Implementing a Custom Spring AOP Annotation

Last modified: March 21, 2020

by baeldung

Spring

Spring Annotations

1. Introduction

In this article, we'll implement a custom AOP annotation using the AOP support in Spring.

First, we'll give a high-level overview of AOP, explaining what it is and its advantages. Following this, we'll implement our annotation step by step, gradually building up a more in-depth understanding of AOP concepts as we go.

The outcome will be a better understanding of AOP and the ability to create our custom Spring annotations in the future.

2. What Is an AOP Annotation?

To quickly summarize, AOP stands for aspect oriented programming. Essentially, **it is a way for adding behavior to existing code without modifying that code**.

For a detailed introduction to AOP, there are articles on AOP *pointcuts* and *advice*. This article assumes we have a basic knowledge already.

The type of AOP that we will be implementing in this article is annotation driven. We may be familiar with this already if we've used the Spring *@Transactional* annotation:

```
@Transactional
public void orderGoods(Order order) {
    // A series of database calls to be performed in a transaction
}
```

The key here is non-invasiveness. By using annotation meta-data, our core business logic isn't polluted with our transaction code. This makes it easier to reason about, refactor, and to test in isolation.

Sometimes, people developing Spring applications can see this as 'Spring Magic', without thinking in much detail about how it's working. In reality, what's happening isn't particularly complicated. However, once we've completed the steps in this article, we will be able to create our own custom annotation in order to understand and leverage AOP.

3. Maven Dependency

First, let's add our Maven dependencies.

For this example, we'll be using Spring Boot, as its convention over configuration approach lets us get up and running as quickly as possible:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
</dependencies>
```

Note that we've included the AOP starter, which pulls in the libraries we need to start implementing aspects.

4. Creating Our Custom Annotation

The annotation we are going to create is one which will be used to log the amount of time it takes a method to execute. Let's create our annotation:

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface LogExecutionTime {

}
```

Although a relatively simple implementation, it's worth noting what the two meta-annotations are used for.

The *@Target* annotation tells us where our annotation will be applicable. Here we are using *ElementType.Method*, which means it will only work on methods. If we tried to use the annotation anywhere else, then our code would fail to compile. This behavior makes sense, as our annotation will be used for logging method execution time.

And *@Retention* just states whether the annotation will be available to the JVM at runtime or not. By default it is not, so Spring AOP would not be able to see the annotation. This is why it's been included.

5. Creating Our Aspect

Now we have our annotation, let's create our aspect. This is just the module that will encapsulate our cross-cutting concern, which is our case is method execution time logging. All it is is a class, annotated with *@Aspect*:

```
@Aspect
@Component
public class ExampleAspect {

}
```

We've also included the *@Component* annotation, as our class also needs to be a Spring bean to be detected. Essentially, this is the class where we will implement the logic that we want our custom annotation to inject.

6. Creating Our Pointcut and Advice

Now, let's create our pointcut and advice. This will be an annotated method that lives in our aspect:

```
@Around("@annotation(LogExecutionTime)")
public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
    return joinPoint.proceed();
}
```

Technically this doesn't change the behavior of anything yet, but there's still quite a lot going on that needs analysis.

First, we have annotated our method with *@Around*. This is our advice, and around advice means we are adding extra code both before and after method execution. There are other types of advice, such as *before* and *after* but they will be left out of scope for this article.

Next, our *@Around* annotation has a point cut argument. Our pointcut just says, 'Apply this advice any method which is annotated with *@LogExecutionTime*'. There are lots of other types of pointcuts, but they will again be left out if scope.

The method *logExecutionTime()* itself is our advice. There is a single argument, *ProceedingJoinPoint*. In our case, this will be an executing method which has been annotated with *@LogExecutionTime*.

Finally, when our annotated method ends up being called, what will happen is our advice will be called first. Then it's up to our advice to decide what to do next. In our case, our advice is doing nothing other than calling *proceed()*, which is the just calling the original annotated method.

7. Logging Our Execution Time

Now we have our skeleton in place, all we need to do is add some extra logic to our advice. This will be what logs the execution time in addition to calling the original method. Let's add this extra behavior to our advice:

```
@Around("@annotation(LogExecutionTime)")
public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
    long start = System.currentTimeMillis();

    Object proceed = joinPoint.proceed();

    long executionTime = System.currentTimeMillis() - start;

    System.out.println(joinPoint.getSignature() + " executed in " + executionTime + "ms");
    return proceed;
}
```

Again, we've not done anything that's particularly complicated here. We've just recorded the current time, executed the method, then printed the amount of time it took to the console. We're also logging the method signature, which is provided to use the *Joinpoint* instance. We would also be able to gain access to other bits of information if we wanted to, such as method arguments.

Now, let's try annotating a method with *@LogExecutionTime*, and then executing it to see what happens. Note that this must be a Spring Bean to work correctly:

```
@LogExecutionTime
public void serve() throws InterruptedException {
    Thread.sleep(2000);
}
```

After execution, we should see the following logged to the console:

```
void org.baeldung.Service.serve() executed in 2010ms
```

8. Conclusion

In this article, we've leveraged Spring Boot AOP to create our custom annotation, which we can apply to Spring beans to inject extra behavior to them at runtime.

The source code for our application is available on over on GitHub; this is a Maven project which should be able to run as is.