Article

What's New in Spring 2.5: Part 1

Posted by Mark Fisher on Nov 19, 2007 09:00 AM
Community Java Topics Web Frameworks Tags Spring

Introduction

Since its inception the Spring Framework has consistently focused on the goal of simplifying enterprise application development while providing powerful, non-invasive solutions to complex problems. With the release of Spring 2.0 just over a year ago, these themes advanced to a new level. XML Schema support and custom namespaces reduce the amount of XML-based configuration. Developers using Java 5 or greater are able to take advantage of Spring libraries that exploit new language features such as generics and annotations. The close integration with AspectJ's expression language enables the non-invasive addition of behavior across well-defined groupings of Spring-managed objects.

The newly released Spring 2.5 continues this trend by offering further simplifications and powerful new features especially for those who are using Java 5 or greater. These features include annotation-driven dependency injection, auto-detection of Spring components on the classpath using annotations rather than XML for metadata, annotation support for lifecycle methods, a new web controller model for mapping requests to annotated methods, support for Junit 4 in the test framework, new additions to the Spring XML namespaces, and more.

This article is the first of a three-part series exploring these new features. The current article will focus on simplified configuration and new annotation-based functionality in the core of the Spring application context. The second article will cover new features available in the web-tier, and the final article will highlight additional features available for integration and testing. Most of the examples depicted within this article series are based upon the Spring PetClinic sample application. That sample has recently been refactored to serve as a showcase of the latest Spring functionality and is included within the Spring 2.5 distribution available at the Spring Framework Download page. View the 'readme.txt' file within the 'samples/petclinic' directory for instructions on building and deploying the PetClinic application. Experimenting with the showcased features in the PetClinic application is probably the best way to master the new techniques discussed here.

Spring support for JSR-250 annotations

The 'Common Annotations for the Java Platform' were introduced with Java EE version 5 and are also included out-of-the-box beginning with Java SE version 6. In May 2006, BEA Systems announced their collaboration with Interface21 on a project called Pitchfork that provided a Spring-based implementation of the Java EE 5 programming model including support for JSR-250 annotations and EJB 3 annotations (JSR-220) for injection, interception, and transactions. As of version 2.5, the Spring Framework *core* now provides support for the following JSR-250 annotations:

- @Resource
- @PostConstruct
- @PreDestroy

With Spring these annotations are supported in any environment - with or without an Application Server - and even for integration testing. Enabling this support is just a matter of registering a single Spring post-processor:

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>
```

The @Resource Annotation

The **@Resource** annotation enables dependency injection of a "named resource". Within a Java EE application, that typically translates to an object bound to the JNDI context. Spring does support this usage of **@Resource** for resolving objects through JNDI lookups, but by default the Spring-managed object whose "bean name" matches the name provided to the **@Resource** annotation will be injected. In the following example, Spring would pass a reference to the Spring-managed object with a bean name of *"dataSource"* to the annotated setter method.

```
@Resource(name="dataSource")
 public void setDataSource(DataSource dataSource) {
   this.dataSource = dataSource;
}
```

It is also possible to annotate a field directly with **@Resource**. By not exposing a setter method, the code is more concise and also provides the additional benefit of enforcing immutability. As demonstrated below, the **@Resource** annotation does not even require an explicit String value. When none is provided, the name of the field will be used as a default.

```
@Resource
private DataSource dataSource; // inject the bean named 'dataSource'
```

When applied to a setter method, the default name would be derived from the corresponding property. In other words, a method named **'setDataSource'** would also resolve to the property named **'dataSource'**.

```
private DataSource dataSource;
@Resource
public void setDataSource(DataSource dataSource) {
   this.dataSource = dataSource;
}
```

When using **@Resource** without an explicitly provided name, if no Spring-managed object is found for the default name, the injection mechanism will fallback to a type-match. If there is exactly one Spring-managed object matching the dependency's type, then it will be injected. This feature can be disabled by setting the **'fallbackToDefaultTypeMatch'** property of the **CommonAnnotationBeanPostProcessor** to 'false' (it is 'true' by default).

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor">
   <property name="fallbackToDefaultTypeMatch" value="false"/>
</bean>
```

As mentioned above, Spring does provide support for JNDI-lookups when resolving dependencies that are annotated with **@Resource**. To force direct JNDI lookups for all dependencies annotated with **@Resource**, set the **'alwaysUseJndiLookup'** flag of the **CommonAnnotationBeanPostProcessor** to 'true' (it is 'false' by default).

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor">
   <property name="alwaysUseJndiLookup" value="true"/>
</bean>
```

Alternatively, to enable lookup based upon global JNDI names specified as 'resource-ref-mappings', provide the **'mappedName'** attribute within the **@Resource** annotation. Even when the target object is actually a JNDI resource, it is the recommended practice to still reference a Spring-managed object thereby providing a level of indirection and hence a lesser degree of coupling. With the namespace additions available since Spring 2.0, a bean definition that delegates to Spring for handling the JNDI lookup is trivial and concise:

```
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/petclinic"/>
```

The advantage of this approach is that the level of indirection provides for greater deployment flexibility. For example, a standalone system test environment should not require a JNDI registry. In such a case, the following alternate bean definition could be provided within the system test configuration:

```
<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}"/>
```

As an aside, in the example above the actual JDBC connection properties are resolved from a properties file where the keys match the provided ${placeholder} tokens. This is accomplished by registering a Spring **BeanFactoryPostProcessor** implementation called **PropertyPlaceholderConfigurer**. This is a commonly used technique for externalizing those properties - often environment-specific ones - that may need to change more frequently than the rest of the configuration.

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
   <property name="location" value="classpath:jdbc.properties"/>
</bean>
```

With the addition of the 'context' namespace in Spring 2.5, a more concise alternative for the property placeholder configuration is available:

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

Lifecycle Annotations: @PostConstruct and @PreDestroy

The **@PostConstruct** and **@PreDestroy** annotations can be used to trigger Spring initialization and destruction callbacks respectively. This feature extends but does not replace the two options for providing such callbacks in Spring versions prior to 2.5. The first option is to implement one or both of Spring's **InitializingBean** and **DisposableBean** interfaces. Each of those interfaces requires a single callback method implementation (**afterPropertiesSet()** and **destroy()** respectively). The interface-based approach takes advantage of Spring's ability to automatically recognize any Spring managed object implementing those interfaces and therefore requires no additional configuration. On the other hand, a key goal of Spring is to be as non-invasive as possible. Therefore instead of implementing Spring-specific interfaces, many Spring users have taken advantage of the second option which is to provide their own initialization and destruction methods. While less invasive, the drawback of that approach is that it requires explicit declaration of **'init-method'** and/or **'destroy-method'** attributes on the **'bean'** element. That explicit configuration is sometimes necessary, such as when the callbacks need to be invoked on code that is outside of the developer's control. The PetClinic application demonstrates this scenario. When it is run with its JDBC configuration, a third party **DataSource** is used, and a **'destroy-method'** is declared explicitly. Also notice that the standalone connection-pooling **DataSource** is yet another deployment option for the **'dataSource'** and does not require any code changes.

```
<bean id="dataSource"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close"
        p:driverClassName="${jdbc.driverClassName}"
        p:url="${jdbc.url}"
        p:username="${jdbc.username}"
        p:password="${jdbc.password}"/>
```

With Spring 2.5, if an object requires invocation of a callback method upon initialization, that method can be annotated with the **@PostConstruct** annotation. For example, imagine a background task that needs to start polling a file directory upon startup.

```
public class FilePoller {

   @PostConstruct
   public void startPolling() {
         ...
   }
   ...
}
```

Similarly, a method annotated with **@PreDestroy** on a Spring-managed object will be invoked when the application context hosting that object is closed.

```
public class FilePoller {

   @PreDestroy
   public void stopPolling() {
         ...
   }
   ...
}
```

With the addition of support for the JSR-250 annotations, Spring 2.5 now combines the advantages of its two previous lifecycle method alternatives. Adding **@PostConstruct** and **@PreDestroy** as method-level annotations is sufficient for triggering the callbacks within a Spring managed context. In other words, no additional XML-based configuration is necessary. At the same time, the annotations are part of the Java language itself (and even included within Java SE as of version 6) and thus require no Spring-specific imports. The annotations have the added benefit of indicating semantics that should be understood within other environments, and over time Java developers can likely expect to see these annotations used more frequently within third-party libraries. Finally, one interesting consequence of the annotation-based lifecycle callbacks is that *more than one* method may carry either annotation, and all annotated methods will be invoked

To enable all of the behavior as described above for the **@Resource**, **@PostConstruct**, and **@PreDestroy** annotations, provide a single bean definition for Spring's **CommonAnnotationBeanPostProcessor** as shown previously. An even more concise option is possible with the new 'context' namespace in 2.5:

```
<context:annotation-config/>
```

Including that single element will not only register a **CommonAnnotationBeanPostProcessor**, but it will also enable the autowiring behavior as described in the section that follows. The **CommonAnnotationBeanPostProcessor** even provides support for **@WebServiceRef** and **@EJB** annotations. These will be covered in the third article of this series along with other new Spring 2.5 features for enterprise integration.

Fine-Grained Autowiring with Annotations

Documentation covering Spring's support for autowiring has often been accompanied with caveats due to the coarse granularity of the autowiring mechanism. Prior to Spring 2.5, autowiring could be configured for a number of different approaches: constructor, setters by type, setters by name, or autodetect (where Spring chooses to autowire either a constructor or setters by type). These various options do offer a large degree of flexibility, but none of them offer very fine-grained control. In other words, prior to Spring 2.5, it has not been possible to autowire a specific subset of an object's setter methods or to autowire some of its properties by type and others by name. As a result, many Spring users have recognized the benefits of autowiring for prototyping and testing, but when it comes to maintaining and supporting systems in production, most agree that the added verbosity of explicit configuration is well worth the clarification it affords.

However, Spring 2.5 dramatically changes the landscape. As described above, the autowiring choices have now been extended with support for the JSR-250 **@Resource** annotation to enable autowiring of named resources on a per-method or per-field basis. However, the **@Resource** annotation alone does have some limitations. Spring 2.5 therefore introduces an **@Autowired** annotation to further increase the level of control. To enable the behavior described in this section, register a single bean definition:

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>
```

Alternatively, the 'context' namespace provides a more concise alternative as shown previously. This will enable *both* post-processors discussed in this article (**AutowiredAnnotationBeanPostProcessor** and **CommonAnnotationBeanPostProcessor**) as well as the annotation-based post-processors that were introduced in Spring 2.0: **RequiredAnnotationBeanPostProcessor** and **PersistenceAnnotationBeanPostProcessor**.

```
<context:annotation-config/>
```

With the **@Autowired** annotation, it is possible to inject dependencies that match by type. This behavior is enabled for fields, constructors, and methods. In fact, autowired methods do not have to be 'setter' methods and can even accept multiple parameters. The following is perfectly acceptable:

```
@Autowired
public void setup(DataSource dataSource, AnotherObject o) { ... }
```

By default, dependencies marked with the **@Autowired** annotation are treated as required. However, it is also possible to declare any of them as optional by setting the **'required'** attribute to **'false'**. In the following example, **DefaultStrategy** will be used if *no* Spring-managed object of type **SomeStrategy** is found within the context.

```
@Autowired(required=false)
private SomeStrategy strategy = new DefaultStrategy();
```

Autowiring by type can obviously result in ambiguities when the Spring context contains more than one object of the expected type. By default, the autowiring mechanism will fail if there is not exactly one bean for a required dependency. Likewise for any optional properties, it will fail if more than one candidate is available (if optional and zero candidates are available, then it will simply skip the property). There are a number of configuration options for avoiding these conflicts.

When there is one *primary* instance of a given type within the context, the bean definition for that type should contain the `'primary'` attribute. This approach works well when other instances may be available in the context, yet those non-primary instances are always explicitly configured.

```
<bean id="dataSource" primary="true" ... />
```

When more control is needed, any autowired field, constructor argument, or method parameter may be further annotated with a **@Qualifier** annotation. The qualifier may contain a **String** value in which case Spring will attempt to match by name.

```
@Autowired
@Qualifier("primaryDataSource")
private DataSource dataSource;
```

The main reason that **@Qualifier** exists as a separate annotation is so that it can be applied at the level of a constructor argument or method parameter while the **@Autowired** annotation is available on the constructor or method itself.

```
@Autowired
public void setup(@Qualifier("primaryDataSource") DataSource dataSource, AnotherObject o) { ... }
```

The fact that **@Qualifier** is a separate annotation provides even more benefits with regard to customization. User-defined annotations may also play the role of qualifier in the autowiring process. The simplest way to accomplish this is to annotate the custom annotation with **@Qualifier** itself as a meta-annotation.

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

Custom annotations may optionally include a value for matching by name but more commonly would be used as "marker" annotations or define a value that provides some further meaning to the qualifier process. For example, the excerpt below depicts a field that should be autowired with a qualified candidate among those that match by type.

```
@Autowired
@VetSpecialty("dentistry")
private Clinic dentistryClinic;
```

When using XML configuration for the target of this dependency resolution, **'qualifier'** sub-elements may be added to the bean definition. In the next section on component-scanning, a non-XML alternative will be presented.

```
<bean id="dentistryClinic" class="samples.DentistryClinic">
   <qualifier type="example.VetSpecialty" value="dentistry"/>
</bean>
```

To avoid any dependency on the **@Qualifier** annotation whatsoever, provide a **CustomAutowireConfigurer** bean definition within the Spring context and register any custom annotation types directly:

```
<bean class="org.springframework.beans.factory.annotation.CustomAutowireConfigurer">
   <property name="customQualifierTypes">
         <set>
                <value>example.VetSpecialty</value>
         </set>
   </property>
</bean>
```

Now that the custom qualifier has been explicitly declared, the **@Qualifier** meta-annotation is no longer required.

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

On a related note, it is even possible to replace the **@Autowired** annotation itself when configuring the **AutowiredAnnotationBeanPostProcessor**.

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor">
   <property name="autowiredAnnotationType" value="example.Injected"/>
</bean>
```

In a majority of cases, the ability to define custom "marker" annotations combined with the options of matching by name or other semantic value should be sufficient for achieving fine-grained control of the autowiring process. However, Spring also provides support for any number of arbitrary attributes on qualifier annotations. For example, the following is a hypothetical example of a very fine-grained qualifier.

```
@SpecializedClinic(species="dog", breed="poodle")
private Clinic poodleClinic;
```

The custom qualifier implementation would define those attributes.

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

   String species();

   String breed();

}
```

The custom qualifier's attributes can then match against **'attribute'** sub-elements of the **'qualifier'** annotation within the XML of a bean definition. These elements are used to provide key/value pairs.

```
<bean id="poodleClinic" class="example.PoodleClinic">
   <qualifier type="example.SpecializedClinic">
         <attribute key="species" value="dog"/>
         <attribute key="breed" value="poodle"/>
   </qualifier>
</bean>
```

All of the autowiring demonstrated so far has been for single instances, but collections are supported as well. Any time it's necessary to get all Spring-managed objects of a certain type within the context, simply add the **@Autowired** annotation to a strongly-typed Collection.

```
@Autowired
private List<Clinic> allClinics;
```

One final feature that is worth pointing out in this section is the use of autowiring in place of Spring's "Aware" interfaces. Prior to Spring 2.5, if an object requires a reference to the Spring context's **ResourceLoader**, it can implement **ResourceLoaderAware** thereby allowing Spring to provide this dependency via the **setResourceLoader(ResourceLoader resourceLoader)** method. This same technique applies for obtaining a reference to the Spring-managed **MessageSource** and even the **ApplicationContext** itself. For Spring 2.5 users, this behavior is now fully supported through autowiring (note that the inclusion of these Spring-specific dependencies should always be carefully considered and typically only used within "infrastructure" code that is clearly separated from business logic).

```
@Autowired
private MessageSource messageSource;

@Autowired
private ResourceLoader resourceLoader;

@Autowired
private ApplicationContext applicationContext;
```

Auto-Detection of Spring Components

Beginning with version 2.0, Spring introduced the concept of "stereotype" annotations with the **@Repository** annotation serving as a marker for data access code. Spring 2.5 adds two new annotations - **@Service** and **@Controller** - to complete the role designations for a common three-tier architecture (data access objects, services, and web controllers). Spring 2.5 also introduces the generic **@Component** annotation which the other stereotypes logically extend. By clearly indicating application roles, these stereotypes facilitate the use of Spring AOP and post-processors for providing additional behavior to the annotated objects based on those roles. For example, Spring 2.0 introduced the **PersistenceExceptionTranslationPostProcessor** to automatically enable data access exception translation for any object carrying the **@Repository** annotation.

These same annotations can also be used in conjunction with another new feature of Spring 2.5: auto-detection of components on the classpath. Although XML has traditionally been the most popular format for Spring metadata, it is not the only option. In fact, the Spring container's internal metadata representation is pure Java, and when XML is used to define Spring-managed objects, those definitions are parsed and converted to Java objects prior to the instantiation process. One significant new capability of Spring 2.5 is the support for reading that metadata from source-level annotations. The autowiring mechanisms described thus far make use of annotation metadata for injecting dependencies but still require registration of at least a minimal "bean definition" in order to provide the implementation class of each Spring-managed object. The component scanning functionality can remove the need for even that minimal bean definition in XML.

As seen above, Spring's annotation-driven autowiring can significantly reduce the amount of XML without sacrificing fine-grained control. The component detection mechanism takes this even further. It is not necessary to completely supplant configuration in XML, rather the component scanning can operate alongside XML metadata to simplify the overall configuration. This possibility of combining XML and annotation-driven techniques can lead to a well-balanced approach as demonstrated by the 2.5 version of the PetClinic sample application. There, the infrastructural components (data source, transaction manager, etc) are defined in XML along with externalized properties as described above. The data access tier objects are also defined partially in XML, but their configuration also takes advantage of the **@Autowired** annotations to simplify the injection of dependencies. Finally, the web tier "controllers" are not explicitly defined in XML at all. Instead the following configuration is used to trigger the auto-detection of all web controllers:

```
<context:component-scan base-package="org.springframework.samples.petclinic.web"/>
```

Notice that the 'base-package' attribute is provided. The default matching rules for component-scanning will recursively detect any of Spring's stereotype annotations on classes within that package (multiple packages can be provided in a comma-separated list). Therefore, the various controller implementations for the PetClinic sample application are all annotated with **@Controller** (one of Spring's *built-in* stereotypes). Here is an example:

```
@Controller
public class ClinicController {

   private final Clinic clinic;

   @Autowired
   public ClinicController(Clinic clinic) {
         this.clinic = clinic;
   }
 ...
```

Auto-detected components are registered with the Spring container just as if they had been defined in XML. As depicted above, those objects can in turn make use of annotation-driven autowiring.

The component scanner's matching rules can also be customized with filters for including or excluding components based on type, annotation, AspectJ expression, or regular expressions for name patterns. The default stereotypes can also be disabled. For example, a test configuration may ignore the default stereotypes and instead auto-detect any class whose name starts with Stub or which includes the **@Mock** annotation:

```
<context:component-scan base-package="example" use-default-filters="false">
   <context:include-filter type="aspectj" expression="example..Stub*"/>
   <context:include-filter type="annotation" expression="example.Mock"/>
</context:component-scan>
```

Type matching restrictions can be controlled with exclusion filters as well. For example, to rely on the default filters *except* for the **@Repository** annotation, then add an **exclude-filter**.

```
<context:component-scan base-package="example">
   <context:exclude-filter type="annotation" expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

Clearly it is possible to extend the component scanning in a number of ways to register your own custom types. The stereotype annotations are the simplest option, so the notion of stereotype is therefore extensible itself. As mentioned earlier, **@Component** is the *generic* stereotype indicator that the **@Repository**, **@Service**, and **@Controller** annotations "logically" extend. It just so happens that **@Component** can be provided as a meta-annotation (i.e. an annotation declared on another annotation), and any custom annotation that has the **@Component** meta-annotation will be automatically detected by the default matching rules of the scanner. An example will hopefully reveal that this is much simpler than it sounds.

Recall the hypothetical background task that was described in the section above covering the **@PostConstruct** and **@PreDestroy** lifecycle annotations. Perhaps an application has a number of such background tasks, and those task instances would typically require XML bean definitions in order to be registered with the Spring context and have their lifecycle methods invoked at the right time. With component scanning, there is no longer a need for those explicit XML bean definitions. If the background tasks all implement the same interface or follow a naming convention, then **include-filters** could be used. However, an even simpler approach is to create an annotation for these task objects and provide the **@Component** meta-annotation.

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface BackgroundTask {
   String value() default "";
}
```

Then provide the custom stereotype annotation in any background task's class definitions.

```
@BackgroundTask
public class FilePoller {

   @PostConstruct
   public void startPolling() {
         ...
   }

   @PreDestroy
   public void stopPolling() {
         ...
   }
   ...
}
```

The generic **@Component** annotation could just as easily have been provided instead, but the custom annotation technique provides an opportunity for using meaningful, domain-specific names. These domain-specific annotations then provide further opportunities such as using an AspectJ pointcut expression to identify all background tasks for the purpose of adding advice that monitors the activity of those tasks.

By default, when a component is detected, Spring will automatically generate a "bean name" using the non-qualified class name. In the previous example, the generated bean name would be "filePoller". However, for any class that is annotated with one of Spring's stereotype annotations (**@Component**, **@Repository**, **@Service**, or **@Controller**) *or* any other annotation that is annotated with `@Component` as a meta-annotation (such as **@BackgroundTask** in the above example), the **'value'** attribute can be explicitly specified for the stereotype annotation, and the instance will then be registered within the context with that value as its "bean name". In the following example the name would be "petClinic" instead of the default generated name of "simpleJdbcClinic".

```
@Service("petClinic")
public class SimpleJdbcClinic {
   ...
}
```

Likewise, the bean name generated for the following revised version of the **FilePoller** would be "poller" instead of "filePoller".

```
@BackgroundTask("poller")
 public class FilePoller {
   ...
}
```

While all Spring-managed objects are treated as *singleton* instances by default, it is sometimes necessary to specify an alternate "scope" for an object. For example, in the web-tier a Spring-managed object may be bound to 'request' or 'session' scope. As of version 2.0, Spring's scope mechanism is even extensible so that custom scopes can be registered with the application context. Within an XML configuration, it's simply a matter of including the **'scope'** attribute and the name of the scope.

```
<bean id="shoppingCart" class="example.ShoppingCart" scope="session">
   ...
</bean>
```

With Spring 2.5, the same can be accomplished for a scanned component by providing the **@Scope** annotation.

```
@Component
@Scope("session")
public class ShoppingCart {
   ...
}
```

One final topic to address here is the simplification of qualifier annotations when using component-scanning. In the previous section, the following object was used as an example of autowiring with a custom qualifier annotation:

```
@VetSpecialty("dentistry")
private Clinic dentistryClinic;
```

That same example then featured use of a **'qualifier'** element within the XML on the intended target bean definition for that dependency. When using component-scanning, the XML metadata is not necessary. Instead, the custom qualifier may be included as a type-level annotation in the target class definition. An alternative example with a scanned **@Repository** instance as the dependency would thus appear as follows:

```
@Repository
@VetSpecialty("dentistry")
public class DentistryClinic implements Clinic {
   ...
}
```

Finally, for the previous example that featured custom qualifier annotations *with attributes*, the non-XML equivalent for that dependency's target would be:

```
@Repository
@SpecializedClinic(species="dog", breed="poodle")
public class PoodleClinic implements Clinic {
   ...
}
```

Conclusion

Spring 2.5 offers significant new functionality in a number of areas. The primary focus of this article has been on simplifying configuration by harnessing the power of Java annotations. Spring supports 'Common Annotations' as defined in JSR-250 while providing additional annotations for even more fine-grained control of the autowiring process. Spring 2.5 also extends the 'stereotype' annotations that began with Spring 2.0's **@Repository**, and all of these stereotypes can be used in conjunction with the new component-scanning functionality. XML-based configuration is still fully supported, and Spring 2.5 introduces a new 'context' namespace that offers more concise syntax for common configuration scenarios. In fact, the support for seamlessly combining XML and annotation-based configuration enables a well-balanced overall approach. Complex configuration of infrastructure can be defined in modular XML files while the progressively higher layers of an application stack can benefit from more annotation-based techniques - all within the same Spring 2.5 application context.

Stay tuned for the next article in this series, which will cover powerful new annotation-based functionality in the Spring web tier.

Bookmark digg+, reddit+, del.icio.us+, dzone+, facebook+ slashdot+

23 comments **Watch Thread** **Reply**

- **Missing/hidden XML** by Matt Raible Posted Nov 19, 2007 9:51 PM
  - **Re: Missing/hidden XML** by Mark Fisher Posted Nov 19, 2007 10:32 PM
    - **Re: Missing/hidden XML** by Diana Plesa Posted Nov 20, 2007 3:05 AM
  - **Re: Missing/hidden XML** by mohan raj Posted Nov 29, 2007 5:11 AM
- **Good Stuff** by Ray Krueger Posted Nov 20, 2007 8:01 AM
  - **Re: Good Stuff** by Mark Fisher Posted Nov 20, 2007 8:35 AM
  - **Re: Good Stuff** by Darryl Pentz Posted Mar 4, 2008 2:53 PM
    - **Re: Good Stuff** by Archie Cobbs Posted May 12, 2008 9:40 AM
- **Grate Guice alternative** by Tom Nichols Posted Nov 20, 2007 8:11 AM
- **Annotations vs. XML** by Geoffrey Wiseman Posted Nov 20, 2007 8:47 AM
  - **Re: Annotations vs. XML** by Rod Johnson Posted Nov 20, 2007 4:47 PM
    - **Re: Annotations vs. XML** by Jörg Gottschling Posted Nov 23, 2007 1:30 AM
      - **Re: Annotations vs. XML** by Rod Johnson Posted Nov 26, 2007 2:34 PM
- **lament the passing of design and OO programming** by James Richardson Posted Nov 27, 2007 6:05 PM
- **Great Job** by Khaled Habiburahman Posted Nov 28, 2007 2:39 AM
  - **Re: Great Job** by mohan raj Posted Nov 29, 2007 5:12 AM
- **the 2nd part and 3rd part** by Nantian Lotus Posted Feb 5, 2008 8:43 AM
  - **Re: the 2nd part and 3rd part** by Lou Sacco Posted Mar 27, 2008 2:06 PM
- **2nd and 3rd parts** by Gregor Morrison Posted Jun 3, 2008 4:58 AM
  - **Re: 2nd and 3rd parts** by Mohammad Naqvi Posted Aug 21, 2008 1:38 PM
    - **the last part** by john wang Posted Sep 8, 2008 3:38 PM
      - **Re: the last part** by Lukasz Budnik Posted Nov 19, 2008 6:00 AM
    - **Re: 2nd and 3rd parts** by Johan Pelgrim Posted Dec 10, 2008 6:03 AM

Sort by date descending

Missing/hidden XML

Nov 19, 2007 9:51 PM by **Matt Raible**

There seems to be a fair amount of missing or hidden XML in this article. If you look at the HTML source, you'll see where some XML has not been properly escaped.

Reply

Re: Missing/hidden XML

Nov 19, 2007 10:32 PM by **Mark Fisher**

Matt, You are right. Someone must have edited this recently, because the XML was showing up properly before. I just sent a mail to the editor. Thanks, Mark

Reply

Re: Missing/hidden XML

Nov 20, 2007 3:05 AM by **Diana Plesa**

Hi Matt, Mark The XML has been fixed now. Best Diana

Reply

Good Stuff

Nov 20, 2007 8:01 AM by **Ray Krueger**

I am really looking forward to using these new features, great job guys. By the way Mark, your sudden and unexplained use of the p: namespace might freak people out. In your explanation of the lifecycle annotations you declare the datasource using the p: namespace trick from http://blog.interface21.com /main/2006/11/25/xml-syntax-sugar-in-spring-20/ Unfortunately, many people probably aren't aware of this feature and it isn't actually mentioned in the article.

Reply

Grate Guice alternative

Nov 20, 2007 8:11 AM by **Tom Nichols**

This is great -- I always liked the annotation-driven style of Guice but didn't want to abandon Spring just for that. Hooray for less XML!

Reply

Re: Good Stuff

Nov 20, 2007 8:35 AM by **Mark Fisher**

Ray, Thanks for pointing out that blog for the 'p' namespace. Hopefully that will serve as a 'footnote' now for anyone who may be confused by its usage. Those examples are taken directly from the PetClinic application by the way. Thanks, Mark

Reply

Annotations vs. XML

Nov 20, 2007 8:47 AM by **Geoffrey Wiseman**

I have to admit, I'm content to use XML for wiring, myself -- although perhaps when I try the annotations, I'll discover more advantages than I expect. I seem to be in the minority here. That said, I was more impressed by the features here than I expected to be, so next time I start some Spring config., I might give this a try. Sorry about the XML; that might have been my fault, I changed some metadata after the initial publishing, and there are some quirks in the publishing process that Diana's better at handling. ;) I'm looking forward to the next two parts of the article.

Reply

Re: Annotations vs. XML

Nov 20, 2007 4:47 PM by **Rod Johnson**

Geoffrey

> I have to admit, I'm content to use XML for wiring, myself -- although perhaps when I try the annotations, I'll discover more advantages than I expect. I seem to be in the minority here.

The goal of Spring is to be the ultimate component model. That component model can be configured in different ways. There is no perfect one size fits all approach to configuration. Different contributions are merged together by the container. Certainly, annotations have an important place. However, my experience in practice (long predating my creation of Spring) has been that you *need* to externalize significant parts of your configuration from Java code. The annotation support in Spring 2.5 is very slick and definitely makes Spring a better product. You can mix and match annotation-driven and XML (and other) configuration so that you can use the appropriate solution for each problem. I'm proud that each version of Spring has made applications easier to build. I recently presented on Configuring the Spring Container at QCon San Francisco, discussing alternative configuration options and best practices. The slides are available as PDF.

Reply

Re: Annotations vs. XML

Nov 23, 2007 1:30 AM by **Jörg Gottschling**

The annotation @Resource is a little confusing in the context of spring. As I saw it I first thought it will be used for ressources and not for beans. Like that: @Ressource("file:out/example.txt") public void setOutput(Ressource ressource) {...}

Reply

Re: Annotations vs. XML

Nov 26, 2007 2:34 PM by **Rod Johnson**

Jorg I agree that @Resource is a poor name for the annotation, but we didn't choose it... Rgds Rod

Reply

lament the passing of design and OO programming

Nov 27, 2007 6:05 PM by **James Richardson**

as now i can turn everything into a FactoryBeanImpl. @Bonkers(Retention.FORALLTIME) afterPropertiesSet()

Reply

Great Job

Nov 28, 2007 2:39 AM by **Khaled Habiburahman**

Great Job, keep it up Thanks

Reply

Re: Missing/hidden XML

Nov 29, 2007 5:11 AM by **mohan raj**

Good Artical...

Reply

Re: Great Job

Nov 29, 2007 5:12 AM by **mohan raj**

Of course...Great

Reply

the 2nd part and 3rd part

Feb 5, 2008 8:43 AM by **Nantian Lotus**

Good material. But where can I find the 2nd part and 3rd part of this series. Thanks.

Reply

Re: Good Stuff

Mar 4, 2008 2:53 PM by **Darryl Pentz**

Not to mention how it's never mentioned how to actually create an instance of the object that is injected by Spring. We see the annotated class, and the configuration XML, but no idea how to construct an instance of the annotated class. Does it appear as if by magic? Is it just me, but why are there no examples of this. Everything is just assumed to originate as some implicit class of the Spring framework, like a controller or some such. I'd like to use this uber-magic of Spring, but how do I create instances of my own 'controller-type' classes automagically injected by Spring. Is this a state secret perhaps? Sorry, just frustrated by the lack of examples, clearly.

Reply

Re: the 2nd part and 3rd part

Mar 27, 2008 2:06 PM by **Lou Sacco**

Agreed...this would be very useful.

Reply

Re: Good Stuff

May 12, 2008 9:40 AM by **Archie Cobbs**

Darryl Pentz writes:

> Not to mention how it's never mentioned how to actually create an instance of the object that is injected by Spring. We see the annotated class, and the configuration XML, but no idea how to construct an instance of the annotated class. Does it appear as if by magic?

I had the same question the first time I read this article (too quickly). The answer lies in the "Auto-Detection of Spring Components". The key "magic" is this tag:

```
<context:component-scan ... />
```

which causes Spring to automatically go hunting through your JARs looking for specially-annotated classes. When it finds them, it auto-constructs instances and adds them to your application context. The net effect is a purely annotation-driven (zero XML) way to add and configure individual beans.

Reply

2nd and 3rd parts

Jun 3, 2008 4:58 AM by **Gregor Morrison**

Mark, great work, are parts 2 and 3 available yet?

Reply

Re: 2nd and 3rd parts

Aug 21, 2008 1:38 PM by **Mohammad Naqvi**

Hey Greg, Did you hear anything about the 2nd and 3rd parts yet?

Reply

the last part

Sep 8, 2008 3:38 PM by **john wang**

"While all Spring-managed objects are treated as singleton instances by default, it is sometimes necessary to specify an alternate "scope" for an object. " I believe with the @PostConstruct @PreDestroy, the default should change to "request" or "prototype", not singleton any more... otherwise it is very possibly not thread-safe.... And is there any plan to support more lifecycle/AOP like annoations? like @PreInvoke @PostInvoke....

Reply

Re: the last part

Nov 19, 2008 6:00 AM by **Lukasz Budnik**

I wrote a series of posts called "Spring for JEE developers" I describe some of Spring 2.5 new features like partial implementations of JSR 220 and JSR 250. If someone is interested here they are: http://jee-bpel-soa.blogspot.com/2008/11/spring-for-jee-developers-jsr-250.html http://jee-bpel-soa.blogspot.com/2008/11/spring-for-jee-developers-jpa.html http://jee-bpel-soa.blogspot.com/2008/11/spring-for-jee-developers-stateless-ejb.html best regards

Reply

Re: 2nd and 3rd parts

Dec 10, 2008 6:03 AM by **Johan Pelgrim**

Here's the second part (haven't found the third part yet) http://www.infoq.com/articles/spring-2.5-ii-spring-mvc

Reply