CODING SHUTTLE

AOP

Spring Proxy and Internal working of AOP

Spring Proxy

Proxies are created only when certain conditions are met, such as when:

- AOP (Aspect-Oriented Programming) is enabled, and there are pointcuts matching specific methods.
- Spring features like @Transactional, @Cacheable, and @Async are used, which internally rely on proxies to apply cross-cutting concerns like transaction management, caching, or asynchronous method execution.

If a Spring bean doesn't need any cross-cutting functionality (like AOP, transaction management, caching, or async processing), no proxy will be created. Spring optimizes its proxy creation to avoid unnecessary overhead when proxies are not needed.

How Proxies are managed?

Proxies are created and managed by the Spring container (the ApplicationContext). When Spring determines that a bean needs to be proxied (due to AOP advice, transaction management, caching, etc.), it creates a proxy for the bean instead of instantiating the bean directly.

- This proxy wraps around the actual bean and adds additional behavior (advice) before and after the actual method calls.
- Thus, proxies are stored in place of the actual beans in the ApplicationContext. Whenever a client retrieves a bean (e.g., via @Autowired or context.getBean()), what they receive is the proxy object, if proxying is enabled for that bean.

How Proxy are created?

Spring AOP typically relies on dynamic proxies (JDK dynamic proxies or CGLIB proxies) to implement the interception of method calls.

- JDK Dynamic Proxies: If the target object implements one or more interfaces, Spring uses JDK dynamic proxies. The proxy is created at runtime, implementing the same interfaces as the target object.
- CGLIB Proxies: If the target object doesn't implement any interfaces, Spring uses CGLIB, which generates a subclass of the target class at runtime to create the proxy.

When a method on the proxy object is called, the proxy intercepts the method call and applies the advice around it.

How Proxies are used?

1. When Spring starts up, it scans the classes for aspects and applies the advice to the relevant beans based on the pointcuts.
2. For each bean, Spring determines whether it needs to be proxied (based on whether any methods match a pointcut).
3. If proxying is needed, Spring creates a proxy object that wraps the original bean.
4. When a client calls a method on the proxied bean, the call is intercepted by the proxy.The proxy determines if there is any advice that should be applied based on the pointcuts.
5. If the method does not match any pointcuts, the proxy delegates the call directly to the target object without applying any advice.

How Proxies are used?

6. If advice exists, the proxy executes the advice (e.g., logging, transaction management) before, after, or around the actual method invocation.
7. The most powerful type of advice is @Around. It allows you to control the execution of the target method by calling ProceedingJoinPoint.proceed(). You can choose to execute code before and after the target method execution or even decide not to execute it at all.
8. After the advice has been executed, control is returned to the original method.
9. Any results returned by the method or exceptions thrown are then passed back to the caller.

Weaving

In addition to dynamic proxies, Spring AOP can use a process called weaving for bytecode manipulation. *Weaving is the process of injecting aspects into the target class bytecode at different points in the lifecycle.*

- Compile-time Weaving: Aspects are woven into the code during the compilation process.
- Load-time Weaving: Aspects are woven when the class is loaded into the JVM.
- Post-compilation Weaving: Aspects are woven after the classes have been compiled.

Spring primarily uses dynamic proxies, but with AspectJ integration, load-time weaving or compile-time weaving can also be used for more complex scenarios.