Spring AOP

Aspect-oriented programming enhances the Object-Oriented Programming concept by providing a different way to structure your code. Spring AOP uses AspectJ internally.

Spring AOP enables Aspect-Oriented Programming in spring applications.

An **Aspect** is simply a common feature present across the code (classes, methods, etc). It is the repeated code or logic which you think can have a better structure to manage instead of scattering it across different classes and methods

Why to use AOP?

- It can be used to provide declarative enterprise services such as declarative transaction management for a particular organization or software.
- It allows users to implement custom elements. These elements can be really helpful for adding some additional features and functionalities that were not present in the software at the beginning.

Problem Statement

Let's say we have around 5 methods in a service layer & we need to perform some notification when a method is called as well as when the control exits the method.

To achieve this, we need to call the notification methods during the start & end in all the 5 methods in the service layer.

So, this is a tedious & repetitive work of writing the same code again & again in all the 5 methods. Also, in the future, if we wanted to remove the notification event for one of the methods, we need to remove the code again. So, there will a problem with maintenance too.

Code Snippets

```
3 public class NotificationService {
4
5     public void startService() {
6         //business logics...
7     }
8
9     public void endService() {
10        //business logics...
11    }
12 }
13
```

```
10 public class TransactionService {
11
12     private NotificationService notificationService;
13
14     public void messagingService() {
15         notificationService.startService();
16         //business logics...
17         notificationService.endService();
18     }
19
20     public void emailService() {
21         notificationService.startService();
22         //business logics...
23         notificationService.endService();
24     }
25
26     public void whatsappService() {
27         notificationService.startService();
28         //business logics...
29         notificationService.endService();
30     }
31
32     public void loggingService() {
33         notificationService.startService();
34         //business logics...
35         notificationService.endService();
36     }
37
38 }
```

So, if you notice in TransactionService class, for each & every method we are calling startService() and endService() method from NotificationService. This acts as a boilerplate coding & repetitive work for the developers. Also, if the client requires to stop calling the NotificationService for emailService() and loggingService(), again we need to remove the code from TransactionService class.

Solution

With the help of Spring AOP, the aspect of calling the NotificationService during the start & end of each method can be centralized which reduces the repetitive work & also will be a good fit for the future with less maintenance.

So, Spring AOP dynamically provides a way to add cross-cutting concerns (i.e., calling the Notification Service in this case) using simple pluggable XML Configuration files or by using Java Annotations.

Important Terminologies:

Aspect is nothing but the concern or the functionality that you are trying to implement generally or centrally. Or An aspect represents a class that contains advices, join points etc like transaction management.

Weaving: The process of linking Aspects with an Advised Object. It can be done at load time, compile time or at runtime time. Spring AOP does weaving at runtime.

Pointcut

A pointcut is nothing but, a kind of regular expression that specifies, what are the method calls that need to be intercepted.

Advice is nothing but, what are the action that needs to be done when a Pointcut is met.

JoinPoints

Joint point represents a point in our application where we can plug-in AOP aspect. It can be method execution, exception handling, field access etc. Spring AOP only supports method execution joint type.It is not limited to only execution of methods, but also when an exception is thrown during Exception Handling Management.

Introduction:

It provides the facility to add new methods or attributes to existing classes.

Target object:

It is an object on which advices are applied. It always be proxied object in spring because Spring AOP is implemented using runtime proxies.

Advanatges of AOP:

- AOP is one of the key components of the Spring Framework. It is important to note that Spring IoC Containers is not dependent on AOP. So, it provides the advantage to developers to whether to use AOP or not.
- As an implementation of cross-cutting concerns, functionalities such as Logging, Notification Management, Authentication, Security, Transaction Management can be kept in a centralized manner & can be used across multiple places in the application.
- As AOP is implemented using Java, there is no need for any special compilation unit or class loaders.
- As the cross-cutting concerns are centralized, Boilerplate coding is reduced.
- It becomes easy for the developers to maintain the system.
- Spring AOP provides XML based configuration as well as advanced Java Annotation configuration.

Supported Pointcut Designators

Spring AOP supports the following Pointcut Designators (PCD).

- **execution** – for matching method execution join points. This is the most widely used PCD.
- **within** – for matching methods of classes within certain types e.g. classes within a package.
- **@within** – for matching to join points within types (target object class) that have the given annotation.
- **this** – for matching to join points (the execution of methods) where the bean reference (Spring AOP proxy) is an instance of the given type.
- **target –** for matching with the target object of the specific instance type.
- **@target** – for matching with the target object annotated with a specific annotation.
- **args** – for matching with methods where its arguments are of a specific type.
- **@args** – for matching with methods where its arguments are annotated with a specific annotation.
- **@annotation –** for matching to join points where the subject (method) of the Joinpoint has the given annotation.
- **bean** (idOrNameOfBean) – This PCD lets you limit the matching of join points to a particular named Spring bean or to a set of named Spring beans (when using wildcards).