org.springframework.context.annotation

Annotation Type Bean

```
@Target(value={METHOD,ANNOTATION_TYPE})
 @Retention(value=RUNTIME)
 @Documented
public @interface Bean
```

Indicates that a method produces a bean to be managed by the Spring container.

Overview

The names and semantics of the attributes to this annotation are intentionally similar to those of the `<bean/>` element in the Spring XML schema. For example:

```
@Bean
public MyBean myBean() {
    // instantiate and configure MyBean obj
    return obj;
}
```

Bean Names

While a `name()` attribute is available, the default strategy for determining the name of a bean is to use the name of the `@Bean` method. This is convenient and intuitive, but if explicit naming is desired, the `name` attribute (or its alias `value`) may be used. Also note that `name` accepts an array of Strings, allowing for multiple names (i.e. a primary bean name plus one or more aliases) for a single bean.

```
@Bean({"b1", "b2"}) // bean available as 'b1' and 'b2', but not 'myBean'
public MyBean myBean() {
    // instantiate and configure MyBean obj
    return obj;
}
```

Scope, DependsOn, Primary, and Lazy

Note that the `@Bean` annotation does not provide attributes for scope, depends-on, primary, or lazy. Rather, it should be used in conjunction with `@Scope`, `@DependsOn`, `@Primary`, and `@Lazy` annotations to achieve those semantics. For example:

```
    @Bean
    @Scope("prototype")
    public MyBean myBean() {
        // instantiate and configure MyBean obj
        return obj;
    }
```

@Bean Methods in @Configuration Classes

Typically, `@Bean` methods are declared within `@Configuration` classes. In this case, bean methods may reference other `@Bean` methods in the same class by calling them *directly*. This ensures that references between beans are strongly typed and navigable. Such so-called *'inter-bean references'* are guaranteed to respect scoping and AOP semantics, just like `getBean()` lookups would. These are the semantics known from the original 'Spring JavaConfig' project which require CGLIB subclassing of each such configuration class at runtime. As a consequence, `@Configuration` classes and their factory methods must not be marked as final or private in this mode. For example:

```
@Configuration
public class AppConfig {

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

    @Bean
    public FooRepository fooRepository() {
        return new JdbcFooRepository(dataSource());
    }

    // ...
}
```

@Bean Lite Mode

`@Bean` methods may also be declared within classes that are *not* annotated with `@Configuration`. For example, bean methods may be declared in a `@Component` class or even in a *plain old class*. In such cases, a `@Bean` method will get processed in a so-called *'lite'* mode.

Bean methods in *lite* mode will be treated as plain *factory methods* by the container (similar to `factory-method` declarations in XML), with scoping and lifecycle callbacks properly applied. The containing class remains unmodified in this case, and there are no unusual constraints for the containing class or the factory methods.

In contrast to the semantics for bean methods in `@Configuration` classes, *'inter-bean references'* are not supported in *lite* mode. Instead, when one `@Bean`-method invokes another `@Bean`-method in *lite* mode, the invocation is a standard Java method invocation; Spring does not intercept the invocation via a CGLIB proxy. This is analogous to inter-`@Transactional` method calls where in proxy mode, Spring does not intercept the invocation — Spring does so only in AspectJ mode.

For example:

```
@Component
public class Calculator {
    public int sum(int a, int b) {
        return a+b;
    }

    @Bean
    public MyBean myBean() {
        return new MyBean();
    }
}
```

Bootstrapping

See @Configuration Javadoc for further details including how to bootstrap the container using AnnotationConfigApplicationContext and friends.

BeanFactoryPostProcessor-returning @Bean methods

Special consideration must be taken for @Bean methods that return Spring BeanFactoryPostProcessor (BFPP) types. Because BFPP objects must be instantiated very early in the container lifecycle, they can interfere with processing of annotations such as @Autowired, @Value, and @PostConstruct within @Configuration classes. To avoid these lifecycle issues, mark BFPP-returning @Bean methods as static. For example:

```
    @Bean
    public static PropertyPlaceholderConfigurer ppc() {
        // instantiate, configure and return ppc ...
    }
```

By marking this method as static, it can be invoked without causing instantiation of its declaring @Configuration class, thus avoiding the above-mentioned lifecycle conflicts. Note however that static @Bean methods will not be enhanced for scoping and AOP semantics as mentioned above. This works out in BFPP cases, as they are not typically referenced by other @Bean methods. As a reminder, a WARN-level log message will be issued for any non-static @Bean methods having a return type assignable to BeanFactoryPostProcessor.

Since:

3.0

Author:

Rod Johnson, Costin Leau, Chris Beams, Juergen Hoeller, Sam Brannen

See Also:

Configuration, Scope, DependsOn, Lazy, Primary, Component, Autowired, Value

Optional Element Summary

Optional Elements

Modifier and Type	Optional Element and Description
`Autowire`	**`autowire`** Are dependencies to be injected via convention-based autowiring by name or type?
`String`	**`destroyMethod`** The optional name of a method to call on the bean instance upon closing the application context, for example a `close()` method on a JDBC `DataSource` implementation, or a Hibernate `SessionFactory` object.
`String`	**`initMethod`** The optional name of a method to call on the bean instance during initialization.
`String[]`	**`name`** The name of this bean, or if several names, a primary bean name plus aliases.
`String[]`	**`value`** Alias for **`name()`**.

Element Detail

value

```
@AliasFor(value="name")
public abstract String[] value
```

Alias for `name()`.

Intended to be used when no other attributes are needed, for example: `@Bean("customBeanName")`.

Since:

`4.3.3`

See Also:

`name()`

Default:

`{}`

name

```
@AliasFor(value="value")
public abstract String[] name
```

The name of this bean, or if several names, a primary bean name plus aliases.

If left unspecified, the name of the bean is the name of the annotated method. If specified, the method name is ignored.

The bean name and aliases may also be configured via the `value()` attribute if no other attributes are declared.

See Also:

`value()`

Default:

`{}`

autowire

```
public abstract Autowire autowire
```

Are dependencies to be injected via convention-based autowiring by name or type?

Note that this autowire mode is just about externally driven autowiring based on bean property setter methods by convention, analogous to XML bean definitions.

The default mode does allow for annotation-driven autowiring. "no" refers to externally driven autowiring only, not affecting any autowiring demands that the bean class itself expresses through annotations.

See Also:

`Autowire.BY_NAME`, `Autowire.BY_TYPE`

Default:

`org.springframework.beans.factory.annotation.Autowire.NO`

initMethod

```
public abstract String initMethod
```

The optional name of a method to call on the bean instance during initialization. Not commonly used, given that the method may be called programmatically directly within the body of a Bean-annotated method.

The default value is "", indicating no init method to be called.

Default:

`""`

destroyMethod

```
public abstract String destroyMethod
```

The optional name of a method to call on the bean instance upon closing the application context, for example a `close()` method on a JDBC `DataSource` implementation, or a Hibernate `SessionFactory` object. The method must have no arguments but may throw any exception.

As a convenience to the user, the container will attempt to infer a destroy method against an object returned from the `@Bean` method. For example, given an `@Bean` method returning an Apache Commons DBCP `BasicDataSource`, the container will notice the `close()` method available on that object and automatically register it as the `destroyMethod`. This 'destroy method inference' is currently limited to detecting only public, no-arg methods named 'close' or 'shutdown'. The method may be declared at any level of the inheritance hierarchy and will be detected regardless of the return type of the `@Bean` method (i.e., detection occurs reflectively against the bean instance itself at creation time).

To disable destroy method inference for a particular `@Bean`, specify an empty string as the value, e.g. `@Bean(destroyMethod="")`. Note that the `DisposableBean` and the `Closeable`/`AutoCloseable` interfaces will nevertheless get detected and the corresponding destroy/close method invoked.

Note: Only invoked on beans whose lifecycle is under the full control of the factory, which is always the case for singletons but not guaranteed for any other scope.

See Also:

`ConfigurableApplicationContext.close()`

Default:

`"(inferred)"`