Chapter 4 –Aspect Oriented Programming (AOP) with Spring

Aspect oriented programming is a way of programming that aims to increase reusability & mobility of cross cutting functionality which are usually tangled with business logic, which otherwise is difficult to segregate – cross cutting functionalities are auditing, transaction, security, logging etc.

*This is achieved by, adding an additional behavior to the existing code while the code is complied, and directly affect byte code. In codding this is done by, adding* ***advice*** *containing code that will be executed in a location name* ***join-points*** *specified as point-cuts.*

The following are the feature that can be implemented using AOP

* Logging
* Data Validation
* Caching
* Internationalization
* Error detection and correction
* Memory management
* Performance Monitoring
* Synchronization

Spring AOP

Spring AOP – cannot manage object those are NOT managed by the spring container. Spring AOP uses dynamic proxies for weaving aspect – once the target code is compiled the bytecode is not change, instead the target code is effectively wrapped by a runtime object that either implements the same interface OR extend the class of that of target object.

Spring AOP framework comprises of following libraries

* **Spring-AOP:** provides AOP alliance API compliant components that can define method interceptors and pointcuts so that code with different responsibilities can be cleanly decoupled.
* **Spring-aspects: Provides integration with AspectJ**
* **spring-instrument**: provides class instrumentation support and classloader implementations that can be used on application servers.

Spring AOP Terminology

1. **Aspect:** A class containing code specific to a cross-cutting concern. A class declaration is recognized in Spring as an aspect if it is annotated with the @Aspect annotation.
2. **Weaving:** A synonym for this word is interlacing, but in software the synonym is linking and it refers to aspects being combined with other types of objects to create an advised object.
3. **Join point:** A point during the execution of a program. In Spring AOP, a join point is always a method execution. Basically, the join point marks the execution point where aspect behavior and target behavior join.
4. **Target object:** An object to which the aspect applies.
5. **Target method:** the advised method.
6. **Advice:** The action taken by an aspect at a join point. In Spring AOP, there are multiple advice types.

* **Before advice:** Methods annotated with **@Before** that will execute before the join point. These methods do not prevent the execution of the target method unless they throw an exception.
* **After returning advice**: Methods annotated with **@AfterReturning** that will execute after a join point completes normally, meaning that the target method returns normally without throwing an exception.
* **After throwing advice:** Methods annotated with **@AfterThrowing** that will execute after a join point execution ends by throwing an exception.
* **After (finally) advice:** Methods annotated with **@After that** will execute after a join point execution, no matter how the execution ended.
* **Around advice:** Methods annotated with **@Around** intercept the target method and surround the join point. This is the most powerful type of advice since can perform custom behavior before and after the invocation. It has the responsibility of choosing to perform the invocation or return its own value, and it provides the option of stopping the propagation of an exception.

1. **Pointcut**: A predicate used to identify join points. Advice definitions are associated with a pointcut expression and the *advice will execute on any join point matching the pointcut expression*. Pointcut expressions are defined using AspectJ Pointcut Expression Language Pointcut expressions can be defined as arguments for Advice annotations or as arguments for the **@Pointcut** annotation.
2. **Introduction:** Declaring additional methods, fields, interfaces being implemented, and annotations on behalf of another type. Spring AOP allows this using a suite of AspectJ @Declare\* annotations that are part of the aspectjrt library.
3. **AOP proxy:** The object created by AOP to implement the aspect contracts. In Spring proxy objects can be **JDK dynamic proxies** or **CGLIB proxies**. By default, the proxy objects are JDK dynamic proxies, and the object being proxied must implement an interface that is also

implemented by the proxy object. But a library like CGLIB can create proxies by subclassing, so an interface is not needed.

# Page # 285 – Date 08-11-2021

By using @EnableAspectJAutoProxy one can enable aspect support , which helps spring to create Proxies for the beans, default spring ONLY creates JDK dynamic proxying which can be alerted to create CGLIB proxies specifying proxyTargetClass attributes which are also know as subclass proxies example - @EnableAspectJAutoProxy(proxyTargetClass = true).

Difference between JDK dynamic proxy and GCLIB proxies.

|  |  |
| --- | --- |
| **JDK Dynamic Proxy** | **GCLIB Proxy** |
| This is suited for the scenarios where the target class implements one or more interfaces. The proxy class will also implement the same interfaces to inject the desired behavior. | This is more suited for the scenarios where the target class doesn’t implement any interface, in that case spring generates a class at the runtime and extend it from the target class to wrapped it with desired behavior. |
| Default when enable @EnableAspectJAutoProxy | One need to explicitly add property proxyTarget = true  @EnableAspectJAutoProxy (proxyTargetClass = true). |

High level steps that need to be performed in order to implements apects in spring

1. Aspect related dependencies needs to be included
2. Configuration calls containing advice needs to be annotated with @Aspect and also declare it as a bean using (@Compoent OR @Bean)
3. Advice method needs to be annotated with one of the many advice annotations.
4. Finally, one needs to add @EnableAspectJAutoProxy to configuration class for enabling aspectJ.

Aspect is available prior to spring 3.0 version, where one needs to enable to aspectJ configuration using XML annotation. AspectJ has its own namespace which has to be used to enabling the annotation.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <beans xmlns=”<http://www.springframework.org/schema/beans>” xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance **xmlns:aop="http://www.springframework.org/schema/aop** xmlns:context="http://www.springframework.org/schema/context"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd">  <!-- Configuration for the aspects that apply to the application -->  <bean id="personRepoMonitor" class="com.apress.cems.aop.PersonMonitor"/>  <context:component-scan base-package="com.apress.cems.aop" />  **<!-- Configure Aspect support -->**  **<aop:aspectj-autoproxy>**  **<aop:include name="personRepoMonitor"/>**  **</aop:aspectj-autoproxy>**  **<!-- Configure advice -->**  **<aop:config>**  **<aop:aspect ref="personRepoMonitor">**  **<aop:before pointcut="execution(public \* com.apress.cems.repos.\*.JdbcPersonRepo+.findById(..))" method="beforeFindById"/>**  **</aop:aspect>**  **</aop:config>**  </beans> |

**PointCuts**

Pointcuts expression – Template

|  |
| --- |
| **execution([Modifiers] [ReturnType] [FullClassName].[MethodName]([Arguments]) throws [ExceptionType])** |

*The expression can contain wildcards like + and \* and can be made of multiple expressions concatenated by boolean operators such as &&, ||, and so forth. The \* wildcard replaces any group of characters when used to match pieces of package names, classes, and methods, and a single character when used to match method parameters. The + wildcard specifies that the method to advise can also be found in subclasses identified by [FullClassName] criteria. The + wildcard works in a similar way when the criteria used is an interface and the pointcut expression matches the methods in all*

*implementations.*

* *The* ***[ReturnType] is mandatory****. If the return type is not a criterion, just use \*. If it is missing the application crashes at boot time throwing an java.lang.IllegalArgumentException with a message explaining that the pointcut is not well-formed.*
* *The* ***[Modifers] is not mandatory*** *and if not specified defaults to public.*
* *The [****MethodName] is not mandatory****, meaning no exception will be thrown at boot time. But if unspecified, the join point where to execute the advice won’t be identified. It’s safe to say that if you want to define a technically useful pointcut expression you need to specify it.*
* *The* ***[Arguments] is mandatory.*** *If it is missing the application crashes at boot time throwing a java.lang.IllegalArgumentException with a message explaining that the pointcut is not well formed. If the arguments are not a criterion, just use (..) which matches a method with 0 or many arguments. If you want the match to be done on a method with no arguments, use (). If you want the match to be done on a method with a single argument, use (\*).*

***For the ease of maintainability and reusability, the point cut expression can be split into multiple smaller expression OR even can be externalized completely.***

***NamePointCuts – single complex pointcut expression can be split into two simple pointcut expression – name of the method then can be used to identify these pointcuts. That’s why they are called a namePointCuts.***

***Example:***

***//NAME POINT-CUT repoFind()***

@Pointcut ("execution(\* com.apress.cems.\*.\*PersonRepo+.findBy\*(..))")

public void repoFind() { }

***//NAME POINT-CUT serviceFind ()***

@Pointcut ("execution (\* com.apress.cems.aop.service.\*Service+.findBy\*(..)))")

public void serviceFind() { }

***//Composite pointcut that refer serviceFind() and repoFind() named pointCuts.***

@Before("repoFind() || serviceFind()")

public void beforeFind(JoinPoint joinPoint) {

..

}

*NOTE : the name pointCuts can be move to different package to make it completely decoupled from the composite pointCuts.*

*Spring PointCut AOP Designator*

*The args() designator is used to identify methods with a parameter configuration defined by it.*

*Limitation of Spring AOP*

* *ONLY public Join Point can be advised.*
* *Aspect can only be applied to springBean*
* *Even if the spring AOP is not set to use CGLIB , if the target class is not implementing any interface , spring will try to use CGLIB proxy instead of JDK dynamic proxy.*
* *If there are two methods one calling another , pointcut expression matching both the method then the advice will be called only for the first method NOT for the second method.*

*WHO write writes the method for repository custom interfaces?*

*Spring defined an aspect that have a pointcut which executes every time when findBy\* method is called, using spring AOP spring provide the necessary method body for those repository methods.*