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 from the 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; pri
vate TreasureMap treasureMap; public Pirate(String name)
{ this.name = name; } @Autowired public void setTreasureM
ap(TreasureMap treasureMap) { this.treasureMap = treasure
Map; } }
```

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 v
alue() 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.pi
rates"> <context:include-filter type="annotation" e
xpression="com.habuma.MyComponent" /> <context:excl
ude-filter type="annotation" expression= "org.sprin
gframework.stereotype.Component" /> </context:compo
nent-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 returned from the method. When applied to a parameter, binds a model attribute to the parameter. table
@RequestMapping	Method, Type	Maps a URL pattern and/or HTTP method to a method or controller type.
@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 Hom
ePage { @RequestMapping(method = RequestMethod.GET) publi
c String showHomePage(Map model) { List<Pirate> pirates =
pirateService. getPirateList(); model.add("pirateList", p
irates); return "home"; } @Autowired PirateService pirate
Service; }
```

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 clas
s AddPirateFormController { @RequestMapping(method = Requ
estMethod.GET) public String setupForm(ModelMap model) {
return "addPirate"; } @ModelAttribute("pirate") public Pi
rate setupPirate() { Pirate pirate = new Pirate(); return
pirate; } @RequestMapping(method = RequestMethod.POST) pr
otected String addPirate(@ModelAttribute("pirate") Pirate
pirate) { pirateService.addPirate(pirate); return "pirate
Added"; } @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 methods.

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="htt
p://www.springframework.org/schema/ beans" xmlns:tx="htt
p://www.springframework.org/schema/tx" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="h
ttp://www.springframework.org/ schema/beans http://www.sp
ringframework.org/schema/beans/ springbeans-2.5.xsd htt
p://www.springframework.org/schema/tx http://www.springfr
amework.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 n
ame="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 Tre
asureRepository { ... @Transactional(propagation=Propagat
ion.REQUIRED, readOnly=false) public void storeTreasure(T
reasure 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 property should be exposed 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.
@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( descr
iption="Get the pirate list") public List<Pirate> getPira
teList(); }
```

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

SECTION 3

Aspect Annotations

For defining aspects, Spring leverages the set of annotations provided by AspectJ.

ANNOTATION	USE	DESCRIPTION
@Aspect	Type	Declares a class to be an aspect.
@After	Method	Declares a method to be called after a pointcut completes.
@AfterReturning	Method	Declares a method to be called after a pointcut returns successfully.
@AfterThrowing	Method	Declares a method to be called after a pointcut throws an exception.
@Around	Method	Declares a method that will wrap the pointcut.
@Before	Method	Declares a method to be called before proceeding to the pointcut.
@DeclareParents	Static Field	Declares that matching types should be given new parents,that is, it introduces new functionality into matching types.
@Pointcut	Method	Declares an empty method as a pointcut placeholder method.

What's important to note, however, is that while you can use AspectJ annotations to define Spring aspects, those aspects will be defined in the context of Spring AOP and will not be handled by the

of Springframework and will not be handled by the AspectJ runtime. This is significant because Spring AOP is limited to proxying method invocations and does not provide for the more exotic pointcuts (constructor interception, field interception, etc.) offered by AspectJ.

Annotating Aspects

To use AspectJ annotations to create Spring aspects, you'll first need to provide a bit of Spring XML plumbing:

```
<beans xmlns="http://www.springframework.org/schema/ bean
s" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop" xs
i:schemaLocation="http://www.springframework.org/ schema/
beans http://www.springframework.org/schema/beans/ spring
-beans-2.5.xsd http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/springaop- 2.5.
xsd"> ... <aop:aspectj-autoproxy/> ... </beans>
```

The <aop:aspectj-autoproxy> element tells Spring to watch for beans annotated with AspectJ annotations and, if it finds any, to use them to create aspects. Then you can annotate bean classes to be aspects:

```
@Aspect public class ChantySinger { @Pointcut("execution
(* Pirate.plunder(..))") public void plunderPC() {} @Befo
re("plunderPC()") public void singYoHo() { ... } @AfterRe
turning("plunderPC()") public void singAPiratesLifeForMe
() { ... } }
```

This simple annotation-based aspect has a pointcut that is triggered by the execution of a plunder() method on the Pirate class. Before the Pirate.plunder() method is executed, the singYoHo() method is called. Then, after the Pirate.plunder() method returns successfully, the singAPiratesLifeForMe() method is invoked. (For more advanced examples of AspectJ annotations, see the AspectJ documentation

at http://www.eclipse.org/aspectj/docs.php.)

Note the rather odd looking plunderPC() method. It is annotated with @Pointcut to indicate that this method is a pointcut placeholder. The key thing here is that the most interesting stuff happens in the annotation itself and not in the method. In fact, pointcut placeholder methods must be empty methods and return void.

SECTION 4

JSR-250 Annotations

In addition to Spring's own set of annotations, Spring also supports a few of the annotations defined by JSR-250, which is the basis for the annotations used in EJB 3.

ANNOTATION	USE	DESCRIPTION
@PostConstruct	Method	Indicates a method to be invoked after a bean has been created and dependency injection is complete. Used to perform any initialization work necessary.
@PreDestroy	Method	Indicates a method to be invoked just before a bean is removed from the Spring context. Used to perform any cleanup work necessary.
@Resource	Method, Field	Indicates that a method or field should be injected with a named resource (by default, another bean).

Wiring Bean Properties with @Resource

Using @Resource, you can wire a bean property by name:

```
public class Pirate { @Resource private TreasureMap treas
ureMap; }
```

In this case, Spring will attempt to wire the

In this case, Spring will attempt to wire the "treasureMap" property with a reference to a bean whose ID is "treasureMap". If you'd rather explicitly choose another bean to wire into the property, specify it to the name attribute:

```
public class Pirate { @Resource(name="mapToSkullIsland")
private TreasureMap treasureMap; }
```

Initialization and Destruction Methods

Using JSR-250's @PostConstruct and @PreDestroy methods, you can declare methods that hook into a bean's lifecycle. For example, consider the following methods added to the Pirate class:

```
public class Pirate { ... @PostConstruct public void wake
Up() { System.out.println("Yo ho!"); } @PreDestroy public
void goAway() { System.out.println("Yar!"); } }
```

As annotated, the wakeUp() method will be invoked just after Spring instantiates the bean and goAway() will be invoked just before the bean is removed from the Spring container.

SECTION 5

Testing Annotations

These annotations are useful for creating unit tests in the JUnit 4 style that depend on Spring beans and/or require a transactional context.

ANNOTATION	USE	DESCRIPTION
@AfterTransaction	Method	Used to identify a method to be invoked after a transaction has completed.
@BeforeTransaction	Method	Used to identify a method to be

@BeforeTransaction	Method	invoked before a transaction starts.
@ContextConfiguration	Type	Configures a Spring application context for a test.
@DirtiesContext	Method	Indicates that a method dirties the Spring container and thus it must be rebuilt after the test completes.
@ExpectedException	Method	Indicates that the test method is expected to throw a specific exception. The test will fail if the exception is not thrown.
@IfProfileValue	Type, Method	Indicates that the test class or method is enabled for a specific profile configuration.
@NotTransactional	Method	Indicates that a test method must not execute in a transactional context.
@ProfileValueSourceConfiguration	Type	Identifies an implementation of a profile value source. The absence of this annotation will cause profile values to be loaded from system properties.
@Repeat	Method	Indicates that the test method must be repeated a specific number of times.
ANNOTATION	**USE**	**DESCRIPTION**
@Rollback	Method	Specifies whether or not the transaction for the annotated method should be rolled back or not.
		Identifies zero or more test

@TestExecutionListeners	Type	execution listeners for a test class.
@Timed	Method	Specifies a time limit for the test method. If the test does not complete before the time has expired, the test will fail.
@TransactionConfiguration	Type	Configures test classes for transactions, specifying the transaction manager and/or the default rollback rule for all test methods in a test class.

Writing a Spring-Aware Test

The key to writing a Spring-aware test is to annotate the test class with @RunWith, specifying SpringJUnit4ClassRunner as the class runner behind the test:

```
@RunWith(SpringJUnit4ClassRunner.class) public class Pira
teTest { ... }
```

In this case, the Spring test runner will try to load a Spring application context from a file named PirateTest-context.xml. If you'd rather specify one or more XML files to load the application context from, you can do that with @ContextConfiguration:

```
@RunWith(SpringJUnit4ClassRunner.class) @ContextConfigura
tion(locations = { "pirates.xml" }) public class PirateTe
st { ... }
```

With test configured to load a Spring application context, you now may request that Spring autowire properties of the test class with beans from the Spring context:

```
@RunWith(SpringJUnit4ClassRunner.class) @ContextConfigura
tion(locations = { "pirates.xml" }) public class PirateTe
st { @Autowired private Pirate pirate; @Autowired private
TreasureMap treasureMap; @Test public void annotatedPrope
rtyShouldBeAutowired() { assertNotNull(pirate.getTreasure
Map()); assertEquals(treasureMap, pirate.getTreasureMap
()); } }
```

In this case, the pirate and treasureMap properties will be wired with the beans whose ID are "pirate" and "treasureMap", respectively.

Accessing the Spring Context in a Test

If you need the Spring application context itself in a test, you can autowire it into the test the same as if it were a bean in the context:

```
@RunWith(SpringJUnit4ClassRunner.class) @ContextConfigura
tion(locations = { "pirates.xml" }) public class PirateTe
st { @Autowired private Pirate pirate; @Autowired private
ApplicationContext applicationContext; @Test public void
annotatedPropertyShouldBeAutowired() { assertNotNull(pira
te.getTreasureMap()); assertEquals(applicationContext. ge
tBean("treasureMap"), pirate .getTreasureMap()); } }
```