Core Spring Annotations

Context Configuration Annotations

These annotations are used by Spring to guide creation and injection of beans.

ANNOTATION	USE	DESCRIPTION
@Autowired	Constructor, Field, Method	Declares a constructor, field, setter method, or configuration metho type. Items annotated with @Autowired do not have to be public.
@Configurable	Type	Used with <context:springconfigured> to declare types whose prope injected, even if they are not instantiated by Spring. Typically used t properties of domain objects.
@Order	Type, Method, Field	Defines ordering, as an alternative to implementing the org. springframework.core.Ordered interface.
@Qualifier	Field, Parameter, Type, Annotation Type	Guides autowiring to be performed by means other than by type.
@Required	Method (setters)	Specifies that a particular property must be injected or else the conf
@Scope	Type	Specifies the scope of a bean, either singleton, prototype, request, se custom scope.

Autowiring Bean Properties

A typical Spring bean might have its properties wired something like this:

```
<bean id="pirate" class="Pirate">
<constructor-arg value="Long John Silver" />
<property name="treasureMap" ref="treasureMap" />
</bean>
```

But it's also possible to have Spring automatically inject a bean's properties from other beans in the context. For example, if the Pirate class were annotated with @Autowired like this...

```
public class Pirate {
private String name;
private TreasureMap treasureMap;
public Pirate(String name) { this.name = name; }
@Autowired
public void setTreasureMap(TreasureMap treasureMap) {
this.treasureMap = treasureMap;
}
}
```

...and if you were to configure annotation configuration in Spring using the <context:annotation-configuration> element like this...

```
<beans ... >
<bean id="pirate" class="Pirate">
<constructor-arg value="Long John Silver" />
</bean>
<bean id="treasureMap" class="TreasureMap" />
<context:annotation-config />
</beans>
```

...then the "treasureMap" property will be automatically injected with a reference to a bean whose type is assignable to TreasureMap (in this case, the bean whose ID is "treasureMap").

Autowiring Without Setter Methods

@Autowired can be used on any method (not just setter methods). The wiring can be done through any method, as illustrated here:

```
@Autowired
public void directionsToTreasure(TreasureMap
treasureMap) {
this.treasureMap = treasureMap;
}
```

And even on member variables:

```
@Autowired
private TreasureMap treasureMap;
```

To resolve any autowiring ambiguity, use the @Qualifier attribute with @Autowired.

```
@Autowired
@Qualifier("mapToTortuga")
private TreasureMap treasureMap;
```

Ensuring That Required Properties are Set

To ensure that a property is injected with a value, use the @Required annotation:

```
@Required
public void setTreasureMap(TreasureMap treasureMap) {
```

```
this.treasureMap = treasureMap;
}
```

In this case, the "treasureMap" property must be injected or else Spring will throw a BeanInitializationException and context creation will fail.

Stereotyping Annotations

These annotations are used to stereotype classes with regard to the application tier that they belong to. Classes that are annotated with one of these annotations will automatically be registered in the Spring application context if <context:component-scan> is in the Spring XML configuration.

In addition, if a PersistenceExceptionTranslationPostProcessor is configured in Spring, any bean annotated with @Repository will have SQLExceptions thrown from its methods translated into one of Spring's unchecked DataAccessExceptions.

ANNOTATION	USE	DESCRIPTION
@Component	Type	Generic stereotype annotation for any Spring-managed component.
@Controller	Type	Stereotypes a component as a Spring MVC controller.
@Repository	Type	Stereotypes a component as a repository. Also indicates that SQLExceptions thrown f component's methods should be translated into Spring DataAccessExceptions.
@Service	Type	Stereotypes a component as a service.

Automatically Configuring Beans

In the previous section, you saw how to automatically wire a bean's properties using the @Autowired annotation. But it is possible to take autowiring to a new level by automatically registering beans in Spring. To get started with automatic registration of beans, first annotate the bean with one of the stereotype annotations, such as @Component:

```
@Component
public class Pirate {
private String name;
private TreasureMap treasureMap;
public Pirate(String name) { this.name = name; }
@Autowired
public void setTreasureMap(TreasureMap treasureMap) {
this.treasureMap = treasureMap;
}
}
```

Then add <context:component-scan> to your Spring XML configuration:

```
<context:component-scan
base-package="com.habuma.pirates" />
```

The base-package annotation tells Spring to scan com.habuma. pirates and all of its subpackages for beans to automatically register.

You can specify a name for the bean by passing it as the value of @Component.

```
@Component("jackSparrow")
public class Pirate { ... }
```

Specifying Scope For Auto-Configured Beans

By default, all beans in Spring, including auto-configured beans, are scoped as singleton. But you can specify the scope using the @Scope annotation. For example:

```
@Component
@Scope("prototype")
public class Pirate { ... }
```

This specifies that the pirate bean be scoped as a prototype bean.

Creating Custom Stereotypes

Autoregistering beans is a great way to cut back on the amount of XML required to configure Spring. But it may bother you that your autoregistered classes are annotated with Spring-specific annotations. If you're looking for a more non-intrusive way to autoregister beans, you have two options:

1. Create your own custom stereotype annotation. Doing so is as simple as creating a custom annotation that is itself annotated with @Component:

   ```
   @Component
   public @interface MyComponent {
   String value() default "";
   }
   ```

2. Or add a filter to <context:component-scan> to scan for annotations that it normally would not:

```
<context:component-scan
base-package="com.habuma.pirates">
<context:include-filter type="annotation"
expression="com.habuma.MyComponent" />
<context:exclude-filter type="annotation"
expression=
"org.springframework.stereotype.Component" />
</context:component-scan>
```

In this case, the @MyComponent custom annotation has been added to the list of annotations that are scanned for, but @Component has been excluded (that is, @Componentannotated classes will no longer be autoregistered).

Regardless of which option you choose, you should be able to autoregister beans by annotating their classes with the custom annotation:

```
@MyComponent
public class Pirate {...}
```

Spring MVC Annotations

These annotations were introduced in Spring 2.5 to make it easier to create Spring MVC applications with minimal XML configuration and without extending one of the many implementations of the Controller interface.

ANNOTATION	USE	DESCRIPTION
@Controller	Type	Stereotypes a component as a Spring MVC controller.
@InitBinder	Method	Annotates a method that customizes data binding.
@ModelAttribute	Parameter, Method	When applied to a method, used to preload the model with the value retur method. When applied to a parameter, binds a model attribute to the para
@RequestMapping	Method, Type	Maps a URL pattern and/or HTTP method to a method or controller type.

ANNOTATION	USE	DESCRIPTION
@RequestParam	Parameter	Binds a request parameter to a method parameter.
@SessionAttributes	Type	Specifies that a model attribute should be stored in the session.

Setting up Spring for Annotated Controllers

Before we can use annotations on Spring MVC controllers, we'll need to add a few lines of XML to tell Spring that our controllers will be annotation-driven. First, so that we won't have to register each of our controllers individually as <bean>s, we'll need a <context:component-scan>:

```
<context:component-scan
base-package="com.habuma.pirates.mvc"/>
```

In addition to autoregistering @Component-annotated beans, <context:component-scan> also autoregisters beans that are annotated with @Controller. We'll see a few examples of @Controller-annotated classes in a moment.

But first, we'll also need to tell Spring to honor the other Spring MVC annotations. For that we'll need <context:annotation-config> :
<context:annotation-config/>

Use a conventions-based view resolver.

If you use a conventions-based view resolver, such as Spring's UrlBasedViewResolver or InternalResourceViewResolver, along with <context:component-scan> and <context:annotation-config>, you can grow your application indefinitely without ever touching the Spring XML again.

Creating a Simple MVC Controller

The following HomePage class is annotated to function as a Spring MVC controller:

```
@Controller
@RequestMapping("/home.htm")
public class HomePage {
@RequestMapping(method = RequestMethod.GET)
public String showHomePage(Map model) {
List<Pirate> pirates = pirateService.
getPirateList();
model.add("pirateList", pirates);
return "home";
}
@Autowired
PirateService pirateService;
}
```

There are several important things to point out here. First, the HomePage class is annotated with @Controller so that it will be autoregistered as a bean by <context:component-scan>. It is also annotated with @RequestMapping, indicating that this controller will respond to requests for "/home.htm".

Within the class, the showHomePage() method is also annotated with @RequestMapping. In this case, @RequestMapping indicates that HTTP GET requests to "/home.htm" will be handled by the showHomePage() method.

Creating a Form-Handling Controller

In a pre-2.5 Spring MVC application, form-processing controllers would typically extend SimpleFormController (or some similar base class). But with Spring 2.5, a form-processing controller just has a method that is annotated to handle the HTTP POST request:

```
@Controller
@RequestMapping("/addPirate.htm")
public class AddPirateFormController {
@RequestMapping(method = RequestMethod.GET)
public String setupForm(ModelMap model) {
return "addPirate";
}
@ModelAttribute("pirate")
public Pirate setupPirate() {
Pirate pirate = new Pirate();
return pirate;
}
@RequestMapping(method = RequestMethod.POST)
protected String addPirate(@ModelAttribute("pirate")
Pirate pirate) {
pirateService.addPirate(pirate);
return "pirateAdded";
}
@Autowired
PirateService pirateService;
}
```

Here the @RequestMapping annotation is applied to two different methods. The setupForm() method is annotated to handle HTTP GET requests while the addPirate() method will handle HTTP POST requests. Meanwhile, the @ModelAttribute is also pulling double duty by populating the model with a new instance of Pirate before the form is displayed and then pulling the Pirate from the model so that it can be given to addPirate() for processing.

Transaction Annotations

The @Transactional annotation is used along with the <tx:annotation-driven> element to declare transactional boundaries and rules as class and method metadata in Java.

ANNOTATION	USE	DESCRIPTION
@Transactional	Method, Type	Declares transactional boundaries and rules on a bean and/or its met

Annotating Transactional Boundaries

To use Spring's support for annotation-declared transactions, you'll first need to add a small amount of XML to the Spring configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/
beans"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/
schema/beans
http://www.springframework.org/schema/beans/
springbeans-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-
2.5.xsd">
<tx:annotation-driven />
...
</beans>
```

The <tx:annotation-driven> element tells Spring to keep an eye out for beans that are annotated with @Transactional. In addition, you'll also need a platform transaction manager bean declared in the Spring context. For example, if your application uses Hibernate, you'll want to include the HibernateTransactionManager:

```
<bean id="transactionManager"
class="org.springframework.orm.hibernate3.
HibernateTransactionManager">
<property name="sessionFactory" ref="sessionFactory"
```

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

With the basic plumbing in place, you're ready to start annotating the transactional boundaries:

```
@Transactional(propagation=Propagation.SUPPORTS,
readOnly=true)
public class TreasureRepositoryImpl implements
TreasureRepository {
...
@Transactional(propagation=Propagation.REQUIRED,
readOnly=false)
public void storeTreasure(Treasure treasure) {...}
...
}
```

At the class level, @Transactional is declaring that all methods should support transactions and be read-only. But, at the method-level, @Transactional declares that the storeTreasure() method requires a transaction and is not read-only. Note that for transactions to be applied to @Transactionalannotated classes, those classes must be wired as beans in Spring.

JMX Annotations

These annotations, used with the <context:mbean-export> element, declare bean methods and properties as MBean operations and attributes.

ANNOTATIONS	USE	DESCRIPTION
@ManagedAttribute	Method	Used on a setter or getter method to indicate that the bean's propert as a MBean attribute.
@ManagedNotification	Type	Indicates a JMX notification emitted by a bean.
@ManagedNotifications	Type	Indicates the JMX notifications emitted by a bean.
@ManagedOperation	Method	Specifies that a method should be exposed as a MBean operation.
@ManagedOperationParameter	Method	Used to provide a description for an operation parameter.
@ManagedOperationParameters	Method	Provides descriptions for one or more operation parameters.

ANNOTATIONS	USE	DESCRIPTION
@ManagedResource	Type	Specifies that all instances of a class should be exposed a MBeans.

Exposing a Spring Bean as a MBean

To get started with Spring-annotated MBeans, you'll need to include <context:mbean-export> in the Spring XML configuration:

```
<context:mbean-export/>
```

Then, you can annotate any of your Spring-managed beans to be exported as MBeans:

```
@ManagedResource(objectName="pirates:name=PirateService")
public interface PirateService {
@ManagedOperation(
description="Get the pirate list")
public List<Pirate> getPirateList();
}
```

Here, the PirateService has been annotated to be exported as a MBean and its getPirateList() method is a managed operation.