Published on *Javalobby* (http://java.dzone.com)

Spring Integration: A Hands-On Tutorial, Part 1

By *wwheeler*

Created *2009/08/18 - 12:26am*

This tutorial is the first in a two-part series on Spring Integration. In this series we're going to build out a lead management system based on a message bus that we implement using Spring Integration. Our first tutorial will begin with a brief overview of Spring Integration and also just a bit about the lead management domain. After that we'll build our message bus. The second tutorial [1] continues where the first leaves off and builds the rest of the bus.

I've written the sample code for this tutorial as a Maven 2 project. I'm using Java 5, Spring Integration 1.0.3 and Spring 2.5.6. The code also works for Java 6. I've used Maven profiles to isolate the dependencies you'll need if you're running Java 5. The tutorials assume that you're comfortable with JEE, the core Spring framework and Maven 2. Also, Eclipse users may find the m2eclipse plug-in helpful.

To complete the tutorial you'll need an IMAP account, and you'll also need access to an SMTP server.

Let's begin with an overview of Spring Integration.

A bird's eye view of Spring Integration

Spring Integration is a framework for implementing a dynamically configurable service integration tier. The point of this tier is to orchestrate independent services into meaningful business solutions in a loosely-coupled fashion, which makes it easy to rearrange things in the face of changing business needs. The service integration tier sits just above the service tier as shown in figure 1.

Following the book Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf (Addison-Wesley), Spring Integration adopts the well-known pipes and filters architectural style as its approach to building the service integration layer. Abstractly, filters are information-processing units (any type of processing—doesn't have to be information filtering per se), and pipes are the conduits between filters. In the context of integration, the network we're building is a messaging infrastructure—a so-called message bus—and the pipes and filters and called message channels and message endpoints, respectively. The network carries messages from one endpoint to another via channels, and the message is validated, routed, split, aggregated, resequenced, reformatted, transformed and so forth as the different endpoints process it.

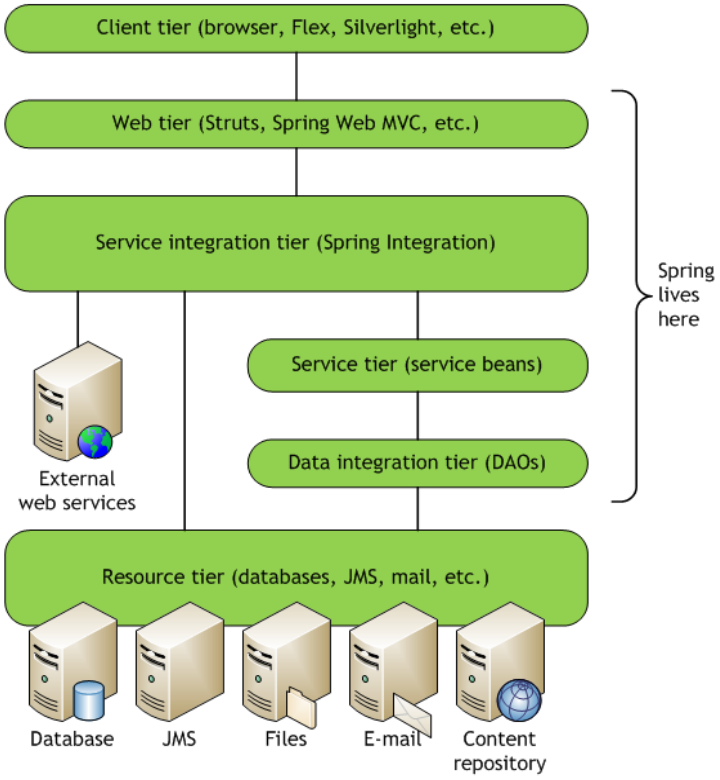


Figure 1. The service integration tier orchestrates the services belowit.

That should give you enough technical context to work through the tutorial. Let's talk about the problem domain for our sample integration, which is enrollment lead management in an online university setting.

Lead management overview

In many industries, such as the mortgage industry and for-profit education, one important component of customer relationship management (CRM) is managing sales leads. This is a fertile area for enterprise integration because there are typically multiple systems that need to play nicely together in order to pull the whole thing off. Examples include front-end marketing/lead generation websites, external lead vendor systems, intake channels for submitted leads, lead databases, e-mail systems (e.g., to accept leads, to send confirmation e-mails), lead qualification systems, sales systems and potentially others.

This tutorial and the next use Spring Integration to integrate several of systems of the kind just mentioned into an overall lead management capability for a hypothetical online university. Specifically we'll integrate the following:

- a CRM system that allows campus and call center staff to create leads directly, as they might do for walk-in or phone-in leads
- a Request For Information (RFI) form on a lead generation ("lead gen") marketing website
- a legacy e-mail based RFI channel
- an external CRM that the international enrollment staff uses to process international leads
- confirmation e-mails

Figure 2 shows what it will look like when we're done with both tutorials. For now focus on the big picture rather than the details.

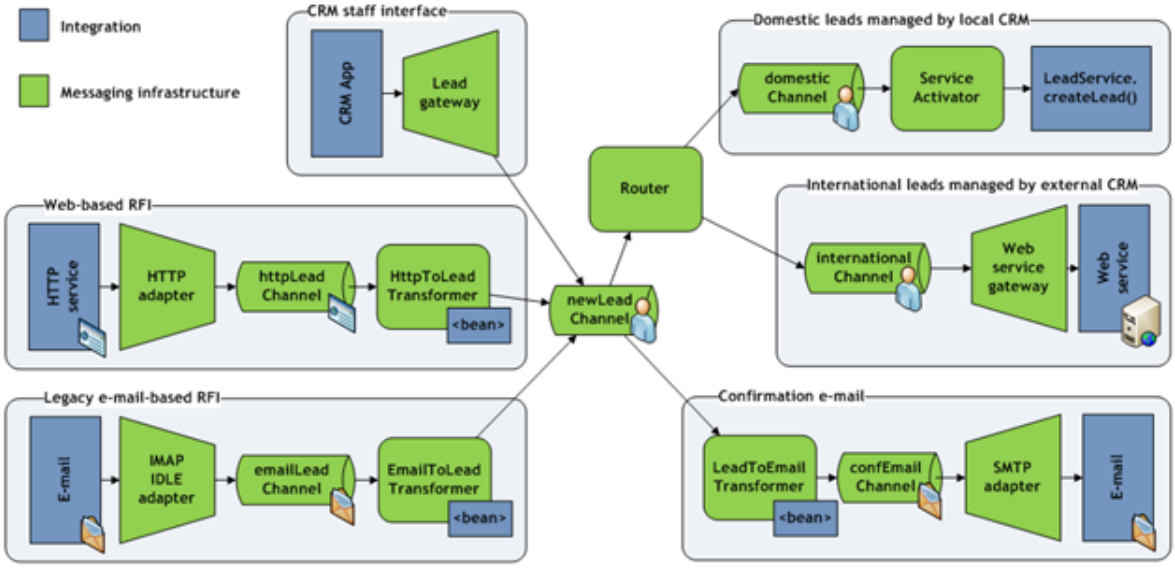


Figure 2. This is the lead management system we'll build.

For this first tutorial we're simply going to establish the base staff interface, the (dummy) backend service that saves leads to a database, and confirmation e-mails. The second tutorial will deal with lead routing, web-based RFIs and e-mail-based RFIs.

Let's dive in. We'll begin with the basic lead creation page in the CRM and expand out from there.

Building the core components

[You can download the source code for this section of the tutorial here [2]]
We're going to start by creating a lead creation HTML form for campus and call center staff. That way, if walk-in or phone-in leads express an interest, we can get them into the system. This is something that might appear as a part of a lead management module in a CRM system, as shown in figure 3.

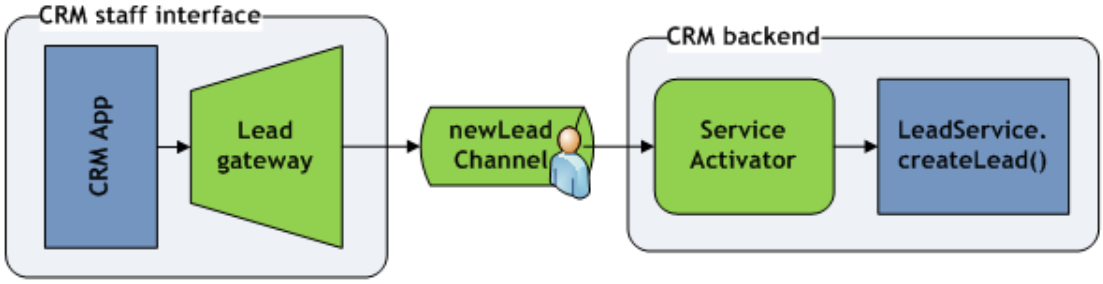


Figure 3. We'll build our lead management module with integration in mind from the beginning.

Because we're interested in the integration rather than the actual app features, we're not really going to save the lead to the database. Instead we'll just call a createLead() method against a local LeadService bean and leave it at that. But we will use Spring Integration to move the lead from the form to the service bean.
Our first stop will be the domain model.

Create the domain model

We'll need a domain object for leads, so listing 1 shows the one we'll use. It's not an industrial-strength representation, but it will do for the purposes of the tutorial.

Listing 1. Lead.java, a basic domain object for leads.

```
package crm.model;

... other imports ...

public class Lead {
    private static DateFormat dateFormat = new SimpleDateFormat();

    private String firstName;
    private String middleInitial;
    private String lastName;
    private String address1;
    private String address2;

    ... other fields ...

    public Lead() { }

    public String getFirstName() { return firstName; }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    ... other getters and setters, and a toString() method ...
}
```

There is nothing special happening here at all. So far the Lead class is just a bunch of getters and setters. You can see the full code listing in the download.
If you thought that was underwhelming, just wait until you see the LeadServiceImpl service bean in listing 2.

Listing 2. LeadServiceImpl.java, a dummy service bean.

```
package crm.service;

import java.util.logging.Logger;
import org.springframework.stereotype.Service;
import crm.model.Lead;

@Service("leadService")
public class LeadServiceImpl implements LeadService {
    private static Logger log = Logger.getLogger("global");

    public void createLead(Lead lead) {
        log.info("Creating lead: " + lead);
    }
}
```

This is just a dummy bean. In real life we'd save the lead to a database. The bean implements a basic LeadService interface that we've suppressed here, but it's available in the code download.
Now that we have our domain model, let's use Spring Integration to create a service integration tier above it.

Create the service integration tier

If you look back at figure 3, you'll see that the CRM app pushes lead data to the service bean by way of a channel called newLeadChannel. While it's possible for the CRM app to push messages onto the channel directly, it's generally more desirable to keep the systems you're integrating decoupled from the underlying messaging infrastructure, such as channels. That allows you to configure service orchestrations dynamically instead of having to go into the code.

Spring Integration supports the Gateway pattern (described in the aforementioned Enterprise Integration Patterns book), which allows an application to push messages onto the message bus without knowing anything about the messaging infrastructure. Listing 3 shows how we do this.

Listing 3. LeadGateway.java, a gateway offering access to the messaging system.

```
package crm.integration.gateways;

import org.springframework.integration.annotation.Gateway;
import crm.model.Lead;

public interface LeadGateway {

    @Gateway(requestChannel = "newLeadChannel")
    void createLead(Lead lead);
}
```

We are of course using the Spring Integration @Gateway annotation to map the method call to the newLeadChannel, but gateway clients don't know that. Spring Integration will use this interface to create a dynamic proxy that accepts a Lead instance, wraps it with an org.springframework.integration.core.Message, and then pushes the Message onto the newLeadChannel. The Lead instance is the Message body, or payload, and Spring Integration wraps the Lead because only Messages are allowed on the bus.
We need to wire up our message bus. Figure 4 shows how to do that with an application context configuration file.

Listing 4. /WEB-INF/applicationContext-integration.xml message bus definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/integration/spring-integration-1.0.xsd">

    <gateway id="leadGateway"
        service-interface="crm.integration.gateways.LeadGateway" />

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

    <service-activator
        input-channel="newLeadChannel"
        ref="leadService"
        method="createLead" />
</beans:beans>
```

The first thing to notice here is that we've made the Spring Integration namespace our default namespace instead of the standard beans namespace. The reason is that we're using this configuration file strictly for Spring Integration configuration, so we can save some keystrokes by selecting the appropriate namespace. This works pretty nicely for some of the other Spring

projects as well, such as Spring Batch and Spring Security.
In this configuration we've created the three messaging components that we saw in figure 3. First, we have an incoming lead gateway to allow applications to push leads onto the bus. We simply reference the interface from listing 3; Spring Integration takes care of the dynamic proxy. Next we create a publish/subscribe ("pub-sub") channel called newLeadChannel. This is the channel that the @Gateway annotation referenced in listing 3. A pub-sub channel can publish a message to multiple endpoints simultaneously. For now we have only one subscriber—a service activator—but we already know we're going to have others, so we may as well make this a pub-sub channel.

The service activator is an endpoint that allows us to bring our LeadServiceImpl service bean onto the bus. We're injecting the newLeadChannel into the input end of the service activator. When a message appears on the newLeadChannel, the service activator will pass its Lead payload to the leadService bean's createLead() method.
Stepping back, we've almost implemented the design described by figure 3. The only part that remains is the lead creation frontend, which we'll address right now.

Create the web tier

Our user interface for creating new leads will be a web-based form that we implement using Spring Web MVC. The idea is that enrollment staff at campuses or call centers might use such an interface to handle walk-in or phone-in traffic. Listing 5 shows our simple @Controller.

Listing 5. LeadController.java, a @Controller to allow staff to create leads

```
package crm.web;

import java.util.Date;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import crm.integration.gateways.LeadGateway;
import crm.model.Country;
import crm.model.Lead;

@Controller
public class LeadController {

    @Autowired
    private LeadGateway leadGateway;

    @RequestMapping(value = "/lead/form.html", method = RequestMethod.GET)
    public void getForm(Model model) {
        model.addAttribute(Country.getCountries());
        model.addAttribute(new Lead());
    }

    @RequestMapping(value = "/lead/form.html", method = RequestMethod.POST)
    public String postForm(Lead lead) {
        lead.setDateCreated(new Date());
        leadGateway.createLead(lead);
        return "redirect:form.html?created=true";
    }
}
```

This isn't an industrial-strength controller as it doesn't do HTTP parameter whitelisting (for example, via an @InitBinder method) and form validation, both of which you would expect from a real implementation. But the main pieces from a Spring Integration perspective are here. We're autowiring the gateway into the @Controller, and we have methods for serving up the empty form and for processing the submitted form. The getForm() method references a

Countries class that we've suppressed (it's in the code download); it just puts a list of countries on the model so the form can present a Country field to the staff member. The postForm() method invokes the createLead() method on the gateway. This will pass the Lead to the dynamic proxy LeadGateway implementation, which in turn will wrap the Lead with a Message and then place the Message on the newLeadChannel.

There are a few other configuration files you will need to put in place, including web.xml, main-servlet.xml and applicationContext.xml. There's also a JSP for the web form. As none of these relates directly to Spring Integration, we won't treat them here. Please see the code download for details.

With that, we've established a baseline system. To try it out, run

```
mvn jetty:run
```

against crm/pom.xml and point your browser at

```
http://localhost:8080/crm/main/lead/form.html
```

You should see a very basic-looking web form for entering lead information. Enter some user information (it doesn't matter what you enter—recall that we don't have any form validation) and press Submit. The console should report that LeadServiceImpl.createLead() created a lead. Congratulations!

Even though we now have a working system, it isn't very interesting. From here on out (this tutorial and the next) we'll be adding some common features to make the lead management system more capable. Our first addition will be confirmation e-mails; the next tutorial will present further additions.

Adding confirmation e-mails

[The source for this section is available here [5]]

After an enrollment advisor (or some other staff member) creates a lead in the system, we want to send the lead an e-mail letting him know that that's happened. Actually—and this is a critical point—we really don't care how the lead was created. Anytime a lead appears on the newLeadChannel, we want to fire off a confirmation e-mail. I'm making the distinction because it points to an important aspect of the message bus: it allows us to control lead processing code centrally instead of having to chase it down in a bunch of different places. Right now there's only one way to create leads, but figure 2 revealed that we'll be adding others. No matter how many we add, they'll all result in sending a confirmation e-mail out to the lead.

Figure 4 shows the new bit of plumbing we're going to add to our message bus.

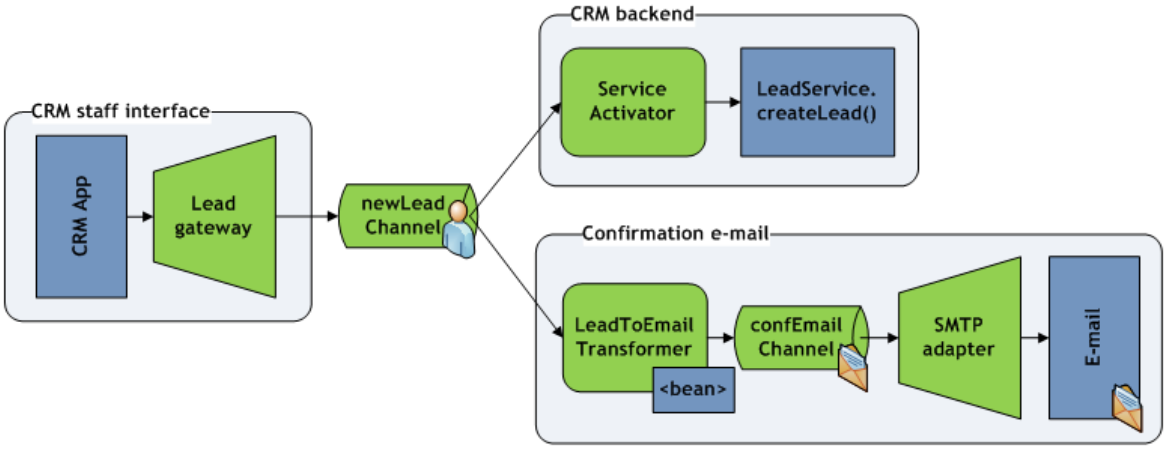


Figure 4. Send a confirmation e-mail when creating a lead.

To do this, we're going to need to make a few changes to the configuration and code.

POM changes

First we need to update the POM. Here's a summary of the changes; see the code download for details:

- Add a JavaMail dependency to the Jetty plug-in.
- Add an org.springframework.context.support dependency.
- Add a spring-integration-mail dependency.
- Set the mail.version property.

These changes will allow us to use JavaMail.

Expose JavaMail sessions through JNDI

We'll also need to add a /WEB-INF/jetty-env.xml configuration to make our JavaMail sessions available via JNDI. Once again, see the code download for details. I've included a /WEB-INF/jetty-env.xml.sample configuration for your convenience. As mentioned previously, you'll need access to an SMTP server.
Besides creating jetty-env.xml, we'll need to update applicationContext.xml. Listing 6 shows the changes we need so we can use JavaMail and SMTP.

Listing 6. /WEB-INF/applicationContext.xml changes supporting JavaMail and SMTP

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    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/spring-context-2.5.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">

    <jee:jndi-lookup id="mailSession"
        jndi-name="mail/Session" resource-ref="true" />

    <bean id="mailSender"
        class="org.springframework.mail.javamail.JavaMailSenderImpl"
        p:session-ref="mailSession" />

    <context:component-scan base-package="crm.service" />
</beans>
```

The changes expose JavaMail sessions as a JNDI resource. We've declared the jee namespace and its schema location, configured the JNDI lookup, and created a JavaMailSenderImpl bean that we'll use for sending mail.
We won't need any domain model changes to generate confirmation e-mails. We will however need to create a bean to back our new transformer endpoint.

Service integration tier changes

First, recall from figure 4 that the newLeadChannel feeds into a LeadToEmailTransformer endpoint. This endpoint takes a lead as an input and generates a confirmation e-mail as an output, and the e-mail gets pipes out to an SMTP transport. In general, transformers transform given inputs into desired outputs. No surprises there.
Figure 4 is slightly misleading since it's actually the POJO itself that we're going to call LeadToEmailTransformer; the endpoint is really just a bean adapter that the messaging infrastructure provides so we can place the POJO on the message bus. Listing 7 presents the LeadToEmailTransformer POJO.

Listing 7. LeadToEmailTransformer.java, a POJO to generate confirmation e-mails

```
package crm.integration.transformers;

import java.util.Date;
import java.util.logging.Logger;
import org.springframework.integration.annotation.Transformer;
import org.springframework.mail.MailMessage;
import org.springframework.mail.SimpleMailMessage;
import crm.model.Lead;

public class LeadToEmailTransformer {
    private static Logger log = Logger.getLogger("global");

    private String confFrom;
    private String confSubj;
    private String confText;

    ... getters and setters for the fields ...

    @Transformer
    public MailMessage transform(Lead lead) {
        log.info("Transforming lead to confirmation e-mail: " + lead);

        String leadFullName = lead.getFullName();
        String leadEmail = lead.getEmail();
        MailMessage msg = new SimpleMailMessage();

        msg.setTo(leadFullName == null ?
                leadEmail : leadFullName + " <" + leadEmail + ">");

        msg.setFrom(confFrom);
        msg.setSubject(confSubj);
        msg.setSentDate(new Date());
        msg.setText(confText);

        log.info("Transformed lead to confirmation e-mail: " + msg);
        return msg;
    }
}
```

Again, LeadToEmailTransformer is a POJO, so we use the @Transformer annotation to select the method that's performing the transformation. We use a Lead for the input and a MailMessage for the output, and perform a simple transformation in between.

When defining backing beans for the various Spring Integration filters, it's possible to specify a Message as an input or an output. That is, if we want to deal with the messages themselves rather than their payloads, we can do that. (Don't confuse the MailMessage in listing 7 with a Spring Integration message; MailMessage represents an e-mail message, not a message bus message.) We might do that in cases where we want to read or manipulate message headers. In this tutorial we don't need to do that, so our backing beans just deal with payloads.

Now we'll need to build out our message bus so that it looks like figure 4. We do this by updating applicationContext-integration.xml as shown in listing 8.

Listing 8. /WEB-INF/applicationContext-integration.xml updates to support confirmation e-mails

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
    xmlns:mail="http://www.springframework.org/schema/integration/mail"
    xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="http://www.springframework.org/schema/integration/mail
          http://www.springframework.org/schema/integration/mail/spring-integration-mail-1.0.xsd
          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/spring-context-2.5.xsd
          http://www.springframework.org/schema/integration
          http://www.springframework.org/schema/integration/spring-integration-1.0.xsd">

      <context:property-placeholder
          location="classpath:applicationContext.properties" />

      <gateway id="leadGateway"
          service-interface="crm.integration.gateways.LeadGateway" />

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

      <service-activator
          input-channel="newLeadChannel"
          ref="leadService"
          method="createLead" />

      <transformer input-channel="newLeadChannel" output-channel="confEmailChannel">
          <beans:bean class="crm.integration.transformers.LeadToEmailTransformer">
              <beans:property name="confFrom" value="${conf.email.from}" />
              <beans:property name="confSubject" value="${conf.email.subject}" />
              <beans:property name="confText" value="${conf.email.text}" />
          </beans:bean>
      </transformer>

      <channel id="confEmailChannel" />

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

  </beans:beans>
```

The property-placeholder configuration loads the various ${...} properties from a properties file; see /crm/src/main/resources/applicationContext.properties in the code download. You don't have to change anything in the properties file.

The transformer configuration brings the LeadToEmailTransformer bean into the picture so it can transform Leads that appear on the newLeadChannel into MailMessages that it puts on the confEmailChannel. As a side note, the p namespace way of specifying bean properties doesn't seem to work here (I assume it's a bug: http://jira.springframework.org/browse/SPR-5990), so I just did it the more verbose way.

The channel definition defines a point-to-point channel rather than a pub-sub channel. That means that only one endpoint can pull messages from the channel.

Finally we have an outbound-channel-adapter that grabs MailMessages from the confEmailChannel and then sends them using the referenced mailSender, which we defined in listing 6.

That's it for this section. We should have working confirmation e-mails. Restart your Jetty instance and go again to
http://localhost:8080/crm/main/lead/form.html
Fill it out and provide your real e-mail address in the e-mail field. A few moments after submitting the form you should receive a confirmation e-mail. If you don't see it, you might check your SMTP configuration in jetty-env.xml, or else check your spam folder.

Summary

In this tutorial we've taken our first steps toward developing an integrated lead management system. Though the current bus configuration is simple, we've already seen some key Spring Integration features, including

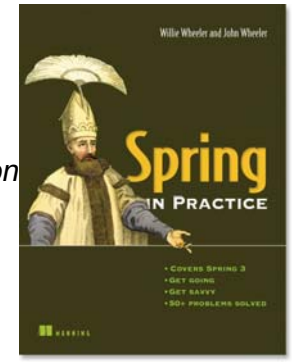
- support for the Gateway pattern, allowing us to connect apps to the message bus without knowing about messages
- point-to-point and pub-sub channels
- service activators to allow us to place service beans on the bus
- message transformers
- outbound SMTP channel adapters to allow us to send e-mail

The second tutorial [1] will continue elaborating what we've developed here, demonstrating the use of several additional Spring Integration features, including

- message routers (including content-based message routers)
- outbound web service gateways for sending SOAP messages
- inbound HTTP adapters for collecting HTML form data from external systems
- inbound e-mail channel adapters (we'll use IMAP IDLE, though POP and IMAP are also possible) for processing incoming e-mails

Enjoy, and stay tuned.

Willie is a solutions architect with 12 years of Java development experience. He and his brother John are coauthors of the upcoming book Spring in Practice by Manning Publications (www.manning.com/wheeler/ [3]). Willie also publishes technical articles (including many on Spring) to wheelersoftware.com/articles/ [6].



Attachment	Size
spring-integration-demo-3.zip [2]	18.84 KB
spring-integration-demo-4.zip [5]	21.61 KB

Source URL: http://java.dzone.com/articles/spring-integration-hands

Links:
[1] http://java.dzone.com/articles/spring-integration-hands-0
[2] http://java.dzone.com/sites/all/files/spring-integration-demo-3.zip
[3] http://www.manning.com/wheeler/
[4] http://mail.dzone.com/Redirect/www.manning.com/
[5] http://java.dzone.com/sites/all/files/spring-integration-demo-4.zip
[6] http://wheelersoftware.com/articles/