SAMPLE CHAPTER

SPRING IN ACTION

Craig Walls
Ryan Breidenbach

MANNING

Spring in Action

by Craig Walls

and

Ryan Breidenbach

Sample Chapter 7

Copyright 2005 Manning Publications

brief contents

PART 1 SPRING ESSENTIALS 1

1 ■ A Spring jump start 3
2 ■ Wiring beans 42
3 ■ Creating aspects 91

PART 2 SPRING IN THE BUSINESS LAYER 131

4 ■ Hitting the database 133
5 ■ Managing transactions 173
6 ■ Remoting 207
7 ■ Accessing enterprise services 240

PART 3 SPRING IN THE WEB LAYER 267

8 ■ Building the web layer 269
9 ■ View layer alternatives 319
10 ■ Working with other web frameworks 346
11 ■ Securing Spring applications 367

Accessing enterprise services

This chapter covers

- Accessing JNDI resources
- Sending and formatting email
- Scheduling tasks
- Integrating with EJBs

There are several enterprise services that Spring doesn't support directly. Instead Spring relies on other APIs to provide the services, but then places them under an abstraction layer so that they're easier to use.

You've already seen a few of Spring's abstraction layers. In chapter 4, you saw how Spring abstracts JDBC and Hibernate. In addition to eliminating the need to write certain boilerplate code, these abstractions eliminated the need for you to catch checked exceptions.

In this chapter, we're going to take a whirlwind tour of the abstraction layers that Spring provides for several enterprise services, including Spring's support for

- Java Naming and Directory Interface (JNDI)
- E-mail
- Scheduling
- Java Message Service (JMS)

We'll begin by looking at Spring's support for JNDI, since this provides the basis for several of the other abstraction layers.

7.1 Retrieving objects from JNDI

JNDI affords Java applications a central repository to store application objects. For example, a typical J2EE application uses JNDI to store and retrieve such things as JDBC data sources and JTA transaction managers.

But why would you want to configure these objects in JNDI instead of in Spring? Certainly, you could configure a `DataSource` object in Spring's configuration file, but you may prefer to configure it in an application server to take advantage of the server's connection pooling. Likewise, if your transactional requirements demand JTA transaction support, you'll need to retrieve a JTA transaction manager from the application server's JNDI repository.

Spring's JNDI abstraction makes it possible to declare JNDI lookups in your application's configuration file. Then you can wire those objects into the properties of other beans as though the JNDI object were just another POJO. Let's take a look at how to use Spring's JNDI abstraction to simplify lookup of objects in JNDI.

7.1.1 Working with conventional JNDI

Looking up objects in JNDI can be a tedious chore. For example, suppose you need to retrieve a `javax.sql.DataSource` from JNDI. Using the conventional JNDI APIs, your might write some code that looks like this:

```
InitialContext ctx = null;
try {
  ctx = new InitialContext();
  DataSource ds =
      (DataSource)ctx.lookup("java:comp/env/jdbc/myDatasource");
} catch (NamingException ne) {
  // handle naming exception
  …
} finally {
  if(ctx != null) {
    try {
      ctx.close();
    } catch (NamingException ne) {}
  }
}
```

At first glance, this may not look like a big deal. But take a closer look. There are a few things about this code that make it a bit clumsy:

- You must create and close an initial context for no other reason than to look up a `DataSource`. This may not seem like a lot of extra code, but it is extra plumbing code that is not directly in line with the goals of your application code.
- You must catch or, at very least, rethrow a `javax.naming.NamingException`. If you choose to catch it, you must deal with it appropriately. If you choose to rethrow it, then the calling code will be forced to deal with it. Ultimately, someone somewhere will have to deal with the exception.
- You code is tightly coupled with a JNDI lookup. All your code needs is a `DataSource`. It doesn't matter whether or not it comes from JNDI. But if your code contains code like that shown earlier, you're stuck retrieving the `DataSource` from JNDI.
- Your code is tightly coupled with a specific JNDI name—in this case `java:comp/env/jdbc/myDatasource`. Sure, you could extract that name into a properties file, but then you'll have to add even more plumbing code to look up the JNDI name from the properties file.

The overall problem with the conventional approach to looking up objects in JNDI is that it is the antithesis of dependency injection. Instead of your code being given an object, your code must go get the object itself. This means that your code is doing stuff that isn't really its job. It also means that your code is unnecessarily coupled to JNDI.

Regardless, this doesn't change the fact that sometimes you need to be able to look up objects in JNDI. `DataSource`s are often configured in an application server, to take advantage of the application server's connection pooling, and then retrieved by the application code to access the database. How can you get all the benefits of JNDI along with all of the benefits of dependency injection?

7.1.2 Proxying JNDI objects

Spring's `JndiObjectFactoryBean` gives you the best of both worlds. It is a factory bean, which means that when it is wired into a property, it will actually create some other type of object that will wire into that property. In the case of `JndiObjectFactoryBean`, it will wire an object retrieved from JNDI.

To illustrate how this works, let's revisit an example from chapter 4 (section 4.1.2). There you used `JndiObjectFactoryBean` to retrieve a `DataSource` from JNDI:

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean"
    singleton="true">
  <property name="jndiName">
    <value>java:comp/env/jdbc/myDatasource</value>
  </property>
</bean>
```

The `jndiName` property specifies the name of the object in JNDI. Here the full JNDI name of `java:comp/env/jdbc/myDatasource` is specified. However, if the object is a Java resource, you may choose to leave off `java:comp/env/` to specify the name more concisely. For example, the following declaration of the `jndiName` property is equivalent to the previous declaration:

```
<property name="jndiName">
  <value>jdbc/myDatasource</value>
</property>
```

With the `dataSource` bean declared, you may now inject it into a `DataSource` property. For instance, you may use it to configure a Hibernate session factory as follows:

```
<bean id="sessionFactory" class="org.springframework.orm.
    ➥ hibernate.LocalSessionFactoryBean">
  <property name="dataSource">
```

```
    <ref bean="dataSource"/>
  </property>
…
</bean>
```

When Spring wires the `sessionFactory` bean, it will inject the `DataSource` object retrieved from JNDI into the session factory's `dataSource` property.

The great thing about using `JndiObjectFactoryBean` to look up an object in JNDI is that the only part of the code that knows that the `DataSource` is retrieved from JNDI is the XML declaration of the `dataSource` bean. The `sessionFactory` bean doesn't know (or care) where the `DataSource` came from. This means that if you decide that you would rather get your `DataSource` from a JDBC driver manager, all you need to do is redefine the `dataSource` bean to be a `DriverManagerDataSource`.

We'll see even more uses of JNDI later in this chapter. But first, let's switch gears a bit and look at another abstraction provided by the Spring framework—Spring's e-mail abstraction layer.

7.2 Sending e-mail

Suppose that the course director of Spring Training has asked you to send her a daily e-mail outlining all of the upcoming courses, including a seat count and how many students have enrolled in the course. She'd like this report to be e-mailed at 6:00 a.m. every day so that she can see it when she first gets to work. Using this report, she'll schedule additional offerings of popular courses and cancel courses that aren't filling up very quickly.

As laziness is a great attribute of any programmer,[1] you decide to automate the e-mail so that you don't have to pull together the report every day yourself.

The first thing to do is to write the code that sends the e-mail (you'll schedule it for daily delivery in section 7.3).

To get started, you'll need a mail sender, defined by Spring's `MailSender` interface. A mail sender is an abstraction around a specific mail implementation. This decouples the application code from the actual mail implementation being used. Spring comes with two implementations of this interface:

[1] The other two attributes of a programmer are impatience and hubris. See *Programming Perl, 3rd Edition*, by Larry Wall et al. (O'Reilly & Associates, 2000).

- `CosMailSenderImpl`—Simple implementation of an SMTP mail sender based on Jason Hunter's COS (`com.oreilly.servlet`) implementation from his *Java Servlet Programming b*ook (O'Rielly, 1998).
- `JavaMailSenderImpl`—A JavaMail API-based implementation of a mail sender. Allows for sending of MIME messages as well as non-SMTP mail (such as Lotus Notes).

Either `MailSender` implementation is sufficient for the purposes of sending the report to the course director. But we'll choose `JavaMailSenderImpl` since it is the more versatile of the two. You'll declare it in your Spring configuration file as follows:

```
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host">
    <value>mail.springtraining.com</value>
  </property>
</bean>
```

The `host` property specifies the host name of the mail server, in this case Spring Training's SMTP server. By default, the mail sender assumes that the port is listening on port 25 (the standard SMTP port), but if your SMTP server is listening on a different port, you can set it using the `port` property of `JavaMailSenderImpl`.

The `mailSender` declaration above explicitly names the mail server that will send the e-mails. However, if you have a `javax.mail.MailSession` in JNDI (perhaps placed there by your application server) you have the option to retrieve it from JNDI instead. Simply use `JndiObjectFactoryBean` (as described in section 7.1) to retrieve the mail session and then wire it into the `mailSession` property as follows:

```
<bean id="mailSession"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/mail/Session</value>
  </property>
</bean>

<bean id="mailSender"
    class="org.springrframework.mail.javamail.JavaMailSenderImpl">
  <property name="session"><ref bean="mailSession"/></property>
</bean>
```

Now that the mail sender is set up, it's ready to send e-mails. But you might want to declare a template e-mail message:

```
<bean id="enrollmentMailMessage"
    class="org.springframework.mail.SimpleMailMessage">
  <property name="to">
    <value>coursedirector@springtraining.com</value>
  </property>
  <property name="from">
    <value>system@springtraining.com</value>
  </property>
  <property name="subject">
    <value>Course enrollment report</value>
  </property>
</bean>
```

Declaring a template e-mail message is optional. You could also create a new instance of `SimpleMailMessage` each time you send the e-mail. But by declaring a template in the Spring configuration file, you won't hard-code the e-mail addresses or subject in Java code.

The next step is to add a `mailSender` property to `CourseServiceImpl` so that `CourseServiceImpl` can use it to send the e-mail. Likewise, if you declared an e-mail template you should add a `message` property that will hold the message template bean:

```
public class CourseServiceImpl implements CourseService {
…
  private MailSender mailSender;
  public void setMailSender(MailSender mailSender) {
    this.mailSender = mailSender;
  }

  private SimpleMailMessage mailMessage;
  public void setMailMessage(SimpleMailMessage mailMessage) {
    this.mailMessage = mailMessage;
  }
…
}
```

Now that `CourseServiceImpl` has a `MailSender` and a copy of the e-mail template, you can write the `sendCourseEnrollementReport()` method (listing 7.1) that sends the e-mail to the course director. (Don't forget to add a declaration of `sendCourse-EnrollmentReport()` to the `CourseService` interface.)

Listing 7.1 Sending the enrollment report e-mail

```
public void sendCourseEnrollmentReport() {
  Set courseList = courseDao.findAll();

  SimpleMailMessage message =
      new SimpleMailMessage(this.mailMessage);      Copy mail
                                                    template
```

```
    StringBuffer messageText = new StringBuffer();
    messageText.append(
        "Current enrollment data is as follows:\n\n");

    for(Iterator iter = courseList.iterator(); iter.hasNext(); ) {
      Course course = (Course) iter.next();
      messageText.append(course.getId() + "    ");
      messageText.append(course.getName() + "    ");
      int enrollment = courseDao.getEnrollment(course);
      messageText.append(enrollment);
    }

    message.setText(messageText.toString());    <-- Set mail text

    try {
      mailSender.send(message);    <-- Send e-mail
    } catch (MailException e) {
      LOGGER.error(e.getMessage());
    }
  }
```

The `sendCourseEnrollmentReport()` starts by retrieving all courses using the `CourseDao`. Then, it creates a working copy of the e-mail template so that the original will remain untouched. It then constructs the message body and sets the message text. Finally, the e-mail is sent using the `mailSender` property.

The final step is to wire the `mailSender` and `enrollmentMailMessage` beans into the `courseService` bean:

```
<bean id="courseService"
    class="com.springinaction.training.service.CourseServiceImpl">
…
  <property name="mailMessage">
    <ref bean="enrollmentMailMessage"/>
  </property>

  <property name="mailSender">
    <ref bean="mailSender"/>
  </property>
</bean>
```

Now that the `courseService` bean has everything it needs to send the enrollment report, the job is half done. Now the only thing left is to set it up on a schedule to send to the course director on a daily basis. Gee, it would be great if Spring had a way to help us schedule tasks...

7.3 Scheduling tasks

Not everything that happens in an application is the result of a user action. Sometimes the software itself initiates an action.

The enrollment report e-mail, for example, should be sent to the course director every day. To make this happen, you have two choices: You can either come in early every morning to e-mail the report manually or you can have the application perform the e-mail on a predefined schedule. (We think we know which one you would choose.)

Two popular scheduling APIs are Java's `Timer` class and OpenSymphony's Quartz scheduler.[2] Spring provides an abstraction layer for both of these schedulers to make working with them much easier. Let's look at both abstractions, starting with the simpler one, Java's `Timer`.

7.3.1 Scheduling with Java's Timer

Starting with Java 1.3, the Java SDK has included rudimentary scheduling functionality through its `java.util.Timer` class. This class lets you schedule a task (defined by a subclass `java.util.TimerTask`) to occur every so often.

Creating a timer task

The first step in scheduling the enrollment report e-mail using Java's `Timer` is to create the e-mail task by subclassing `java.util.TimerTask`, as shown in listing 7.2.

Listing 7.2 A timer task to e-mail the enrollment report

```
public class EmailReportTask extends TimerTask {
  public EmailReportTask() {}

  public void run() {
    courseService.sendCourseEnrollmentReport();    <- Send the report
  }

  private CourseService courseService;
  public void setCourseService(CourseService courseService) {
    this.courseService = courseService;                  Inject the
  }                                                      CourseService
}
```

[2] Quartz is an open source job scheduling system from the OpenSymphony project. You can learn more about Quartz at http://www.opensymphony.com/quartz/.

The `run()` method defines what to do when the task is run. In this case, it calls the `sendCourseEnrollmentReport()` of the `CourseService` (see listing 7.1) to send the enrollment e-mail. As for the `CourseService`, it will be supplied to `EmailReportTask` via dependency injection.

Declare the `EmailReportTask` in the Spring configuration file like this:

```
<bean id="reportTimerTask"
    class="com.springinaction.training.schedule.EmailReportTask">
  <property name="courseService">
    <ref bean="courseService"/>
  </property>
</bean>
```

By itself, this declaration simply places the `EmailReportTask` into the application context and wires the `courseService` bean into the `courseService` property. It won't do anything useful until you schedule it.

Scheduling the timer task

Spring's `ScheduledTimerTask` defines how often a timer task is to be run. Since the course director wants the enrollment report e-mailed to her every day, a `ScheduledTimerTask` should be wired as follows:

```
<bean id="scheduledReportTask"
    class="org.springframework.scheduling.timer.ScheduledTimerTask">
  <property name="timerTask">
    <ref bean="reportTimerTask"/>
  </property>
  <property name="period">
    <value>86400000</value>
  </property>
</bean>
```

The `timerTask` property tells the `ScheduledTimerTask` which `TimerTask` to run. Here it is wired with a reference to the `reportTimerTask` bean, which is the `EmailReportTask`. The `period` property is what tells the `ScheduledTimerTask` how often the `TimerTask`'s `run()` method should be called. This property, specified in milliseconds, is set to `86400000` to indicate that the task should be kicked off every 24 hours.

Starting the timer

The final step is to start the timer. Spring's `TimerFactoryBean` is responsible for starting timer tasks. Declare it in the Spring configuration file like this:

```
<bean class="org.springframework.scheduling.timer.TimerFactoryBean">
  <property name="scheduledTimerTasks">
```

```
      <list>
        <ref bean="scheduledReportTask"/>
      </list>
    </property>
  </bean>
```

The `scheduledTimerTasks` property takes an array of timer tasks that it should start. Since you only have one timer task right now, the list contains a single reference to the `scheduledReportTask` bean.

Unfortunately, even though the task will be run every 24 hours, there is no way to specify what time of the day it should be run. `ScheduledTimerTask` does have a `delay` property that lets you specify how long to wait before the task is first run. For example, to delay the first run of `EmailReportTask` by an hour:

```
  <bean id="scheduledReportTask"
      class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <property name="timerTask">
      <ref bean="reportTimerTask"/>
    </property>
    <property name="period">
      <value>86400000</value>
    </property>
    <property name="delay">
      <value>3600000</value>
    </property>
  </bean>
```

Even with the delay, however, the time that the `EmailReportTask` will run will be relative to when the application starts. How can you have it sent at 6:00 a.m. every morning as requested by the course director (aside from starting the application at 5:00 a.m.)?

Unfortunately, that's a limitation of using Java's `Timer`. You can specify how often a task runs, but you can't specify exactly when it will be run. In order to specify precisely when the e-mail is sent, you'll need to use the Quartz scheduler instead.

7.3.2 Using the Quartz scheduler

The Quartz scheduler provides richer support for scheduling jobs. Just as with Java's `Timer`, you can use Quartz to run a job every so many milliseconds. But Quartz goes beyond Java's `Timer` by enabling you to schedule a job to run at a particular time and/or day.

For more information about Quartz, visit the Quartz home page at http://www.opensymphony.com/quartz.

Let's start working with Quartz by defining a job that sends the report e-mail.

Creating a job

The first step in defining a Quartz job is to create the class that defines the job. For that, you'll subclass Spring's `QuartzJobBean`, as shown in listing 7.3.

Listing 7.3 Defining a Quartz job

```
public class EmailReportJob extends QuartzJobBean {

  public EmailReportJob() {}

  protected void executeInternal(JobExecutionContext context)
      throws JobExecutionException {

    courseService.sendCourseEnrollmentReport();   <-- Send enrollment report
  }

  private CourseService courseService;
  public void setCourseService(CourseService courseService) {
    this.courseService = courseService;                 Inject
  }                                              CourseService
}
```

A `QuartzJobBean` is the Quartz equivalent of a Java `TimerTask`. It is an implementation of the `org.quartz.Job` interface. The `executeInternal()` method defines the actions that the job does when its time comes. Here, just as with `EmailReportTask`, you simply call the `sendCourseEnrollmentReport()` method on the `courseService` property.

Declare the job in the Spring configuration file as follows:

```
<bean id="reportJob"
    class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass">
    <value>com.springinaction.training.
        schedule.EmailReportJob</value>
  </property>
  <property name="jobDataAsMap">
    <map>
      <entry key="courseService">
        <ref bean="courseService"/>
      </entry>
    </map>
  </property>
</bean>
```

Notice that you don't declare an `EmailReportJob` bean directly. Instead you declare a `JobDetailBean`. This is an idiosyncrasy of working with Quartz.

JobDetailBean is a subclass of Quartz's org.quartz.JobDetail, which requires that the Job object be set through the jobClass property.

Another quirk of working with Quartz's JobDetail is that the courseService property of EmailReportJob is set indirectly. JobDetail's jobDataAsMap takes a java.util.Map that contains properties that are to be set on the jobClass. Here, the map contains a reference to the courseService bean with a key of courseService. When the JobDetailBean is instantiated, it will inject the courseService bean into the courseService property of EmailReportJob.

Scheduling the job

Now that the job is defined, you'll need to schedule the job. Quartz's org.quartz.Trigger class decides when and how often a Quartz job should run. Spring comes with two triggers, SimpleTriggerBean and CronTriggerBean. Which trigger should you use? Let's take a look at both of them, starting with SimpleTriggerBean.

SimpleTriggerBean is similar to ScheduledTimerTask. Using it, you can specify how often a job should run and (optionally) how long to wait before running the job for the first time. For example, to schedule the report job to run every 24 hours, with the first run starting after one hour, declare it as follows:

```
<bean id="simpleReportTrigger"
    class="org.springframework.scheduling.quartz.SimpleTriggerBean">
  <property name="jobDetail">
    <ref bean="reportJob"/>
  </property>
  <property name="startDelay">
    <value>3600000</value>
  </property>
  <property name="repeatInterval">
    <value>86400000</value>
  </property>
</bean>
```

The jobDetail property is wired to the job that is to be scheduled, here the reportJob bean. The repeatInterval property tells the trigger how often to run the job (in milliseconds). Here, we've set it to 86400000 so that it gets triggered every 24 hours. And the startDelay property can be used (optionally) to delay the first run of the job. We've set it to 3600000 so that it waits an hour before firing off for the first time.

Scheduling a `cron` job

Although you can probably think of many applications for which SimpleTriggerBean is perfectly suitable, it isn't sufficient for e-mailing the enrollment report. Just as with ScheduledTimerTask, you can only specify how often the job is run—not exactly when it is run. Therefore, you can't use SimpleTriggerBean to send the enrollment report to the course directory at 6:00 a.m. every day.

CronTriggerBean, however, gives you more precise control over when your job is run. If you're familiar with the Unix cron tool, then you'll feel right at home with CronTriggerBean. Instead of declaring how often a job is run you get to specify exact times (and days) for the job to run. For example, to run the report job every day at 6:00 a.m., declare a CronTriggerBean as follows:

```
<bean id="cronReportTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail">
    <ref bean="reportJob"/>
  </property>
  <property name="cronExpression">
    <value>0 0 6 * * ?</value>
  </property>
</bean>
```

As with SimpleTriggerBean, the jobDetail property tells the trigger which job to schedule. Again, we've wired it with a reference to the reportJob bean. The cronExpression property tells the trigger when to fire. If you're not familiar with cron, this property may seem a bit cryptic, so let's examine this property a bit closer.

A cron expression has at least 6 (and optionally 7) time elements, separated by spaces. In order from left to right, the elements are defined as follows:

1. Seconds (0–59)
2. Minutes (0–59)
3. Hours (0–23)
4. Day of month (1–31)
5. Month (1–12 or JAN–DEC)
6. Day of week (1–7 or SUN–SAT)
7. Year (1970–2099)

Each of these elements can be specified with an explicit value (e.g., 6), a range (e.g., 9–12), a list (e.g., 9,11,13), or a wildcard (e.g., *). The day of the month and day of the week elements are mutually exclusive, so you should also indicate which one of these fields you don't want to set by specifying it with a question mark (?). Table 7.1 shows some example cron expressions and what they mean.

Table 7.1 Some sample `cron` expressions

Expression	What it means
0 0 10,14,16 * * ?	Every day at 10 a.m., 2 p.m., and 4 p.m.
0 0,15,30,45 * 1–10 * ?	Every 15 minutes on the first 10 days of every month
30 0 0 1 1 ? 2012	30 seconds after midnight on January 1, 2012
0 0 8-5 ? * MON–FRI	Every working hour of every business day

In the case of `cronReportTrigger`, we've set `cronExpression` to `0 0 6 * * ?` You can read this as "at the zero second of the zero minute of the sixth hour on any day of the month of any month (regardless of the day of the week), fire the trigger." In other words, the trigger is fired at 6:00 a.m. every day.

Using `CronTriggerBean`, you are able to adequately meet the course director's expectations. Now all that's left is to start the job.

Starting the job

Spring's `SchedulerFactoryBean` is the Quartz equivalent to `TimerFactoryBean`. Declare it in the Spring configuration file as follows:

```
<bean class="org.springframework.scheduling.
        ➥ quartz.SchedulerFactoryBean">
  <property name="triggers">
    <list>
      <ref bean="cronReportTrigger"/>
    </list>
  </property>
</bean>
```

The `triggers` property takes an array of triggers. Since you only have a single trigger at this time, you simply need to wire it with a list containing a single reference to the `cronReportTrigger` bean.

At this point, you've satisfied the requirements for scheduling the enrollment report e-mail. But in doing so, you've done a bit of extra work. Before we move on, let's take a look at a slightly easier way to schedule the report e-mail.

7.3.3 Invoking methods on a schedule

In order to schedule the report e-mail you had to write the `EmailReportJob` bean (or the `EmailReportTask` bean in the case of timer tasks). But this bean does little more than make a simple call to the `sendCourseEnrollmentReport()` method of `CourseService`. In this light, `EmailReportTask` and `EmailReportJob` both seem a bit superfluous. Wouldn't it be great if you could specify that the `sendCourse-EnrollmentReport()` method be called without writing the extra class?

Good news! You can schedule single method calls without writing a separate TimerTask or QuartzJobBean class. To accomplish this, Spring has provided MethodInvokingTimerTaskFactoryBean and MethodInvokingJobDetailFactoryBean to schedule method calls with Java's timer support and the Quartz scheduler, respectively.

For example, to schedule a call to sendCourseEnrollmentReport() using Java's timer service, re-declare the scheduledReportTask bean as follows:

```
<bean id="scheduledReportTask">
    class="org.springframework.scheduling.timer.
          ➥ MethodInvokingTimerTaskFactoryBean">
  <property name="targetObject">
    <ref bean="courseService"/>
  </property>
  <property name="targetMethod">
    <value>sendCourseEnrollmentReport</value>
  </property>
</bean>
```

Behind the scenes, MethodInvokingTimerTaskFactoryBean creates a TimerTask that calls the method specified by the targetMethod property on the object specified by the targetObject property. This is effectively the same as the EmailReportTask.

With scheduledReportTask declared this way, you can now eliminate the EmailReportTask class and its declaration in the reportTimerTask bean.

MethodInvokingTimerTaskFactoryBean is good for making simple one-method calls when you are using a ScheduledTimerTask. But you're using Quartz's CronTriggerBean so that the report will be sent every morning at 6:00 a.m. So instead of using MethodInvokingTimerTaskFactoryBean, you'll want to re-declare the reportJob bean as follows:

```
<bean id="courseServiceInvokingJobDetail">
    class="org.springframework.scheduling.quartz.
          MethodInvokingJobDetailFactoryBean">
  <property name="targetObject">
    <ref bean="courseService"/>
  </property>
  <property name="targetMethod">
    <value>sendCourseEnrollmentReport</value>
  </property>
</bean>
```

MethodInvokingJobDetailFactoryBean is the Quartz equivalent of MethodInvokingTimerTaskFactoryBean. Under the covers, it creates a Quartz JobDetail object that makes a single method call to the object and method specified in the

`targetObject` and `targetMethod` properties. Using `MethodInvokingJobDetail-FactoryBean` this way, you can eliminate the superfluous `EmailReportJob` class.

7.4 Sending messages with JMS

Most operations that take place in software are performed synchronously. In other words, when a routine is called the program flow is handed off to that routine to perform its functionality. Upon completion, control is returned to the calling routine and the program proceeds. Figure 7.1 illustrates this.

But sometimes, it's not necessary (or even desirable) to wait for the called routine to complete. For example, if the routine is slow, it may be preferable to send a message to a routine and then just assume that the routine will process the message or to check on its progress sometime later.

When you send a message to a routine and do not wait for a result, it is said to be *asynchronous*. Asynchronous program flow is illustrated in figure 7.2.

The Java Messaging Service (JMS) is a Java API for asynchronous processing. JMS supports two types of messaging: point-to-point and publish-subscribe.

A point-to-point message is placed into a message *queue* by the message producer and later pulled off the queue by the message consumer. Once the message is pulled from the queue, it is no longer available to any other message consumer that is watching the queue. This means that even though several consumers may observe a queue, a single consumer will consume each point-to-point message.

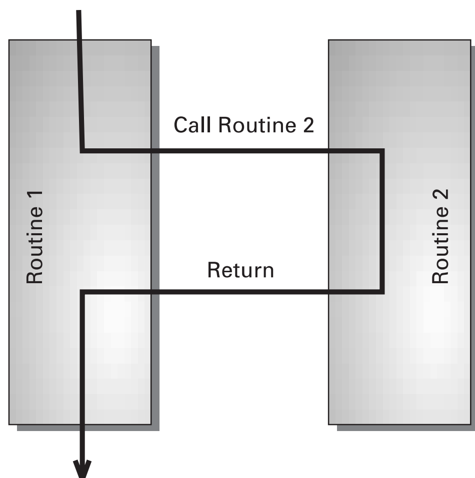


Figure 7.1 Synchronous program flow

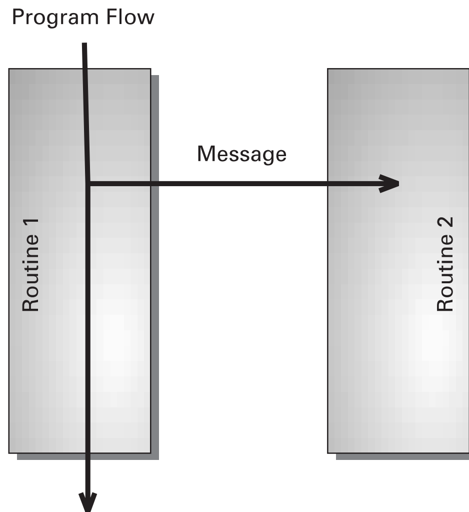


Figure 7.2 Asynchronous program flow

The publish-subscribe model invokes images of a magazine publisher who sends out copies of its publication to multiple subscribers. This is, in fact, a good analogy of how publish-subscribe messaging works. Multiple consumers subscribe to a message *topic*. When a message producer publishes a message to the topic, all subscribers will receive the message and have an opportunity to process it.

Spring provides an abstraction for JMS that makes it simple to access a message queue or topic (abstractly referred to as a *destination*) and publish messages to the destination. Moreover, Spring frees your application from dealing with `javax.jms.JMSException` by rethrowing any JMS exceptions as unchecked `org.springframework.jms.JmsExceptions`.

Let's see how to apply Spring's JMS abstraction.

7.4.1 Sending messages with JMS templates

In chapter 6 you learned to use Spring's remoting support to perform credit card authorization against the Spring Training payment service. Now you're ready to settle the account and receive payment.

When authorizing payment, it was necessary to wait for a response from the credit card processor, because you needed to know whether or not the credit card's issuing bank would authorize payment. But now that authorization has been granted, payment settlement can be performed asynchronously. There's no need to wait for a response—you can safely assume that the payment will be settled.

The credit card processing system accepts an asynchronous message, sent via JMS, for the purposes of payment settlement. The message it accepts is a `javax.jms.MapMessage` containing the following fields:

- `authCode`—The authorization code received from the credit card processor
- `creditCardNumber`—The credit card number
- `customerName`—The card holder's name
- `expirationMonth`—The month that the credit card expires
- `expirationYear`—The year that the credit card expires

Spring employs a callback mechanism to coordinate JMS messaging. This callback is reminiscent of the JDBC callback described in chapter 4. The callback is made up of two parts: a message creator that constructs a JMS message (`javax.jms.Message`) and a JMS template that actually sends the message.

Using the template

The first thing to do is to equip the `PaymentServiceImpl` class with a `JmsTemplate` property:

```
private JmsTemplate jmsTemplate;
public void setJmsTemplate(JmsTemplate jmsTemplate) {
  this.jmsTemplate = jmsTemplate;
}
```

The `jmsTemplate` property will be wired with an instance of `org.springframework.jms.core.JmsTemplate` using setter injection. We'll show you how to wire this a little bit later. First, however, let's implement the service-level method that sends the settlement message.

`PaymentServiceImpl` will need a `sendSettlementMessage()` method to send the settlement message to the credit card processor. Listing 7.4 shows how `sendSettlementMessage()` uses the `JmsTemplate` to send the message. (The `PaySettlement` argument is a simple JavaBean containing the fields needed for the message.)

Listing 7.4 Sending a payment settlement via the JMS callback

```
public void sendSettlementMessage(final PaySettlement settlement) {
  jmsTemplate.send(    <— Send message

      new MessageCreator() {    <— Define message creator
        public Message createMessage(Session session)
            throws JMSException {
```

```
            MapMessage message = session.createMapMessage();
            message.setString("authCode",
                settlement.getAuthCode());
            message.setString("customerName",
                settlement.getCustomerName());
            message.setString("creditCardNumber",
                settlement.getCreditCardNumber());
            message.setInt("expirationMonth",
                settlement.getExpirationMonth());
            message.setInt("expirationYear",
                settlement.getExpirationYear());

            return message;
          }
        }
    );
  }
```

Construct message

The sendSettlementMessage() method uses the JmsTemplate's send() method to send the message. This method takes an instance of org.springframework.jms.core.MessageCreator, here defined as an anonymous inner class, which constructs the Message to be sent. In this case, the message is a javax.jms.MapMessage. To construct the message, the MessageCreator retrieves values from the PaySettlement bean's properties and uses them to set fields on the MapMessage.

Wiring the template

Now you must wire a JmsTemplate into the PaymentServiceImpl. The following XML from the Spring configuration file will do just that:

```
<bean id="paymentService"
    class="com.springinaction.training.service.PaymentServiceImpl">
…
  <property name="jmsTemplate">
    <ref bean="jmsTemplate"/>
  </property>
<bean>
```

The declaration of the jmsTemplate bean is as follows:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory">
    <ref bean="jmsConnectionFactory"/>
  </property>
  <property name="defaultDestination">
    <ref bean="destination"/>
```

```
    </property>
  </bean>
```

Notice that the `jmsTemplate` bean is wired with a JMS connection factory and a default destination. The `connectionFactory` property is mandatory because it is how `JmsTemplate` gets a connection to a JMS provider. In the case of the Spring Training application, the connection factory is retrieved from JNDI, as shown in the following declaration of the `connectionFactory` bean:

```
<bean id="jmsConnectionFactory"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>connectionFactory</value>
  </property>
</bean>
```

Wired this way, Spring will use `JndiObjectFactoryBean` (see section 7.1) to look up the connection factory in JNDI using the name `java:comp/env/connectionFactory`. (Of course, this assumes that you have a JMS implementation with an instance of `JMSConnectionFactory` registered in JNDI.)

The `defaultDestination` property defines the default JMS destination (an instance of `javax.jms.Destination`) that the message will be published to. Here it is wired with a reference to the `destination` bean. Just as with the `connectionFactory` bean, the `destination` bean will be retrieved from JNDI using a `JndiObjectFactoryBean`:

```
<bean id="destination"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>creditCardQueue</value>
  </property>
</bean>
```

The `defaultDestination` property is optional. But because there's only one JMS destination for credit card messages, it is set here for convenience. If you do not set a default destination, then you must pass a `Destination` instance or the JNDI name of a `Destination` when you call `JmsTemplate`'s `send()` method. For example, you'd use this to specify the JNDI name of the JMS destination in the call to `send()`:

```
jmsTemplate.send(
    "creditCardQueue", new MessageCreator() { … });
```

Working with JMS 1.0.2

Until now, the jmsTemplate bean has been declared to be an instance of JmsTemplate. Although it isn't very apparent, this implies that the JMS provider implementation adheres to version 1.1 of the JMS specification. If your JMS provider is 1.0.2-compliant and not 1.1-compliant, then you'll want to use JmsTemplate102 instead of JmsTemplate.

The big difference between JmsTemplate and JmsTemplate102 is that JmsTemplate102 needs to know whether you're using point-to-point or publish-subscribe messaging. By default, JmsTemplate102 assumes that you'll be using point-to-point messaging, but you can specify publish-subscribe by setting the pubSubDomain property to true:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
…
  <property name="pubSubDomain">
    <value>true</value>
  </property>
</bean>
```

Other than that, you use JmsTemplate102 the same as you would JmsTemplate.

Handling JMS exceptions

An important thing to notice about using JmsTemplate is that you weren't forced to catch a javax.jms.JMSException. Many of JmsTemplate's methods (including send()) catch any JMSException that is thrown and converts it to an unchecked runtime org.springframework.jms.JmsException.

7.4.2 Consuming messages

Now suppose that you are writing the code for the receiving end of the settlement process. You're going to need to receive the message, convert it to a PaySettlement object, and then pass it on to be processed. Fortunately, JmsTemplate can be used for receiving messages as well as sending messages.

Listing 7.5 demonstrates how you might use JmsTemplate to receive a settlement message.

Listing 7.5 Receiving a PaySettlement message

```
public PaySettlement processSettlementMessages() {
  Message msg = jmsTemplate.receive("creditCardQueue");    <- Receive message
```

```
    try {
      MapMessage mapMessage = (MapMessage) msg;
      PaySettlement paySettlement = new PaySettlement();

      paySettlement.setAuthCode(mapMessage.getString("authCode"));
      paySettlement.setCreditCardNumber(
          mapMessage.getString("creditCardNumber"));
      paySettlement.setCustomerName(
          mapMessage.getString("customerName"));
      paySettlement.setExpirationMonth(
          mapMessage.getInt("expirationMonth"));
      paySettlement.setExpirationYear(
          mapMessage.getInt("expirationYear"));

      return paySettlement;
    } catch (JMSException e) {
      throw JmsUtils.convertJmsAccessException(e);
    }
  }
```

Map message to PaySettlement

The `receive()` method of `JmsTemplate` attempts to receive a `Message` from the specified `Destination`. As used earlier, `receive()` will try to receive a message from the `Destination` that has a JNDI name of `creditCardQueue`.

Once the `Message` is received, it is cast to a `MapMessage` and a `PaySettlement` object is initialized with the values from the fields of the `MapMessage`.

By default, `receive()` will wait indefinitely for the message. However, it may not be desirable to have your application block while it waits to receive a message. It'd be nice if you could set a timeout period so that `receive()` will give up after a certain time.

Fortunately, you can specify a timeout by setting the `receiveTimeout` property on the `jmsTemplate` bean. For example:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
  <property name="receiveTimeout">
    <value>10000</value>
  </property>
</bean>
```

The `receiveTimeout` property takes a value that is the number of milliseconds to wait for a message. Setting it to `10000` specifies that the `receive()` method should give up after 10 seconds. If no message is received in 10 seconds, the `JmsTemplate` will throw an unchecked `JmsException` (which you may choose to catch or ignore).

7.4.3 Converting messages

In listing 7.4, the `MessageCreator` instance was responsible for mapping the properties of `PaySettlement` to fields in a `MapMessage`. The `processSettlement()` message in listing 7.5 performs the reverse mapping of a `Message` to a `PaySettlement` object. That'll work fine, but it does result in a lot of mapping code that may end up being repeated every time you need to send or receive a `PaySettlement` message.

To avoid repetition and to keep the send and receive code clean, it may be desirable to extract the mapping code to a separate utility object.

Converting PaySettlement messages

Although you could write your own utility object to handle message conversion, Spring's `org.springframework.jms.support.converter.MessageConverter` interface defines a common mechanism for converting objects to and from JMS `Messages`.

To illustrate, `PaySettlementConverter` (listing 7.6) implements `MessageConverter` to accommodate the conversion of `PaySettlement` objects to and from JMS `Message` objects.

Listing 7.6 Convert a `PaySettlement` to and from a JMS `Message`

```
public class PaySettlementConverter implements MessageConverter {
  public PaySettlementConverter() {}

  public Object fromMessage(Message message)
      throws MessageConversionException {
    MapMessage mapMessage = (MapMessage) message;
    PaySettlement settlement = new PaySettlement();

    try {
      settlement.setAuthCode(mapMessage.getString("authCode"));
      settlement.setCreditCardNumber(
          mapMessage.getString("creditCardNumber"));
      settlement.setCustomerName(
          mapMessage.getString("customerName"));
      settlement.setExpirationMonth(
          mapMessage.getInt("expirationMonth"));
      settlement.setExpirationYear(
          mapMessage.getInt("expirationYear"));
    } catch (JMSException e) {
      throw new MessageConversionException(e.getMessage());
    }

    return settlement;
  }
```

Convert Message to PaySettlement

Rethrow as runtime exception

```
    public Message toMessage(Object object, Session session)
        throws JMSException, MessageConversionException {

      PaySettlement settlement = (PaySettlement) object;
      MapMessage message = session.createMapMessage();
      message.setString("authCode", settlement.getAuthCode());
      message.setString("customerName",
          settlement.getCustomerName());
      message.setString("creditCardNumber",
          settlement.getCreditCardNumber());
      message.setInt("expirationMonth",
          settlement.getExpirationMonth());
      message.setInt("expirationYear",
          settlement.getExpirationYear());

      return message;
    }
  }
```

Convert PaySettlement to Message

As its name implies, the `fromMessage()` method is intended to take a `Message` object and convert it to some other object. In this case, the `Message` is converted to a `PaySettlement` object by pulling the fields out of the `MapMessage` and setting properties on the `PaySettlement` object.

The conversion is performed in reverse by the `toMessage()` method. This method takes an `Object` (in this case, assumed to be a `PaySettlement` bean) and sets elements in the `MapMessage` from the properties of the `Object`.

Wiring a message converter

To use the message converter, you first must declare it as a bean in the Spring configuration file:

```
<bean id="settlementConverter" class="com.springinaction.
      ➥ training.service.PaySettlementConverter">
…
</bean>
```

Next, the `JmsTemplate` needs to know about the message converter. You tell it about the `PaySettlementConverter` by wiring it into `JmsTemplate`'s `messageConverter` property:

```
<bean id="jmsTemplate"
    class="org.springframework.jms.core.JmsTemplate">
…
  <property name="messageConverter">
    <ref bean="settlementConverter"/>
  </property>
</bean>
```

Now that JmsTemplate knows about PaySettlementConverter, you're ready to send messages converted from PaySettlement objects.

Sending and receiving converted messages

With a converted message wired into PayServiceImpl, the implementation of sendSettlementMessage() becomes significantly simpler:

```
public void sendSettlementMessage(PaySettlement settlement) {
  jmsTemplate.convertAndSend(settlement);
}
```

Instead of calling JmsTemplate's send() method and using a MessageCreator to construct the Message object, you simply call JmsTemplate's convertAndSend() method passing in the PaySettlement object. Under the covers, the convertAndSend() method creates its own MessageCreator instance that uses PaySettlementConverter to create a Message object from a PaySettlement object.

Likewise, to receive converted messages, you call the JmsTemplate's receiveAndConvert() method (instead of the receive() method) passing the name of the JMS message queue:

```
PaySettlement settlement = (PaySettlement)
    jmsTemplate.receiveAndConvert("creditCardQueue");
```

Other than automatically converting Message objects to application objects, the semantics of receiveAndConvert() are the same as receive().

Using SimpleMessageConverter

Spring comes with one prepackaged implementation of the MessageConverter interface. SimpleMessageConverter converts MapMessages, TextMessages, and ByteMessages to and from java.util.Map collections, Strings, and byte arrays, respectively.

To use SimpleMessageConverter to convert PaySettlement objects to and from JMS Messages, replace the settlementConverter bean declaration with the following declaration:

```
<bean id="settlementConverter" class="org.springframework.jms.
      ➥ support.converter.SimpleMessageConverter">
…
</bean>
```

Although this converter's function is quite simple, it may prove useful when your messages are simple and do not correspond directly to an object in your application's domain.

7.5 Summary

Even though Spring provides functionality that eliminates much of the need to work with EJBs, there are still many enterprise services that Spring doesn't provide direct replacements for. In those cases, Spring provides abstraction layers that make it easy to wire those services into your Spring-enabled applications.

In this chapter, you've seen how to obtain references to objects that are kept in JNDI. These references could then be wired into bean properties as though they were locally defined beans. This proved to be useful throughout the chapter as you used Spring's JNDI abstraction to look up such things as mail sessions and JMS connection factories.

You've also seen how to send e-mails using Spring's e-mail abstraction and how to schedule tasks using either Java's `Timer` or OpenSymphony's Quartz scheduler.

Finally, you saw how to send and receive asynchronous messages using Spring's JMS abstraction.

In the next chapter, we'll move our focus to the presentation layer of our application, learning how to use Spring's MVC framework to develop web applications.

JAVA

SPRING IN ACTION

Craig Walls • Ryan Breidenbach

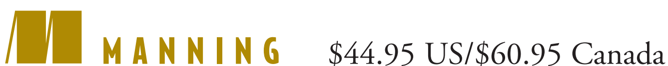
Spring is a fresh breeze blowing over the Java landscape. Based on a design principle called Inversion of Control, Spring is a powerful but lightweight J2EE framework that does not require the use of EJBs. Spring greatly reduces the complexity of using interfaces, and speeds and simplifies your application development. You get the power and robust features of EJB *and* get to keep the simplicity of the non-enterprise JavaBean.

Spring in Action introduces you to the ideas behind Spring and then quickly launches into a hands-on exploration of the framework. Combining short code snippets and an ongoing example developed throughout the book, it shows you how to build simple and efficient J2EE applications. You will see how to solve persistence problems using the leading open-source tools, and also how to integrate your application with the most popular web frameworks. You will learn how to use Spring to manage the bulk of your infrastructure code so you can focus on what really matters—your critical business needs.

What's Inside

- Persistence using Hibernate, JDO, iBatis, OJB, and JDBC
- Declarative transactions and transaction management
- Integration with web frameworks:
  Struts, WebWork, Tapestry, Velocity
- Accessing J2EE services such as JMS and EJB
- Addressing cross-cutting concerns with AOP
- Enterprise applications best practices

Craig Walls is a software developer with over 10 years' experience and co-author of *XDoclet in Action*. He has sucessfully implemented a number of Spring applications. Craig lives in Denton, Texas. An avid supporter of open source Java technologies, **Ryan Breidenbach** has developed Java web applications for the past five years. He lives in Coppell, Texas.



"... a great way of explaining Spring topics... I enjoyed the entire book."

—Christian Parker
President Adigio Inc.

"... no other book can compare with the practical approach of this one."

—Olivier Jolly
J2EE Architect, Interface SI

"I thoroughly enjoyed the way Spring is presented."

—Norman Richards
co-author of *XDoclet in Action*

"I highly recommend it!"

—Jack Herrington, author of
Code Generation in Action

Ask the Authors

Ebook edition

www.manning.com/walls2

ISBN 1-932394-35-4