Spring Aspect Orientated Programming (AOP)

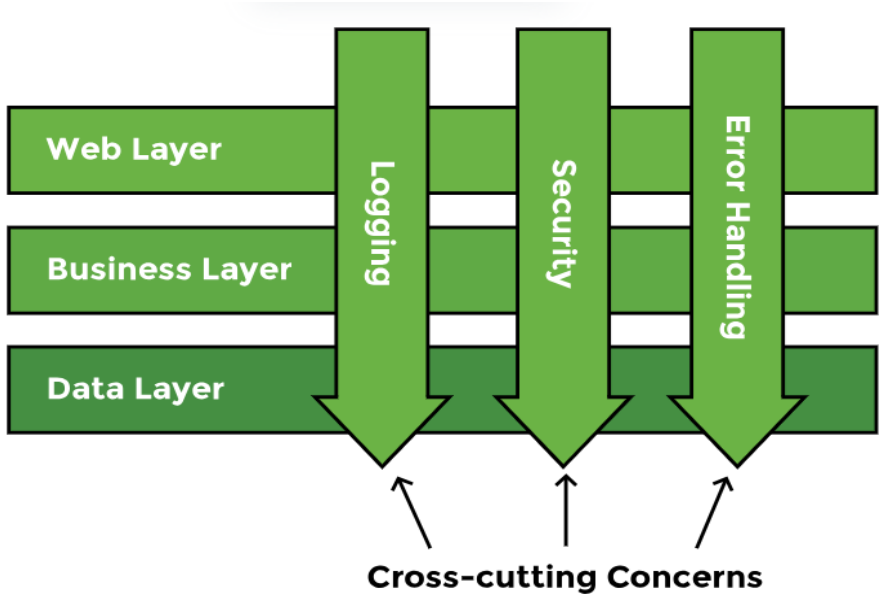
AGENDA

- Aspect Oriented Programming
- Terminology
- Defining an Aspect
- Pointcut Expressions
- After Aspect
- Around Aspect
- JoinPoint Configuration File
- Defining a Custom Annotation for Aspects
- Conclusion

Aspect Oriented Programming

Spring AOP is a powerful feature of the Spring Framework that allows you to modularize cross-cutting concerns in your applications.

Cross-cutting concerns are those aspects of a program that affect multiple components, such as **security**, **logging**, **transaction management**, **auditing**, **error handling**, **performance tracking**, **etc**. By using **Spring AOP**, you can separate these concerns from your core business logic and apply them declaratively or programmatically.



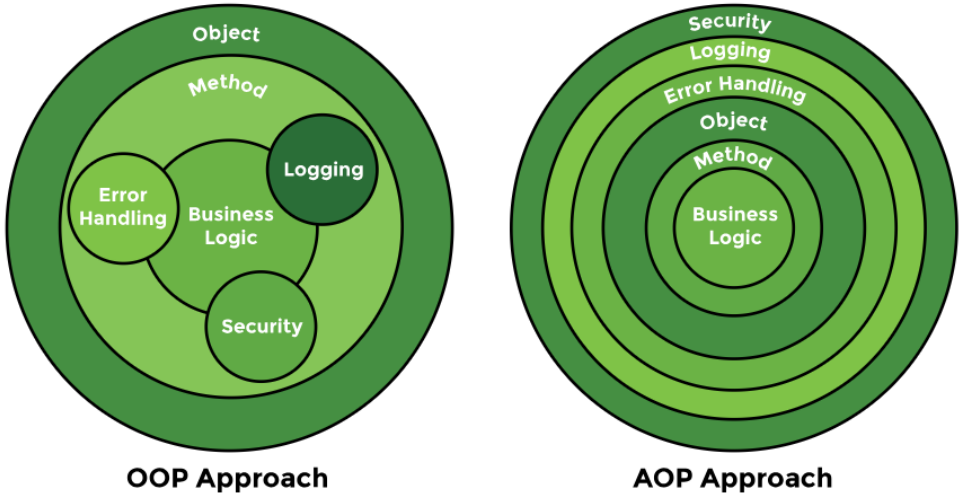
Aspect Oriented Programming

Some of the benefits of using Spring AOP are:

- **It improves the readability and maintainability of your code by separating the cross-cutting concerns from the core logic**.
- **It reduces code duplication and boilerplate code by applying the cross-cutting behavior in a centralized way**.
- **It enables you to change the cross-cutting behavior without affecting the core logic, and vice versa**.
- **It allows you to reuse the cross-cutting functionality across different modules and applications**.

Aspect Oriented Programming

AOP lets you add cross-cutting concerns like logging or performance tracking to any method in any layer. For example, you can log methods for a feature in the web, business, or data layer. With OOP, you have to call the logger from these methods. But with AOP, you can define logging separately and apply it to any method easily. You can also change or remove logging without changing the source code, just by editing the config files.



Aspect Oriented Programming

spring-aop is a popular implementation of AOP provided by Spring. It is not as powerful as AspectJ**.**

Let's now understand AOP with an example. We will create a book recommender application. There are two classes in the business layer implementing two techniques for finding books to recommend. The data layer contains two classes namely User and Book. We proceed as follow :

- **Create a Spring Boot application using the Spring Initializr.** We do not need to add any dependency to the project. Once the project has been created and imported in the IDE, we will add the spring-aop dependency.
- **Create two business classes (into the service layer), FilteringTechnique1 and FilteringTechnique2, and mark them with the @Service annotation.**
- **create two classes, Book and User, in the repository layer and mark them with the @Repository annotation.**
- **We will have the SpringAopDemoApplication class** implement the CommandLineRunner interface which will enable us to write dynamic code

We can start intercepting method call

Terminology

❑ Aspect

An aspect is a class annotated with @Aspect that defines cross-cutting concerns, such as logging, performance, or transaction, and specifies how and when to apply them to other methods.

❑ Pointcut

Pointcuts are expressions with a specific syntax that decide which method calls to intercept and whether to execute an advice or not.

❑ Advice

Advices are methods in an @Aspect class that define the cross-cutting tasks to perform before, around, or after a method call that matches a pointcut.

Terminology

- Joinpoint

Joinpoints are points in the program execution where an aspect can be applied, and they contain the name of the method call that is intercepted by a pointcut and an advice.

- Weaving

Weaving is the process of creating an advised object that links an aspect with a method call in the application and executes the aspect at the right time.

- Weaver

Weaver is the framework that ensures that an aspect is invoked at the right time

Defining an Aspect

Now, We will create a separate package for aspects that log the output of the business layer methods by intercepting their calls before and after execution. To define an aspect for a cross-cutting concern, we will proceed as follow :

- **Aspect** : create a class called AccessCheckAspect and place it in the aspect package and mark it with @Configuration and @Aspect annotations.
- **Advice :** Next, we define a method that contains the logic of the steps that need to be carried out when a method call gets intercepted. We called it userAccess.

- **Pointcut expression :** We need the @Before annotation on our method. It ensures that the advice is run before the method is executed. @Before needs an argument which specifies the method calls that will be intercepted. This is called the pointcut.

- **Jointpoint :** we use a join point as an argument to the method in order to find out which method calls have been intercepted . It contains the name of the method that is intercepted.

Pointcut Expressions

Pointcuts are defined in the following format:

execution(* PACKAGE.*.*(..))

The pointcut expression starts with a key word called a designator, which tells Spring AOP what to match. execution is the primary designator which matches method execution joinpoints.

- The first * in the expression corresponds to the return type. * means any return type.
- Then comes the package name followed by class and method names.
- The first * after package means any class and the second * means any method. Instead of *, we could specify the class name and method name to make the pointcut expression specific.
- Lastly, parentheses correspond to arguments. (…) means any kind of argument.

Pointcut Expressions

The way pointcuts are defined is important because it decides the method calls that will be intercepted

❑ Intercepting all method calls in a package

We have the following pointcut expression :

@Before("execution(* com.aop.springaopdemo.service.*.*(..))")

This pointcut matches any method call in the service package, but if we change the package to repository, it will match only the calls in the repository package.

❑ Intercepting all method calls

The pointcut expression becomes :

@Before("execution(* com.aop.springaopdemo..*.*(..))")

This pointcut matches all method calls in the springaopdemo package, including the SpringAopDemoApplication.run method and the methods in the service and repository layer.

Pointcut Expressions

❑ Intercepting calls using return type

If want to intercept calls to all methods that return a String value. The pointcut expression becomes :

@Before("execution(String com.aop.springaopdemo..*.*(..))")

This pointcut matches any String-returning method call in the springaopdemo package and its subpackages, including two methods in the service package and two in the repository package.

❑ Intercepting calls to a specific method

If we want to intercept calls to all methods that have the word Filtering in it, we will use the following pointcut expression:

Before("execution(String com.aop.springaopdemo..*.*Filtering(..))")

This pointcut matches any method with Filtering in its name, such as contentBasedFiltering() and collaborativeFiltering(). If we change Filtering to Filter, it will match no methods.

Pointcut Expressions

❑ Intercepting calls with specific method arguments

Suppose we have the following pointcut expression :

@Before("execution(* com.aop.springaopdemo..*.*(String))")

This pointcut will match method calls having one parameter of String type. We can modify this expression to match all method calls with String as the first argument as follows:

@Before("execution(* com.aop.springaopdemo..*.*(String, ..))")

❑ Combining pointcut expressions

We can used && , || and ! to combine different pointcut expressions. Here is an example :

Before("execution(* com.aop.springaopdemo..*.*Filtering(..)) || execution(* com.aop.springaopdemo..*.*(..))")

This pointcut matches any method that returns a String or has Filtering in its name, intercepting four methods that meet either condition.

After Aspect

Since Logging is a cross-cutting concern for which aspects can be created. Here we will create another aspect that will log the values returned *after* the methods have been executed . We will proceed as follow:

- **We create a class called LoggingAspect** and place it in the aspect package and mark it with @Configuration and @Aspect annotations.
- **Next, we define a method LogAfterExecution**, which will print a message if the method is successfully executed. We use the @AfterReturning annotation.
- **Next, we define a method LogAfterException**, which will get the result of the exception using the throwing tag. We use the @AfterThrowing annotation.
- **We can also define a method LogAfterMethod** that is used in both scenarios, whether the method execution is successful or results in an exception. We use the @After annotation.

Around Aspect

Here we look at another type of aspect, the around aspect, which is executed around the intercepted method call. It is useful if we want to perform a task before the intercepted method starts execution and after the method has returned.
An example of an @Around aspect is measuring the method execution time. We can record the time before and after the method runs, and calculate the difference. The @Around annotation combines the @Before and @After annotations. We will implement it as follow:

- **We create a class called ExecutionTimeAspect** and place it in the aspect package and mark it with @Configuration and @Aspect annotations.
- **Next, we define a method calculateExecutionTime**, the parameter type of this method will be ProceedingJoinPoint which will allows the continuation of the execution. We use the @AfterReturning annotation.
- **Then, we use the @Around annotation** to define a pointcut for method calls for which we want the execution time to be tracked.

Pointcut Expressions

Creating aspects with the same pointcuts repeatedly is a tedious task for applications with many aspects.
A best practice for AOP is to define all the pointcuts in a separate config file. This makes them easy to manage and use in any aspect.

❑ For a specific layer

We will proceed as follow :

- **We create a class called JoinPointConfig** and define pointcuts using the @pointcut annotation.
- **Next, we use the method that defines this pointcut** in the configuration
- **In the same manner, we can create a pointcut configuration** for intercepting method calls in the service layer

The fully qualified name of serviceLayerPointcut can be used in place of the pointcut definition in AccessCheckAspect without affecting the output

Pointcut Expressions

- For multiple layers

We can also combine pointcuts using the AND (&&), (OR) ||, and (NOT) ! operators. We create a method called allLayerPointcut will intercept calls belonging to either the service layer or the repository layer.

When this pointcut is used in AccessCheckAspect, four method calls are intercepted.

- For a bean

We can also define a pointcut to intercept calls belonging to a particular bean. Say we want to log the execution of all methods belonging to beans that have the word Book in their name.

When this pointcut is used in AccessCheckAspect, it will intercept calls from the Book bean.

Defining a Custom Annotation for Aspects

An aspect can measure the method execution time for a layer, or we can use a custom annotation for any method. Spring AOP lets us create and implement our own annotations with aspects.

Suppose we want to call our annotation @MeasureTime. We will proceed as follow:

- **Create this annotation** in the same folder as the other aspects. This creates an interface
- **We will restrict the use of this annotation to methods only**. This can be achieved using the @Target annotation, with ElementType set to METHOD
- **We would also like the annotation information to be available at runtime**. We will use the @Retention annotation to define a retention policy
- **Now that we have defined our annotation, we can add it to the JoinPointConfig file** and create a pointcut

We can now use this pointcut to calculate the execution time of only chosen methods.

Conclusion

We have learned how to use Spring Aspect Oriented Programming, Here is the GitHub repository Of the code we used

https://github.com/Dilane-Kamga/Spring-Aspect-Oriented-Programming

MERCI

Pour votre attention