covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

3 Minimizing XML configuration in Spring

This chapter covers

- Automatic bean wiring
- Automatic bean discovery
- Annotation-oriented bean wiring
- Java-based Spring configuration

So far, we've seen how to declare beans using the `<bean>` element and inject `<bean>` with values using either the `<constructor-arg>` or `<property>` element. That's all well and good for a small application where you only have a handful of beans. But as your application grows, so will the amount of XML configuration you'll write.

Fortunately, Spring offers a few tricks to help cut down on the amount of XML configuration required:

- Autowiring helps reduce or even eliminate the need for `<property>` and `<constructor-arg>` elements by letting Spring automatically figure out how to wire bean dependencies.
- Autodiscovery takes autowiring a step further by letting Spring figure out which classes should be configured as Spring beans, reducing the need for the `<bean>` element.

When used together, autowiring and autodiscovery can dramatically reduce the amount of XML Spring configuration. Often you'll need only a handful of lines of XML, regardless of how many beans are in your Spring application context.

We'll start this chapter by looking at how to take advantage of Spring autowiring and autodiscovery to cut down on the amount of XML needed to configure a Spring application. Then to wrap up the chapter, we'll look at Spring's Java-based configuration, which relies on good old Java code instead of XML to configure a Spring application.

3.1 Automatically wiring bean properties

If I were to say "The moon is bright tonight," it's not likely that you'd respond by asking "Which moon?" That's because we both reside on the Earth and in that context it's obvious that I'm talking about Luna, the Earth's moon. If I were to say the same thing while we were standing on Jupiter, you'd be justified in asking which of the planet's 63 natural satellites I had in mind. But on Earth, there's no ambiguity.[1] Similarly, when it comes to wiring bean properties, it's sometimes quite obvious which bean reference should be wired into a given property. If your application context only has one bean of type `javax.sql.DataSource`, then any bean that needs a `DataSource` will certainly need *that* `DataSource`. After all, it's the only `DataSource` to be had.

Taking advantage of such obvious wirings, Spring offers autowiring. Rather than explicitly wiring bean properties, why not let Spring sort out those cases when there's no question about which bean reference should be wired?

3.1.1 The four kinds of autowiring

When it comes to automatically wiring beans with their dependencies, Spring has a lot of clues to work from. As a result, Spring provides four flavors of autowiring:

- `byName`—Attempts to match all properties of the autowired bean with beans that have the same name (or ID) as the properties. Properties for which there's no matching bean will remain unwired.
- `byType`—Attempts to match all properties of the autowired bean with beans whose types are assignable to the properties. Properties for which there's no matching bean will remain unwired.
- `constructor`—Tries to match up a constructor of the autowired bean with beans whose types are assignable to the constructor arguments.
- `autodetect`—Attempts to apply `constructor` autowiring first. If that fails, `byType` will be tried.

Each of these options has its pros and cons. Let's first look at how to have Spring autowire a bean's properties using the names of those properties as a guide.

[1] Of course, if we were standing on Jupiter, the brightness of any of its moons would be of little concern given the intense atmospheric pressure and all of the unbreathable methane.

AUTOWIRING BY NAME

In Spring, everything is given a name. Thus bean properties are given names, as are the beans that are wired into those properties. Suppose that the name of a property happens to match the name of the bean that's to be wired into that property. That happy coincidence could serve as a hint to Spring that the bean should be automatically wired into the property.

For example, let's revisit the `kenny` bean from the previous chapter:

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Here you've explicitly configured Kenny's `instrument` property using `<property>`. For a moment, let's pretend that you declared the `Saxophone` as a `<bean>` with an `id` of `instrument`:

```
<bean id="instrument"
      class="com.springinaction.springidol.Saxophone" />
```

If this were the case, the `id` of the `Saxophone` bean would be the same as the name of the `instrument` property. Spring can take advantage of this to automatically configure Kenny's `instrument` by setting the `autowire` property:

```
<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist"
    autowire="byName">
  <property name="song" value="Jingle Bells" />
</bean>
```

`byName` autowiring establishes a convention where a property will automatically be wired with a bean of the same name. In setting the `autowire` property to `byName`, you're telling Spring to consider all properties of `kenny` and look for beans that are declared with the same names as the properties. In this case, Spring finds that the `instrument` property is eligible for autowiring through setter injection. As illustrated in figure 3.1, if there's a bean in the context whose `id` is `instrument`, it'll be autowired into the `instrument` property.

The downside of using `byName` autowiring is that it assumes that you'll have a bean whose name is the same as the name of the property of another bean. In our example, it would require creating a bean whose name is `instrument`. If multiple `Instrumentalist` beans are configured to be autowired by name, then all of them will be playing the same instrument. This may not be a problem in all circumstances, but it's something to keep in mind.

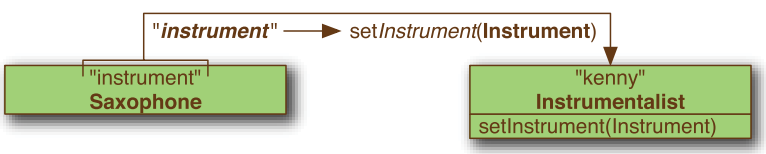

Figure 3.1 When autowiring by name, a bean's name is matched against properties that have the same name.

AUTOWIRING BY TYPE

Autowiring using `byType` works in a similar way to `byName`, except that instead of considering a property's name, the property's type is examined. When attempting to autowire a property by type, Spring will look for beans whose type is assignable to the property's type.

For example, suppose that the `kenny` bean's `autowire` property is set to `byType` instead of `byName`. The container will search itself for a bean whose type is `Instrument` and wire that bean into the `instrument` property. As shown in figure 3.2, the `saxophone` bean will be automatically wired into Kenny's `instrument` property because both the `instrument` property and the `saxophone` bean are of type `Instrument`.

But there's a limitation to autowiring by type. What happens if Spring finds more than one bean whose type is assignable to the `autowired` property? In such a case, Spring isn't going to guess which bean to `autowire` and will instead throw an exception. Consequently, you're allowed to have only one bean configured that matches the `autowired` property. In the *Spring Idol* competition, there are likely to be several beans whose types are subclasses of `Instrument`.

To overcome ambiguities with autowiring by type, Spring offers two options: you can either identify a primary candidate for autowiring or you can eliminate beans from autowiring candidacy.

To identify a primary autowire candidate, you'll work with the `<bean>` element's `primary` attribute. If only one autowire candidate has the `primary` attribute set to `true`, then that bean will be chosen in favor of the other candidates.

But here's the weird side of the `primary` attribute: it defaults to `true`. That means that all autowire candidates will be primary (and thus none will be preferred). So, to use `primary`, you'll need to set it to `false` for all of the beans that are *not* the primary choice. For example, to establish that the `saxophone` bean isn't the primary choice when autowiring `Instrument`s:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      primary="false" />
```

The `primary` attribute is only useful for identifying a preferred autowire candidate. If you'd rather eliminate some beans from consideration when autowiring, then you can set their `autowire-candidate` attribute to `false`, as follows:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      autowire-candidate="false" />
```

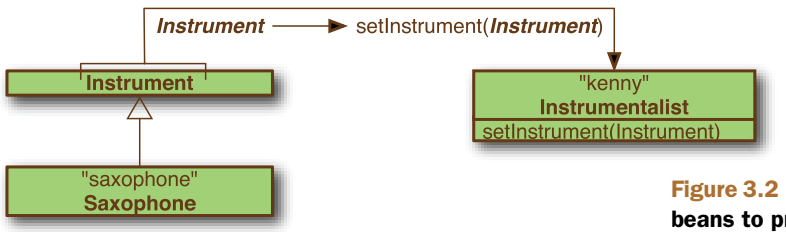

Figure 3.2 Autowiring by type matches beans to properties of the same type.

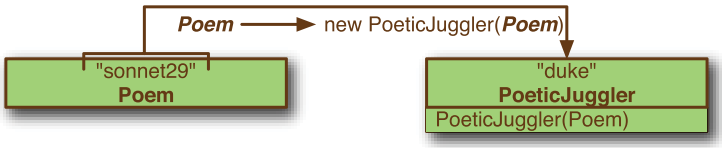

Figure 3.3 When autowired by constructor, the `dukePoeticJuggler` is instantiated with the constructor that takes a `Poem` argument.

Here, we've asked Spring to disregard the `saxophone` bean as a candidate when performing autowiring.

AUTOWIRING CONSTRUCTORS

If your bean is configured using constructor injection, you may choose to put away the `<constructor-arg>` elements and let Spring automatically choose constructor arguments from beans in the Spring context.

For example, consider the following redeclaration of the `duke` bean:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="constructor" />
```

In this new declaration of `duke`, the `<constructor-arg>` elements are gone and the `autowire` attribute has been set to `constructor`. This tells Spring to look at `PoeticJuggler`'s constructors and try to find beans in the Spring configuration to satisfy the arguments of one of the constructors. You've already declared the `sonnet29` bean, which is a `Poem` and matches the constructor argument of one of `PoeticJuggler`'s constructors. Therefore, Spring will use that constructor, passing in the `sonnet29` bean, when constructing the `duke` bean, as expressed in figure 3.3.

Autowiring by constructor shares the same limitations as `byType`. Spring won't attempt to guess which bean to autowire when it finds multiple beans that match a constructor's arguments. Furthermore, if a class has multiple constructors, any of which can be satisfied by autowiring, Spring won't attempt to guess which constructor to use.

BEST-FIT AUTOWIRING

If you want to autowire your beans, but you can't decide which type of autowiring to use, have no fear. You can set the `autowire` attribute to `autodetect` to let Spring make the decision for you. For example:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="autodetect" />
```

When a bean has been configured to autowire by `autodetect`, Spring will attempt to autowire by `constructor` first. If a suitable constructor-to-bean match can't be found, then Spring will attempt to autowire by type.

3.1.2 Default autowiring

If you find yourself putting the same `autowire` attribute on every bean in your application context (or even most of them), you can simplify things by asking Spring to apply the same autowiring style to all beans that it creates. All you need to do is add a `default-autowire` attribute to the root `<beans>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"
    default-autowire="byType">

</beans>
```

By default, `default-autowire` is set to `none`, indicating that no beans should be autowired unless they're individually configured for autowiring with the `autowire` attribute. Here we've set it to `byType` to indicate that we want the properties of every bean to be automatically wired using that style of autowiring. But you can set `default-autowire` to any of the valid autowiring types to be applied to all beans in a Spring configuration file.

Note that I said that `default-autowire` would be applied to all beans in a given Spring configuration file; I didn't say that it would be applied to all beans in a Spring application context. You could have multiple configuration files that define a single application context, each with their own default autowiring setting.

Also, just because you've defined a default autowiring scheme, that doesn't mean that you're stuck with it for all of your beans. You can still override the default on a bean-by-bean basis using the `autowire` attribute.

3.1.3 Mixing auto with explicit wiring

Just because you choose to autowire a bean, that doesn't mean you can't explicitly wire some properties. You can still use the `<property>` element on any property just as if you hadn't set `autowire`.

For example, to explicitly wire Kenny's `instrument` property even though he's set to autowire by type, use this code:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
      <property name="song" value="Jingle Bells" />
      <property name="instrument" ref="saxophone" />
</bean>
```

As illustrated here, mixing automatic and explicit wiring is also a great way to deal with ambiguous autowiring that might occur when autowiring using `byType`. There may be several beans in the Spring context that implement `Instrument`. To keep Spring from throwing an exception due to the ambiguity of several `Instrument`s to choose from, we can explicitly wire the `instrument` property, effectively overriding autowiring.

We mentioned earlier that you could use `<null/>` to force an autowired property to be `null`. This is a special case of mixing autowiring with explicit wiring. For example, if you wanted to force Kenny's `instrument` to be `null`, you'd use the following configuration:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
      <property name="song" value="Jingle Bells" />
      <property name="instrument"><null/></property>
</bean>
```

This is just for illustration's sake, of course. Wiring `null` into `instrument` will result in a `NullPointerException` being thrown when the `perform()` method is invoked.

One final note on mixed wiring: when using constructor autowiring, you must let Spring wire all of the constructor arguments—you can't mix `<constructor-arg>` elements with constructor autowiring.

3.2 Wiring with annotations

Since Spring 2.5, one of the most interesting ways of wiring beans in Spring has been to use annotations to automatically wire bean properties. Autowiring with annotations isn't much different than using the `autowire` attribute in XML. But it does allow for more fine-grained autowiring, where you can selectively annotate certain properties for autowiring.

Annotation wiring isn't turned on in the Spring container by default. So, before we can use annotation-based autowiring, we'll need to enable it in our Spring configuration. The simplest way to do that is with the `<context:annotation-config>` element from Spring's `context` configuration namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:annotation-config />

  <!-- bean declarations go here -->

</beans>
```

`<context:annotation-config>` tells Spring that you intend to use annotation-based wiring in Spring. Once it's in place you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.

Spring 3 supports a few different annotations for autowiring:

- Spring's own `@Autowired` annotation
- The `@Inject` annotation from JSR-330
- The `@Resource` annotation from JSR-250

We'll look at how to use Spring's `@Autowired` first. Then we'll try out standards-based dependency injection with JSR-330's `@Inject` and JSR-250's `@Resource`.

3.2.1 Using @Autowired

Suppose that you want to use `@Autowired` to have Spring autowire the `instrument` property of the `Instrumentalist` bean. You could annotate the `setInstrument()` method like this:

```
@Autowired
public void setInstrument(Instrument instrument) {
  this.instrument = instrument;
}
```

Now you can get rid of the `<property>` element that wires the `Instrumentalist` with an instrument. When Spring sees that you've annotated `setInstrument()` with `@Autowired` it'll try to perform `byType` autowiring on the method.

What's especially interesting about `@Autowired` is that you don't have to use it with a setter method. You can use it on any method to automatically wire in bean references:

```
@Autowired
public void heresYourInstrument(Instrument instrument) {
  this.instrument = instrument;
}
```

The `@Autowired` annotation can even be used on constructors:

```
@Autowired
public Instrumentalist(Instrument instrument) {
  this.instrument = instrument;
}
```

When used with constructors, `@Autowired` indicates that the constructor should be autowired when creating the bean, even if no `<constructor-arg>` elements are used to configure the bean in XML.

What's more, you can directly annotate properties and do away with the setter methods altogether:

```
@Autowired
private Instrument instrument;
```

As you can see, `@Autowired` won't even be thwarted by the `private` keyword. Even though the `instrument` property is `private`, it'll still be autowired. Is there no limit to `@Autowired`'s reach?

Actually, there are a couple of circumstances that could keep `@Autowired` from getting its job done. Specifically, there must be exactly one bean that's applicable for wiring into the `@Autowired` property or parameter. If there are no applicable beans or if multiple beans could be autowired, then `@Autowired` will run into some trouble.

Fortunately, there's a way that we can help `@Autowired` out in those circumstances. First, let's look at how to keep `@Autowired` from failing when there isn't a matching bean.

OPTIONAL AUTOWIRING

By default, `@Autowired` has a strong contract, requiring that the thing it annotates is wired. If no bean can be wired into the `@Autowired`-annotated property or argument,

then autowiring fails (with a nasty `NoSuchBeanDefinitionException`). That may be what you want—to have Spring fail early when autowiring goes bad rather than later with a `NullPointerException`.

But it's also possible that the property being wired is truly optional and a `null` value is acceptable. In that case, you can configure optional autowiring by setting `@Autowired`'s `required` attribute to `false`. For example:

```
@Autowired(required=false)
private Instrument instrument;
```

Here, Spring will try to wire the `instrument` property. But if no bean of type `Instrument` can be found, then no problem. The property will be left `null`.

Note that the `required` attribute can be used anywhere `@Autowired` can be used. But when used with constructors, only one constructor can be annotated with `@Autowired` and `required` set to `true`. All other `@Autowired`-annotated constructors must have `required` set to `false`. Moreover, when multiple constructors are annotated with `@Autowired`, Spring will choose the constructor which has the most arguments that can be satisfied.

QUALIFYING AMBIGUOUS DEPENDENCIES

On the other hand, maybe the problem's not a lack of beans for Spring autowiring to choose from. Maybe it's an abundance of (or at least two) beans, each of which is equally qualified to be wired into a property or parameter.

For example, suppose you have two beans that implement `Instrument`. In that event, there's no way for `@Autowired` to choose which one you really want. So, instead of guessing, a `NoSuchBeanDefinitionException` will be thrown and wiring will fail.

To help `@Autowired` figure out which bean you want, you can accompany it with Spring's `@Qualifier` annotation.

For example, to ensure that Spring selects a guitar for the `eddie` bean to play, even if there are other beans that could be wired into the `instrument` property, you can use `@Qualifier` to specify a bean named `guitar`:

```
@Autowired
@Qualifier("guitar")
private Instrument instrument;
```

As shown here, the `@Qualifier` annotation will try to wire in a bean whose ID matches `guitar`.

On the surface, it would seem that using `@Qualifier` is a means of switching `@Autowired`'s by-type autowiring into explicit by-name wiring. And, as used here, that's effectively what's happening. But it's important to know that `@Qualifier` is really about narrowing the selection of autowire candidate beans. It just so happens that specifying a bean's ID is one way to narrow the selections down to a single bean.

In addition to narrowing by a bean's ID, it's also possible to narrow by a qualifier that's applied to a bean itself. For example, suppose that the `guitar` bean were declared in XML as follows:

```
<bean class="com.springinaction.springidol.Guitar">
  <qualifier value="stringed" />
</bean>
```

Here the `<qualifier>` element qualifies the `guitar` bean as a stringed instrument. But instead of specifying the qualifier in XML, you could have also annotated the `Guitar` class itself with the `@Qualifier` annotation:

```
@Qualifier("stringed")
public class Guitar implements Instrument {
  ...
}
```

Qualifying autowired beans with `String` identifiers, whether they're the bean's ID or some other qualifier, is simple enough. But you can take qualifiers so much further. In fact, you can even create your own custom qualifier annotations.

CREATING CUSTOM QUALIFIERS

To create a custom qualifier annotation, all you need to do is to define an annotation that's itself annotated with `@Qualifier`. For example, let's create our own `@StringedInstrument` annotation to serve as a qualifier. The following listing shows the custom qualifier annotation.

Listing 3.1 Use `@Qualifier` to create your own qualifier annotation.

```
package com.springinaction.springidol.qualifiers;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.springframework.beans.factory.annotation.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument {
}
```

With the `@StringedInstrument` annotation defined, you can now use it instead of `@Qualifier` to annotate `Guitar`:

```
@StringedInstrument
public class Guitar implements Instrument {
  ...
}
```

Then, you can qualify the `@Autowiredinstrument` property with `@StringedInstrument`:

```
@Autowired
@StringedInstrument
private Instrument instrument;
```

When Spring tries to autowire the `instrument` property, it'll narrow the selection of all `Instrument` beans down to just those that are annotated with `@StringedInstrument`. As long as only one bean is annotated with `@StringedInstrument`, it'll be wired into the `instrument` property.

If there's more than one `@StringedInstrument`-annotated bean, then you'll need to provide further qualification to narrow it down. For example, suppose that in addition to the `Guitar` bean, you also have a `HammeredDulcimer` bean which is also annotated with `@StringedInstrument`. One key difference between a guitar and a hammered dulcimer is that guitars are strummed whereas hammered dulcimers are hit with small wooden sticks (called hammers).

So, to qualify the `Guitar` class further, you could define another qualifier annotation called `@Strummed`:

```
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Strummed {
}
```

Now you can annotate the `instrument` property with `@Strummed` to narrow the selection down to strummed string instruments:

```
@Autowired
@StringedInstrument
@Strummed
private Instrument instrument;
```

If the `Guitar` class is the only class annotated with `@Strummed` and `@StringedInstrument`, then it'll be the one injected into `instrument`.

I suppose we could discuss the implications of adding a `Ukelele` or a `Mandolin` bean to the mix, but we have to end this somewhere. Suffice it to say that you'd need further qualification to deal with these additional strummed and stringed instruments.

Spring's `@Autowired` annotation is one way to cut down on the amount of Spring configuration XML. But it does create a Spring-specific dependency within the classes that use it (even if that dependency is just an annotation). Fortunately, Spring also supports a standard Java alternative to `@Autowired`. Let's look at how to use `@Inject` from the Dependency Injection for Java specification.

3.2.2 Applying standards-based autowiring with @Inject

In an effort to unify the programming model among the various dependency injection frameworks, the Java Community Process recently published the Dependency Injection for Java specification. Known in the Java Community Process as JSR-330 or more commonly as *at inject*, this specification brings a common dependency injection model to Java. As of Spring 3, Spring supports the at inject model.[2]

The centerpiece of JSR-330 is the `@Inject` annotation. This annotation is an almost complete drop-in replacement for Spring's `@Autowired` annotation. So, instead of using the Spring-specific `@Autowired` annotation, you might choose to use `@Inject` on the `instrument` property:

[2] Spring isn't alone in its support for JSR-330. Google Guice and Picocontainer also support the JSR-330 model.

```
@Inject
private Instrument instrument;
```

Just like `@Autowired`, `@Inject` can be used to autowire properties, methods, and constructors. Unlike `@Autowired`, `@Inject` doesn't have a `required` attribute. Therefore, `@Inject`-annotated dependencies are expected to be fulfilled, failing with an exception if they're not.

JSR-330 has another trick up its sleeve in addition to the `@Inject` annotation. Rather than inject a reference directly, you could ask `@Inject` to inject a `Provider`. The `Provider` interface enables, among other things, lazy injection of bean references and injection of multiple instances of a bean.

For example, let's say you have a `KnifeJuggler` class that needs to be injected with one or more instances of `Knife`. Assuming that the `Knife` bean is declared as having prototype scope, the following `KnifeJuggler` constructor will be able to retrieve five `Knife` beans:

```
private Set<Knife> knives;

@Inject
public KnifeJuggler(Provider<Knife> knifeProvider) {
  knives = new HashSet<Knife>();
  for (int i = 0; i < 5; i++) {
    knives.add(knifeProvider.get());
  }
}
```

Instead of receiving a `Knife` instance at construction, `KnifeJuggler` will receive a `Provider<Knife>`. At this point, only the provider is injected. No actual `Knife` object will be injected until the `get()` method is called on the provider. In this case, the `get()` method is called five times. And since the `Knife` bean is a prototype, we know that the `Set` of `knives` will be given five distinct `Knife` objects to work it.

QUALIFYING @INJECTED PROPERTIES

As you've seen, `@Inject` and `@Autowired` have a lot in common. And like `@Autowired`, the `@Inject` annotation is prone to ambiguous bean definitions. `@Inject`'s answer to the `@Qualifier` annotation is the `@Named` annotation.

The `@Named` annotation works much like Spring's `@Qualifier`, as you can see here:

```
@Inject
@Named("guitar")
private Instrument instrument;
```

The key difference between Spring's `@Qualifier` and JSR-330's `@Named` is one of semantics. Whereas `@Qualifier` helps narrow the selection of matching beans (using the bean's ID by default), `@Named` specifically identifies a selected bean by its ID.

CREATING CUSTOM JSR-330 QUALIFIERS

As it turns out, JSR-330 has its own `@Qualifier` annotation in the `javax.inject` package. Unlike Spring's `@Qualifier`, the JSR-330 version isn't intended to be used on its

own. Instead, you're expected to use it to create custom qualifier annotations, much as we did with Spring's `@Qualifier`.[3]

For example, the following listing shows a new `@StringedInstrument` annotation that's created using JSR-330's `@Qualifier` instead of Spring's `@Qualifier`.

Listing 3.2 Creating a custom qualifier using JSR-330's `@Qualifier`

```
package com.springinaction.springidol;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument {
}
```

As you can see, the only real difference between listing 3.2 and 3.1 is the `import` statement for the `@Qualifier` annotation. In listing 3.1 we used the one from the `org.springframework.beans.factory.annotation` package. But this time, we're using the standards-friendly `@Qualifier` from the `javax.inject` package. Otherwise, they're virtually the same.

Annotation-based autowiring is great for wiring bean references and reducing `<property>` elements in our Spring XML configuration. But can annotations be used to wire values into `String` and other primitive values?

3.2.3 Using expressions with annotation injection

As long as you're using annotations to autowire bean references into your Spring beans, you may want to also use annotations to wire simpler values. Spring 3.0 introduced `@Value`, a new wiring annotation that lets you wire primitive values such as `int`, `boolean`, and `String` using annotations.

The `@Value` annotation is simple to use but, as you'll soon see, is also powerful. To use it, annotate a property, method, or method parameter with `@Value` and pass in a `String` expression to be wired into the property. For example:

```
@Value("Eruption")
private String song;
```

Here we're wiring a `String` value into a `String` property. But the `String` parameter passed into `@Value` is just an expression—it can evaluate down to any type and thus `@Value` can be applied to just about any kind of property.

Wiring hardcoded values using `@Value` is interesting, but not all that necessary. If you're hardcoding the values in Java code, then why not disregard `@Value` altogether

[3] In fact, the `@Named` annotation is just an annotation that's itself annotated with `@Qualifier`.

and just hardcode the value directly on the property? `@Value` seems like extra baggage in that case.

As it turns out, simple values aren't where `@Value` shines. Instead, `@Value` finds its power with SpEL expressions. Recall that SpEL lets you dynamically evaluate complex expressions, at runtime, into values to be wired into bean properties. That makes `@Value` a powerful wiring option.

For example, rather than hardcoding a static value into the `song` property, let's use SpEL to pull a value from a system property:

```
@Value("#{systemProperties.myFavoriteSong}")
private String song;
```

Now `@Value` shows its stuff. It's not just a courier of static values—it's an effective, annotation-driven method of wiring dynamically evaluated SpEL expressions.

As you can see, autowiring is a powerful technique. Letting Spring automatically figure out how to wire beans together can help you reduce the amount of XML configuration in your application. What's more, autowiring can take decoupling to a whole new level by decoupling bean declarations from each other.

Speaking of rising to new levels, let's now look at bean autodiscovery to see how we can rely on Spring to not only wire beans together, but also automatically figure out which beans should be registered in a Spring context in the first place.

3.3 Automatically discovering beans

When you added `<context:annotation-config>` to your Spring configuration, you told Spring that you wanted it to honor a certain set of annotations in the beans that you declared and to use those beans to guide bean wiring. Even though `<context:annotation-config>` can go a long way toward eliminating most uses of `<property>` and `<constructor-arg>` elements from your Spring configuration, you still must explicitly declare beans using `<bean>`.

But Spring has another trick up its sleeve. The `<context:component-scan>` element does everything that `<context:annotation-config>` does, plus it configures Spring to automatically discover beans and declare them for you. What this means is that most (or all) of the beans in your Spring application can be declared and wired without using `<bean>`.

To configure Spring for autodiscovery, use `<context:component-scan>` instead of `<context:annotation-config>`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/context
     http://www.springframework.org/schema/context/spring-context-3.0.xsd">
```

```
  <context:component-scan
      base-package="com.springinaction.springidol">
  </context:component-scan>

</beans>
```

The `<context:component-scan>` element works by scanning a package and all of its subpackages, looking for classes that could be automatically registered as beans in the Spring container. The `base-package` attribute tells `<context:component-scan>` the package to start its scan from.

So, how does `<context:component-scan>` know which classes to register as Spring beans?

3.3.1 Annotating beans for autodiscovery

By default, `<context:component-scan>` looks for classes that are annotated with one of a handful of special stereotype annotations:

- `@Component`—A general-purpose stereotype annotation indicating that the class is a Spring component
- `@Controller`—Indicates that the class defines a Spring MVC controller
- `@Repository`—Indicates that the class defines a data repository
- `@Service`—Indicates that the class defines a service
- Any custom annotation that is itself annotated with `@Component`

For example, suppose that our application context only has the `eddie` and `guitar` beans in it. We can eliminate the explicit `<bean>` declarations from the XML configuration by using `<context:component-scan>` and annotating the `Instrumentalist` and `Guitar` classes with `@Component`.

First, let's annotate the `Guitar` class with `@Component`:

```
package com.springinaction.springidol;

import org.springframework.stereotype.Component;

@Component
public class Guitar implements Instrument {
  public void play() {
    System.out.println("Strum strum strum");
  }
}
```

When Spring scans the `com.springinaction.springidol` package, it'll find that `Guitar` is annotated with `@Component` and will automatically register it in Spring. By default, the bean's ID will be generated by camel-casing the class name. In the case of `Guitar` that means that the bean ID will be `guitar`.

Now let's annotate the `Instrumentalist` class:

```
package com.springinaction.springidol;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
@Component("eddie")
public class Instrumentalist implements Performer {
  // ...
}
```

In this case, we've specified a bean ID as a parameter to `@Component`. The bean ID would've been "instrumentalist," but to keep it consistent with the previous examples, we've explicitly named it `eddie`.

Annotation-based autodiscovery is just one option available when using `<context:component-scan>`. Let's see how to configure `<context:component-scan>` to look for bean candidates using other means.

3.3.2 Filtering component-scans

As it turns out, `<context:component-scan>` is flexible with regard to how it scans for bean candidates. By adding `<context:include-filter>` and/or `<context:exclude-filter>` subelements to `<context:component-scan>`, you can tweak component-scanning behavior to your heart's content.

To demonstrate component-scan filtering, consider what it would take to have `<context:component-scan>` automatically register all classes that are implementations of `Instrument` using the annotation-based strategy. We'd have to visit the source code for each of the `Instrument` implementations and annotate them with `@Component` (or one of the other stereotype annotations). At the least, that'd be inconvenient. And if we were working with a third-party implementation of `Instrument` we may not even have access to the source code to be able to add that annotation.

So, instead of relying on annotation-based component scanning, you can ask `<context:component-scan>` to automatically register all classes that are assignable to `Instrument` by adding an include filter, as follows:

```
<context:component-scan
    base-package="com.springinaction.springidol">
  <context:include-filter type="assignable"
      expression="com.springinaction.springidol.Instrument"/>
</context:component-scan>
```

The `type` and the `expression` attributes of `<context:include-filter>` work together to define a component-scanning strategy. In this case, we're asking for all classes that are assignable to `Instrument` to be automatically registered as Spring beans. But you can choose from other kinds of filtering, as cataloged in table 3.1.

Table 3.1 Component scanning can be customized using any of five kinds of filters.

Filter type	Description
`annotation`	Filters scan classes looking for those annotated with a given annotation at the type level. The annotation to scan for is specified in the `expression` attribute.
`assignable`	Filters scan classes looking for those that are assignable to the type specified in the `expression` attribute.

Table 3.1 Component scanning can be customized using any of five kinds of filters. *(continued)*

Filter type	Description
`aspectj`	Filters scan classes looking for those that match the AspectJ type expression specified in the `expression` attribute.
`custom`	Uses a custom implementation of `org.springframework.core.type.TypeFilter`, as specified in the `expression` attribute.
`regex`	Filters scan classes looking for those whose class names match the regular expression specified in the `expression` attribute.

Just as `<context:include-filter>` can be used to tell `<context:component-scan>` what it should register as beans, you can use `<context:exclude-filter>` to tell it what not to register. For example, to register all `Instrument` implementations except for those annotated with a custom `@SkipIt` annotation:

```
<context:component-scan
    base-package="com.springinaction.springidol">
  <context:include-filter type="assignable"
      expression="com.springinaction.springidol.Instrument"/>
  <context:exclude-filter type="annotation"
      expression="com.springinaction.springidol.SkipIt"/>
</context:component-scan>
```

When it comes to filtering `<context:component-scan>`, the possibilities are virtually endless. But you'll find that the default annotation-based strategy is the most commonly used. And it'll be the one you'll see most often throughout this book.

3.4 Using Spring's Java-based configuration

Believe it or not, not all developers are fans of XML. In fact, some are card-carrying members of the He-Man XML Haters Club. They'd love nothing more than to rid the world of the dreaded angle bracket. Spring's long history of using XML in its configuration has turned off a few of those who oppose XML.

If you're one who abhors XML, then Spring 3 has something special for you. Now you have the option of configuring a Spring application with almost no XML, using pure Java. And even if you don't hate XML, you may want to try out Spring's Java-based configuration because, as you'll soon see, the Java-based configuration knows a few tricks that its XML counterpart doesn't.

3.4.1 Setting up for Java-based configuration

Even though Spring's Java configuration option enables you to write most of your Spring configuration without XML, you'll still need a minimal amount of XML to bootstrap the Java configuration:

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

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:component-scan
      base-package="com.springinaction.springidol" />

</beans>
```

We've already seen how `<context:component-scan>` automatically registers beans that are annotated with certain stereotype annotations. But it also automatically loads in Java-based configuration classes that are annotated with `@Configuration`. In this case, the `base-package` attribute tells Spring to look in `com.springinaction.springidol` to find classes that are annotated with `@Configuration`.

3.4.2 Defining a configuration class

When we first started looking at Spring's XML-based configuration, I showed you a snippet of XML with the `<beans>` element from Spring's `beans` namespace at its root. The Java-based equivalent to that XML is a Java class annotated with `@Configuration`. For example:

```
package com.springinaction.springidol;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringIdolConfig {

  // Bean declaration methods go here

}
```

The `@Configuration` annotation serves as a clue to Spring that this class will contain one or more Spring bean declarations. Those bean declarations are just methods that are annotated with `@Bean`. Let's see how to use `@Bean` to wire beans using Spring's Java-based configuration.

3.4.3 Declaring a simple bean

In the previous chapter, we used Spring's `<bean>` element to declare a `Juggler` bean whose ID was `duke`. Had we chosen Java-based configuration to wire up the *Spring Idol* beans, the `duke` bean would be defined in a method that's annotated with `@Bean`:

```
@Bean
public Performer duke() {
  return new Juggler();
}
```

This simple method is the Java configuration equivalent of the `<bean>` element we created earlier. The `@Bean` tells Spring that this method will return an object that should be registered as a bean in the Spring application context. The bean will get its ID from the method name. Everything that happens in the method ultimately leads to the creation of the bean.

In this case, the bean declaration is simple. The method creates and returns an instance of `Juggler`. That object will be registered in the Spring application context with an ID of `duke`.

Although this bean declaration method is largely equivalent to the XML version, it illustrates one strength that Spring's Java configuration has over its XML counterpart. In the XML version, both the bean's type and its ID were identified by `String` attributes. The downside of `String` identifiers is that they don't lend themselves to compile-time checking. If we were to rename the `Juggler` class, we may forget to change the XML configuration to match.

In Spring's Java-based configuration, there are no `String` attributes. Both the bean's ID and its type are expressed as part of a method signature. The actual creation of the bean is defined in the method body. Because it's all Java, you gain some benefit in terms of compile-time checking to ensure that your bean's type is a real type and that its ID is unique.

3.4.4 Injecting with Spring's Java-based configuration

If declaring beans with Spring's Java-based configuration is nothing more than writing a method that returns an instance of a class, then how does dependency injection work in Java-based configuration? It's actually simple, following common Java idioms.

For example, let's first look at how to inject values into a bean. Earlier, we saw how to create a `Juggler` bean that juggles 15 beanbags by using the `<constructor-arg>` element in XML configuration. In the Java-based configuration, we can just pass the number directly into the constructor:

```
@Bean
public Performer duke15() {
  return new Juggler(15);
}
```

As you can see, the Spring Java–based configuration feels natural, as it lets you define your beans using Java the way you always have. Setter injection is also natural Java:

```
@Bean
public Performer kenny() {
  Instrumentalist kenny = new Instrumentalist();
  kenny.setSong("Jingle Bells");
  return kenny;
}
```

Wiring simple values is straightforward enough. What about wiring in references to other beans? It's just as easy.

To illustrate, let's first set things up by declaring a `sonnet29` bean in Java:

```
@Bean
private Poem sonnet29() {
  return new Sonnet29();
}
```

This is another simple Java-based bean declaration, not much different than what we've already done with the `duke` bean. Now, let's create a `PoeticJuggler` bean, wiring the `sonnet29` bean in through its constructor:

```
@Bean
public Performer poeticDuke() {
  return new PoeticJuggler(sonnet29());
}
```

Wiring in another bean is a simple matter of referring to that bean's method. But don't let the simplicity fool you. More is going on here than meets the eye.

In Spring Java Configuration, referring to a bean through its declaration method isn't the same as calling the method. If it were, then each time we call `sonnet29()`, we'd get a new instance of that bean. Spring is more clever than that.

By annotating the `sonnet29()` method with `@Bean`, we're telling Spring that we want that method to define a bean to be registered in the Spring application context. Then, whenever we refer to that method in another bean declaration method, Spring will intercept the call to the method and try to find the bean in its context instead of letting the method create a new instance.

3.5 Summary

Over the years, Spring has taken a lot of flak for XML verbosity. Despite the great strides in simplicity that Spring has brought to enterprise Java, a lot of developers haven't been able to look past all those angle brackets.

To answer the critics, Spring offers several ways of reducing and even eliminating Spring configuration XML. In this chapter we've seen how `<property>` and `<constructor-arg>` elements can be replaced with autowiring. Entire `<bean>` configuration elements can be handled automatically by Spring using component scanning. We've also seen how Spring configuration can be expressed in Java instead of XML, eliminating XML from Spring applications altogether.

At this point we've seen several ways to declare beans in Spring and wire their dependencies. In the next chapter, we'll take a look at how Spring supports aspect-oriented programming and see how AOP can be used to embellish beans with behavior that, although important to the functionality of an application, isn't a core concern of the beans that the aspects affect.