**SPRING AOP**

**What is the concept of AOP? Which problem does it solve? What is a cross cutting concern?**

Layered applications have a number of concerns like logging, security (validation), Auditing, Exception handling, API data analytics or performance tracing. These concerns that cut across layers (web, business and data) are called cross-cutting concerns. Ideally, you would want to implement the cross-cutting concerns in one place and apply them across multiple layers. The programming paradigm that allows you to do that is called AOP - Aspect Oriented Programming. Spring AOP and AspectJ are the preferred options to implement cross-cutting concerns.

1) AOP Prevents code scattering and code changes at multiple classes.

Implementing cross-cutting concerns in each layer (web, service or repo) would make them difficult to maintain. Ideally, you would want to implement the cross-cutting concerns in one place and reuse them across multiple layers.

2) Prevents code tangling by separating AOP logic from main business logic. Separation of concerns is achieved between different Aspects (classes) by having separate aspect/class for logging , separate aspect/class for security etc.

3) AOP is configurable - AOP PointCut expressions allow us to apply cross-cutting concerns selectively to different parts of the application.

4) AOP apply Proxy Design Pattern. Good spies listening to normal Java code calls and perform cross cutting concerns.

**What are important AOP Terminologies?**

***1) An Advice -***

The feature that we want to implement can be logging, security (validation), Auditing, Exception handling, API data analytics or performance tracing. This is called as advice. What action is taken and when it should be applied?

AspectJ provides a number of interception points (advices) for method calls , including the following:

* @Before: Before executing a method.
* @After: After executing a method. This executed even if the method throws an exception. Also called as After Finally since it runs after finally block is executed.
* @AfterReturning: After successful execution of a method.
* @AfterThrowing: After a method call which resulted in an exception being thrown.
* @Around: Take complete control of method execution. @Around advice can be used to do something before and after the method execution.

***2) A PointCut -***

Where do you want to apply this advice? To every method in web, business, and data layers. How do you identify these methods? You would need to define a PointCut. A PointCut identifies which methods an advice should be applied to. It is a predicate expression for where advice should be applied.

***3) An Aspect -***

The combination of a PointCut and an advice is called an aspect. Aspect is a class or module of code for cross cutting concern.

***4) A JoinPoint -***

A JoinPoint is the runtime result of AOP. A specific instance where the PointCut matches and an advice is being executed is called a JoinPoint. You can get the details of the method name, method signature and arguments by using JoinPoint.

***5) A ProceedingJoinPoint -***

A ProceedingJoinPoint exposes the proceed method to support @Around advice.

***6) Weaving -***

Basically Weaving refers to the process of injecting functionality into an existing program. Weaving is the process of linking aspects with other outsider application types or objects to create an advised object. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime only. In contrast, the AspectJ framework supports both compile-time and runtime weaving.

**What is difference between Spring AOP and AspectJ?**

1. AspectJ is the most popular AOP framework for Java. AspectJ provides compile-time and runtime weaving. Spring AOP provides integration with AspectJ and a few basic AOP features of its own, and Spring AOP does runtime weaving .
2. AspectJ is faster whereas Spring AOP is slower than AspectJ.
3. AspectJ Join points are method, constructor, field level. However, Spring AOP only supports method level join points.
4. AspectJ has complete "AOP Support" and is complex. Spring AOP is light implementation which is simpler to use, but you can only intercept method calls on Spring beans.
5. If you are working with Spring beans and want to intercept method calls on Spring beans, then Spring AOP is sufficient. If you want to intercept method calls on objects that are not managed by the Spring container, you will need to use the complete AOP framework—AspectJ.
6. Spring AOP uses some of the AspectJ annotations and classes. Even though we are using Spring AOP, we still need AspectJ jar files.

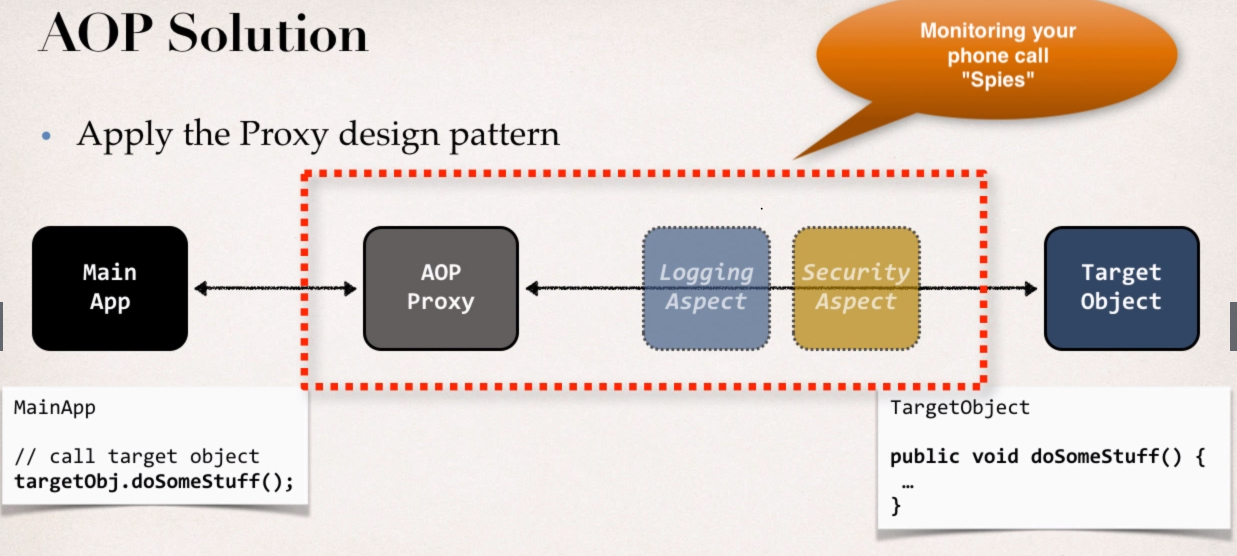
**What is Spring AOP Proxy?**

A proxy is a well-used design pattern. To put it simply, a proxy is an object that looks like another object, but adds special functionality behind the scene.

Spring AOP is proxy-based. AOP proxy is an object created by the AOP framework in order to implement the aspect contracts in runtime.

Spring AOP defaults to using standard JDK dynamic proxies for AOP proxies. This enables any interface (or set of interfaces) to be proxied. It involves creating some class that implements the same interface as the proxied bean

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes, rather than interfaces. It involves creating subclass for the proxied bean.



**Which are the limitations of the two proxy-types?**

***Common limitations:***

1)Proxying works only when calls to methods are beyond the limits of a bean. If one method calls another on internally in a bean – that call will not be intercepted.

2)Proxied objects must be created by the Spring IoC container.

3)Proxies are not serializable.

***JDK dynamic proxying has the following limitations:***

1)Can be applied only on objects that implement an interface

***CGLib proxying limitations:***

1)Final methods cannot be advised (remember you have to create a subclass for the proxied object so it won’t be possible because of the final modifier)

2)The proxied class must have a default constructor

3)Resulting proxy object is final – you won’t be able to proxy a proxy

**What visibility must Spring bean methods have to be proxied using Spring AOP?**

Only public methods of Spring beans will be proxied. Additionally, the call to the public method must originate from outside of the Spring bean.

**What do you have to do to enable the detection of the @Aspect annotation?**

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and autoproxying beans based on whether or not they are advised by those aspects.

To enable @AspectJ support with Java @Configuration add the @EnableAspectJAutoProxy annotation:

***@Configuration***

***@EnableAspectJAutoProxy***

***public class AppConfig {***

***}***

To apply AOP, Spring creates proxies. By default, it creates JDK dynamic proxies, which are interface-based. It’s possible to create proxies by relying on CGLIB. To enable CGLIB, you need to set the attribute proxyTargetClass=true on the @EnableAspectJAutoProxy annotation.

To enable @AspectJ support with XML based configuration use the <aop:aspectj-autoproxy/> element:

***<aop:aspectj-autoproxy/>***

***Spring Boot*** provides a special AOP starter library that removes a little of the hassle of configuration. The @EnableAspectJAutoProxy(proxyTarget Class = true) annotation is no longer needed because the AOP Spring support is already enabled by default. The attribute does not have to be set anywhere either because Spring Boot automatically detects what type of proxies you need.

**What is PointCut expression? Give some examples of PointCut expressions?**

PointCut Expression is a predicate expression defined with advice to define where this advice should be applied.

***@Before("execution(public void Addacount())")***

***public void beforeAdvice(JoinPoint joinPoint) { .... }***

Execute method beforeAdvice() before execution of target method defined in PointCut Expression (method Addacount())

***PE Syntax ---***

***execution(modifiers ? return-type declaring type ? method-name (parameter) throws ?)***

All ? are optional -- modifiers, declaring type and throws

optional arguments when not present treated as any (\*)

***Some Examples ---***

***execution(public void com.dao.AccDAO.addAccount())*** -- match addAccount() method in com.dao pkg and AccDAO class with 'public' modifier and 'void' return type throwing any exception (0 or more)

* modifiers - public
* return-type - void
* declaring type - com.dao.AccDAO
* method-name(parameter) - addAccount() method with no parameters
* throws - Optional - (0 or more)

***execution(public void addAccount())*** -- match addAccount() method in any pkg and any class with 'public' modifier and 'void' return type throwing any exception (0 or more)

* modifiers - public
* return-type - void
* declaring type - Optional - any
* method-name(parameter) - addAccount() method with no parameters
* throws - Optional - (0 or more)

***execution(\* com.mastering.spring.aop.repo.\*Dao.\*(..))*** -- match any method in com.mastering.spring.aop.repo package in class name ending with "Dao" with any return type with zero of more arguments with any modifier throwing any exception (0 or more)

* modifiers - Optional - any
* return-type - \* (any)
* declaring type - com.mastering.spring.aop.repo.\*Dao
* method-name(parameter) - \*(..) -- any method with zero of more parameters
* throws - Optional - (0 or more)

***execution(public void add\*())*** -- match method starting with add in any pkg and any class with 'public' modifier and 'void' return type throwing any exception (0 or more)

* modifiers - public
* return-type - void
* declaring type - Optional - any
* method-name(parameter) - \*(..) -- any method with zero of more parameters
* throws - Optional - (0 or more)

***execution(public void addAccount(\*))*** -- match addAccount(\*) method in any pkg and any class with 'public' modifier and 'void' return type throwing any exception (0 or more)

* modifiers - public
* return-type - void
* declaring type - Optional - any
* method-name(parameter) - addAccount(\*) method with one parameter of any type
* throws - Optional - (0 or more)

***execution(public void addAccount(com.demo.Account))*** -- match addAccount(com.demo.Account) method in any pkg and any class with 'public' modifier and 'void' return type throwing any exception (0 or more)

* modifiers - public
* return-type - void
* declaring type - Optional - any
* method-name(parameter) - addAccount(com.demo.Account) method with one parameters of type com.demo.Account - Here param type must be fully qualified class name
* throws - Optional - (0 or more)

Match all methods defined in any class inside package com.howtodoinjava

***within(com.howtodoinjava.\*)***

Match all methods defined in classes inside package com.howtodoinjava and classes inside all sub-packages as well -- For including, sub-packages use two dots.

***within(com.howtodoinjava..\*)***

***PointCut Declarations***

To reuse PointCut expression in multiple advices, create a PointCut declaration using @PointCut annotation once and apply it to multiple advices.

***@Aspect***

***@Configuration***

***public class MyAspect {***

***// Define PointCut declaration in Aspect class and reuse it for different advices***

***@PointCut("\* com.aop.dao.\*.\*(..))")***

***private void anymethodName() {} // Name of PointCut declaration***

***@Before("anymethodName()")***

***public void beforeAddAccountAdvice(){.....}***

***@Before("anymethodName()")***

***public void peformDataAnalytics(){.....}***

***}***

Common PointCut Declarations can be combined in a separate Utility class with @Aspect annotation. These PointCut declarations must be public in order to be accessed from external aspects/classes. We need to use fully qualified class name of Utility class followed by PointCut Declaration name in order to access it in external aspects/classes.

***@Aspect***

***public class Utility {***

***@Pointcut("execution(\* com.mastering.spring.ch03aopwithspring.OrderDao.\*(..))")***

***public void orderDaoExecution() {}***

***@Pointcut("execution(\* com.mastering.spring.ch03aopwithspring.StockDao.\*(..))")***

***public void stockDaoExecution() {}***

***@Pointcut("execution(\* com.mastering.spring.ch03aopwithspring.business.\*.\*(..))")***

***public void allBusinessLayerMethods() {}***

***@Pointcut("execution(\* com.mastering.spring.ch03aopwithspring.data.\*.\*(..))")***

***public void allDataLayerMethods() {}***

***}***

We can use Logical operators like &&, || and ! when using PointCuts

***@Before("pdOne() && (!(pdTwo() || pdThree()))")***

**How to control order of advices?**

When advices match on same PointCut Expression, or condition then order of execution is undefined (random) by default.

To solve this issue, we can use @Order annotation.

1. Place such conflicting advices in separate Aspects/classes
2. Control order on Aspects using @Order(<number>) annotation to Aspects/classes

***Some Rules --***

* Lower number - Higher precedence
* Numbers can be negative too
* No need of consecutive numbers
* It two orders are same numbers then order of execution is undefined (random)

**How @After, @AfterReturning, @AfterThrowing and @Around annotations work together?**

1. @AfterReturning - Executed on Successful method return
2. @AfterThrowing - Executed when method throws an exception
3. @After - Always executed either in success or an exception. It is executed after finally block but before @AfterReturning or @AfterThrowing advices.
4. @After advice do not have access to Exception object. If you need access to Exception object, use @AfterThrowing advice.
5. @AfterThrowing advice do not interfer with normal flow of exception propogation. If you want to stop the exception propagation, use @Around advice

***try {***

***result = theProceedingJoinPoint.proceed();***

***} catch(Exception ex) {***

***// Handle, swallo or stop exception in @Around advice - calling program would never know that exception occured - Give default value to result object instead of returning null***

***// OR simply log exception in @Around advice and rethrow the exception for calling program to handle it again***

***}***