##### **Supported Pointcut Designators**

Spring AOP supports the following AspectJ pointcut designators (PCD) for use in pointcut expressions:

* execution: For matching method execution join points. This is the primary pointcut designator to use when working with Spring AOP.
* within: Limits matching to join points within certain types (the execution of a method declared within a matching type when using Spring AOP).
* this: Limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type.
* target: Limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type.
* args: Limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types.
* @target: Limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type.
* @args: Limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given types.
* @within: Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP).
* @annotation: Limits matching to join points where the subject of the join point (the method being executed in Spring AOP) has the given annotation.

Other pointcut types

The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring: call, get, set, preinitialization, staticinitialization, initialization, handler, adviceexecution, withincode, cflow,cflowbelow, if, @this, and @withincode. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP results in an IllegalArgumentException being thrown.

The set of pointcut designators supported by Spring AOP may be extended in future releases to support more of the AspectJ pointcut designators.

Because Spring AOP limits matching to only method execution join points, the preceding discussion of the pointcut designators gives a narrower definition than you can find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and, at an execution join point, both this and target refer to the same object: the object executing the method. Spring AOP is a proxy-based system and differentiates between the proxy object itself (which is bound to this) and the target object behind the proxy (which is bound to target).

###### **Passing Parameters to Advice**

We have already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of args. If you use a parameter name in place of a type name in an args expression, the value of the corresponding argument is passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of DAO operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**public** **void** validateAccount(Account account) {

*// ...*

}

The args(account,..) part of the pointcut expression serves two purposes. First, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of Account. Second, it makes the actual Account object available to the advice through the account parameter.

Another way of writing this is to declare a pointcut that “provides” the Account object value when it matches a join point, and then refer to the named pointcut from the advice. This would look as follows:

@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() && args(account,..)")

**private** **void** accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")

**public** **void** validateAccount(Account account) {

*// ...*

}

See the AspectJ programming guide for more details.

The proxy object ( this), target object ( target), and annotations ( @within, @target, @annotation, and @args) can all be bound in a similar fashion. The next two examples show how to match the execution of methods annotated with an @Auditableannotation and extract the audit code:

The first of the two examples shows the definition of the @Auditable annotation:

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

**public** @interface Auditable {

AuditCode value();

}

The second of the two examples shows the advice that matches the execution of @Auditable methods:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)")

**public** **void** audit(Auditable auditable) {

AuditCode code = auditable.value();

*// ...*

}

###### **Advice Parameters and Generics**

Spring AOP can handle generics used in class declarations and method parameters. Suppose you have a generic type like the following:

**public** **interface** **Sample**<T> {

**void** sampleGenericMethod(T param);

**void** sampleGenericCollectionMethod(Collection<T> param);

}

You can restrict interception of method types to certain parameter types by typing the advice parameter to the parameter type for which you want to intercept the method:

@Before("execution(\* ..Sample+.sampleGenericMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(MyType param) {

*// Advice implementation*

}

This approach does not work for generic collections. So you cannot define a pointcut as follows:

@Before("execution(\* ..Sample+.sampleGenericCollectionMethod(\*)) && args(param)")

**public** **void** beforeSampleMethod(Collection<MyType> param) {

*// Advice implementation*

}

To make this work, we would have to inspect every element of the collection, which is not reasonable, as we also cannot decide how to treat null values in general. To achieve something similar to this, you have to type the parameter to Collection<?> and manually check the type of the elements.

###### **Determining Argument Names**

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in advice and pointcut method signatures. Parameter names are not available through Java reflection, so Spring AOP uses the following strategy to determine parameter names:

* If the parameter names have been explicitly specified by the user, the specified parameter names are used. Both the advice and the pointcut annotations have an optional argNames attribute that you can use to specify the argument names of the annotated method. These argument names are available at runtime. The following example shows how to use the argNamesattribute:
* @Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",
* argNames="bean,auditable")
* **public** **void** audit(Object bean, Auditable auditable) {
* AuditCode code = auditable.value();
* *// ... use code and bean*

}

If the first parameter is of the JoinPoint, ProceedingJoinPoint, or JoinPoint.StaticPart type, you can leave out the name of the parameter from the value of the argNames attribute. For example, if you modify the preceding advice to receive the join point object, the argNames attribute need not include it:

@Before(value="com.xyz.lib.Pointcuts.anyPublicMethod() && target(bean) && @annotation(auditable)",

argNames="bean,auditable")

**public** **void** audit(JoinPoint jp, Object bean, Auditable auditable) {

AuditCode code = auditable.value();

*// ... use code, bean, and jp*

}

The special treatment given to the first parameter of the JoinPoint, ProceedingJoinPoint, and JoinPoint.StaticPart types is particularly convenient for advice instances that do not collect any other join point context. In such situations, you may omit the argNames attribute. For example, the following advice need not declare the argNames attribute:

@Before("com.xyz.lib.Pointcuts.anyPublicMethod()")

**public** **void** audit(JoinPoint jp) {

*// ... use jp*

}

* Using the 'argNames' attribute is a little clumsy, so if the 'argNames' attribute has not been specified, Spring AOP looks at the debug information for the class and tries to determine the parameter names from the local variable table. This information is present as long as the classes have been compiled with debug information ( '-g:vars' at a minimum). The consequences of compiling with this flag on are: (1) your code is slightly easier to understand (reverse engineer), (2) the class file sizes are very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables is not applied by your compiler. In other words, you should encounter no difficulties by building with this flag on.
* If the code has been compiled without the necessary debug information, Spring AOP tries to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method takes only one parameter, the pairing is obvious). If the binding of variables is ambiguous given the available information, an AmbiguousBindingException is thrown.
* If all of the above strategies fail, an IllegalArgumentException is thrown.

###### **Proceeding with Arguments**

We remarked earlier that we would describe how to write a proceed call with arguments that works consistently across Spring AOP and AspectJ. The solution is to ensure that the advice signature binds each of the method parameters in order. The following example shows how to do so:

@Around("execution(List<Account> find\*(..)) && " +

"com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +

"args(accountHolderNamePattern)")

**public** Object preProcessQueryPattern(ProceedingJoinPoint pjp,

String accountHolderNamePattern) **throws** Throwable {

String newPattern = preProcess(accountHolderNamePattern);

**return** pjp.proceed(**new** Object**[]** {newPattern});

}

In many cases, you do this binding anyway (as in the preceding example).

#### Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

You can make an introduction by using the @DeclareParents annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface named UsageTracked and an implementation of that interface named DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface (to expose statistics via JMX for example):

@Aspect

**public** **class** **UsageTracking** {

@DeclareParents(value="com.xzy.myapp.service.\*+", defaultImpl=DefaultUsageTracked.class)

**public** **static** UsageTracked mixin;

@Before("com.xyz.myapp.SystemArchitecture.businessService() && this(usageTracked)")

**public** **void** recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

}

The interface to be implemented is determined by the type of the annotated field. The value attribute of the @DeclareParentsannotation is an AspectJ type pattern. Any bean of a matching type implements the UsageTracked interface. Note that, in the before advice of the preceding example, service beans can be directly used as implementations of the UsageTracked interface. If accessing a bean programmatically, you would write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

#### Aspect Instantiation Models

By default, there is a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles. Spring supports AspectJ’s perthis and pertarget instantiation models ( percflow, percflowbelow, and pertypewithin are not currently supported).

You can declare a perthis aspect by specifying a perthis clause in the @Aspect annotation. Consider the following example:

@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")

**public** **class** **MyAspect** {

**private** **int** someState;

@Before(com.xyz.myapp.SystemArchitecture.businessService())

**public** **void** recordServiceUsage() {

*// ...*

}

}

In the preceding example, the effect of the 'perthis' clause is that one aspect instance is created for each unique service object that executes a business service (each unique object bound to 'this' at join points matched by the pointcut expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it executes at matched join points, but only when the service object is the one with which this aspect is associated. See the AspectJ Programming Guide for more information on perclauses.

The pertarget instantiation model works in exactly the same way as perthis, but it creates one aspect instance for each unique target object at matched join points.

#### 5.4.7. An AOP Example

Now that you have seen how all the constituent parts work, we can put them together to do something useful.

The execution of business services can sometimes fail due to concurrency issues (for example, a deadlock loser). If the operation is retried, it is likely to succeed on the next try. For business services where it is appropriate to retry in such conditions (idempotent operations that do not need to go back to the user for conflict resolution), we want to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer and, hence, is ideal for implementing through an aspect.

Because we want to retry the operation, we need to use around advice so that we can call proceed multiple times. The following listing shows the basic aspect implementation:

@Aspect

**public** **class** **ConcurrentOperationExecutor** **implements** Ordered {

**private** **static** **final** **int** DEFAULT\_MAX\_RETRIES = 2;

**private** **int** maxRetries = DEFAULT\_MAX\_RETRIES;

**private** **int** order = 1;

**public** **void** setMaxRetries(**int** maxRetries) {

this.maxRetries = maxRetries;

}

**public** **int** getOrder() {

**return** this.order;

}

**public** **void** setOrder(**int** order) {

this.order = order;

}

@Around("com.xyz.myapp.SystemArchitecture.businessService()")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

**int** numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

**do** {

numAttempts++;

**try** {

**return** pjp.proceed();

}

**catch**(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} **while**(numAttempts <= this.maxRetries);

**throw** lockFailureException;

}

}

Note that the aspect implements the Ordered interface so that we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties are both configured by Spring. The main action happens in the doConcurrentOperation around advice. Notice that, for the moment, we apply the retry logic to each businessService(). We try to proceed, and if we fail with a PessimisticLockingFailureException, we try again, unless we have exhausted all of our retry attempts.

The corresponding Spring configuration follows:

<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor" class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

To refine the aspect so that it retries only idempotent operations, we might define the following Idempotent annotation:

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

We can then use the annotation to annotate the implementation of service operations. The change to the aspect to retry only idempotent operations involves refining the pointcut expression so that only @Idempotent operations match, as follows:

@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +

"@annotation(com.xyz.myapp.service.Idempotent)")

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

...

}

Schema Declaration

Aspect

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

...

</aop:aspect>

</aop:config>

<bean id="aBean" class="...">

...

</bean>

Point Cut

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

</aop:config>

If you use the schema based declaration style, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression. Another way of defining the above pointcut would be as follows:

<aop:config>

<aop:pointcut id="businessService"

expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>

Then declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut, as the following example shows:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

...

</aop:aspect>

</aop:config>

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) &amp;&amp; this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

The advice must be declared to receive the collected join point context by including parameters of the matching names, as follows:

**public** **void** monitor(Object service) {

...

}

When combining pointcut sub-expressions, && is awkward within an XML document, so you can use the and, or, and notkeywords in place of &&, ||, and !, respectively. For example, the previous pointcut can be better written as follows:

<aop:config>

<aop:aspect id="myAspect" ref="aBean">

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and this(service)"/>

<aop:before pointcut-ref="businessService" method="monitor"/>

...

</aop:aspect>

</aop:config>

Declaring Advise

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

...

</aop:aspect>

Here, dataAccessOperation is the id of a pointcut defined at the top (<aop:config>) level. To define the pointcut inline instead, replace the pointcut-ref attribute with a pointcut attribute, as follows:

<aop:aspect id="beforeExample" ref="aBean">

<aop:before

pointcut="execution(\* com.xyz.myapp.dao.\*.\*(..))"

method="doAccessCheck"/>

<aop:after-returning

pointcut-ref="dataAccessOperation"

method="doAccessCheck"/>

<aop:after-returning

pointcut-ref="dataAccessOperation"

returning="retVal"

method="doAccessCheck"/>

<aop:after-throwing

pointcut-ref="dataAccessOperation"

method="doRecoveryActions"/>

<aop:after-throwing

pointcut-ref="dataAccessOperation"

throwing="dataAccessEx"

method="doRecoveryActions"/>

<aop:after

pointcut-ref="dataAccessOperation"

method="doReleaseLock"/>

<aop:around

pointcut-ref="businessService"

method="doBasicProfiling"/>

...

</aop:aspect>

**public** **void** doAccessCheck(Object retVal) {...

**public** **void** doRecoveryActions(DataAccessException dataAccessEx) {...

**public** Object doBasicProfiling(ProceedingJoinPoint pjp) **throws** Throwable {

*// start stopwatch*

Object retVal = pjp.proceed();

*// stop stopwatch*

**return** retVal;

}

##### **Advice Parameters**

The schema-based declaration style supports fully typed advice in the same way as described for the @AspectJ support — by matching pointcut parameters by name against advice method parameters. See [Advice Parameters](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-params) for details. If you wish to explicitly specify argument names for the advice methods (not relying on the detection strategies previously described), you can do so by using the arg-names attribute of the advice element, which is treated in the same manner as the argNames attribute in an advice annotation (as described in [Determining Argument Names](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-params-names)). The following example shows how to specify an argument name in XML:

<aop:before

pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"

method="audit"

arg-names="auditable"/>

The arg-names attribute accepts a comma-delimited list of parameter names.

The following slightly more involved example of the XSD-based approach shows some around advice used in conjunction with a number of strongly typed parameters:

**package** x.y.service;

**public** **interface** **PersonService** {

Person getPerson(String personName, **int** age);

}

**public** **class** **DefaultFooService** **implements** FooService {

**public** Person getPerson(String name, **int** age) {

**return** **new** Person(name, age);

}

}

Next up is the aspect. Notice the fact that the profile(..) method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call. The presence of this parameter is an indication that the profile(..) is to be used as around advice, as the following example shows:

**package** x.y;

**import** org.aspectj.lang.ProceedingJoinPoint;

**import** org.springframework.util.StopWatch;

**public** **class** **SimpleProfiler** {

**public** Object profile(ProceedingJoinPoint call, String name, **int** age) **throws** Throwable {

StopWatch clock = **new** StopWatch("Profiling for '" + name + "' and '" + age + "'");

**try** {

clock.start(call.toShortString());

**return** call.proceed();

} **finally** {

clock.stop();

System.out.println(clock.prettyPrint());

}

}

}

Finally, the following example XML configuration effects the execution of the preceding advice for a particular join point:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="

http://www.springframework.org/schema/beans https://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/aop https://www.springframework.org/schema/aop/spring-aop.xsd">

*<!-- this is the object that will be proxied by Spring's AOP infrastructure -->*

<bean id="personService" class="x.y.service.DefaultPersonService"/>

*<!-- this is the actual advice itself -->*

<bean id="profiler" class="x.y.SimpleProfiler"/>

<aop:config>

<aop:aspect ref="profiler">

<aop:pointcut id="theExecutionOfSomePersonServiceMethod"

expression="execution(\* x.y.service.PersonService.getPerson(String,int))

and args(name, age)"/>

<aop:around pointcut-ref="theExecutionOfSomePersonServiceMethod"

method="profile"/>

</aop:aspect>

</aop:config>

</beans>

Consider the following driver script:

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** x.y.service.PersonService;

**public** **final** **class** **Boot** {

**public** **static** **void** main(**final** String**[]** args) **throws** Exception {

BeanFactory ctx = **new** ClassPathXmlApplicationContext("x/y/plain.xml");

PersonService person = (PersonService) ctx.getBean("personService");

person.getPerson("Pengo", 12);

}

}

With such a Boot class, we would get output similar to the following on standard output:

StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0

-----------------------------------------

ms % Task name

-----------------------------------------

00000 ? execution(getFoo)

##### **Advice Ordering**

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in [Advice Ordering](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-advice-ordering). The precedence between aspects is determined by either adding the Order annotation to the bean that backs the aspect or by having the bean implement the Ordered interface.

#### 5.5.4. Introductions

Introductions (known as inter-type declarations in AspectJ) let an aspect declare that advised objects implement a given interface and provide an implementation of that interface on behalf of those objects.

You can make an introduction by using the aop:declare-parents element inside an aop:aspect. You can use the aop:declare-parents element to declare that matching types have a new parent (hence the name). For example, given an interface named UsageTracked and an implementation of that interface named DefaultUsageTracked, the following aspect declares that all implementors of service interfaces also implement the UsageTracked interface. (In order to expose statistics through JMX for example.)

<aop:aspect id="usageTrackerAspect" ref="usageTracking">

<aop:declare-parents

types-matching="com.xzy.myapp.service.\*+"

implement-interface="com.xyz.myapp.service.tracking.UsageTracked"

default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

<aop:before

pointcut="com.xyz.myapp.SystemArchitecture.businessService()

and this(usageTracked)"

method="recordUsage"/>

</aop:aspect>

The class that backs the usageTracking bean would then contain the following method:

**public** **void** recordUsage(UsageTracked usageTracked) {

usageTracked.incrementUseCount();

}

The interface to be implemented is determined by the implement-interface attribute. The value of the types-matchingattribute is an AspectJ type pattern. Any bean of a matching type implements the UsageTracked interface. Note that, in the before advice of the preceding example, service beans can be directly used as implementations of the UsageTracked interface. To access a bean programmatically, you could write the following:

UsageTracked usageTracked = (UsageTracked) context.getBean("myService");

#### 5.5.5. Aspect Instantiation Models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

#### 5.5.6. Advisors

The concept of “advisors” comes from the AOP support defined in Spring and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean and must implement one of the advice interfaces described in [Advice Types in Spring](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-api-advice-types). Advisors can take advantage of AspectJ pointcut expressions.

Spring supports the advisor concept with the <aop:advisor> element. You most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring. The following example shows an advisor:

<aop:config>

<aop:pointcut id="businessService"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:advisor

pointcut-ref="businessService"

advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">

<tx:attributes>

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

As well as the pointcut-ref attribute used in the preceding example, you can also use the pointcut attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the order attribute to define the Ordered value of the advisor.

#### 5.5.7. An AOP Schema Example

This section shows how the concurrent locking failure retry example from [An AOP Example](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#aop-ataspectj-example) looks when rewritten with the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, a deadlock loser). If the operation is retried, it is likely to succeed on the next try. For business services where it is appropriate to retry in such conditions (idempotent operations that do not need to go back to the user for conflict resolution), we want to transparently retry the operation to avoid the client seeing a PessimisticLockingFailureException. This is a requirement that clearly cuts across multiple services in the service layer and, hence, is ideal for implementing through an aspect.

Because we want to retry the operation, we need to use around advice so that we can call proceed multiple times. The following listing shows the basic aspect implementation (which is a regular Java class that uses the schema support):

**public** **class** **ConcurrentOperationExecutor** **implements** Ordered {

**private** **static** **final** **int** DEFAULT\_MAX\_RETRIES = 2;

**private** **int** maxRetries = DEFAULT\_MAX\_RETRIES;

**private** **int** order = 1;

**public** **void** setMaxRetries(**int** maxRetries) {

this.maxRetries = maxRetries;

}

**public** **int** getOrder() {

**return** this.order;

}

**public** **void** setOrder(**int** order) {

this.order = order;

}

**public** Object doConcurrentOperation(ProceedingJoinPoint pjp) **throws** Throwable {

**int** numAttempts = 0;

PessimisticLockingFailureException lockFailureException;

**do** {

numAttempts++;

**try** {

**return** pjp.proceed();

}

**catch**(PessimisticLockingFailureException ex) {

lockFailureException = ex;

}

} **while**(numAttempts <= this.maxRetries);

**throw** lockFailureException;

}

}

Note that the aspect implements the Ordered interface so that we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The maxRetries and order properties are both configured by Spring. The main action happens in the doConcurrentOperation around advice method. We try to proceed. If we fail with a PessimisticLockingFailureException, we try again, unless we have exhausted all of our retry attempts.

|  |  |
| --- | --- |
|  | This class is identical to the one used in the @AspectJ example, but with the annotations removed. |

The corresponding Spring configuration is as follows:

<aop:config>

<aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..))"/>

<aop:around

pointcut-ref="idempotentOperation"

method="doConcurrentOperation"/>

</aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"

class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">

<property name="maxRetries" value="3"/>

<property name="order" value="100"/>

</bean>

Notice that, for the time, being we assume that all business services are idempotent. If this is not the case, we can refine the aspect so that it retries only genuinely idempotent operations, by introducing an Idempotent annotation and using the annotation to annotate the implementation of service operations, as the following example shows:

@Retention(RetentionPolicy.RUNTIME)

**public** @interface Idempotent {

*// marker annotation*

}

The change to the aspect to retry only idempotent operations involves refining the pointcut expression so that only @Idempotent operations match, as follows:

<aop:pointcut id="idempotentOperation"

expression="execution(\* com.xyz.myapp.service.\*.\*(..)) and

@annotation(com.xyz.myapp.service.Idempotent)"/>