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

Spring Integration: A Hands-On Tutorial Part 2

By *wwheeler*

Created *2009/08/26 - 4:53am*

This is the second of a two-part series of tutorials on Spring Integration. The first tutorial [1] provided a high-level overview of Spring Integration along with a quick introduction to the lead management domain. It also showed how to take some initial steps into the world of Spring Integration, and we built a simple lead entry form with a confirmation e-mail. In this second installment we'll continue building the message bus we started in the first. Once again our example domain will be enrollment lead management in the context of an online university.

In part 1 new leads flowed to a service activator that called a local LeadService.createLead() service bean method, which then pretended to save the lead to a database. Let's revisit that.

Routing international leads to an external CRM

[The source code for this section is available here [2] [3]. The funny numbering scheme is due to the fact that this section was originally section 5 of a single tutorial.]

As just mentioned, the bus currently drops enrollment leads into the CRM's lead database. Let's say, however, that we actually have a couple of different teams involved in processing enrollment leads: one team for domestic (U.S.) leads and another for international leads. The international team already uses a separate legacy CRM that eventually we plan to phase out, but for now we need to use it. So the problem is to route domestic leads to the CRM that the domestic team uses, and international teams to the legacy CRM.

Figure 1 shows how we're going to use a router to solve this problem.

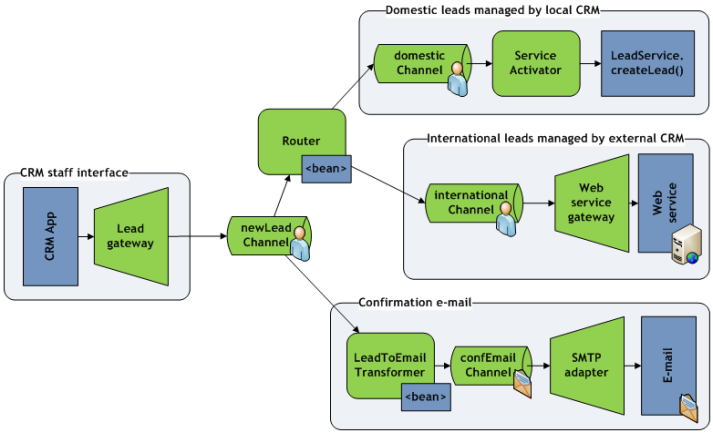


Figure 1. Send domestic leads to the local CRM and international leads to a different CRM.

To put all of this in place we need to take care of several things, including creating a dummy external CRM for international leads, updating the local CRM POM, updating the Lead class to support content-based routing and serialization to XML, and adding further configuration to applicationContext-integration.xml to build out the new parts of the pipeline.

We'll take care of the dummy external CRM first, since that's self-contained.

Create a dummy external CRM for international leads

We won't cover the details here, but in the sample code we've added a new Maven module called xcrm (for "External CRM") to the top-level POM. All it contains is are some configuration files and a dummy web service endpoint—implemented as a plain old servlet—to log the HTTP request payload to the console. That's our "CRM." It will allow us to see the SOAP message we'll be sending it shortly, which is of course the real point of our including it.

Now we need to make several updates to the local CRM. We'll start with the POM and go from there.

Update the CRM POM

There are several POM dependencies we need to add to crm/pom.xml, mostly surrounding XML and SOAP web services. See the code download for details. I've used a Maven profile to support Java 5. If you know for sure that you're using Java 5, you can simply add those dependencies to the dependencies section of the POM instead of adding them to a profile. Alternatively, if you know for sure that you're using Java 6, you can omit them completely, as they're included with the Java 6 SDK.

You'll also need to add the spring.ws.version, jaxb.version and saaj.version properties.

Now let's check out the Lead class. There are a couple of different things we'll need to do.

Elaborate the Lead class and annotate it for OXM

Listing 1 shows our updated Lead class. Please see the code download for the full version.

Listing 1. Annotating Lead.java to support object/XML mapping

```
package crm.model;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlTransient;
import javax.xml.bind.annotation.XmlType;

@XmlRootElement
@XmlType(propOrder = {
    "firstName", "lastName",
    "address1", "address2", "city", "stateOrProvince", "postalCode", "country",
    "homePhone", "workPhone", "mobilePhone", "email" })
public class Lead {

    ... various fields (firstName, etc.) ...

    public Lead() { }

    @XmlElement(name = "givenName")
    public String getFirstName() { return firstName; }

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

    @XmlTransient
```

```
    public String getMiddleInitial() { return middleInitial; }

    public void setMiddleInitial(String middleInitial) {
        this.middleInitial = middleInitial;
    }

    @XmlElement(name = "surname")
    public String getLastName() { return lastName; }

    public void setLastName(String lastName) { this.lastName = lastName; }

    public String getFullName() { return firstName + " " + lastName; }

    ... various getters and setters ...

    @XmlElement
    public String getCountry() { return country; }

    public void setCountry(String country) { this.country = country; }

    public boolean isInternational() {
        boolean unknown = (country == null);
        boolean domestic = ("US".equals(country));
        return !(unknown || domestic);
    }

    ... even more getters and setters ...
}
```

The first thing to discuss is the new isInternational() method. This returns a boolean indicating whether the lead is an international lead. If the country is null or else "US", it's not international; otherwise, it is. (The sample code includes a unit test for this method.) We're adding this method because we're going to create a content-based router in just a few moments. The router will query the lead's isInternational() method and make a routing decision based on that result.

The other change is that we've added several JAXB annotations to support object/XML mapping (OXM). This will allow us to generate a lead document that we can send to the external CRM as a SOAP message. We're not doing anything especially sophisticated here but it gets the basic point across. There's some Spring Integration configuration we'll need to do in order to activate the OXM, and we'll see that in section 1.5 below. Before that, though, let's create the content router.

Add lead routing

In figure 1 there's a router that sits between the newLeadChannel on the one hand and the two CRMs on the other. This router is backed by the POJO appearing in listing 2 below.

Listing 2. CountryRouter.java, a router POJO

```
package crm.integration.routers;

import java.util.logging.Logger;
import org.springframework.integration.annotation.Router;
import crm.model.Lead;

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

    @Router
    public String route(Lead lead) {
        String destination = (lead.isInternational() ?
                "internationalChannel" : "domesticChannel");
        log.info("Lead country is " +
                lead.getCountry() + "; routing to " + destination);
        return destination;
    }
```

```
}
```

This is a simple component, with a single route() method that we've annotated with a @Router annotation so that Spring Integration will know which method to use for routing. It should be pretty clear what's going on. As mentioned earlier, we're doing content-based routing here, meaning that we're using the actual message payload (i.e., the Lead instance) to drive the routing decision rather than using message headers. This ability to inspect message payloads is key as it allows us to move the system's "intelligence" into the bus itself instead of forcing clients to have to provide hints (e.g., in the form of message headers) that steer processing in the right direction. This reduces the burden on clients and makes the bus more generally usable.

The last step is to tie it all together in our message bus configuration.

Update the message bus configuration

Listing 3 presents some updates to our message bus.

Listing 3. /WEB-INF/applicationContext-integration.xml updates to support lead routing

```
<?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:ws="http://www.springframework.org/schema/integration/ws"
    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:util="http://www.springframework.org/schema/util"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
        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/integration/ws
        http://www.springframework.org/schema/integration/ws/spring-integration-ws-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/util
        http://www.springframework.org/schema/util/spring-util-2.5.xsd">

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

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

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

    <router input-channel="newLeadChannel">
        <beans:bean id="countryRouter"
            class="crm.integration.routers.CountryRouter" />
    </router>

    <channel id="domesticChannel" />

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

    <channel id="internationalChannel" />

    <util:list id="classesToBeBound">
        <beans:value>crm.model.Lead</beans:value>
    </util:list>
```

```
    <beans:bean id="marshaller"
        class="org.springframework.oxm.jaxb.Jaxb2Marshaller"
        p:classesToBeBound-ref="classesToBeBound" />

    <ws:outbound-gateway
        request-channel="internationalChannel"
        uri="${external.crm.uri}"
        marshaller="marshaller" />

    <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>
```

There are a few things going on here; so we'll take them in turn. It's helpful to look at figure 1 while looking at this configuration.

First, notice that we've added the util and ws namespace declarations and schema locations to the top-level beans element. The ws prefix identifies elements from Spring Integration's web services schema.

Next, we've inserted a router in between the newLeadChannel and the service activator that was previously connected to the newLeadChannel. We've defined the CountryRouter bean as an inner bean to keep its definition hidden from the rest of the context.

After the router, messages can go in one of two directions as we've already discussed. The domestic branch is really just what we had at the end of the previous tutorial. The international branch, however, is new. There's an internationalChannel that feeds into an outbound SOAP web service gateway. We have to give the gateway a marshaller (we've chosen a JAXB marshaller, though other options are available) so it can turn message payloads into SOAP messages, and that's what the marshaller configuration is all about.

You'll also need to add the following property to applicationContext.properties to specify the URI for the external CRM:

external.crm.uri=http://localhost:9091/xcrm/main/leads.xml

Once you've done all that, go ahead and start up both CRMs by running

mvn jetty:run

against the appropriate module's POM. After doing that, try creating some leads using the staff UI:

http://localhost:8080/crm/main/lead/form.html [6]

When you create a domestic lead it should go to the local CRM, and when you create an international lead it should go to the external CRM (where you'll see the SOAP message displayed on the command line). In either event the system will send a confirmation e-mail to the lead.

The next two sections will show how to integrate so-called requests for information (RFIs), which allow leads to (yep) request information. First we'll do a web-based RFI form, and after

that we'll do a legacy e-mail-based RFI.

Integrating web-based RFI forms

[The code for this section is available here [7]]

Request for information forms, or RFIs, are an important piece of the lead management domain. They allow prospective customers to find out more, and they result in a lead being created in the lead management system. Qualifying reps or else salespersons can then pull the lead out of the CRM and follow up, providing the requested information. In the context of our hypothetical online university, enrollment advisors might for example contact prospective students with information about academic programs, financial aid and so on.

Figure 2 shows how our web-based RFI fits into the overall scheme of things.

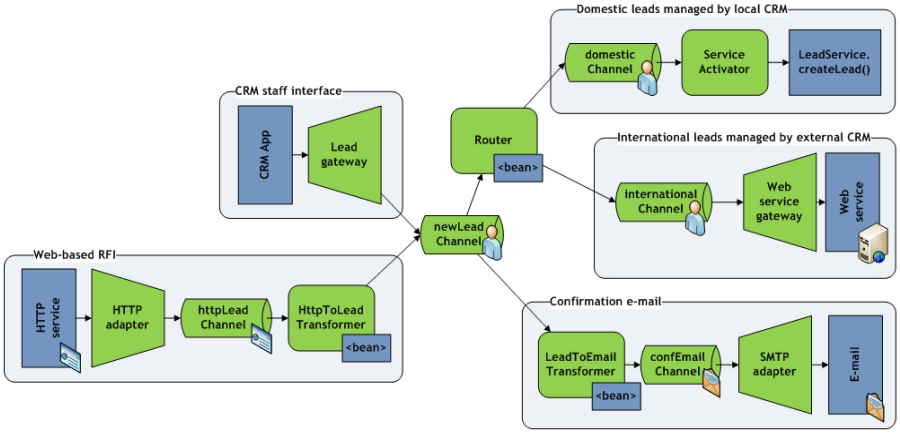


Figure 2: We're going to accept leads from web-based RFI forms.

Let's go through the various steps involved in adding the new web-based RFI to the overall integration.

Add the lead generation website

RFI forms live on customer-facing lead generation websites, so we're going to need a separate Maven module. See the code download for the leadgen module. Normally this would be an entire website that describes university events, academic programs, student and faculty profiles, accreditation information and that sort of thing. Here though we're interested in only the RFI form itself, so that's all our sample lead gen site has.

When the user submits the RFI, it goes to a web MVC controller (part of the lead gen site), which in turn passes it to a service bean (still part of the lead gen site). The service bean uses HttpClient to send an HTTP POST request through the firewall to the CRM, which would typically live on the internal network. So as this example shows, we can integrate systems in the DMZ with those on the internal network.

Once the HTTP request makes it onto the bus (and we'll show how that works shortly), we have to have a way to transform it into a Lead object that we can place on the newLeadChannel. The next section shows how to do that.

Transform the HTTP POST request parameters to a Lead

In the first tutorial we created a transformer bean to convert Leads into MailMessages so we could send prospective students a confirmation e-mail. Here we're going to create another transformer bean, this time to map the HTTP POST request to a Lead. Listing 4 shows how it's done.

We've already covered the mechanics of creating a transformer bean, so there's no need to

repeat that. In this case, the HTTP channel adapter (we'll see it momentarily) creates a Message with a Map of HTTP parameters as its payload, so that's why our @Transformer method takes a Map as an input.

The last step is once again to configure the message bus.

Configure the message bus to add the new pipeline

There's not much of a change to applicationContext-integration.xml, so instead of showing the whole configuration, listing 5 concentrates on the parts that change.

Listing 5. Updates to /WEB-INF/applicationContext-integration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/integration"
    xmlns:http="http://www.springframework.org/schema/integration/http"

    ... other namespace declarations ...

    xsi:schemaLocation="http://www.springframework.org/schema/integration
        http://www.springframework.org/schema/integration/spring-integration-1.0.xsd

        ... other schema locations ...

        http://www.springframework.org/schema/integration/http
        http://www.springframework.org/schema/integration/http/spring-integration-http-1.0.xsd">

    <beans:bean
        class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />

    <http:inbound-channel-adapter
        name="/leads"
        channel="httpLeadChannel"
        supported-methods="POST"/>

    <channel id="httpLeadChannel" />

    <transformer input-channel="httpLeadChannel" output-channel="newLeadChannel">
        <beans:bean class="crm.integration.transformers.HttpToLeadTransformer" />
    </transformer>

    ... other message bus configuration ...

</beans:beans>
```

We've added the new http namespace and schema location, and we're using it to declare an inbound HTTP channel adapter that handles requests coming to the /leads path, thanks to the BeanNameUrlHandlerMapping. (See Spring Web MVC if you're unfamiliar with how that works.) The full URI is

http://localhost:8080/crm/main/leads [8]

and that's the destination URI for the HTTP POST requests that the lead gen website's HttpClient sends. The inbound channel adapter listens for HTTP POST requests mapping to that URI and then pushes a Message containing the parameter map onto the httpLeadChannel. The bus takes over from there.

We have a transformer definition, but we've already seen one of those so we won't cover it again.

To try it out, start up your CRM and lead gen site, and direct your browser to

http://localhost:9090/leadgen/main/rfi/form.html [9]

The RFI form as it currently stands doesn't provide any way to specify the country, so the country will be null and RFI leads will all route through the domestic channel and end up in the

local CRM. So you don't need to start up the external CRM.

One more before we call it a day: an e-mail based RFI.

Integrating legacy e-mail-based RFIs

[The code for this section is available here [10] [11]]

The last piece of our integration will be a legacy channel for e-mail based RFIs. See Figure 3 below, which represents the final product.

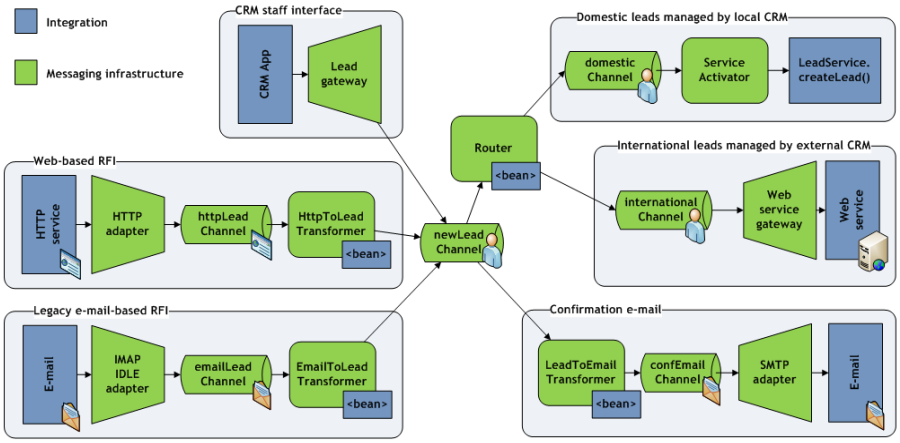


Figure 3: The final product: an integrated lead management module.

The approach here is to connect our message bus up to an IMAP mailbox, and treat the mailbox like a message queue. When people submit e-mails to the mailbox in question, our message bus will pick them up and process them.

WARNING: THIS WILL DELETE ALL MESSAGES IN YOUR MAILBOX! Do NOT use your personal e-mail account unless you don't mind having all of your messages deleted!

I learned that lesson the hard way, thinking that because it's an IMAP mailbox, the mail will stay on the server. That's the way IMAP works, but it's not IMAP eating up the e-mails. It's Spring Integration (and it's by design). Anyway, you've been warned!

Here are the changes we'll need to make.

Update Lead.java to support notes

First, because e-mail is mostly unstructured, we're going to need a way to capture whatever it is that the prospective student wants to say. So we'll add a notes property to Lead.java. Additionally, we'll add a method to extract the first and last names from the sender's full name. See listing 6 below.

Listing 6. Updates to Lead.java

```
package crm.model;

import java.util.List;
import org.springframework.util.StringUtils;

... various imports ...

... same annotations as before ...
public class Lead {
    private List<String> notes;
```

```
    ... various fields and methods ...

    @XmlTransient
    public List<String> getNotes() { return notes; }

    public void setNotes(List<String> notes) { this.notes = notes; }

    public void guessNamesFromFullName(String fullName) {
        if (fullName == null) { return; }
        String[] tokens = fullName.trim().split("\\s+");
        int len = tokens.length;
        if (len == 0) {
            return;
        } else if (len == 1) {
            if (!tokens[0].equals("")) { setFirstName(tokens[0]); }
        } else {
            StringBuilder builder = new StringBuilder();
            for (int i = 0; i < len - 1; i++) {
                builder.append(tokens[i] + " ");
            }
            setFirstName(builder.toString().trim());
            setLastName(tokens[len - 1]);
        }
    }

    ... minor toString() update (see code download) ...
}
```

here's enough logic in the guessNamesFromFullName() method to warrant a unit test, and indeed the sample code includes a unit test that covers this method. Basically it would break a name like "Willie Wheeler" down into "Willie" and "Wheeler".

We're going to need to transform e-mails into leads, and we'll look at that now.

Create EmailToLeadTransformer.java

Listing 7 contains the transformer bean for converting e-mails into leads.

Listing 7. EmailToLeadTransformer.java

```
package crm.integration.transformers;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import org.springframework.integration.annotation.Transformer;
import crm.model.Lead;

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

    @Transformer
    public Lead transform(MimeMessage email) {
        log.info("Transforming e-mail to lead");
        try {
            InternetAddress from = (InternetAddress) email.getFrom()[0];
            String fullName = from.getPersonal();
```

```
            Lead lead = new Lead();
            lead.guessNamesFromFullName(fullName);
            lead.setEmail(from.getAddress());
            lead.setDateCreated(email.getSentDate());

            StringBuilder builder = new StringBuilder("Full name: " + fullName);
            builder.append("\nSubject: " + email.getSubject());

            // FIXME Doesn't work with MimeMultipart. Output looks like this:
            // javax.mail.internet.MimeMultipart@598d00
            builder.append("\n\n" + email.getContent().toString());

            List<String> notes = new ArrayList<String>();
            notes.add(builder.toString());
            lead.setNotes(notes);

            log.info("Transformed e-mail to lead: " + lead);
            return lead;
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The code in listing 7 isn't anywhere near industrial-strength: it doesn't, for example, handle the common case of multipart e-mails. We can call that one an exercise left to the reader. (Ha ha.) But it does feature the basics of extracting the name, e-mail address, date, subject and yes, even the body if you're not sending a multipart e-mail.

The piece we really care about, though, is how to get the e-mail onto the message bus in the first place, and that's the topic of the next section.

3.3 Getting e-mail messages onto the message bus

There are at least three different options for getting e-mail messages onto the message bus:

- Poll a POP3 mailbox
- Poll an IMAP mailbox
- Receive e-mail pushes from an IMAP mailbox that supports the IMAP IDLE [12] feature (basically an e-mail push mechanism)

We're going to do the third one since it's easy to get an IMAP mailbox that supports IDLE (e.g., Gmail). To do that, we're going to need to update our configuration in a couple of ways. First, we'll need to create an imap.properties file with the relevant mailbox configuration. See the sample file at /crm/src/resources/imap.properties.sample for an example of that.

I'll repeat my previous warning here: the messages in the mailbox you choose will be deleted. Don't use your personal e-mail account or any other e-mail account where you don't want the whole thing to be wiped out.

OK. Listing 8 shows the updates we need to make to applicationContext-integration.xml.

Listing 8. Updates to /WEB-INF/applicationContext-integration.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans
    ... various namespace and schema location declarations ... >
```

```
    <!-- Note: This requires Spring Integration 1.0.3 or higher. -->
    <context:property-placeholder location="classpath:*.properties" />

    <!-- Use this if your server supports IMAP IDLE. Requires JavaMail 1.4.1
         or higher on the client side. -->
    <mail:imap-idle-channel-adapter
        channel="emailLeadChannel"
        store-uri="${email.store.uri}"
        should-delete-messages="true" />

    <!-- Use this if your server doesn't support IMAP IDLE. -->
    <!--
    <mail:inbound-channel-adapter
        channel="emailLeadChannel"
        store-uri="${email.store.uri}">
        <poller max-messages-per-poll="3">
            <interval-trigger interval="30" time-unit="SECONDS" />
        </poller>
    </mail:inbound-channel-adapter>
    -->

    <channel id="emailLeadChannel" />

    <transformer input-channel="emailLeadChannel" output-channel="newLeadChannel">
        <!-- Use inner bean instead of ref because no one else uses this bean -->
        <beans:bean class="crm.integration.transformers.EmailToLeadTransformer" />
    </transformer>

    ... other stuff ...

</beans:beans>
```

The change to the property placeholder piece is just that we now have two properties files to load instead of one. (Recall that we just added imap.properties.)

The more significant change is the addition of the IMAP IDLE channel adapter, which basically sets the message bus up to listen for e-mail notifications from the IMAP mailbox. I've included the should-delete-messages="true" attribute here just to emphasize its existence; it is however unnecessary since the default value is true. (As an aside, I'm not sure I like the default value being true. I understand that this needs to be true if we want to use the mailbox as a message queue, but man, it's a bummer to accidentally delete four or five years' worth of e-mail you've been saving. I might suggest that the Spring Integration team should remove the default value altogether and make this attribute required. Anyway.) If your mailbox doesn't support IDLE, you can use the alternative adapter configuration, which polls on an interval that you can set. And again, there's a POP3 adapter as well if you want that.

The rest of it is stuff we've already seen.

Once you have it all in, start up the CRM app, and send an e-mail to the IMAP mailbox you specified. You should receive a confirmation e-mail shortly thereafter.

Summary

This tutorial completes a two-part series on the basics of Spring Integration. In the first tutorial [1] we provided a bird's eye view of the framework, and then got started with some simple integration. In this second tutorial we completed the work we started. The end result is a fairly capable bus, and one that can be easily reconfigured.

The bus we build is really only the starting point. Besides the router, transformer, gateway, channel and channel adapter components we've seen here, there are several others we haven't covered, including:

- Filter: Decide whether to drop a message
- Splitter: Split a message into multiple messages

- Aggregator: Collapse multiple messages into a single message
- Resequencer: Orders a group of messages
- Delayer: Delays the propagation of messages
- Message Handler Chain: Provides a convenient way to sequence a series of endpoints
- Messaging Bridge: Connect two channels or channel adapters

In addition to the components above, there are several gateways and channel adapters that you might explore on your own, including those for

- File support
- JMS support
- Web services support (both inbound and outbound)
- RMI support
- HttpInvoker support
- Stream support
- ...and more

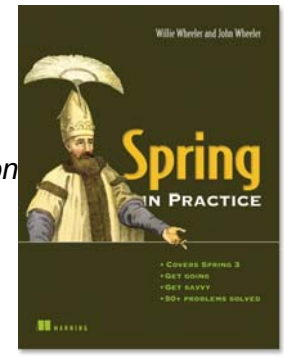
Needless to say, there's a good deal of capability here, and over time I would expect to see more patterns from *Enterprise Integration Patterns* [13] make their way into the framework.

Some other frameworks/platforms you might find interesting are:

- Apache Camel: http://camel.apache.org/ [14]
- Apache ServiceMix: http://servicemix.apache.org/home.html [15]
- Apache Synapse: http://synapse.apache.org/ [16]
- JBossESB: http://jboss.org/jbossesb/ [17]
- Mule: http://www.mulesource.org/display/MULE/Home [18]

Until next time, have fun!

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/ [4]). Willie also publishes technical articles (including many on Spring) to wheelersoftware.com/articles/ [19].



*Don't forget your voucher code, **dzone30** if you purchase the book.*

If you enjoyed this article, you'll like Part 1 [1] of the series as well as Willie's other article covering Spring Batch [20].

Attachment	Size
spring-integration-demo-5.zip [2]	29.93 KB
spring-integration-demo-6.zip [7]	42.65 KB
spring-integration-demo-7.zip [10]	44.91 KB

Source URL: http://java.dzone.com/articles/spring-integration-hands-0

Links:
[1] http://java.dzone.com/articles/spring-integration-hands
[2] http://java.dzone.com/sites/all/files/spring-integration-demo-5.zip
[3] http://willie.s3.amazonaws.com/articles/spring-integration/spring-integration-demo-5.zip
[4] http://www.manning.com/wheeler/
[5] http://mail.dzone.com/Redirect/www.manning.com/
[6] http://localhost:8080/crm/main/lead/form.html
[7] http://java.dzone.com/sites/all/files/spring-integration-demo-6.zip
[8] http://localhost:8080/crm/main/leads
[9] http://localhost:9090/leadgen/main/rfi/form.html
[10] http://java.dzone.com/sites/all/files/spring-integration-demo-7.zip
[11] http://willie.s3.amazonaws.com/articles/spring-integration/spring-integration-demo-7.zip
[12] http://en.wikipedia.org/wiki/IMAP_IDLE
[13] http://www.amazon.com/Enterprise-Integration-Patterns-Designing-Addison-Wesley/dp/0321200683
[14] http://camel.apache.org/
[15] http://servicemix.apache.org/home.html
[16] http://synapse.apache.org/
[17] http://jboss.org/jbossesb/
[18] http://www.mulesource.org/display/MULE/Home
[19] http://wheelersoftware.com/articles/
[20] http://java.dzone.com/articles/getting-started-spring-batch