Recipe: Creating an Integration-Ready Ticketing System Shell

**Key technologies**
Spring Integration, Spring Web MVC

**Background**
This recipe is part of a series of recipes that incrementally builds an integrated technical support ticketing system to support an external-facing, internally-developed university portal.

**Problem**
You want to implement a ticketing system in such a way that it’s easy to integrate it with other systems, including self-service ticketing and external ticketing systems used by escalated support teams.

**Solution**
We’re going to use Spring Integration as a basis for our integration efforts. In this recipe, we’re not going to do any integrations—we’re just going to implement a basic ticketing shell against a simple message bus. That way it will be easier to perform the systems integrations.

Figure 1 shows what we’re going to build in this recipe.
Figure 1 The basic functionality of our sample tech support ticketing system: creating and escalating tickets.

Let’s go over figure 1 in some detail since this is our first exposure to Spring Integration. We’ll revisit some of these points later on in our application context configuration, but it will help to begin our discussion with a lay of the land.

First, we’re developing an application for internal tech support staff. Reps might conceivably use the system for several reasons, but we’ll focus on two use cases:

- A student or faculty user calls the help desk and reports a portal issue. A rep at the help desk uses the ticketing system to create a new ticket on the user’s behalf.
- If a ticket looks like a development issue, then the rep uses the ticketing system to escalate the ticket to the portal development team.

In figure 1, we can see the relationship between the customer, the rep, and the ticketing system. You may notice something a little unusual. Normally, we’d just inject our TicketService directly into the UI (for example, into any Spring Web MVC controller that wants to use it), and then the UI would call service methods such as createTicket() and escalateTicket() as needed. But we’re not doing that here. Instead, we’ve inserted a bunch of messaging components—collectively forming a message bus—between the UI and services. The reason is that we want to define ticket creation and ticket escalation as standardized, bus-accessible services, and the first step to doing that is to create the bus and attach the services to the bus. Then our ticketing system UI accesses those services through the bus.

The TicketGateway interface hides the details of the messaging system away from UI. This allows the UI to use the bus without having to know anything about messaging concepts such as channels and messages. We’ll see in listing 5 that the TicketGateway interface is expressed entirely in ticketing domain terms.

The message flow out of the TicketGateway depends on whether the rep is creating or escalating a ticket. Let’s take the ticket creation fork first. When the rep creates a ticket, the gateway will wrap the ticket in an org.springframework.integration.core.Message and drop the Message onto the newTicketChannel. The reason for wrapping the ticket is that we’re in the messaging domain now, and the messaging components making up the messaging system need to know how to talk to one another; hence Messages. Each Message has a payload (in the present case, the ticket is the payload) and an arbitrary set of headers that can be used for routing and processing.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/wheeler/
Our newTicketChannel is a publish/subscribe (pub/sub) channel. (The underlying class is PublishSubscribeChannel.) Any number of endpoints can subscribe to a pub/sub channel and, when messages appear on the channel, all subscribed endpoints receive it. This stands in contrast to direct channels, which allow only a single receiving endpoint.¹

The subscriber essentially is our ticket creation service (as represented by TicketService.createTicket()). We wrap the ticket creation service with a service activator endpoint, which allows us to access the service from the messaging system by adapting the service interface to the messaging endpoint interface.

 Entirely analogous comments apply to the ticket escalation fork, so we won’t work through all the gory details. Instead let’s start looking at some code.

Listing 1 shows the domain object for tickets.

Listing 1 Ticket.java, a domain object representing a simple ticket

```java
package ticketing.model;
import java.util.Date;
public class Ticket {
    private String custName;
    private String custEmail;
    private String issueCategory;
    private String issueDesc;
    private Date dateCreated;
    ... getters and setters for the above,
    along with a descriptive toString() method ...
}
```

Our Ticket class isn’t anything special, but it will do the job we need it to do.

We’ll now build our equally basic service tier. Our goal is to keep it focused on business logic and, in particular, free from integration code. This offers greater flexibility during integration efforts because we can configure and aggregate services as we wish.

Building the service tier

In listing 2 we have a basic ticket service interface.

Listing 2 TicketService.java service interface

```java
package ticketing.service;
import ticketing.model.Ticket;
public interface TicketService {
    void createTicket(Ticket ticket);
    void escalateTicket(Ticket ticket);
    Ticket findTicket(long id);
}
```

Besides the createTicket() and escalateTicket() methods, we have a findTicket() method that we can use to look up tickets prior to escalating or doing other things with them.

Listing 3 presents a dummy implementation of our TicketService interface. Take a look and we’ll discuss it afterward.

Listing 3 TicketServiceImpl.java service implementation

```java
package ticketing.service;
import java.util.Date;
import java.util.logging.Logger;
import org.springframework.stereotype.Service;
```

¹ Pub/sub and direct channels are roughly analogous to JMS topics and queues, respectively.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/wheeler/](http://www.manning.com/wheeler/)
import ticketing.model.Ticket;

@Service("ticketService")
public class TicketServiceImpl implements TicketService {
    private static Logger log =
            Logger.getLogger(Logger.GLOBAL_LOGGER_NAME);

    public void createTicket(Ticket ticket) {
        log.info("TicketService is creating ticket...");
    }

    public void escalateTicket(Ticket ticket) {
        log.info("TicketService is escalating ticket...");
    }

    public Ticket findTicket(long id) {
        Ticket ticket = new Ticket();
        ticket.setCustomerName("Frank Black");
        ticket.setCustomerEmail("frankblack@pixies.com");
        ticket.setIssueCategory("library");
        ticket.setIssueDescription("Can’t download MP3s in the library.");
        ticket.setDateCreated(new Date());
        return ticket;
    }
}

#1 Returns dummy ticket

The createTicket() and escalateTicket() methods are just placeholder methods. In a typical implementation, createTicket() would probably save the ticket in a database, and it might even send out a confirmation email. In our case, it would be fine to save the ticket in the database, but we would not send out a confirmation because we consider sending confirmation emails a good candidate for a reusable service.\(^2\)

Similarly, a typical escalateTicket() implementation might update the ticket with a note concerning the escalation, save the ticket, and then call a web service that creates a new ticket in the escalation ticketing system. But, once again, that ticket creation is a good service candidate, so we shouldn’t include it inside the escalateTicket() method.

These considerations aren’t to say that we don’t want to send confirmation emails and create escalated tickets. We do. We just want to handle the more general definition of the ticket creation and ticket escalation processes in the integration layer instead of inside TicketServiceImpl itself. That allows us to achieve the integration we want without inflexibly coupling TicketServiceImpl to external systems.

Finally, our findTicket() method (#1) simply returns a dummy ticket. We won’t do anything with this ticket other than escalate it; in particular, we won’t send frankblack@pixies.com any confirmation emails.

Listing 4 shows our applicationContext.xml. All it does is find and create the TicketServiceImpl bean, but stay tuned; we’ll look at another application context file shortly.

Listing 4 /WEB-INF/applicationContext.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/[CA]spring-context-2.5.xsd">
    <context:component-scan base-package="ticketing.service" />
</beans>
```

\(^2\) Indeed, we wrote exactly this type of functionality in chapter 4 for new user registrations.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/wheeler/
That’s it for the service tier. Clearly we’ve met our goal of avoiding integration logic in our service bean. We’ll turn now to the integration tier, where the integration concern lives.

**Building the integration tier**

We’ve already mentioned that we’re just going to set the integration stage in this recipe. Listing 5 presents a gateway interface into our messaging system.

**Listing 5 TicketGateway.java gateway interface hides messaging from the app**

```java
class TicketGateway {
    @Gateway(requestChannel = "newTicketChannel")
    void createTicket(Ticket ticket);

    @Gateway(requestChannel = "escalatedTicketChannel")
    void escalateTicket(Ticket ticket);
}
```

**#1 A gateway method**

The purpose of this interface is to hide messaging concepts away from applications that want to use the messaging system. Here that means defining the interface in terms of Tickets rather than Messages.

While it would be possible to implement the interface manually, we’ll see in a bit how to use Spring Integration to generate dynamic proxies for the interface. We use the optional @Gateway annotation to assist with that. The proxy will wrap the Ticket with a Message. (That is, the Ticket will be the Message payload.) The requestChannel element specifies where the proxy should drop the Message. So at [#1] we’re saying that when an app calls createTicket(), we want to place a Message containing that Ticket on the newTicketChannel. Table 1 shows the options for the @Gateway annotation.

**Table 1 @Gateway annotation and available elements**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>requestChannel</td>
<td>Target channel for request messages (messages wrapping the argument)</td>
</tr>
<tr>
<td>requestTimeout</td>
<td>Timeout for sending messages on the requestChannel</td>
</tr>
<tr>
<td>replyChannel</td>
<td>Source channel for reply messages (messages wrapping the return value)</td>
</tr>
<tr>
<td>replyTimeout</td>
<td>Timeout for receiving messages on the replyChannel</td>
</tr>
</tbody>
</table>

In listing 5 our methods are request-only (no reply) since their return values are void, so we don’t specify a replyChannel.

Listing 6 is a Spring application context configuration that defines our simple message bus.

**Listing 6 /WEB-INF/applicationContext-integration.xml message bus definition**

```xml
<?xml version="1.0" encoding="UTF-8"?>
```

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/wheeler/](http://www.manning.com/wheeler/)
We're using a separate application context configuration to define our message bus as this approach helps keep the messaging concern separate. We begin by making the Spring Integration namespace the default namespace (#1), which saves a few keystrokes. Accordingly, we define an explicit namespace prefix for the beans namespace (#2). Note also the schema location for the Spring Integration namespace (#3).

Refer back to figure 1 to get a visual sense of what we're building here. First, we use the <gateway> element (#4) to define the TicketGateway dynamic proxy that we discussed above. Besides the service-interface attribute specifying the gateway interface, <gateway> supports several attributes that describe the gateway’s interaction with its request and reply channels, including default-request-channel, default-request-timeout, default-reply-channel and default-reply-timeout, each having the semantics you'd expect given what we laid out in table 1. We aren't using the attributes here, however, having opted instead to specify the request channel using the @Gateway annotation as we saw in listing 5.

At #5, we define a publish/subscribe channel called newTicketChannel. Recall that our gateway will place a Message on the newTicketChannel whenever somebody calls createTicket() on the TicketGateway. Note also that newTicketChannel will be a focal point for the integrations we do in recipes 3 and 4 when we add support for ticket creation via inbound web forms and e-mails, respectively. Because it's a pub/sub channel, it can have multiple subscribers, and each subscriber receives a copy of any message that appears on the channel.

Finally, at #6, we have an endpoint called a service activator. The purpose of a service activator is to adapt service methods to the message bus. In the general case, a service method has an argument and a return value, both of which need to be mapped to messages in order to play nicely with the bus. That's where the service activator comes in. It wraps a service method such that payloads are extracted from incoming messages (messages arriving on the input-channel) and passed into the service method. The return value is wrapped with a new message and dropped onto the output-channel. In this case, the TicketService.createTicket() method doesn't have a return value, so there's no output-channel definition.

The same statements apply to the service activator for the escalateTicket() method.

Believe it or not, that's all there is to the integration tier for now. So far, we haven't done anything that we couldn't have done just as easily (well, OK, more easily) without Spring Integration, but the idea is that, as we start hooking up external dependencies, the value of separating business code from integration code will grow.

Let's knock out the web tier.

**Building the web tier**

We're going to use Spring Web MVC for the web tier. Listing 7 shows how we access our integration tier from our controller.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/wheeler/](http://www.manning.com/wheeler/)
The TicketController depends on both the TicketService (#1) and the TicketGateway (#2). The POST version of our create() method (#3) uses the gateway to perform the ticket creation. While we could certainly call createTicket() directly, doing so would mean that we’d have to put any future integration code inside the service itself instead of isolating it and wiring it up to the bus. So we use the gateway instead. Notice also that the gateway has indeed allowed us to use the message bus without referencing messaging constructs like channels and messages.

In the POST version of our escalate() method, we use the service and the gateway together. We use the service to look up the Ticket (#4) and then pass the Ticket along to the gateway (#5) so the gateway can wrap it with a Message and put it on the bus.

Our simple app has four JSPs. Two of them are just “thank you” pages, so we’ll ignore those here (check the code download). The other two are forms for ticket creation and escalation. First, listing 8 presents the primary part of the ticket creation form.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/wheeler/
Listing 9 shows a ticket escalation form snippet. We’ve hardcoded a fake ID into the form. You may have noticed in listing 3 that the actual ID doesn’t matter.

Listing 9 /WEB-INF/jsp/ticket/escalate.jsp form for escalating tickets

Listing 10 contains a completely pedestrian DispatcherServlet application context.

Listing 10 /WEB-INF/main-servlet.xml

Finally, listing 11 is just a standard web.xml configuration, which we’re including just for completeness.
We should now have a working Spring Integration app. Assuming you have the Maven code download, enter http://localhost:8080/ticketing/main/ticket/create into your browser. You should see a web form called Create a trouble ticket. When you submit the form, you should see that TicketService.createTicket() generates a console message that says INFO: TicketService is creating ticket...
If so, pat yourself on the back, because you’ve made it through this recipe.

Discussion

This recipe, though lengthy, established an integration-ready web application. We learned how to invoke service beans using a service bus.

One major advantage of building an application using a message bus is that it makes integration with other applications simpler. The reason is that Spring Integration provides channel adapters for pulling several different kinds of external channel into the bus infrastructure. External systems can communicate with our ticketing system over those external channels. So, even though we haven’t yet performed any actual integration, we’ve created a bus that will serve as a solid foundation once you start doing the integrations.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/wheeler/
Here are some other Manning titles you might be interested in:

**Spring in Action, Third Edition**  
Craig Walls

**Spring Batch in Action**  
Thierry Templier, Arnaud Cogoluegnes, Gary Gregory, and Olivier Bazoud

**Spring Integration in Action**  
Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld

Last updated: February 3, 2012

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/wheeler/](http://www.manning.com/wheeler/)