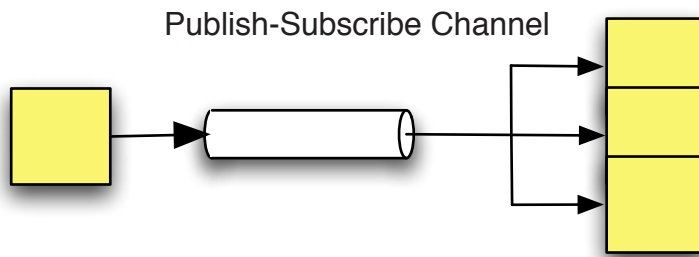


Figure .5.

Figure 1.5 A Publish/Subscribe Channel

Contrary to 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.

### .1.1. 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

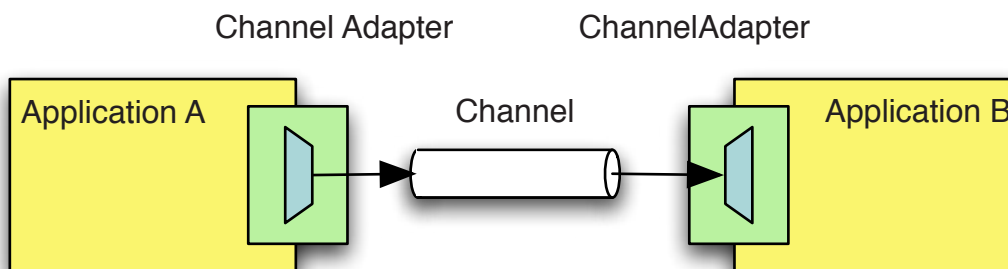


Figure .6.

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

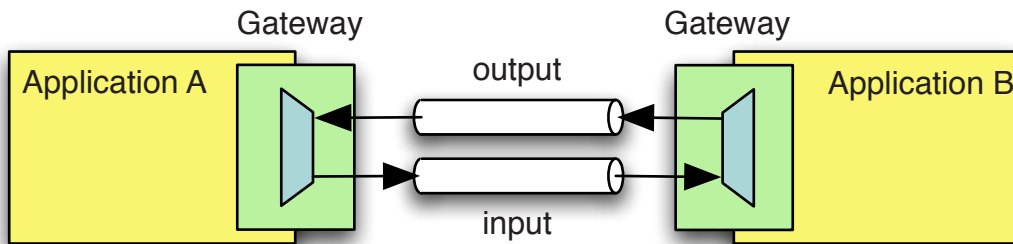


Figure 7.

Figure 1.7 Messaging Gateway

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 stay 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 Gateway a bit in comparison to the EIP book.

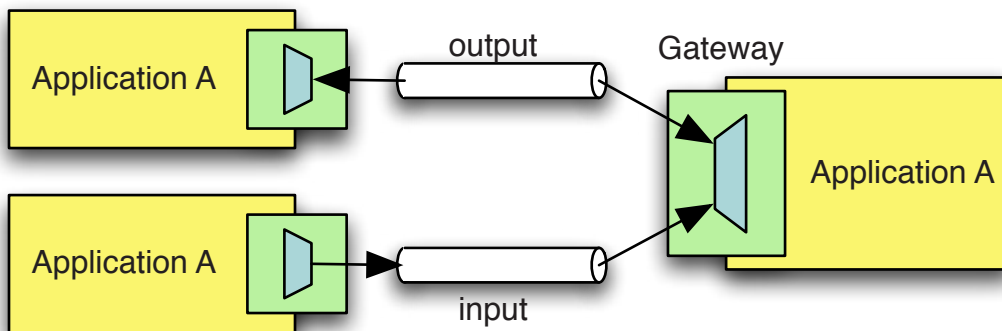


Figure 8.

Figure 1.8 Gateway and Channel Adapters

## Service Activator

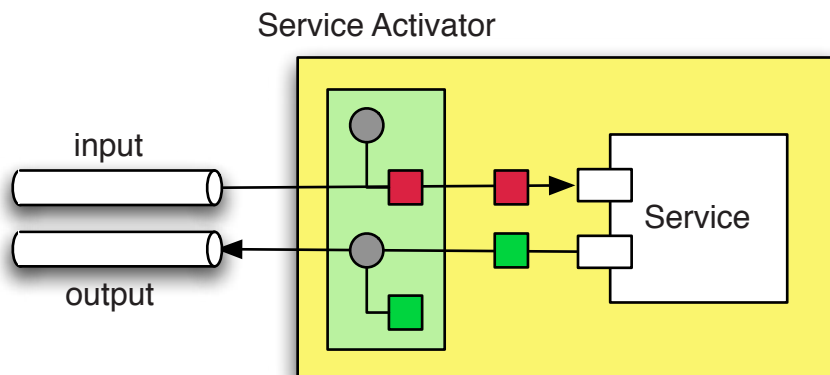


Figure 9.

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

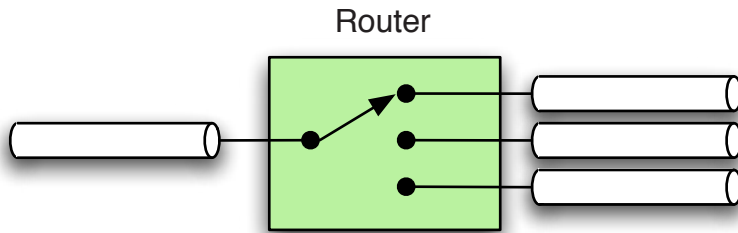


Figure .10.

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

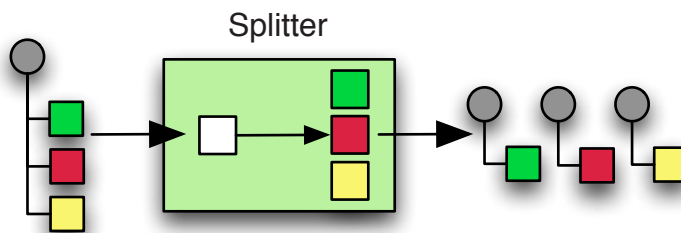


Figure .11.

Figure 1.11 Splitter

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

## Aggregator

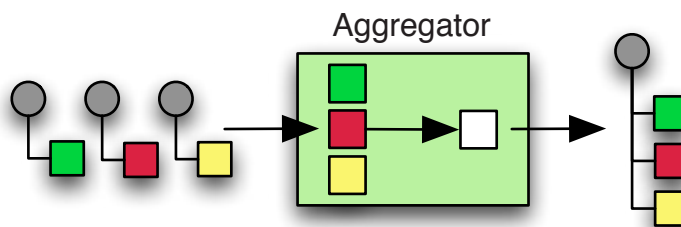


Figure .12.

Figure 1.12 Aggregator

An aggregator waits for a sequence of messages that are correlated, and it will merge them together when the

sequence 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 sequence. 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 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, there are 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 "Return Address" that we will see later. And finally, there are patterns that 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.

## 2. 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:

```
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 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 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. In 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 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="currentQuote"/> </jms:message-listener-container>
    <bean id="quoteService"
        CLASS="example.QuoteService"/>
```

Whatever client is passing request Messages to the 'quoteRequest' destination could also provide a JMSReplyTo property on that request Message. Spring's listener container 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.

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. Listing 1.1 Scheduling a Task programmatically

```
Runnable task = new Runnable() { public void run() {
    File file = 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 .2. Listing 1.2 Scheduling Tasks declaratively with Spring

```
<task:scheduled-tasks scheduler="myScheduler">
    <task:scheduled ref="filePoller" method="poll"
```

```

    fixed-rate="60000"/> </task:scheduled-tasks>
    <task:scheduler id="myScheduler"
    pool-size="5"/>

```

As you can see, this provides a 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 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.

## 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 {
    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 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.HelloWorld"/>

```

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 Message 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 3. Listing 1.3 Hello World with Spring Integration

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

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 {
    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="helloService" default-request-channel="names"/>
<channel id="names"/> <service-activator input-channel="names"
    ref="helloService" method="sayHello"/> <beans:bean
    id="helloService" class="ssia.ch1.HelloWorld"/>
```

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 wants with the return value. Here, we will just move the console printing to the caller side. The service instance would be reusable in a number of situations. The revised main method now has no direct dependencies on Spring Integration.

### Listing 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/context.xml"); HelloService
        helloService = context.getBean("helloGateway", HelloService.class);
        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 that you now have a basic understanding of how to apply 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

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. 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 .13.**

Figure 1.21 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 (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] See Appendix A for an overview of the 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 over it". In software we rely on things being a bit more definite.

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