org.springframework.context.annotation

Annotation Type EnableAspectJAutoProxy

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Import(value=org.springframework.context.annotation.AspectJAutoProxyRegistrar.class)
public @interface EnableAspectJAutoProxy
```

Enables support for handling components marked with AspectJ's `@Aspect` annotation, similar to functionality found in Spring's `<aop:aspectj-autoproxy>` XML element. To be used on `@Configuration` classes as follows:

```
@Configuration
@EnableAspectJAutoProxy
public class AppConfig {

    @Bean
    public FooService fooService() {
        return new FooService();
    }

    @Bean
    public MyAspect myAspect() {
        return new MyAspect();
    }
}
```

Where `FooService` is a typical POJO component and `MyAspect` is an `@Aspect`-style aspect:

```
public class FooService {

    // various methods
}
```

```
@Aspect
public class MyAspect {

    @Before("execution(* FooService+.*(..))")
    public void advice() {
        // advise FooService methods as appropriate
    }
}
```

In the scenario above, `@EnableAspectJAutoProxy` ensures that `MyAspect` will be properly processed and that `FooService` will be proxied mixing in the advice that it contributes.

Users can control the type of proxy that gets created for `FooService` using the `proxyTargetClass()` attribute. The following enables CGLIB-style 'subclass' proxies as opposed to the default interface-based JDK proxy approach.

```
@Configuration
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class AppConfig {
    // ...
}
```

Note that `@Aspect` beans may be component-scanned like any other. Simply mark the aspect with both `@Aspect` and `@Component`:

```
package com.foo;

@Component
public class FooService { ... }

@Aspect
@Component
public class MyAspect { ... }
```

Then use the `@ComponentScan` annotation to pick both up:

```
@Configuration
@ComponentScan("com.foo")
@EnableAspectJAutoProxy
public class AppConfig {

    // no explicit @Bean definitions required
}
```

Since:

3.1

Author:

Chris Beams, Juergen Hoeller

See Also:

`Aspect`

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
boolean	**exposeProxy** Indicate that the proxy should be exposed by the AOP framework as a `ThreadLocal` for retrieval via the **`AopContext`** class.
boolean	**proxyTargetClass** Indicate whether subclass-based (CGLIB) proxies are to be created as opposed to standard Java interface-based proxies.

Element Detail

proxyTargetClass

```
public abstract boolean proxyTargetClass
```

Indicate whether subclass-based (CGLIB) proxies are to be created as opposed to standard Java interface-based proxies. The default is `false`.

Default:

`false`

exposeProxy

```
public abstract boolean exposeProxy
```

Indicate that the proxy should be exposed by the AOP framework as a `ThreadLocal` for retrieval via the `AopContext` class. Off by default, i.e. no guarantees that `AopContext` access will work.

Since:

`4.3.1`

Default:

`false`