covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

14 Odds and ends

This chapter covers

- Externalizing configuration
- Wiring JNDI resources in Spring
- Sending email messages
- Scheduling tasks
- Asynchronous methods

I don't know about your house, but many houses (including mine) have a so-called junk drawer. Despite its name, the contents of a junk drawer are often handy or even necessary. Things such as screwdrivers, ballpoint pens, paper clips, and extra keys often call the junk drawer their home. It's not that they're truly junk and have no value—it's that they don't have a place otherwise.

We've covered a lot of ground so far in this book and have explored several corners of working with Spring. Each topic has had a chapter of its own. But there are a few more Spring tricks that I'd like to show you, and none of them were big enough to justify a chapter of their own.

This is the junk drawer of the book. But don't think that the topics here are useless. You'll find valuable techniques here. We'll see how to externalize Spring

configuration, encrypt property values, work with JNDI objects, send emails, and configure methods to run in the background—all using Spring.

First up, let's look at how to move property value configuration out of Spring configuration and into external properties files that can be managed without repackaging and redeploying your applications.

14.1 Externalizing configuration

For the most part, it's possible to configure your entire application in a single bean-wiring file. But sometimes you may find it beneficial to extract certain pieces of that configuration into a separate property file. For example, a configuration concern that's common to many applications is configuring a data source. In Spring, you could configure a data source with the following XML in the bean-wiring file:


```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="org.hsqldb.jdbcDriver"
      p:url="jdbc:hsqldb:hsql://localhost/spitter/spitter"
      p:username="spitterAdmin"
      p:password="t0ps3cr3t" />
```

As you can see, everything you need to do to connect to the database is available in this bean declaration. This has two implications:

- If you need to change the database URL or the username and password, you'll have to edit the Spring configuration file; then recompile and redeploy the application.
- The username and password are sensitive details that you wouldn't want to fall into the wrong hands.

In situations like this, it might be better to not directly configure these details in the Spring application context. Spring comes with a couple of options for externalizing Spring configuration details into property files that can be managed outside of the deployed application:

- *Property placeholder configurers* replace placeholder variables placed in property values with values from an external properties file.
- *Property overriders* override bean property values with values from an external properties file.

In addition, the open source Jasypt project[1] offers alternative implementations of Spring's property placeholder configurer and overrider that can pull those values from encrypted properties files.

We'll look at all of these options, starting with the basic property placeholder configurer that comes with Spring.

[1] http://www.jasypt.org

14.1.1 Replacing property placeholders

In versions of Spring prior to version 2.5, configuring a property placeholder configurer in Spring involved declaring `PropertyPlaceholderConfigurer` as a `<bean>` in the Spring context definition. Although that wasn't terribly complex, Spring 2.5 made it even easier with a new `<context:property-placeholder>` element in the `context` configuration namespace. Now a placeholder configurer can be configured like this:

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

Here, the placeholder configurer is configured to pull property values from a file named db.properties that resides at the root of the classpath. But it could just as easily pull configuration data from a properties file on the file system:

```
<context:property-placeholder
    location="file:///etc/db.properties" />
```

As for the contents of the db.properties file, it would contain (at a minimum) the properties needed by the `DriverManagerDataSource`:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://localhost/spitter/spitter
jdbc.username=spitterAdmin
jdbc.password=t0ps3cr3t
```

Now we can replace the hardcoded values in the Spring configuration with placeholder variables based on the properties in db.properties:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="${jdbc.driverClassName}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}" />
```

For what it's worth, a property placeholder configurer's power isn't limited to bean property configuration in XML. You can also use it to configure `@Value`-annotated properties. For example, if you have a bean that needs the JDBC URL, you can use the `${jdbc.url}` placeholder with `@Value` like this:

```
@Value("${jdbc.url}")
String databaseUrl;
```

What's more, you can even use placeholder variables in the properties file itself. For example, you could define the `jdbc.url` property using placeholder variables to break its configuration into multiple parts:

```
jdbc.protocol=hsqldb:hsql
db.server=localhost
db.name=spitter
jdbc.url=jdbc:${jdbc.protocol}://${db.server}/${db.name}/${db.name}
```

Here I've defined three properties, `jdbc.protocol`, `db.server`, and `db.name`. And I've also defined a fourth property that uses the other properties to construct the database URL.

That describes the essentials of property placeholder replacement in Spring. But we can do a few more things with a property placeholder configurer. First, let's see how to cope with property placeholder variables for which no property is defined.

REPLACING MISSING PROPERTIES

What would happen if a property placeholder variable referred to a property that hasn't been defined? Or worse, what if the `location` attribute pointed to a properties file that doesn't exist?

Well, what happens by default is that an exception will be thrown as the Spring context is being loaded and the beans are being created. But you can configure it to fail silently, without incident, by setting the `<context:property-placeholder>`'s `ignore-resource-not-found` and `ignore-unresolvable` attributes:

```
<context:property-placeholder
  location="file:///etc/myconfig.properties"
  ignore-resource-not-found="true"
  ignore-unresolvable="true"
  properties-ref="defaultConfiguration"/>
```

By setting these properties to `true`, the property placeholder configurer will withhold exceptions when the placeholder variable can't be resolved or if the properties file doesn't exist. Instead, the placeholders will remain unresolved.

Okay, but if the placeholders are unresolved, isn't that a bad thing? After all, `${jdbc.url}` can't be used to access a database. It's not a valid JDBC URL.

Instead of wiring useless placeholder variables, it'd be better to wire in default values. That's where the `properties-ref` attribute comes in handy. This attribute is set to the ID of a `java.util.Properties` bean that contains the properties to use by default. For our database properties, the following `<util:properties>` will hold the default database configuration values:

```
<util:properties id="defaultConfiguration">
  <prop key="jdbc.url">jdbc:hsqldb:hsql://localhost/spitter/spitter</prop>
  <prop key="jdbc.driverClassName">org.hsqldb.jdbcDriver</prop>
  <prop key="jdbc.username">spitterAdmin</prop>
  <prop key="jdbc.password">t0ps3cr3t</prop>
</util:properties>
```

Now, if any of the placeholder variables can't be found in the db.properties file, then the default values in the `defaultConfiguration` bean will be used.

RESOLVING PLACEHOLDER VARIABLES FROM SYSTEM PROPERTIES

At this point, we've seen how to resolve placeholder variables from a properties file and from a `<util:properties>` definition. But it's also possible to resolve them from system properties. All we must do is set the `system-properties-mode` attribute of `<component:property-placeholder>`. For example:

```
<context:property-placeholder
  location="file:///etc/myconfig.properties"
  ignore-resource-not-found="true"
  ignore-unresolvable="true"
  properties-ref="defaultConfiguration"
  system-properties-mode="OVERRIDE"/>
```

Here, the `system-properties-mode` has been set to `OVERRIDE` to indicate that `<component:property-placeholder>` should prefer system properties over those in db.properties or in the `defaultConfiguration` bean. `OVERRIDE` is just one of three values that the `system-properties-mode` attribute accepts:

- `FALLBACK`—Resolve placeholder variables from system properties if they can't be resolved from the properties file.
- `NEVER`—Never resolve placeholder variables from system properties.
- `OVERRIDE`—Prefer system properties over those in a properties file.

The default behavior of `<component:property-placeholder>` is to try to resolve placeholder variables from a properties file, but to fall back to system properties, if available—using the `FALLBACK` value of the `system-properties-mode` attribute.

14.1.2 Overriding properties

Another approach to external configuration in Spring is to override bean properties with those from a property file. In this case, no placeholders are required. Instead, the bean properties are either wired with default values or are left unwired altogether. If an external property matches a bean property, then the external value will be used instead of the one explicitly wired in Spring.

For example, consider the `dataSource` bean as it was before we learned about property placeholders. As a reminder, this is what it looked like with hardcoded values:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="org.hsqldb.jdbcDriver"
      p:url="jdbc:hsqldb:hsql://localhost/spitter/spitter"
      p:username="spitterAdmin"
      p:password="t0ps3cr3t" />
```

In the previous section, I showed you how to declare default values using `<util:properties>` along with a property placeholder configurer. But with a property overrider, you can leave the default values in the bean properties—the overrider will take care of the rest.

Configuring a property overrider is much the same as configuring a property placeholder configurer. The difference is that instead of using `<component:property-placeholder>`, we'll use `<component:property-override>`:

```
<context:property-override
    location="classpath:/db.properties" />
```

In order for a property overrider to know which property in db.properties goes to which bean property in the Spring application context, you must map the bean and property name to the property name in the properties file. Figure 14.1 breaks down how this works.

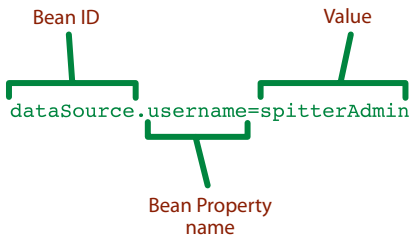

Figure 14.1 A property overrider determines which bean properties to override by mapping keys from a properties file to a bean ID and a property name.

As you can see, the property key in the external properties file is made up of a bean ID and a property name, separate by a period. If you flip back to the beginning of section 14.1, you'll see that the properties defined in db.properties were close, but not quite right. The properties all began with *jdbc.*. which would only work if our data source bean were given an ID of `jdbc`. But its ID is `dataSource`, so we'll need to make some adjustments to the db.properties file:

```
dataSource.driverClassName=org.hsqldb.jdbcDriver
dataSource.url=jdbc:hsqldb:hsql://localhost/spitter/spitter
dataSource.username=spitterAdmin
dataSource.password=t0ps3cr3t
```

Now the keys in db.properties match up with the `dataSource` bean and its properties. In the absence of the db.properties file, the explicitly wired values in the Spring configuration will be in play. But if the db.properties file exists and contains the properties we just defined, those properties will take precedence over those in the Spring XML configuration.

You may be interested to know that `<context:property-override>` can be configured with the same set of attributes as `<context:property-placeholder>`. You can set it up to resolve properties from a `<util:properties>` or from system properties.

At this point we've seen two options for externalizing property values. It should be easy to change the database URL or password without having to rebuild and redeploy the application. But one thing still isn't quite right. Even though the database password is no longer in the Spring context definition, it's still laying out in the open in some properties file somewhere. Let's see how to use Jasypt's property placeholder configurer and overrider to be able to encrypt the password stored in the external properties file.

14.1.3 Encrypting external properties

The *Jasypt project* is a wonderful library that simplifies encryption in Java. It does many things that are beyond the scope of this book. But germane to the topic of externalizing bean property configuration, Jasypt comes with special implementations of Spring's property placeholder configurer and property overrider that can read properties that are encrypted in the external property file.

As I mentioned earlier, Spring 2.5 introduced the `context` namespace and, in doing so, the `<context:property-placeholder>` and `<context:property-overrider>`

elements. Prior to that, you would have to configure `PropertyPlaceholderConfigurer` and `PropertyOverrideConfigurer` as `<bean>`s to get the same functionality.

Jasypt's implementations of the property placeholder configurer and property overrider don't currently have a special configuration namespace. Therefore, much like their pre-2.5 Spring counterparts, Jasypt's placeholder configurer and overrider must be configured as `<bean>` elements.

For example, the following `<bean>` configures a Jasypt property placeholder configurer:

```
<bean class=
   "org.jasypt.spring.properties.EncryptablePropertyPlaceholderConfigurer"
   p:location="file:///etc/db.properties">
  <constructor-arg ref="stringEncrypter" />
</bean>
```

Or, if a property overrider suits you better, then this `<bean>` will do the trick:

```
<bean class=
     "org.jasypt.spring.properties.EncryptablePropertyOverrideConfigurer"
    p:location="file:///etc/db.properties">
  <constructor-arg ref="stringEncrypter" />
</bean>
```

Whichever you choose, either will need to be configured with the location of the properties file through its `location` property. And both require a string encryptor object as a constructor argument.

In Jasypt, a *string encryptor* is a strategy class that handles the chore of encrypting `String` values. The placeholder configurer/overrider will use the string encryptor to decrypt any encrypted values it finds in the external properties file. For our purposes, the `StandardPBEStringEncryptor` that comes with Jasypt is perfectly sufficient:

```
<bean id="stringEncrypter"
      class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor"
      p:config-ref="environmentConfig" />
```

The only things that `StandardPBEStringEncryptor` really needs to do its job are the algorithm and the password used to encrypt the data. If you look at the Javadoc for `StandardPBEStringEncryptor`, you'll see that it has `algorithm` and `password` properties—so we could configure those directly in the `stringEncryptor` bean.

But if we leave the encryption password in the Spring configuration, then have we really secured access to the database? Figuratively speaking, we'd be locking the keys to the database in a box and leaving the keys to the box on the table next to it. At best we've made it slightly more inconvenient, but certainly not secure.

Instead of configuring the password directly in Spring, I've configured `StandardPBEStringEncryptor`'s `config` property with an `EnvironmentStringPBEConfig`. `EnvironmentStringPBEConfig` will let us configure encryption details, such as the encryption password, in environment variables. The `EnvironmentStringPBEConfig` is just another bean declared like this:

```
<bean id="environmentConfig" class=
     "org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig"
      p:algorithm="PBEWithMD5AndDES"
      p:passwordEnvName="DB_ENCRYPTION_PWD" />
```

I didn't mind configuring the algorithm in the Spring configuration—I've configured it as `PBEWithMD5AndDES`. The encryption password is what I want stored outside of Spring in an environment variable. Here, that environment variable is named `DB_ENCRYPTION_PWD`.

So you may be wondering how moving the encryption password to an environment variable makes this arrangement any more secure. Can't a hacker read an environment variable just as easily as they can read a Spring configuration file? The answer to that question is, yes. But the idea here is that the environment variable would be set by a system administrator just before the application is started and then unset once the application is underway. By then the data source properties will have been set and the environment variable will no longer be needed.

Externalizing bean property values is one way to manage configuration details that are sensitive and/or may need to be changed after the application has been deployed. Another way to cope with those situations is to externalize entire objects in JNDI and configure Spring to retrieve those objects into the Spring context. That's what we'll look at next.

14.2 Wiring JNDI objects

The *Java Naming and Directory Interface*, or *JNDI* as it's known to its friends, is a Java API that enables lookup of objects by name in a directory (often but not necessarily an LDAP directory). JNDI provides Java applications with access to a central repository for storing and retrieving applications objects. JNDI is typically used in Java EE applications to store and retrieve JDBC data sources and JTA transaction managers. You'll also find that EJB 3 session beans frequently find their home in JNDI.

But if some of our application objects are configured in JNDI, external to Spring, how can we inject them into the Spring-managed objects that need them?

In this section, we'll look at how Spring supports JNDI by providing a simplified abstraction layer above the standard JNDI API. Spring's JNDI abstraction makes it possible to declare JNDI lookup information in your Spring context definition file. Then you can wire a JNDI-managed object into the properties of other Spring beans as though the JNDI object were just another bean in the Spring application context.

To gain a deeper appreciation of what Spring's JNDI abstraction provides, let's look up an object from JNDI without Spring.

14.2.1 Working with conventional JNDI

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

```
InitialContext ctx = null;
try {
  ctx = new InitialContext();

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

If you've ever written JNDI lookup code before, you're probably familiar with what's going on in this code snippet. You may have written a similar incantation dozens of times before to raise an object out of JNDI. Before you repeat it again, take a closer look at what is going on:

- 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's extra plumbing that's not directly in line with the goal of retrieving a data source.
- You must catch or, at the least, rethrow a `javax.naming.NamingException`. If you choose to catch it, you must also deal with it appropriately. If you choose to rethrow it, the calling code will be forced to deal with it. Ultimately, someone somewhere will have to handle this exception.
- Your code is tightly coupled with a JNDI lookup. All your code needs is a `DataSource`. It doesn't matter whether it comes from JNDI or somewhere else. But if your code contains code like this, then 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/SpitterDatasource`. 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.

Upon closer inspection we find that most of the code is boilerplate JNDI lookup that looks much the same for all JNDI lookups. Only one line is directly responsible for retrieving the data source:

```
DataSource ds =
    (DataSource) ctx.lookup("java:comp/env/jdbc/SpitterDatasource");
```

Even more disquieting than boilerplate JNDI code is the fact that the application knows where the data source comes from. It's coded to *always* retrieve a data source from JNDI. As illustrated in figure 14.2, the DAO that uses the data source will be coupled to JNDI. This makes it almost impossible to use this code in a setting where JNDI isn't available or desirable.

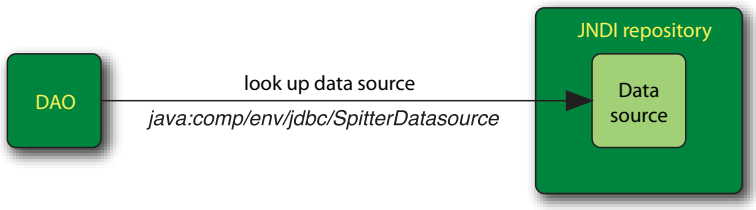

Figure 14.2
Using conventional JNDI to retrieve dependencies means that an object is coupled to JNDI, making it difficult to use the object anywhere that JNDI isn't available.

For instance, imagine that the data source lookup code is embedded in a class that's being unit tested. In an ideal unit test, we're testing an object in isolation without any direct dependence on specific objects. Although the class is decoupled from the data source through JNDI, it's coupled to JNDI itself. Therefore, our unit test has a direct dependence on JNDI and a JNDI server must be available for the unit test to run.

Regardless, this doesn't change the fact that sometimes you need to be able to look up objects in JNDI. `DataSources` 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 your code get an object from JNDI without being dependent on JNDI?

The answer is found in dependency injection. Instead of asking for a data source from JNDI, you should write your code to accept a data source from anywhere—your code should have a `DataSource` property that's injected. Where the object comes from is of no concern to the class that needs it.

The data source object still lives in JNDI. So how can we configure Spring to inject an object that's stored in JNDI?

14.2.2 Injecting JNDI objects

Spring's `jee` configuration namespace holds the answer to working with JNDI in a loosely coupled manner. Within that namespace you'll find the `<jee:jndi-lookup>` element, which makes simple work of wiring a JNDI object into Spring.

To illustrate how this works, let's revisit an example from chapter 5. There, we used `<jee:jndi-lookup>` to retrieve a `DataSource` from JNDI:

```
<jee:jndi-lookup id="dataSource"
                 jndi-name="/jdbc/SpitterDS"
                 resource-ref="true" />
```

The `jndi-name` attribute specifies the name of the object in JNDI. By default, this is the name used to look up the object in JNDI. But if the lookup is occurring in a Java EE container, then a `java:comp/env/` prefix may need to be added. You could manually add the prefix when specifying the value in `jndi-name`. But setting `resource-ref` to `true` will tell `<jee:jndi-lookup>` to do it for you.

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

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.annotation.
                                          ➥ AnnotationSessionFactoryBean">
```

```
  <property name="dataSource" ref="dataSource" />
  ...
</bean>
```

As shown in figure 14.3, when Spring wires the `sessionFactory` bean, it'll inject the `DataSource` object retrieved from JNDI into the session factory's `dataSource` property.

The great thing about using `<jee:jndi-lookup>` 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'd rather get your `DataSource` from a JDBC driver manager, all you need to do is redefine the `dataSource` bean to be a `DriverManagerDataSource`.

Now our data source is retrieved from JNDI and then injected into the session factory. No more explicit JNDI lookup code! Whenever we need it, the data source is always handy in the Spring application context as the `dataSource` bean.

As you've seen, wiring a JNDI-managed bean in Spring is fairly simple. Now let's explore a few ways that we can influence when and how the object is retrieved from JNDI, starting with caching.

CACHING JNDI OBJECTS

Often, the objects retrieved from JNDI will be used more than once. A data source, for example, will be needed every time you access the database. It'd be inefficient to repeatedly retrieve the data source from JNDI every time that it's needed. For that reason, `<jee:jndi-lookup>` caches the object that it retrieves from JNDI by default.

Caching is good in most circumstances. But it precludes hot redeployment of objects in JNDI. If you were to change the object in JNDI, the Spring application would need to be restarted so that the new object can be retrieved.

If your application is retrieving an object from JNDI that will change frequently, you'll want to turn caching off for `<jee:jndi-lookup>`. To turn caching off, you'll need to set the `cache` attribute to `false`:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    cache="false"
    proxy-interface="javax.sql.DataSource" />
```

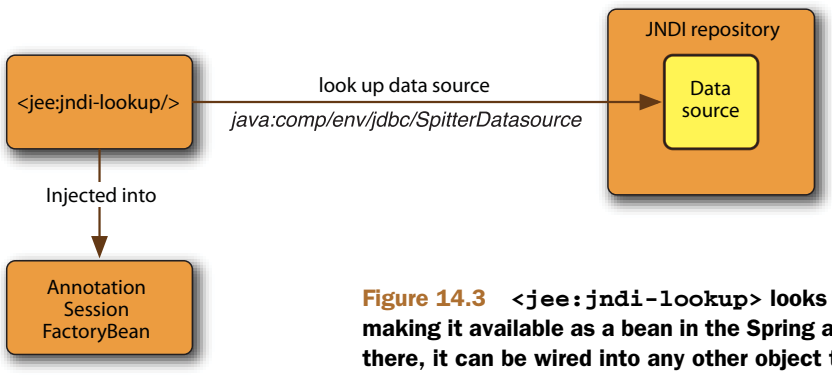

Figure 14.3 `<jee:jndi-lookup>` looks up an object from JNDI, making it available as a bean in the Spring application context. From there, it can be wired into any other object that depends on it.

Setting the `cache` attribute to `false` tells `<jee:jndi-lookup>` to always fetch the object from JNDI. Note that the `proxy-interface` attribute has also been set. Since the JNDI object can be changed at any time, there's no way for `<jee:jndi-lookup>` to know the actual type of the object. The `proxy-interface` attribute specifies a type that's expected for the object retrieved from JNDI.

LAZILY LOADING JNDI OBJECTS

Sometimes your application won't need to retrieve the JNDI object right away. For instance, suppose that a JNDI object is only used in an obscure branch of your application's code. In that situation, it may not be desirable to load the object until it's actually needed.

By default, `<jee:jndi-lookup>` fetches objects from JNDI when the application context is started. Nevertheless, you can configure it to wait to retrieve the object until it's needed by setting the `lookup-on-startup` attribute to `false`:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    lookup-on-startup="false"
    proxy-interface="javax.sql.DataSource" />
```

As with the `cache` attribute, you'll need to set the `lookup-on-startup` attribute when setting `lookup-on-startup` to `false`. That's because `<jee:jndi-lookup>` won't know the type of the object being retrieved until it's actually retrieved. The `proxy-interface` attribute tells it what type to expect from the fetched object.

FALLBACK OBJECTS

You now know how to wire JNDI objects in Spring and have a JNDI-loaded data source to show for it. Life is good. But what if the object can't be found in JNDI?

For instance, maybe your application can count on a data source being available in JNDI when running in a production environment. But that arrangement may not be practical in a development environment. If Spring is configured to retrieve its data source from JNDI for production, the lookup will fail in development. How can we make sure that a data source bean is always available from JNDI in production and explicitly configured in development?

As you've seen, `<jee:jndi-lookup>` is great for retrieving objects from JNDI and wiring them in a Spring application context. But it also has a fallback mechanism that can account for situations where the requested object can't be found in JNDI. All you must do is configure its `default-ref` attribute.

For example, suppose that you've declared a data source in Spring using `DriverManagerDataSource` as follows:

```
<bean id="devDataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      lazy-init="true">
  <property name="driverClassName"
            value="org.hsqldb.jdbcDriver" />
  <property name="url"
```

```
            value="jdbc:hsqldb:hsql://localhost/spitter/spitter" />
  <property name="username" value="sa" />
  <property name="password" value="" />
</bean>
```

This is the data source that you'll use in development. But in production, you'd rather use a data source configured in JNDI by the system administrators. If that's the case, you'll configure the `<jee:jndi-lookup>` element like this:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    default-ref="devDataSource" />
```

Here, we've wired the `default-ref` attribute with a reference to the `devDataSource` bean. If `<jee:jndi-lookup>` can't find an object in JNDI at `jdbc/SpitterDS`, it'll use the `devDataSource` bean as its object. And because the fallback datasource bean has `lazy-init` set to `true`, it won't be created unless it's needed.

As you can see, it's reasonably simple to use `<jee:jndi-lookup>` to wire JNDI-managed objects into a Spring application context. As it turns out, `<jee:jndi-lookup>` can also be used to wire EJB session beans into a Spring application context. Let's see how to do that.

14.2.3 Wiring EJBs in Spring

In EJB 3, session beans are just objects stored away in JNDI, much like any other object in JNDI. Therefore, `<jee:jndi-lookup>` is perfectly sufficient for retrieving EJB 3 session beans. But what if you want to wire an EJB 2 session bean into the Spring application context?

To access an EJB 2 stateless session bean, you start by retrieving an object from JNDI. But that object is an implementation of the EJB's home interface, not the EJB itself. To get a handle to the EJB, you have to call the `create()` method on the home interface.

Fortunately, you don't need to deal with those details when using Spring to access EJB 2 session beans. Instead of using `<jee:jndi-lookup>`, Spring offers two other elements in the `jee` namespace that are expressly for accessing EJBs:

- `<jee:local-slsb>` to access local stateless session beans
- `<jee:remote-slsb>` to access remote stateless session beans

Both of these elements work in a way very similar to `<jee:jndi-lookup>`. For example, to declare a reference to a remote stateless session bean in Spring, use `<jee:remote-slsb>` like this:

```
<jee:remote-slsb id="myEJB"
    jndi-name="my.ejb"
    business-interface="com.habuma.ejb.MyEJB" />
```

The `jndi-name` attribute is the JNDI name used to lookup the EJB's home interface. Meanwhile, the `business-interface` specifies the business interface the EJB

implements. With an EJB reference declared like this, the `myEJB` bean can then be wired into any other bean's property that is of the type `com.habuma.ejb.MyEJB`.

Similarly, a reference to a local stateless session bean can be declared with the `<jee:local-slsb>` element like this:

```
<jee:local-slsb id="myEJB"
    jndi-name="my.ejb"
    business-interface="com.habuma.ejb.MyEJB" />
```

Here, we've discussed using the `<jee:local-slsb>` and `<jee:remote-slsb>` elements to declare EJB 2 session beans in Spring. But what's especially interesting about these elements is that they can also be used to wire EJB 3 session beans. They're smart enough to retrieve the object requested from JNDI and determine whether they're dealing with an EJB 2 home interface or an EJB 3 session bean. If it's an EJB 2 home interface, they'll call `create()` for you. Otherwise, they'll assume that they're dealing with an EJB 3 bean and make that object available in the Spring context.

Looking up objects in JNDI comes in handy when you need access to objects that are configured external to Spring. As you've seen, data sources may be configured through an application server and accessed through JNDI. And as you'll see next, Spring's JNDI lookup capability can be useful when sending email. Let's take a look at Spring's email abstraction layer next.

14.3 Sending email

In chapter 12, we used Spring messaging support to asynchronously queue up jobs to send spittle alerts to other Spitter application users. Now we're ready to use Spring's email support to send the emails.

Spring comes with an email abstraction API that makes simple work of sending emails.

14.3.1 Configuring a mail sender

At the heart of Spring's email abstraction is the `MailSender` interface. As its name implies and as illustrated in figure 14.4, a `MailSender` implementation sends email. Spring comes with one implementation of the `MailSender` interface, `JavaMailSenderImpl`.

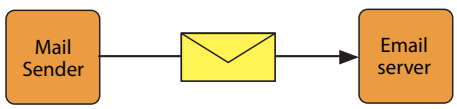

Figure 14.4 Spring's `MailSender` interface is the primary component of Spring's email abstraction API. It sends an email to a mail server for delivery.

WHAT ABOUT COSMAILSENDERIMPL? Older versions of Spring, up to and including Spring 2.0, included another implementation of `MailSender` called `CosMailSenderImpl`. That implementation was removed in Spring 2.5. If you're still using it, you'll need to switch to `JavaMailSenderImpl` before moving up to Spring 2.5 or Spring 3.0.

To use `JavaMailSenderImpl`, we'll declare it as a `<bean>` in the Spring application context:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}" />
```

The `host` property specifies the hostname for the mail server that we'll use to send the email. Here it's configured with a placeholder variable so that we can manage the mail server configuration outside of Spring. By default, `JavaMailSenderImpl` assumes that the mail server is listening on port 25 (the standard SMTP port). If your mail server is listening on a different port, specify the correct port number using the `port` property. For example:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}"
      p:port="${mailserver.port}"/>
```

Likewise, if the mail server requires authentication, you'll also want to set values for the `username` and `password` properties:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}"
      p:port="${mailserver.port}"
      p:username="${mailserver.username}"
      p:password="${mailserver.password}" />
```

This shows how to fully configure the mail sender in Spring with all of the details it'll need to access the mail server. Optionally, you may prefer to use an existing mail session configured in JNDI. Let's see how to configure `JavaMailSenderImpl` to use a mail session that's resident in JNDI.

USING A JNDI MAIL SESSION

You may already have a `javax.mail.MailSession` configured in JNDI (or perhaps one was placed there by your application server). If so then Spring's `JavaMailSenderImpl` offers you an option to use the `MailSender` that you already have ready to use from JNDI.

We've seen how to retrieve objects from JNDI using Spring's `<jee:jndi-lookup>` element. So let's use that to reference a mail session from JNDI:

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

With the mail sender in hand, we can now wire it into the `mailSender` bean like this:

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

By wiring the mail session into the `session` property of `JavaMailSenderImpl`, we've completely replaced the explicit server (and username/password) configuration from before. Now the mail session is completely configured and managed in JNDI. `JavaMailSenderImpl` can focus on sending emails and not dealing with the mail server itself.

WIRING THE MAIL SENDER INTO A SERVICE BEAN

Now that the mail sender has been configured, it's time to wire it into the bean that will use it. In the Spitter application, the `SpitterEmailServiceImpl` class is the most appropriate place to send the email from. This class has a `mailSender` property that's annotated with `@Autowired`:

```
@Autowired
JavaMailSender mailSender;
```

When Spring creates `SpitterEmailServiceImpl` as a bean, it'll try to find a bean that implements `MailSender` that it can wire into the `mailSender` property. It should find our `mailSender` bean and use that. With the `mailSender` bean wired in, we're ready to construct and send emails.

14.3.2 Constructing the email

Since we want to send an email to a Spitter user to alert them of new spittles that their friends may have written, we'll need a method that, given an email address and a `Spittle` object, will send that email. The `sendSimpleSpittleEmail()` method uses the mail sender to do just that.

Listing 14.1 Sending an email with Spring using a `MailSender`

```
public void sendSimpleSpittleEmail(String to, Spittle spittle) {
  SimpleMailMessage message = new SimpleMailMessage();

  String spitterName = spittle.getSpitter().getFullName();
  message.setFrom("noreply@spitter.com");
  message.setTo(to);
  message.setSubject("New spittle from " + spitterName);

  message.setText(spitterName + " says: " +
          spittle.getText());

  mailSender.send(message);
}
```

Construct message

Address email

Set message text

Send email

The first thing that `sendSimpleSpittleEmail()` does is construct an instance of a `SimpleMailMessage`. This mail message object, as its name implies, is perfect for sending no-nonsense email messages.

Next, the details of the message are set. The sender and recipient of the email are specified via the `setFrom()` and `setTo()` methods on the mail message. After setting the subject with `setSubject()`, the virtual "envelope" has been addressed. All that's left is to call `setText()` to set the message's content.

The last step is to pass the message to the mail sender's `send()` method and the email is on its way.

Simple emails are a fine start. But what if you want to add an attachment? Or what if you want the body of the email to have a polished look? Let's see how to dress up our emails sent by Spring, starting with simply adding an attachment.

ADDING ATTACHMENTS

The trick to sending emails with attachments is to create multipart messages—emails composed of multiple parts, one of which is the body of the email and the other parts being the attachments.

The `SimpleMailMessage` class is too… well… simple for sending attachments. To send multipart emails, you need to create a *MIME (Multipurpose Internet Mail Extensions)* message. The mail sender object's `createMimeMessage()` method can get you started:

```
MimeMessage message = mailSender.createMimeMessage();
```

There you go. We now have a MIME message to work with. It seems that all we need to do is give it to and from addresses, a subject, some text, and an attachment. Though that's true, it's not as straightforward as you might think. The `javax.mail.internet.MimeMessage` class has an API that's too cumbersome to use on its own. Fortunately, Spring provides `MimeMessageHelper` to lend a hand.

To use `MimeMessageHelper`, instantiate an instance of it, passing in the `MimeMessage` to its constructor:

```
MimeMessageHelper helper = new MimeMessageHelper(message, true);
```

The second parameter to the constructor, a Boolean `true` as shown here, indicates that this message is to be a multipart message.

From the `MimeMessageHelper` instance, we're ready to assemble our email message. The only major difference here is that you'll provide the email specifics through methods on the helper instead of on the message itself:

```
String spitterName = spittle.getSpitter().getFullName();
helper.setFrom("noreply@spitter.com");
helper.setTo(to);
helper.setSubject("New spittle from " + spitterName);
helper.setText(spitterName + " says: " + spittle.getText());
```

The only thing needed before you can send the email is to attach the coupon image. To do that, you'll need to load the image as a resource and then pass that resource in as you call the helper's `addAttachment()` method:

```
FileSystemResource couponImage =
      new FileSystemResource("/collateral/coupon.png");
helper.addAttachment("Coupon.png", couponImage);
```

Here, you're using Spring's `FileSystemResource` to load coupon.png from within the application's classpath. From there, you call `addAttachment()`. The first parameter is the name to be given to the attachment in the email. The second parameter is the image's resource.

The multipart email has been constructed. Now you're ready to send it. The complete `sendSpittleEmailWithAttachment()` method is shown next.

Listing 14.2 `MimeMessageHelper` simplifies sending emails with attachments.

```
public void sendSpittleEmailWithAttachment(
           String to, Spittle spittle) throws MessagingException {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper =
        new MimeMessageHelper(message, true);

  String spitterName = spittle.getSpitter().getFullName();
  helper.setFrom("noreply@spitter.com");
  helper.setTo(to);
  helper.setSubject("New spittle from " + spitterName);

  helper.setText(spitterName + " says: " + spittle.getText());

  FileSystemResource couponImage =
        new FileSystemResource("/collateral/coupon.png");
  helper.addAttachment("Coupon.png", couponImage);

  mailSender.send(message);
}
```

Construct message helper

Add attachment

Adding attachments to an email is only one thing you can do with multipart emails. In addition, by specifying that the body of the email is HTML, you can produce polished emails that look much nicer than flat text. Let's see how to send attractive-looking emails using Spring's `MimeMessageHelper`.

SENDING EMAILS WITH RICH CONTENT

Sending a rich email isn't much different than sending plain-text emails. The key is to set the message's text as HTML. Doing that is as simple as passing in an HTML string to the helper's `setText()` method and `true` as the second parameter:

```
helper.setText("<html><body><img src='cid:spitterLogo'>" +
    "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
    "<i>" + spittle.getText() + "</i>" +
        "</body></html>", true);
```

The second parameter indicates that the text passed into the first parameter is HTML, so that the message part's content type will be set accordingly.

Note that the HTML passed in has an `<img>` tag to display the Spitter application's logo as part of the email. The `src` attribute could be set to a standard `http:` URL to pull the Spitter logo from the web. But here, we've embedded the logo image in the email itself. The value `cid:spitterLogo` indicates that there will be an image in one of the message's parts identified as `spitterLogo`.

Adding the embedded image to the message is much like adding an attachment. Instead of calling the helper's `addAttachment()` method, you must call the `addInline()` method:

```
ClassPathResource image = new ClassPathResource("spitter_logo_50.png");
helper.addInline("spitterLogo", image);
```

The first parameter to `addInline` specifies the identity of the inline image—which is the same as was specified by the `<img>`'s `src` attribute. The second parameter is the resource reference for the image, created here using Spring's `ClassPathResource` to retrieve the image from the application's classpath.

Aside from the slightly different call to `setText()` and the use of the `addInline()` method, sending an email with rich content is much like how you sent a plain-text message with attachments. For sake of comparison, here's the new `sendRichSpitterEmail()` method.

```
public void sendRichSpitterEmail(String to, Spittle spittle) throws Messaging
  Exception {
  MimeMessage message = mailSender.createMimeMessage();
  MimeMessageHelper helper = new MimeMessageHelper(message, true);
  helper.setFrom("noreply@spitter.com");
  helper.setTo("craig@habuma.com");
  helper.setSubject("New spittle from " +
          spittle.getSpitter().getFullName());
  helper.setText("<html><body><img src='cid:spitterLogo'>" +
      "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
      "<i>" + spittle.getText() + "</i>" +
          "</body></html>", true);
  ClassPathResource image = new ClassPathResource("spitter_logo_50.png");
  helper.addInline("spitterLogo", image);
  mailSender.send(message);
}
```

Set HTML body

Add inline image

And now you're sending emails with rich content and embedded images! You could stop here and call your email code complete. But it bugs me that the email's body was created by using string concatenation to construct an HTML message. Before we put the email topic to rest, let's see how to replace that string-concatenated message with a template.

CREATING EMAIL TEMPLATES

The problem with constructing an email message using string concatenation is that it's not clear what the resulting email will look like. It's hard enough to mentally parse HTML markup to imagine how it might appear when rendered. But mixing that HTML up within Java code compounds the issue. Moreover, it might be nice to extract the email layout into a template that a graphic designer (who has an aversion to Java code) can produce.

What we need is a way to express the email layout in something close to what the resulting HTML will look like and then transform that template into a `String` to be passed into the `setText()` method on the message helper. When it comes to transforming templates into strings, Apache Velocity[2] is one of the best options available.

[2] http://velocity.apache.org

To use Velocity to lay out our email messages, we'll first need to wire a `VelocityEngine` into `SpitterEmailServiceImpl`. Spring provides a handy factory bean called `VelocityEngineFactoryBean` that will produce a `VelocityEngine` in the Spring application context. The declaration for `VelocityEngineFactoryBean` looks like this:

```
<bean id="velocityEngine"
      class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
  <property name="velocityProperties">
    <value>
resource.loader=class
class.resource.loader.class=org.apache.velocity.runtime.resource.loader.Class
     pathResourceLoader
    </value>
  </property>
</bean>
```

The only property that needs to be set on `VelocityEngineFactoryBean` is `velocityProperties`. In this case, we're configuring it to load Velocity templates from the classpath (see the Velocity documentation for more details on how to configure Velocity).

Now we can wire the Velocity engine into `SpitterEmailServiceImpl`. Since `SpitterEmailServiceImpl` is automatically registered with the component scanner, we can use `@Autowired` to automatically wire a `velocityEngine` property:

```
@Autowired
  VelocityEngine velocityEngine;
```

Now that the `velocityEngine` property is available, we can use it to transform a Velocity template into a `String` to send as our email text. To help out with that, Spring comes with `VelocityEngineUtils` to make simple work of merging a Velocity template and some model data into a `String`. Here's how we might use it:

```
Map<String, String> model = new HashMap<String, String>();
    model.put("spitterName", spitterName);
    model.put("spittleText", spittle.getText());
    String emailText = VelocityEngineUtils.mergeTemplateIntoString(
            velocityEngine, "emailTemplate.vm", model );
```

In preparation for processing the template, we start by creating a `Map` to hold the model data used by the template. In our previous string-concatenated code, we needed the full name of the spitter and the text of their spittle, so we'll need that here as well. To produce the merged email text, we then just need to call `VelocityEngineUtils`'s `mergeTemplateIntoString()` method, passing in the Velocity engine, the path to the template (relative to the root of the classpath), and the model map.

All that's left to be done in the Java code is to hand off the merged email text to the message helper's `setText()` method:

```
helper.setText(emailText, true);
```

As for the template itself, that's sitting at the root of the classpath in a file called emailTemplate.vm, which looks like this:

```
<html>
  <body>
    <img src='cid:spitterLogo'>
    <h4>${spitterName} says...</h4>
    <i>${spittleText}</i>
  </body>
</html>
```

As you can see, the template file is a lot easier to read than the string-concatenated version from before. Consequently, it's also easier to maintain and edit. Figure 14.5 gives a sample of the kind of email it might produce.

After looking at figure 14.5, I see a lot of opportunity left to dress up the template so that the email appears much nicer. But, as they say, I'll leave that as an exercise for the reader.

But for now, we have one more Spring attraction to look at. And I've saved one of the best for last! Let's see how to make short work of running jobs in the background using Spring.

14.4 Scheduling and background tasks

The better part of the functionality in most applications happens in response to something that the application's users have done. A user fills in a form and then clicks a button, and the application reacts by processing the data, persisting it to a database, and producing some output.

But sometimes applications have work of their own to do, without the user being involved. While the users click the buttons, the application can be handling background jobs that don't involve user interaction.

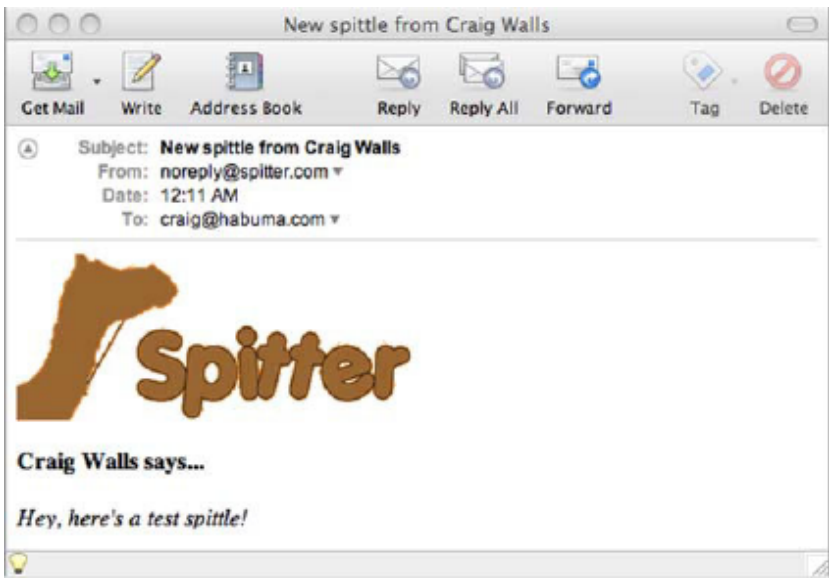

Figure 14.5 A Velocity template and some embedded images can dress up an otherwise ho-hum email.

There are two kinds of background jobs to choose from:

- Scheduled jobs
- Asynchronous methods

Scheduled jobs involve functionality that takes place every so often, either at some specified period or at some specific time. Asynchronous methods, on the other hand, are methods that are called, but that return immediately so that the caller can proceed—while the asynchronous method continues running in the background.

Regardless of which kind of background job you need, you'll need to add a single line of configuration to the Spring application context:

```
<task:annotation-driven/>
```

The `<task:annotation-driven/>` element sets Spring up to automatically support scheduled and asynchronous methods. These methods are identified with the `@Scheduled` and `@Async` methods, respectively.

Let's see how to use these annotations, starting with using the `@Scheduled` annotation to fire off methods on a schedule.

14.4.1 Declaring scheduled methods

If you've been working with Spring for awhile, then you know that Spring has supported scheduling of method invocations for a long time. But until recently, the Spring configuration required to schedule methods was involved. In the second edition of this book, I spent 10 pages showing how to invoke methods periodically.

Spring 3 changes that with the new `@Scheduled` annotation. What used to take several lines of XML and a handful of beans now can be done with the `<task :annotation-driven>` element and a single annotation. I definitely won't need 10 pages to show you how it works.

To schedule a method, all you have to do is to annotate it with `@Scheduled`. For example, to have Spring automatically invoke a method every 24 hours (86,400,000 milliseconds):

```
@Scheduled(fixedRate=86400000)
public void archiveOldSpittles() {
  // ...
}
```

The `fixedRate` attribute indicates that the method should be invoked periodically, every so many milliseconds. In this case, 86,400,000 milliseconds will pass between the start of each invocation. If you'd rather specify the time that passes in between invocations (between the completion of one invocation and the start of the next), then use the `fixedDelay` attribute instead:

```
@Scheduled(fixedDelay=86400000)
public void archiveOldSpittles() {
  // ...
}
```

Running a task at a given interval can be handy. But you may want more precise control over when a method is invoked. With `fixedRate` and `fixedDelay`, you can only control how often a method is invoked, but not *when* it happens. To be specific about the times a method should be called, use the `cron` attribute:

```
@Scheduled(cron="0 0 0 * * SAT")
public void archiveOldSpittles() {
  // ...
}
```

The value given to the `cron` attribute is a Cron expression. For those who aren't so well-versed in Cron expressions, let's break down the `cron` attribute. The Cron expression is made up of six (or possibly seven) 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 (`6`), a range (`9-12`), a list (`9,11,13`), or a wildcard (for example, `*`). The day of the month and day of the week elements are mutually exclusive, so you should also indicate which one of the fields you don't want to set by specifying it with a question mark (`?`). Table 14.1 shows some example Cron expressions that could be used with the `cron` attribute.

In the example, I've specified that old `Spittle`s should be archived every Saturday at midnight. But since this method is scheduled using a Cron expression, the scheduling options are virtually limitless. Where `fixedRate` and `fixedDelay` are limited to fixed time periods, a Cron-scheduled method could be scheduled to run at odd times. I'm sure you can dream up some interesting Cron expressions to schedule methods with.

Table 14.1 Some sample Cron expressions

Cron 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 * 1-30 * ?`	Every 15 minutes on the first 30 days of the month
`30 0 0 1 1 ? 2012`	30 seconds after midnight on January 1, 2012
`0 0 8-17 ? * MON-FRI`	Every working hour of every business day

14.4.2 Declaring asynchronous methods

When it comes to dealing with the human users of an application, there are two kinds of application performance: actual and perceived. The *actual performance* of an application (the discrete measurement of how long it takes to perform an operation) is certainly important. But even if the actual performance is less than ideal, its effect on the users can be mitigated with the perceived performance.

Perceived performance is exactly what it sounds like. Who cares how long it takes to do something, as long as the user sees something happening immediately? For example, let's suppose that the act of adding a `Spittle` is a costly operation. If handled synchronously, the perceived performance would be a function of the actual performance. The user would have to wait while the `Spittle` is saved.

But if there were only some way for the `SpitterService`'s `saveSpittle()` method to be handled asynchronously, then the application could be presenting a new page to the user while the persistence logic is handled in the background. That's what the `@Async` annotation is for.

`@Async` is a simple annotation and has no attributes to set. All you need to do is use it to annotate a bean method and that method becomes asynchronous. It couldn't be any simpler than that.

For example, here's roughly what the `SpittleServiceImpl`'s `saveSpittle()` method might look like as an asynchronous method:

```
@Async
public void addSpittle(Spittle spittle) {
    ...
}
```

That's really all there is to it. When the `saveSpittle()` method is called, control will return to the caller immediately. Meanwhile, the `saveSpittle()` method will continue running in the background.

You might be wondering what would happen if an asynchronous method needs to return something to the caller. If the method returns immediately, then how can it possibly pass results back to the caller?

Since Spring asynchronous methods are based on Java's concurrency API, they can return an object that implements `java.util.concurrent.Future`. This interface represents a holder for some value that will eventually be available at some point after the method returns, but not necessarily at the point that the method returns. Spring comes with a convenient implementation of `Future` called `AsyncResult` that makes it easy to work with future values.

For example, suppose that you have an asynchronous method that attempts to perform some complex and long-running calculation. You want the method to run in the background, but once it's finished you want to be able to see what the results were. In that case, you might write the method something like this:

```
@Async
public Future<Long> performSomeReallyHairyMath(long input) {
    // ...

    return new AsyncResult<Long>(result);
}
```

This method can take as long as it needs to produce the result, while the caller can go about any other business that needs to be done. The caller will receive a `Future` object (actually, an `AsyncResult`) to hold on to while the result is calculated.

Once the result is ready, the caller can retrieve it by calling the `get()` method on the `Future` object. Until then, the caller can check in on the status of the result by calling `isDone()` and `isCancelled()`.

14.5 Summary

In this chapter, we've covered a mixed bag of topics—Spring features that, by themselves, didn't have a home in any other chapter.

We started by looking at how to externalize bean property values using property placeholder configurers and overriders. We also learned how to not only externalize properties, but to encrypt them so that prying eyes won't gain access to the sensitive configuration details of our application.

We then took externalization up a notch by pushing entire objects into JNDI and then configuring Spring to pull those objects into the Spring context where they can be wired into other beans as if they were beans themselves.

Then we looked at sending emails with Spring. Although Spring's email abstraction is hardly the most exciting thing that Spring can do, it beats writing code to send email without Spring. We saw how to send simple emails, HTML-based emails, and emails with attachments and embedded content.

Finally, we tinkered with background jobs in Spring. We started by annotating methods to run on a specific schedule. Then we annotated methods to run asynchronously with our application so that the perceived performance of the application could be improved.

14.6 The end...?

I hate to admit it, but we've come to the end of the book. That's not to say that there's nothing else to learn about Spring. As I stated in the preface, I could literally write *volumes* about Spring. But if I did that, this book would never have made it into your hands and I'd never know the satisfaction of a full night's sleep.

Although some tough decisions had to be made with regard to the scope of this book, I think we've covered the most important topics that you'll need to build applications with Spring. And now you're equipped to explore those other topics on your own.

So, although this chapter ends *Spring in Action*, your journey in Spring is just beginning. I encourage you to leverage what you've learned here to dig more into the other

areas of Spring such as Spring Integration, Spring Batch, Spring Dynamic Modules, and (my personal favorite) Spring Roo. Fortunately, Manning has *in Action* books on each of these topics to help you explore further:

- *Spring Integration in Action* by Mark Fisher, Jonas Partner, Marius Bogoevici, and Iwein Fuld
- *Spring Batch in Action* by Thierry Templier and Arnaud Cogoluègnes
- *Spring Dynamic Modules in Action* by Arnaud Cogoluègnes, Thierry Templier, and Andy Piper
- *Roo in Action* by Gordon Dickens and Ken Rimple

And you can always hang out at the Spring forums—http://forum.springframework.org—to learn about these and other Spring-related projects.

It's been fun for me. I hope it's been fun for you.

This page intentionally left blank