Spring Start Here

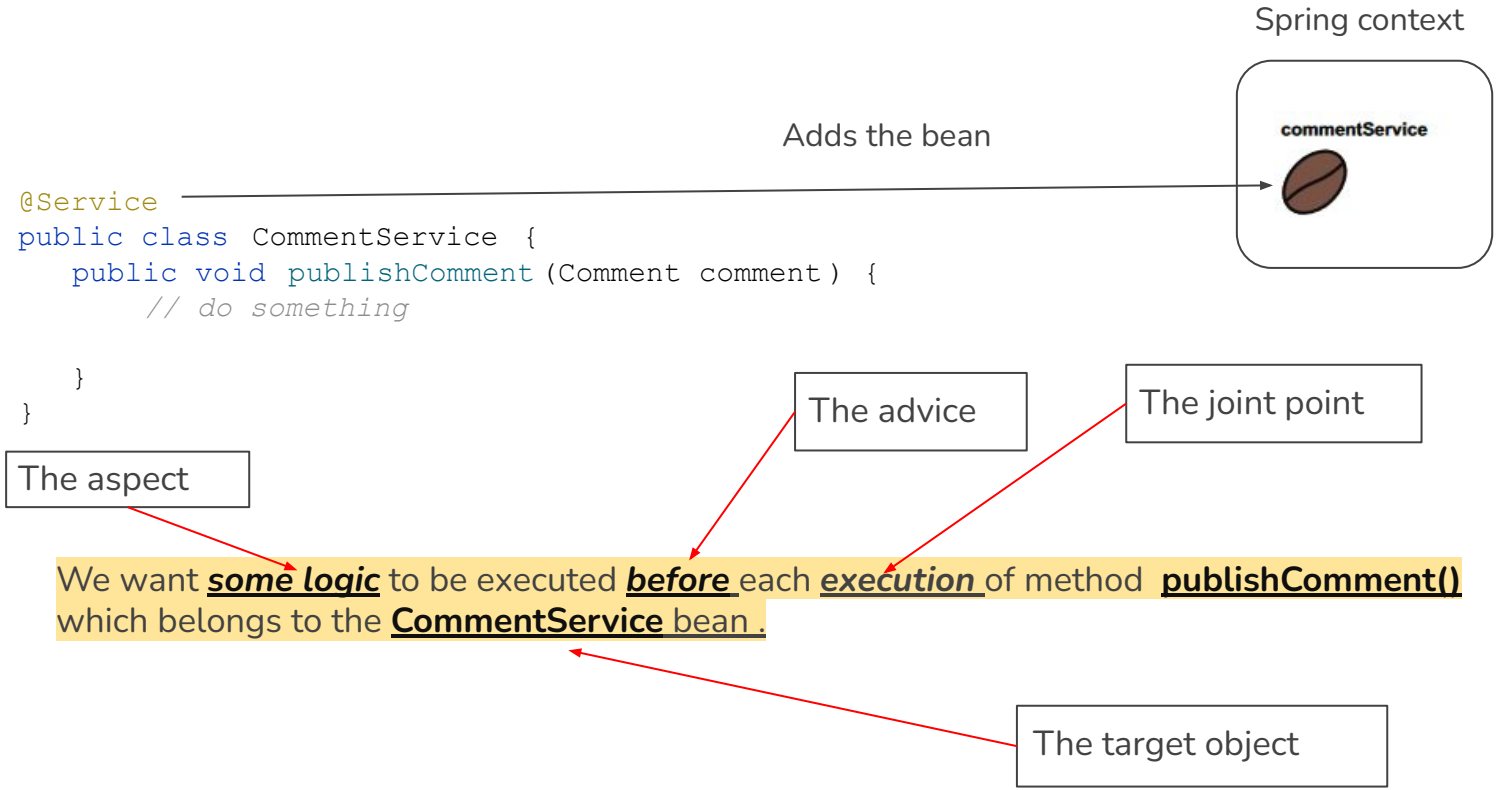
Chapter-6:
Using aspect with Spring AOP

Upcode Software
Engineer Team

-2024-

CONTENT

How aspects work in Spring (1/7)



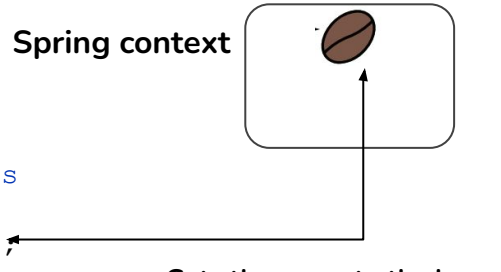
How aspects work in Spring (2/7)

An aspect is simply a piece of logic the framework executes when you call specific methods of your choice. When designing an aspect, you define the following:

- What code you want Spring to execute when you call specific methods. This is named an ***aspect***
- When the app should execute this logic of the aspect (e.g., before or after the method call, instead of the method call). This is named the ***advice***
- Which methods the framework needs to intercept and execute the aspect for them. This is named a ***pointcut***

How aspects work in Spring (3/7)

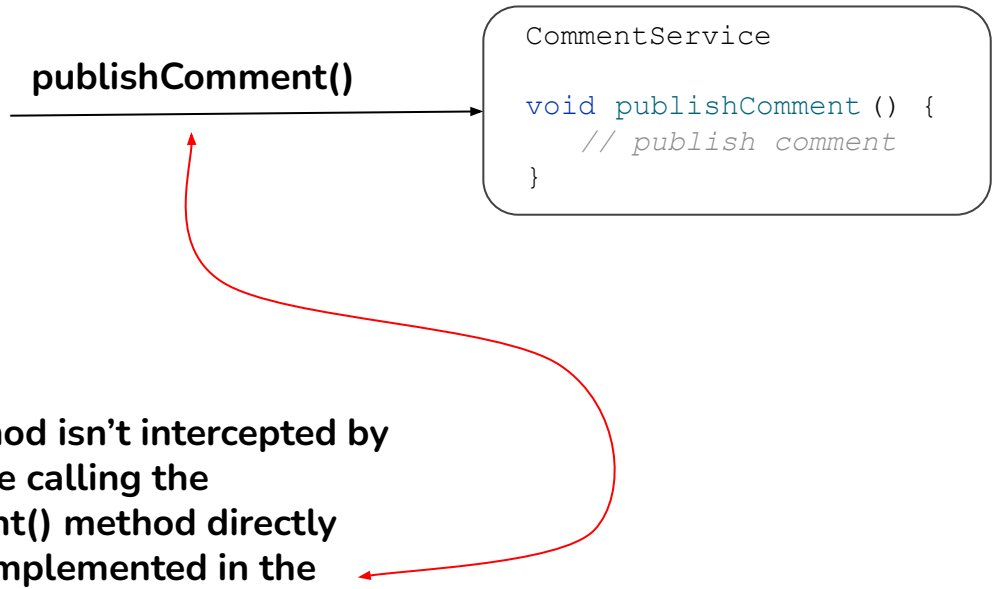
```
public class Main {
    public static void main(String[] args) {
        var c = new
AnnotationConfigApplicationContext(ProjectConfig. clas
s);
        var service = c.getBean(CommentService.class);
        System.out.println(service.getClass());
    }
}
```



If the bean is an aspect target, Spring doesn't provide you a reference to the actual object. Instead, Spring gives you a reference to a proxy object that can manage each call to the intercepted method and apply the aspect logic

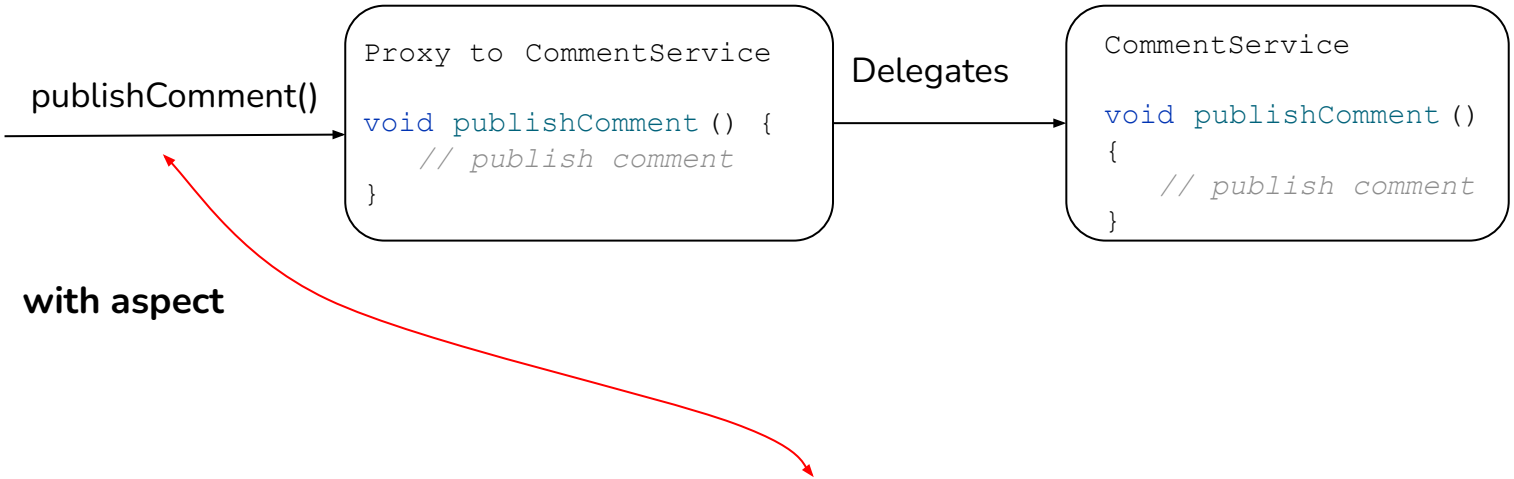
How aspects work in Spring (4/7)

Without aspect



When the method isn't intercepted by spects, someone calling the publishComment() method directly calls the logic implemented in the CommentService class.

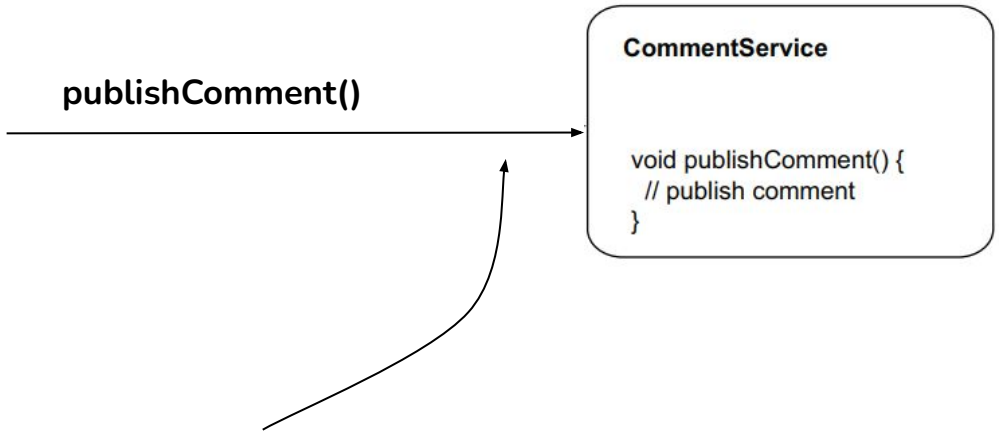
How aspects work in Spring (5/7)



When we define an aspect for the method, someone calls the method through the proxy Spring provides. The proxy applies the aspect logic and then further delegates the call to the actual method

How aspects work in Spring (6/7)

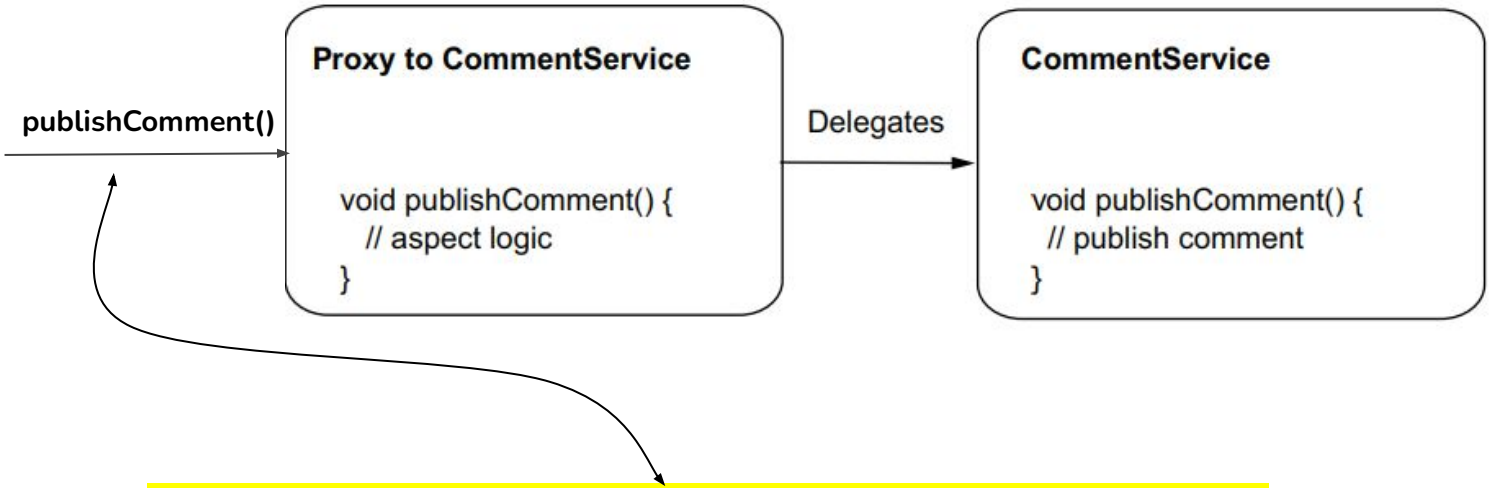
Without aspect



When the method isn't intercepted by aspects, someone calling the publishComment() method directly calls the logic implemented in the CommentService class

How aspects work in Spring (7/7)

With aspect



When we define an aspect for the method, someone calls the method through the proxy Spring provides. The proxy applies the aspect logic and then further delegates the call to the actual method

Implementing a simple aspect(1/7)

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>5.3.19</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>5.2.8.RELEASE</version>
</dependency>
```

We need this dependency to implement the aspects

Implementing a simple aspect(2/7)

```
@Data
public class Comment {
  private String comment;
  private String author;
}
```

We use the stereotype annotation to make this a bean in the Spring context.

To log a message in the app's console every time someone calls the use case, we use a logger object.

This method defines the use case for our demonstration.

```
@Service
public class CommentService {

   private Logger logger =
Logger.getLogger(CommentService.class.getName());

   public  void publishComment(Comment comment) {
       logger.info(" Publish Comment : " + comment.getComment());
   }
}
```

Implementing a simple aspect(3/7)

```
@Configuration
@ComponentScan(basePackages = "com.example.demo.service")
public class ProjectConfig {


}
```

We use @ComponentScan to tell Spring where to search for classes annotated with stereotype annotations.

Implementing a simple aspect(4/7)

```
@SpringBootApplication
public class Demo4Application {

    public static void main(String[] args) {
    ApplicationContext context = SpringApplication.run(Demo4Application.class,args);
        CommentService service = context.getBean(CommentService.class);

        Comment comment = new Comment();
        comment.setComment("This is a comment");
        comment.setAuthor("John Doe");

        service.publishComment(comment);
    }
}
```

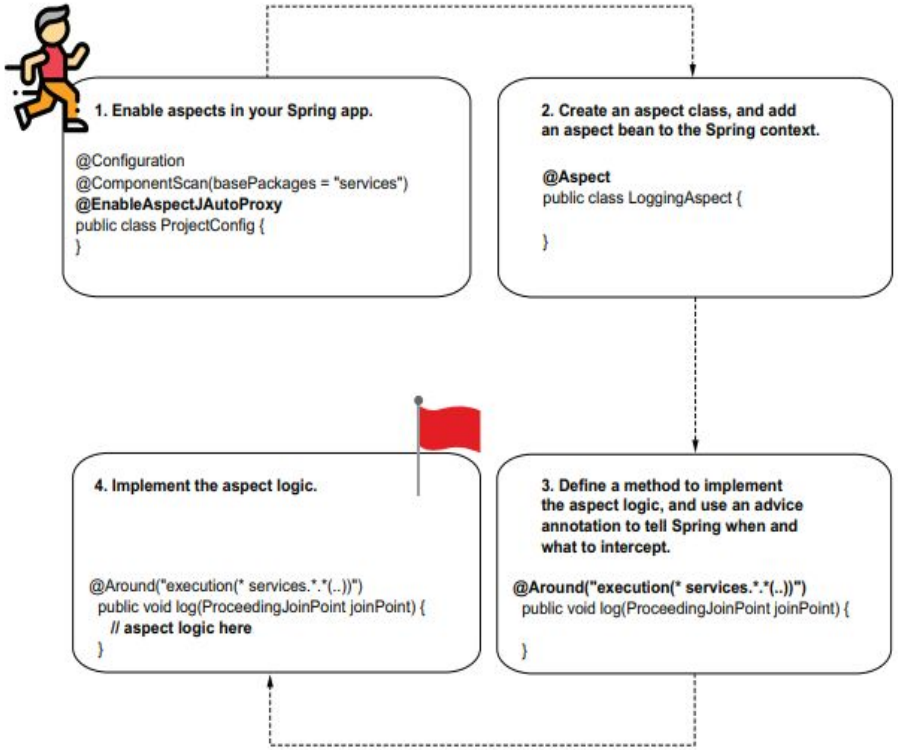
Gets the commentService bean from the context

Creates a Comment instance to give as a parameter to the publishComment() method

Calls the publishComment() method

```
: Started Demo4Application in 2.015 seconds (process running for 2.702)
:  Publish Comment : This is a comment
```

Implementing a simple aspect(5/7)



Implementing a simple aspect(6/7)

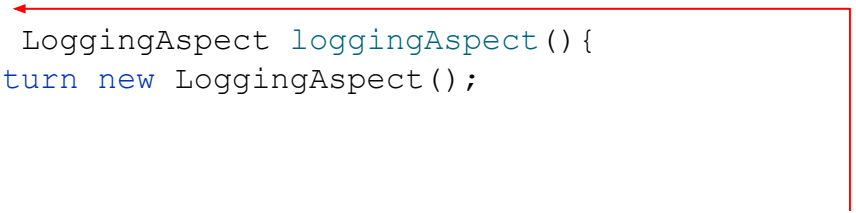
```
@Configuration
@ComponentScan(basePackages = "com.example.demo.service")
@EnableAspectJAutoProxy
public class ProjectConfig {

}
```

Enables the aspects mechanism in our Spring app

```
@Aspect
@Component
public class LoggingAspect {
  public void log(){
      // to implement later
  }
}
```

Implementing a simple aspect(7/7)

```
@Configuration
@ComponentScan(basePackages = "com.example.demo.service")
@EnableAspectJAutoProxy
public class ProjectConfig {

    @Bean
    public LoggingAspect loggingAspect(){
        return new LoggingAspect();
    }

}
```



Adds an instance of the LoggingAspect class to the Spring context

USE AN ADVICE ANNOTATION TO TELL SPRING WHEN AND WHICH METHOD CALLS TO INTERCEPT

Defines which are the intercepted methods

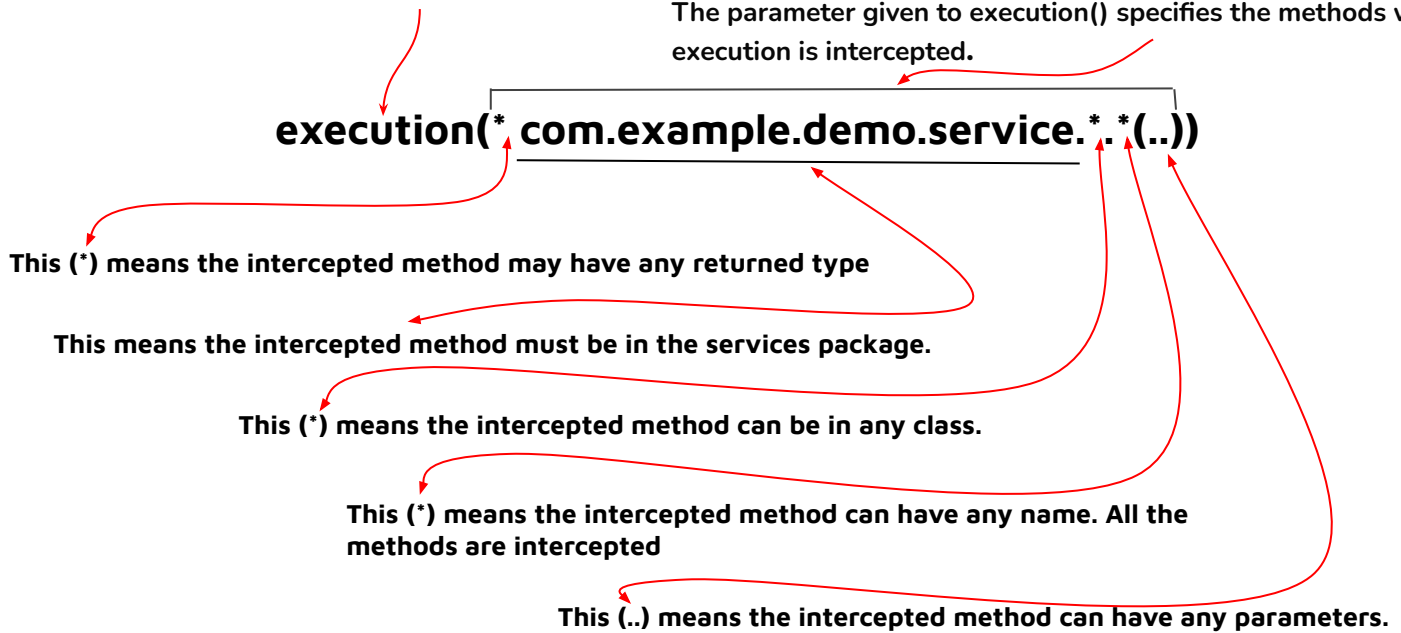
```
@Aspect
public class LoggingAspect {
    @Around("execution(* com.example.demo.service.*.*(..))")
    public void  log(ProceedingJoinPoint joinPoint)throws Throwable {
      joinPoint.proceed();
    }
}
```

Delegates to the actual intercepted method

USE AN ADVICE ANNOTATION TO TELL SPRING WHEN AND WHICH METHOD CALLS TO INTERCEPT

execution() is equivalent to saying “When the method is called . . .”

The parameter given to execution() specifies the methods whose execution is intercepted.



Implementing the aspect logic (1/2)

Prints a message in the console before the intercepted method's execution

```
@Aspect
public class LoggingAspect {
    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());
    @Around("execution(* com.example.demo.service.*.*(..)) ")
    public void  log(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info("Method will execute");
        joinPoint.proceed();
        logger.info("Method done");
    }
}
```

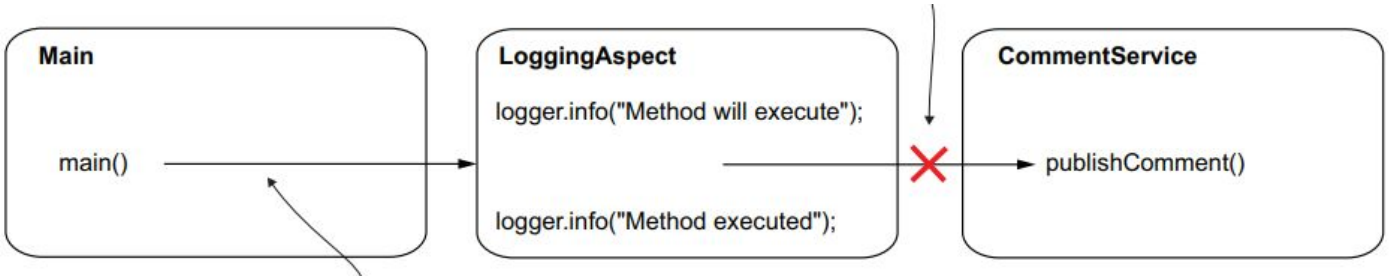
Calls the intercepted method

Prints a message in the console after the intercepted method's execution

The method proceed() of the ProceedingJoinPoint parameter calls the intercepted method, publishComment(), of the CommentService bean. If you don't call proceed(), the aspect never delegates further to the intercepted method

Implementing the aspect logic (2/2)

If you don't call the proceed() method of the ProceedingJoinPoint parameter, the aspect never delegates to the intercepted method



The aspect executes its logic and directly returns to the main() method. For the main() method, it still looks like publishComment() method executed.

Obtaining the method name and parameters in the aspect logic(1/5)

```
@Aspect
public class LoggingAspect {
   private Logger logger = Logger.getLogger(LoggingAspect.class.getName());
   @Around("execution(* com.example.demo.service.*.*(..))")
   public Object  log(ProceedingJoinPoint joinPoint) throws Throwable {

        String methodName = joinPoint.getSignature().getName();
        Object[] arguments = joinPoint.getArgs();

        logger.info("Method " + methodName + " with parameters " + Arrays.asList(arguments) +
"will execute");

        Object returnedByMethod = joinPoint.proceed();
        logger.info("Method executed and returned " + returnedByMethod);
        return returnedByMethod;

   }
}
```

Obtains the name and parameters of the intercepted method

Calls the intercepted method

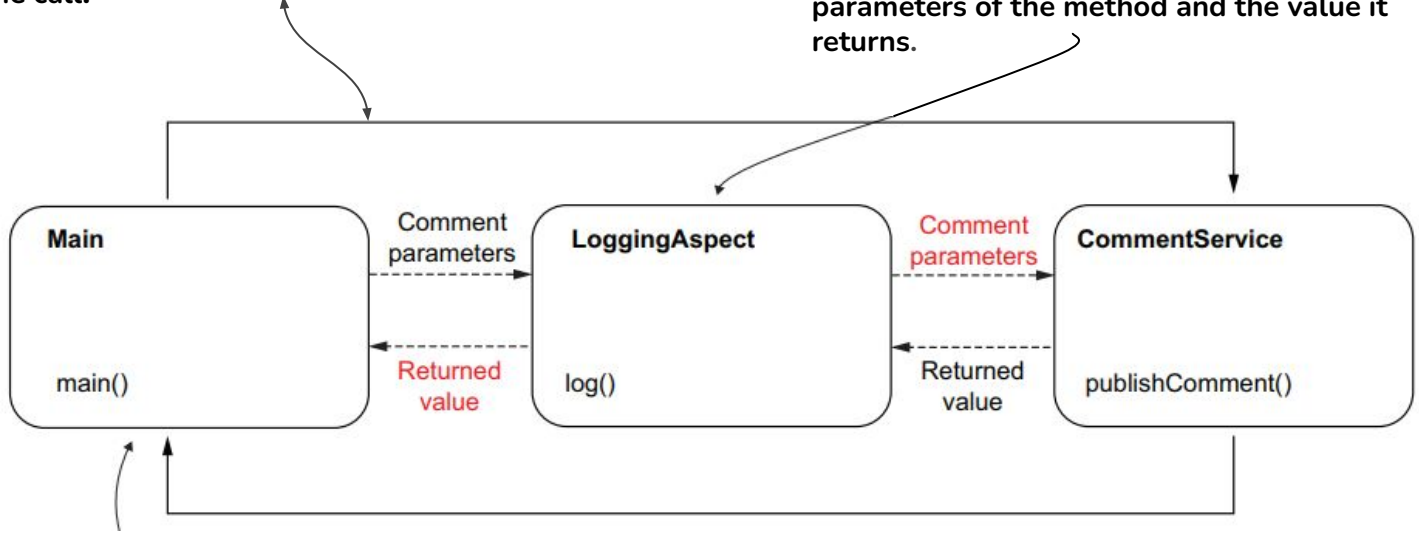
Logs the name and parameters of the intercepted method

Returns the value returned by the intercepted method

Obtaining the method name and parameters in the aspect logic(2/5)

The main() method calls publishComment() of the CommentService bean, but an aspect intercepts the call.

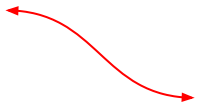
The aspect logs the call as well as the parameters of the method and the value it returns.



The main() method is unaware of the aspect's existence. From its side, it looks like it directly calls the publishComment() method of the CommentService bean

Obtaining the method name and parameters in the aspect logic(3/5)

```
@SpringBootApplication
public class Demo4Application {

    private static final Logger log = LoggerFactory.getLogger(Demo4Application.class);

    public static void main(String[] args) {

        ApplicationContext context = SpringApplication.run(Demo4Application.class, args);
        CommentService service = context.getBean(CommentService.class);

        Comment comment = new Comment();
        comment.setComment("This is a comment");
        comment.setAuthor("John Doe");

        String value = service.publishComment(comment);
        log.info(value);
    }
}
```



Prints the value returned by the publishComment() method

Obtaining the method name and parameters in the aspect logic(4/5)

```
: Method publishComment with parameters [Comment(comment=This is a comment, author=John Doe)]will execute
:  Publish Comment : This is a comment John Doe
: Method executed and returned SUCCESS
: SUCCESS
```

Parameters printed by the aspect

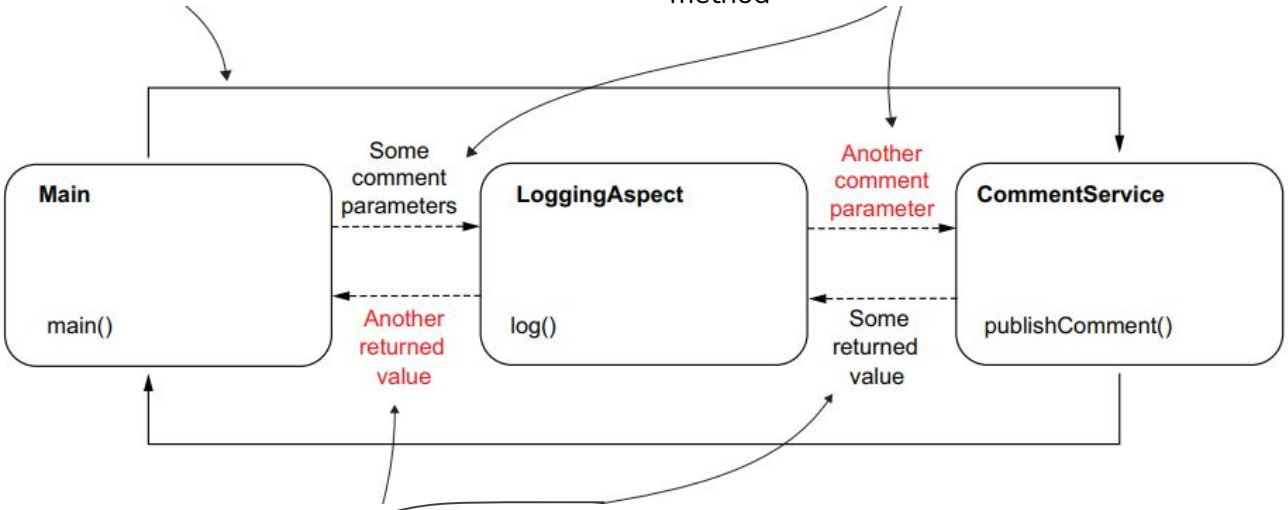
Message printed by the intercepted method

Returned value printed by the aspect

Obtaining the method name and parameters in the aspect logic(5/5)

The **main()** method calls **publishComment()** of the CommentService bean, but an aspect intercepts the call.

When calling the **publishComment()** method, **main()** sent a parameter, but the aspect changed the value of this parameter when further calling the intercepted method



The **publishComment()** method returned a value, but the aspect changed this value when returning it to **main()**. For the **main()** method it looks like the changed value comes directly from **publishComment()**

Altering the parameters and the returned value

```
@Aspect
public class LoggingAspect {
    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

    @Around("execution(* com.example.demo.service.*.*(..))")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {

        String methodName = joinPoint.getSignature().getName();
        Object[] arguments = joinPoint.getArgs();

        logger.info("Method " + methodName +
                " with parameters " + Arrays.asList(arguments) +
                "will execute");

        Comment comment = new Comment();
        comment.setComment("Some other text ! ");
        Object[] newArguments = {comment};

        Object returnedByMethod = joinPoint.proceed(newArguments);

        logger.info("Method executed and returned " + returnedByMethod);
        return "FAILED";

    }
}
```

We send a different comment instance as a value to the method's parameter

We log the value returned by the intercepted method, but we return a different value to the caller

Intercepting annotated methods (1/4)

1.Define a custom annotation.

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

2.Use an AspectJ pointcut expression to tell the aspect to intercept the method with the newly created annotation.

```
@Aspect
public class LoggingAspect {
    @Around("@annotation(ToLog)")
    public Object log(ProceedingJoinPoint jp) {
        // Omitted code
    }
}
```

The steps for intercepted annotated methods. You need to create a custom annotation you want to use to annotate the methods your aspect needs to intercept. Then you use a different AspectJ pointcut expression to configure the aspect to intercept the methods annotated with the custom annotation you created

Intercepting annotated methods (2/4)

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

Enables the annotation to be intercepted at runtime

Restricts this annotation to only be used with methods

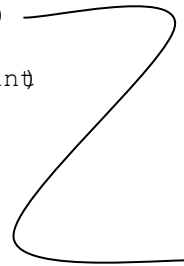
Intercepting annotated methods (3/4)

```
@Service
public class CommentService {

   private Logger logger =
Logger.getLogger(CommentService.class.getName());

   public  String publishComment(Comment comment) {
       logger.info(" Publish Comment : " + comment.getComment()
);
       return "SUCCESS";
   }
   @ToLog
   public void deleteComment(Comment comment) {
       logger.info(" Delete Comment : " + comment.getComment();
   }
   @ToLog
   public  void editComment(Comment comment) {
       logger.info(" Edit Comment : " + comment.getComment() );
   }
}
```

We use the custom annotation for the methods we want the aspect to intercept.

Intercepting annotated methods (4/4)

```
@Aspect
public class LoggingAspect {
   private Logger logger =

Logger.getLogger(LoggingAspect.clas
s.getName());
   @Around("@annotation(ToLog)")
   public Object
log(ProceedingJoinPoint joinPoint)
throws Throwable {
       // Omitted code
   }
}
```



```
@Service
public class CommentService {

   private Logger logger =
Logger.getLogger(CommentService.class.getName())
;

   public  String publishComment(Comment comment)
{
       logger.info(" Publish Comment : "+
comment.getComment() );
       return "SUCCESS";
   }
   @ToLog
   public void deleteComment(Comment comment) {
       logger.info(" Delete Comment : "+
comment.getComment() );
   }
   public  void editComment(Comment comment) {
       logger.info(" Edit Comment : "+
comment.getComment() );
   }
}
```

Other advice annotations you can use

@AfterReturning—Calls the method defining the aspect logic after the method successfully returns, and provides the returned value as a parameter to the aspect method. The aspect method isn't called if the intercepted method throws an exception.

Optionally, when you use @AfterReturning, you can get the value returned by the intercepted method. In this case, we add the "returning" attribute with a value that corresponds to the name of the method's parameter where this value will be provided.

```
@Aspect
public class LoggingAspect {
   private Logger logger = Logger.getLogger(LoggingAspect.class.getName());
   @AfterReturning(value = "@annotation(ToLog)" ,
                           returning = "returnedValue")
   public void log(Object returnedValue) {
       logger.info("Returned value: " + returnedValue);

   }
}
```

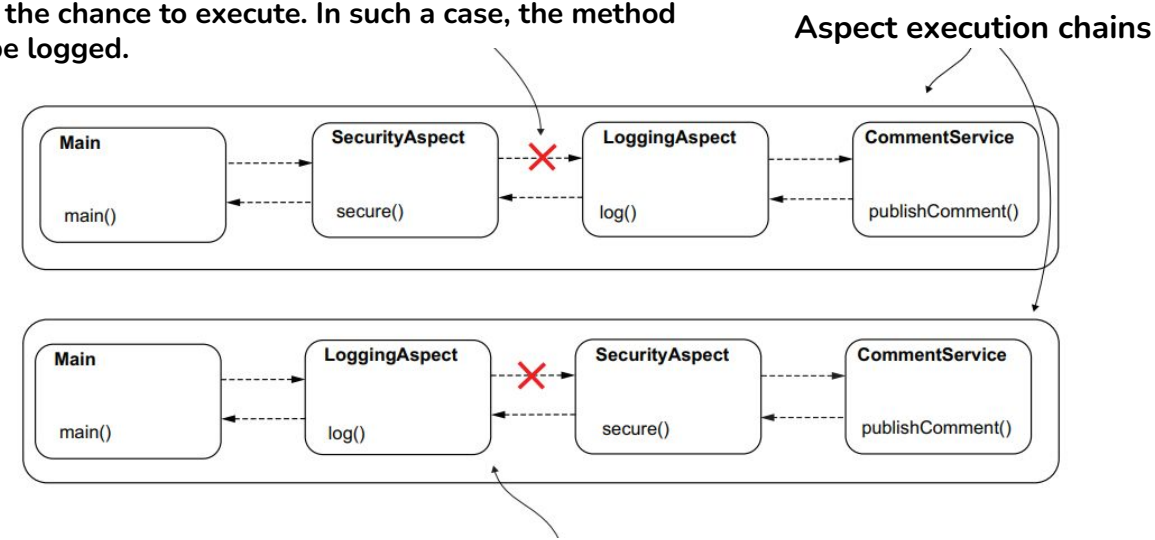
The parameter name should be the same as the value of the "returning" attribute of the annotation or missing if we don't need to use the returned value.

The aspect execution chain(1/8)

- **SecurityAspect**—Applies the security restrictions. This aspect intercepts the method, validates the call, and in some conditions doesn't forward the call to the intercepted method (the details about how the SecurityAspect works aren't relevant for our current discussion; just remember that sometimes this aspect doesn't call the intercepted method).
- **LoggingAspec**t—Logs the beginning and end of the intercepted method execution.

The aspect execution chain(2/8)

In some cases, the SecurityAspect doesn't further delegate. So if the SecurityAspect is executed first, the LoggingAspect won't always have the chance to execute. In such a case, the method calls won't be logged.



If we expect the LoggingAspect to log all the calls, even those that were rejected by the SecurityAspect, we need to make sure the LoggingAspect executes first

The aspect execution chain(3/8)

```
@Aspect
@Component
public class LoggingAspect {
   private static final Logger logger =
Logger.getLogger(LoggingAspect.class.getName());

   @Around(value = "@annotation(ToLog)")
   public Object secure(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
       logger.info("Security aspect : Calling the intercepted method");

       Object returnValue = proceedingJoinPoint.proceed();
       logger.info("Security Aspect : Method executed and returned " + returnValue);
       return returnValue;

   }
```

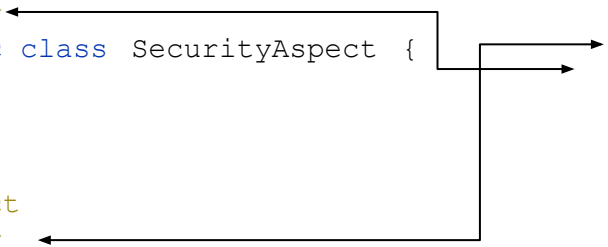
The proceed() method here delegates further in the aspect execution chain. It can call either the next aspect or the intercepted method

The aspect execution chain(4/8)

```
@Service
public class CommentService {

    private Logger logger = Logger.getLogger(CommentService.class.getName());
    @ToLog
    public  String publishComment(Comment comment) {
        logger.info(" Publish Comment : " + comment.getComment() );
        return "SUCCESS";
    }
}
```

```
@Aspect
@Order
public class SecurityAspect {

}
```

```
@Aspect
@Order
public class LoggingAspect {
    // Omitted code
}
```

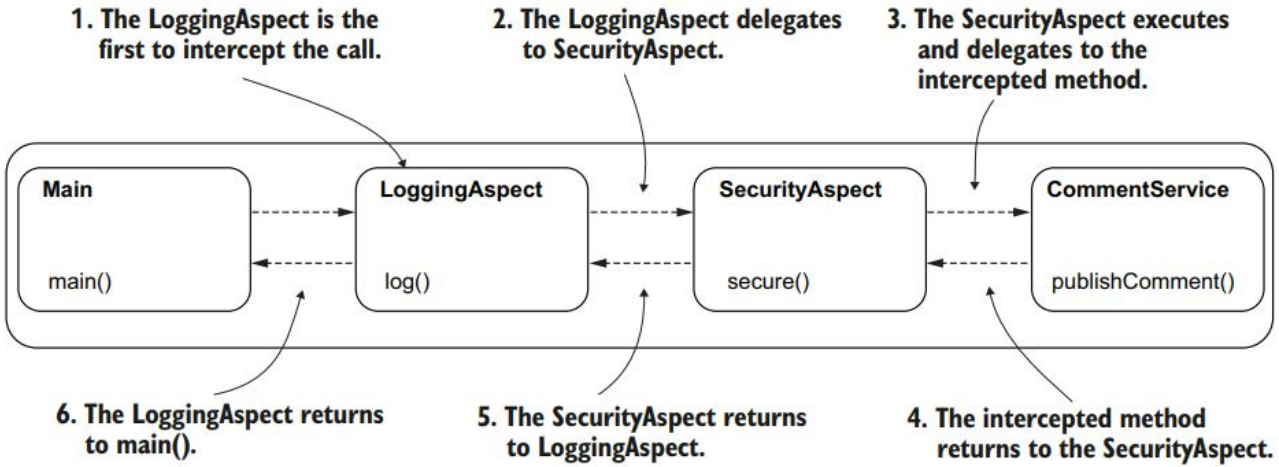


Gives an execution order position to the aspect

The aspect execution chain(5/8)

```
: Logging Aspect : Calling the intercepted method
:  Publish Comment : This is a comment
: Logging Aspect : Method executed and returned SUCCESS
```

The aspect execution chain(6/8)



The aspect execution chain(7/8)

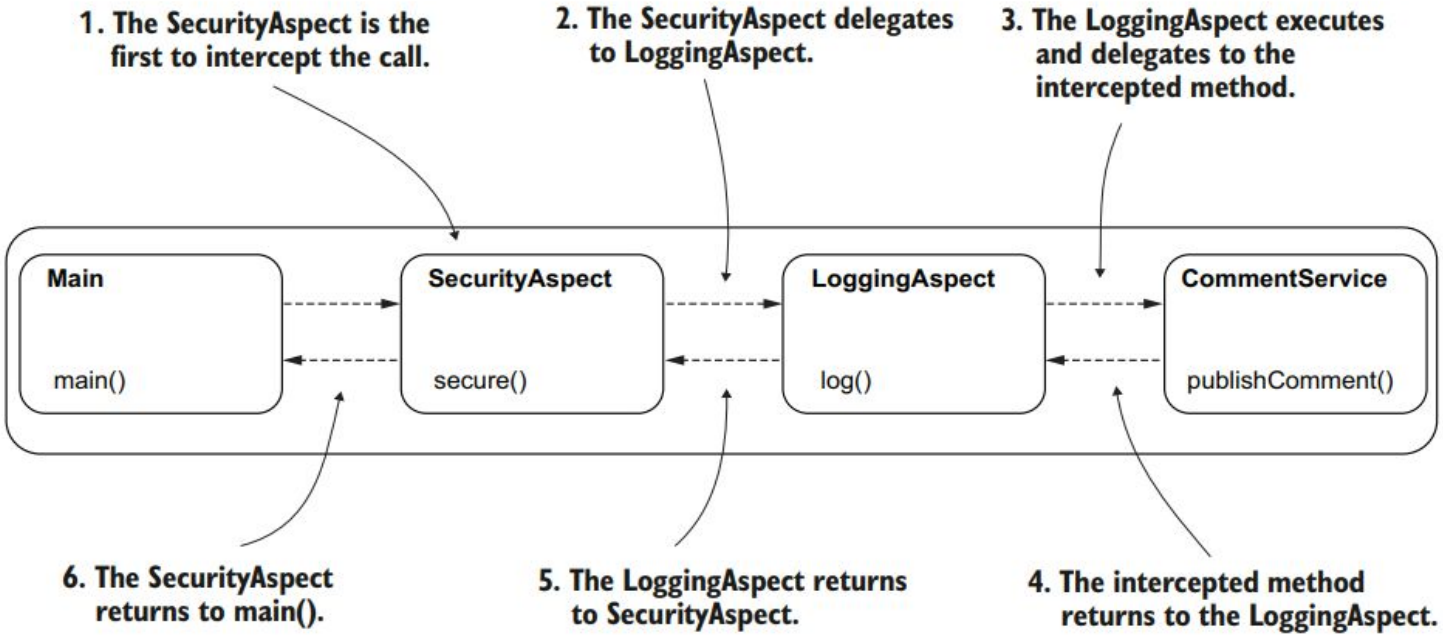
```
@Aspect
@Order(1)
public class SecurityAspect {
    // Omitted code
}
```

Gives an execution order position to the aspect

```
@Aspect
@Order(2)
public class LoggingAspect {
    // Omitted code
}
```

Places the LoggingAspect as second to be executed

The aspect execution chain(8/8)



Summary(1/3)

- An aspect is an object that intercepts a method call and can execute logic before, after, and even instead of executing the intercepted method. This helps you decouple part of the code from the business implementation and makes your app easier to maintain.

- Using an aspect, you can write logic that executes with a method execution while being completely decoupled from that method. This way, someone who reads the code only sees what’s relevant regarding the business implementation

Summary(2/3)

- However, aspects can be a dangerous tool. Overengineering your code with aspects will make your app less maintainable. You don’t need to use aspects everywhere. When using them, make sure they really help your implementation.

- Aspects support many essential Spring capabilities like transactions and securing methods.

- To define an aspect in Spring, you annotate the class implementing the aspect logic with the @Aspect annotation. But remember that Spring needs to manage an instance of this class, so you need to also add a bean of its type in the Spring context

Summary(3/3)

- To tell Spring which methods an aspect needs to intercept, you use AspectJ pointcut expressions. You write these expressions as values to advice annotations. Spring offers you five advice annotations: @Around, @Before, @After, @AfterThrowing, and @AfterReturning. In most cases we use @Around, which is also the most powerful.

- Multiple aspects can intercept the same method call. In this case, it’s recommended that you define an order for the aspects to execute using the @Order annotation

REFERENCE

1: Spring Start Here

Thank you!

Presented by

Tokhirjon Sadullaev

(tohirjonsadullayev387@gmail.com)