

# 1 Sources

- <https://redips789.github.io/spring-certification/Spring-Certification.html>
- <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
- <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
- <https://www.baeldung.com/spring-bean-names>
- <https://www.baeldung.com/spring-core-annotations>
- <https://www.baeldung.com/spring-bean-annotations>
- <https://www.baeldung.com/spring-component-scanning>
- <https://www.baeldung.com/spring-annotations-resource-inject-autowire>
- <https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle>

TBD <https://www.baeldung.com/spring-annotations-resource-inject-autowire>

## 2 Bean Lifecycle

### 2.1 Overview

From a bird's eye, everything that happens before a bean is ready to use can be assigned to one of three phases (see fig. 1):

- Loading and maybe modifying bean definitions
- Instantiating beans
- Initializing beans

Figure 2 focuses on pre-initialization.

On the other hand, fig. 4 zooms in on post-instantiation.

See <https://www.digitalocean.com/community/tutorials/spring-bean-life-cycle> for code to display the order of invocations.

#### 2.1.1 Load bean definitions, creating an ordered graph

In this step, all the configuration files – @Configuration classes or XML files – are processed. For annotation-based configuration, all the classes annotated with @Components are scanned to load the bean definitions.

Bean definitions are passed to a BeanFactory, each under its id and type. For example, ApplicationContext is a BeanFactory.

Then, BeanFactoryPostProcessors are run.

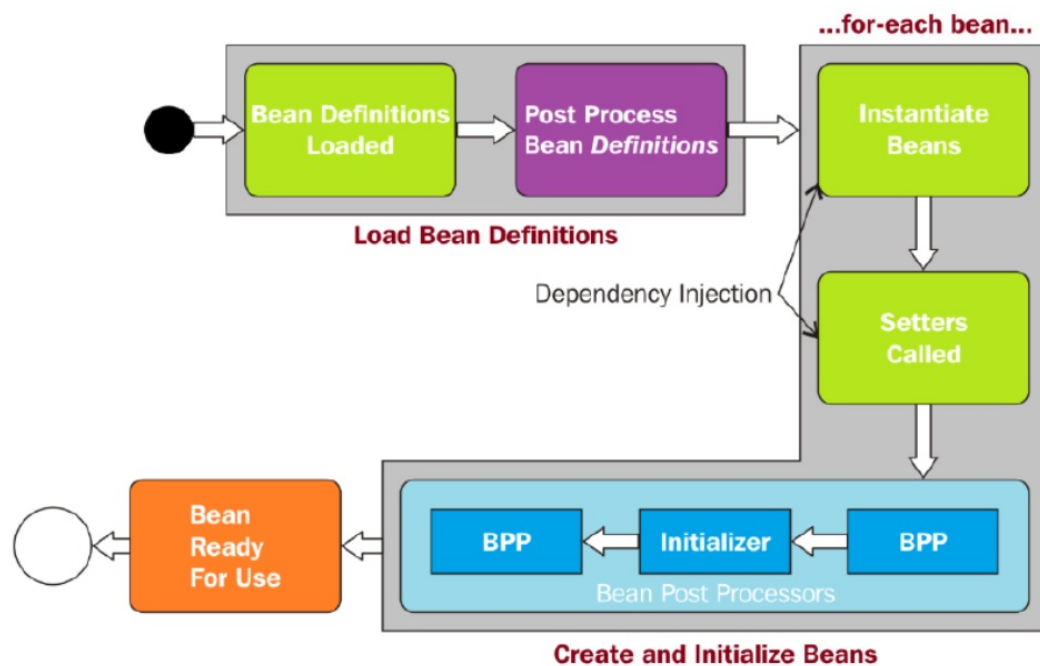


Figure 1: Lifecycle overview

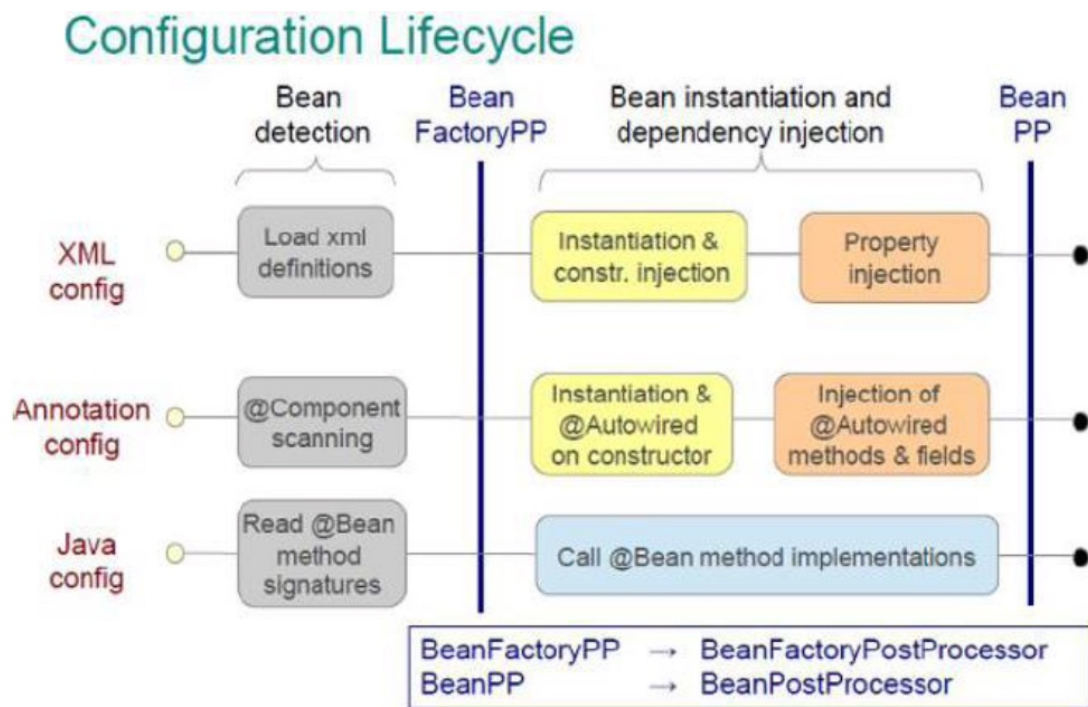


Figure 2: Zooming in on pre-instantiation

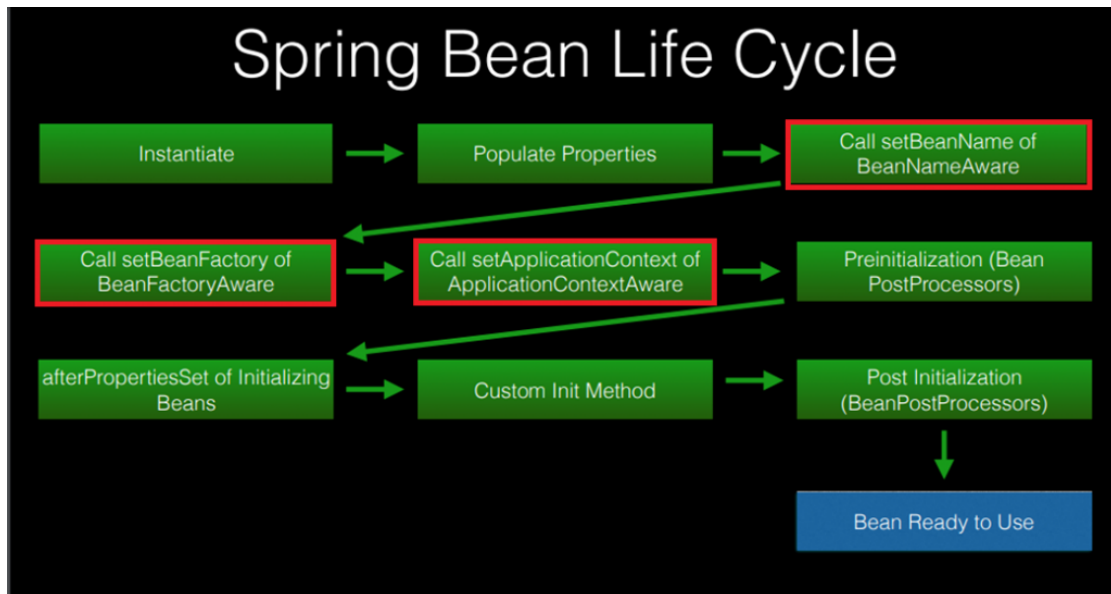


Figure 3: Zooming in on post-instantiation

### 2.1.2 Instantiate and run BeanFactoryPostProcessors

In a Spring application, a `BeanFactoryPostProcessor` can modify the definition of any bean. The `BeanFactory` object is passed as an argument to the `postProcess()` method of the `BeanFactoryPostProcessor`. `BeanFactoryPostProcessor` then works on the bean definitions or the configuration metadata of the bean before the beans are actually created. Spring provides several useful implementations of `BeanFactoryPostProcessor`, such as reading properties and registering a custom scope. We can write your own implementation of the `BeanFactoryPostProcessor` interface. To influence the order in which bean factory post processors are invoked, their bean definition methods may be annotated with the `@Order` annotation. If you are implementing your own bean factory post processor, the implementation class can also implement the `Ordered` interface.

### 2.1.3 Instantiate beans

Injects values and bean references into beans' properties.

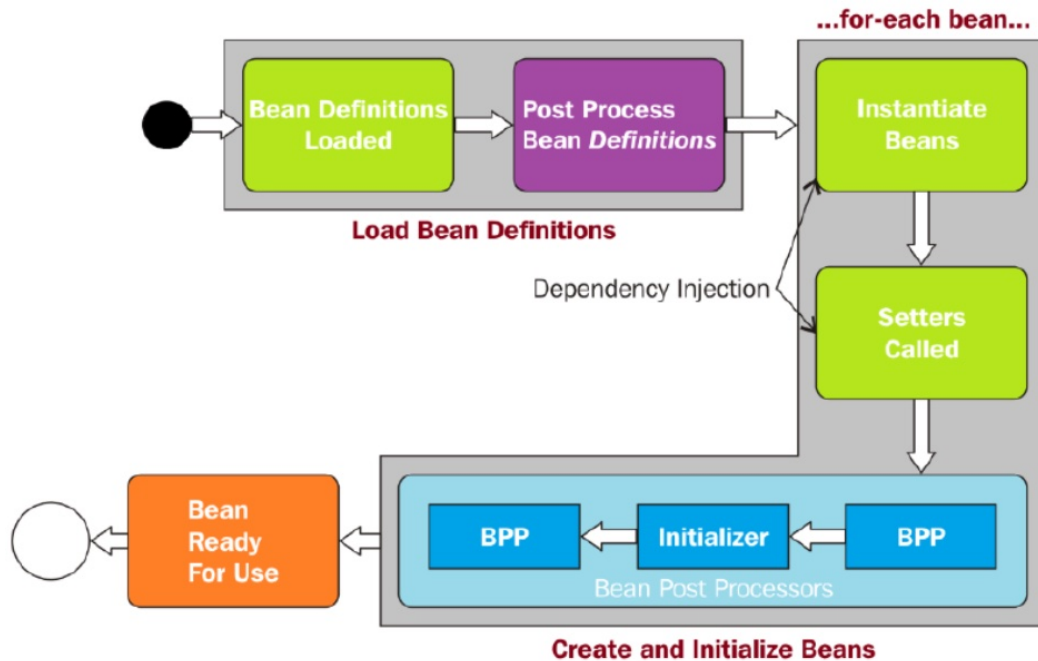


Figure 4:

- 2.1.4 Call `BeanNameAware`'s `setBeanName()` for each bean implementing it
- 2.1.5 Call `BeanFactoryAware`'s `setBeanFactory()` passing the bean factory for each bean implementing it
- 2.1.6 Call `ApplicationContextAware`'s `setApplicationContext` for each bean implementing it
- 2.1.7 Before initialization: Run pre-initialization `BeanPostProcessors`

The Application context calls `postProcessBeforeInitialization()` for each bean implementing `BeanPostProcessor`.

```

1  public interface BeanPostProcessor {
2
3      /**
4       * Apply this {@code BeanPostProcessor} to the given new
5       * bean instance before any bean's initialization
6       * callbacks (like InitializingBean's afterPropertiesSet
7       * or a custom init-method).
8       */
9      @Nullable
10     default Object postProcessBeforeInitialization(Object
11         bean, String beanName) throws BeansException {
12
13     
```

## Example: CustomBeanPostProcessor

← `@Component` Can be found by component-scanner, like any other bean

```
public class CustomBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName) {
        // Some code
        return bean; // Remember to return your bean or you'll lose it!
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) {
        // Some code
        return bean; // Remember to return your bean or you'll lose it!
    }
}
```

vmware Confidential | ©2021 VMware, Inc. 20

Figure 5: Custom bean postprocessor

```
9         return bean;
10    }
11
12    /**
13     * Apply this {@code BeanPostProcessor} to the given new
14     * bean instance after any bean initialization
15     * callbacks (like InitializingBean's afterPropertiesSet
16     * or a custom init-method).
17     */
18    @Nullable
19    default Object postProcessAfterInitialization(Object
20        bean, String beanName) throws BeansException {
21        return bean;
22    }
23 }
```

In `postProcessBeforeInitialization` and `postProcessAfterInitialization`, a bean implementing `BeanPostProcessor` can return anything it wants - even something completely different!

Figure 5 shows a no-op implementation.

### 2.1.8 Initialization: Call InitializingBean's afterPropertiesSet()

If a bean implements the InitializingBean interface, Spring calls its afterPropertiesSet() method. Used to initialize processes, load resources, etc. This approach is simple to use but it's not recommended because it will create tight coupling with the Spring framework in our bean implementations.

```
1 public interface InitializingBean {
2
3     /**
4      * Invoked by the containing BeanFactory after it has set
5      * all bean properties.
6      * This method allows the bean instance to perform
7      * validation of its overall configuration and final
8      * initialization when all bean properties have been set.
9      */
10    void afterPropertiesSet() throws Exception;
11 }
```

### 2.1.9 Initialization: Init Method, @PostConstruct

Instead of implementing InitializingBean, you can use the init-method of the bean tag, the initMethod attribute of the @Bean annotation, and JSR 250's @PostConstruct annotation. Here we use the init-method attribute:

```
1 <bean name="myEmployeeService"
2     class="com.journaldev.spring.service.MyEmployeeService"
3     init-method="init" destroy-method="destroy">
4     <property name="employee" ref="employee"></property>
5 </bean>
```

Using init-method is a solution when you don't own the class (and so, can't annotate it).

And here, the @PostConstruct annotation.

```
1 @PostConstruct
2 public void init() {
3     System.out.println("MyService init method called");
4 }
```

@PostConstruct and init-method are enabled by Spring's CommonAnnotationBeanPostProcessor. This is a BeanPostProcessor implementation that supports common Java

annotations out of the box, in particular the JSR-250 annotations in the `javax.annotation` package.

It includes support for the `javax.annotation.PostConstruct` and `javax.annotation.PreDestroy` annotations - as init annotation and destroy annotation, respectively - through inheriting from `InitDestroyAnnotationBeanPostProcessor` with pre-configured annotation types.

```
1  public class CommonAnnotationBeanPostProcessor extends
    InitDestroyAnnotationBeanPostProcessor
2  implements InstantiationAwareBeanPostProcessor ,
    BeanFactoryAware , Serializable { ... }
```

#### 2.1.10 After initialization: Run post-initialization BeanPostProcessors

The application context calls `postProcessAfterInitialization()` for each bean implementing `BeanPostProcessor`.

#### 2.1.11 Bean ready to use

Your beans remain live in the application context until it is closed by calling the `close()` method of the application context.

#### 2.1.12 Custom destruction

If a bean implements the `DisposableBean` interface, Spring calls its `destroy()` method to destroy any process or clean up the resources of your application. There are other methods to achieve this step-for example, you can use the `destroy`-method of the tag, the `destroyMethod` attribute of the `'@Bean'` annotation, and JSR 250's `'@PreDestroy'` annotation.

## 3 Dependency injection

### 3.1 Constructor-based

In the case of constructor-based dependency injection, the container will invoke a constructor with arguments each representing a dependency we want to set. This is the recommended way.

```
1  @Configuration
2  public class AppConfig {
3      @Bean
4      public Item item1() {
5          return new ItemImpl1();
6      }
7      @Bean
```

```

8         public Store store() {
9             return new Store(item1());
10        }
11    }

```

Resp.

```

1    <bean id="item1" class="org.baeldung.store.ItemImpl1"
    />
2    <bean id="store" class="org.baeldung.store.Store">
3    <constructor-arg type="ItemImpl1" index="0"
    name="item" ref="item1" />
4    </bean>

```

### 3.2 Method-based

For setter-based DI, the container will call setter methods of our class after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

```

1    @Bean
2    public Store store() {
3        Store store = new Store();
4        store.setItem(item1());
5        return store;
6    }

```

Resp.

```

1    <bean id="store" class="org.baeldung.store.Store">
2    <property name="item" ref="item1" />
3    </bean>

```

### 3.3 Field-based

In field-based DI, we can inject the dependencies by marking them with an `@Autowired` annotation. (This even works for private fields.) Field-based injection is not recommended - e.g., it makes testing harder.

```

1    public class Store {
2        @Autowired // deprecated
3        private Item item;

```



```
4     }
```

## 4 Configuration: Implicit vs. Explicit

Also referred to as Java-based (decoupled) and annotation-based.

### 4.1 Java-based

Takes place completely in @Configuration classes. E.g.,

```
1     @Configuration
2     public class MyConfig {
3         @Bean
4         public AccountRepo AccountRepo() {}
5     }
```

### 4.2 Java-based

Bean definition and wiring take place completely in POJOs. For this to work, we need to enable component scanning.

```
1     @Configuration
2     @ComponentScan
3     public class MyConfig {}
4
5     @Component
6     public class AccountRepo {}
```

## 5 Annotations

### 5.1 Annotations for dependency injection

#### 5.1.1 @Autowired

@Autowired marks a dependency which Spring is going to resolve and inject. We can use this annotation with constructor, setter, or field injection. E.g.,

```
1     class Car {
2         @Autowired
3         Engine engine;
```

```
4      }
```

Starting with version 4.3, we don't need to annotate constructors with `@Autowired` explicitly unless we declare at least two constructors.

`@Autowired` matches by type. If there are several classes matching the required type (e.g., implementing the same interface), `@Autowired` needs to be supplemented by `@Qualifier`:

```
1      @Component("Repo1")
2      class Repo1 implements Repo {}
3
4      @Component("Repo2")
5      class Repo2 implements Repo {}
6
7      @Component
8      public class Service1 implements ServiceX {
9          public Service1(@Qualifier("Repo2") Repo) {}
10
11     }
```

If there is no `@Qualifier` given, `@Autowired` looks for a matching bean name (= bean id). Here, Spring will look for a bean named `x`:

```
1      // constructor injection
2      @Autowired
3      public MyBean(X x) {}
4
5      // method injection
6      @Autowired
7      public setX(X x) {}
8
9      // field injection
10     @Autowired
11     private X x;
```

### 5.1.2 @Bean

`@Bean` marks a factory method which instantiates a Spring bean.

```
1      @Bean
2      Engine engine() {
3          return new Engine();
4      }
```

```
4     }
```

Spring calls these methods when a new instance of the return type is required. All methods annotated with `@Bean` must be in `@Configuration` classes.

### 5.1.3 @Resource

The `@Resource` annotation matches by name, type, or qualifier (in this order). It is applicable to setter and field injection. Here's an example injecting a field. Note that the bean id and the corresponding reference attribute value must match:

```
1     @Configuration
2     public class MyAppContext {
3         @Bean(name="namedFile")
4         public File namedFile() {
5             File namedFile = new File("namedFile.txt");
6             return namedFile;
7         }
8     }
9
10    @ContextConfiguration(
11        loader=AnnotationConfigContextLoader.class,
12        classes= MyAppContext.class)
13    public class Xxx {
14        @Resource(name="namedFile")
15        private File defaultFile;
16    }
```

### 5.1.4 @Inject

The `@Inject` annotation matches by type, qualifier, or name (in this order). It is applicable to setter and field injection. With `@Inject`, the class reference variable's name and the bean name don't have to match.

To use the `@Inject` annotation, declare the `javax.inject` library as a Gradle or Maven dependency.

```
1     public class MyAppContext {
2         @Bean
3         // no bean name specified - method name is used
4         public File getSomeFile() {
5             File namedFile = new File("namedFile.txt");
6             return namedFile;
7         }
8     }
```

```

8     }
9
10    @ContextConfiguration(
11        loader=AnnotationConfigContextLoader.class,
12        classes= MyAppContext.class)
13    public class Xxx {
14        @Inject
15        private File defaultFile;
16    }

```

### 5.1.5 @Value

We can use @Value for injecting property values into beans. It's compatible with constructor, setter, and field injection. E.g.,

```

1    Engine(@Value("8") int cylinderCount) {
2        this.cylinderCount = cylinderCount;
3    }

```

This is an alternative to making explicit use of Spring's Environment bean. E.g.

```

1    public DataSource dataSource(
2        @Value("${db.driver}") String driver,
3        ...
4    )
5    }

```

### 5.1.6 @DependsOn

We can use this annotation to make Spring initialize other beans before the annotated one. Usually, this behavior is automatic, based on the explicit dependencies between beans. We only need this annotation when the dependencies are implicit, for example, JDBC driver loading or static variable initialization. E.g.,

```

1    @Bean
2    @DependsOn("fuel")
3    Engine engine() {
4        return new Engine();
5    }

```

### 5.1.7 @Lazy

This annotation behaves differently depending on where exactly we place it.

- In an @Bean-annotated bean factory method, it is used to delay the method call (hence the bean creation)
- With an @Configuration class, all contained @Bean methods will be affected
- For all other @Component classes, they will be initialized lazily when so annotated.
- @Autowired constructors, setters, and fields will be loaded lazily (via proxy).

```
1      @Configuration
2      @Lazy
3      class VehicleFactoryConfig {
4
5          @Bean
6          @Lazy( false )
7          Engine engine() {
8              return new Engine();
9          }
10     }
```

### 5.1.8 @Scope

@Scope is used to define the scope of a @Component class or a @Bean definition. It can be either singleton, prototype, request, session, globalSession or some cust@Component.

## 5.2 Context Configuration Annotations

### 5.2.1 @Import

With @import, we can use specific @Configuration classes without component scanning.

```
1      @Import( VehiclePartSupplier.class )
2      class VehicleFactoryConfig {}
```

### 5.2.2 @ImportResource

We can import XML configurations with @ImportResource. We can specify the XML file locations with the locations argument, or with its alias, the value argument:

```

1      @Configuration
2      @ImportResource("classpath:/annotations.xml")
3      class VehicleFactoryConfig {}

```

### 5.2.3 @PropertySource

With this annotation, we define property files for application settings.

```

1      @Configuration
2      @PropertySource("classpath:/annotations.properties")
3      @PropertySource("classpath:/vehicle-factory.properties")
4      class VehicleFactoryConfig {}

```

These properties can be used by Spring's Environment bean, in addition to environment variables and Java system properties.

Allowed prefixes are classpath:, file:, and http:.

## 5.3 Bean annotations

### 5.3.1 @Profile

Profiles are a way to group bean definitions, for example:

- dev, test, prod environment
- jdbc, jpa [implementations]

The @Profile annotation may be used in any of the following ways:

- At class level in @Configuration classes.
- At class level in classes annotated with @Component or annotated with any other annotation that in turn is annotated with @Component.
- On methods annotated with the @Bean annotation.

To define alternative beans with different profile conditions, use distinct Java method names pointing to the same bean name via the @Bean name attribute:

```

1      @Bean("dataSource")
2      @Profile("development")
3      public DataSource standaloneDataSource() {
4
5      @Bean("dataSource")
6      @Profile("production")

```

```
7      public DataSource jndiDataSource() throws Exception {}
```

Spring uses two separate properties when determining which profiles are active, `spring.profiles.active` and `spring.profiles.default`:

- If `spring.profiles.active` is set, then its value determines which profiles are active.
- If `spring.profiles.active` isn't set, then Spring looks to `spring.profiles.default`.
- If neither `spring.profiles.active` nor `spring.profiles.default` is set, only those beans that aren't defined as being in a profile are created.

These properties can be set on the command line:

```
1      -Dspring.profiles.active=embedded.jpa
```

, programmatically:

```
1      System.setProperty("spring.profiles.active",  
                          "embedded.jpa");
```

, or via an annotation (`@ActiveProfiles`; integration tests only).

### 5.3.2 @ComponentScan

The `@ComponentScan` annotation is used together with `@Configuration`.

`@ComponentScan` can be used with and without arguments.

Without arguments, `@ComponentScan` tells Spring to scan the current package and all of its sub-packages.

With arguments, `@ComponentScan` tells which packages or classes to scan. E.g., specifying packages:

```
1      @Configuration  
2      @ComponentScan(basePackages =  
                     "com.baeldung.annotations")  
3      class VehicleFactoryConfig {}
```

Or else, specifying classes:

```
1      @Configuration  
2      @ComponentScan(basePackageClasses =  
                     VehicleFactoryConfig.class)  
3      class VehicleFactoryConfig {}
```

We can specify multiple package names, using spaces, commas, or semicolons as a separator.

```
1 @ComponentScan( basePackages =
    "com.baeldung.componentscan.springapp.animals;com.baeldung.compone
2 @ComponentScan( basePackages =
    "com.baeldung.componentscan.springapp.animals ,com.baeldung.compone
3 @ComponentScan( basePackages =
    "com.baeldung.componentscan.springapp.animals
    com.baeldung.componentscan.springapp.flowers") )
```

We could also apply a filter, choosing from a range of filter types. For example:

```
1 @ComponentScan(excludeFilters =
2 @ComponentScan.Filter(type=FilterType.REGEX,
3 pattern="com\\.baeldung\\.componentscan\\.springapp\\.flowers\\.*"))
```

Or:

```
1 @ComponentScan(excludeFilters =
2 @ComponentScan.Filter(type =
    FilterType.ASSIGNABLE_TYPE, value = Rose.class))
```

### 5.3.3 @Component

@Component is a class-level annotation. During component scan, Spring automatically detects classes annotated with @Component.

```
1      @Component
2      class CarUtility {
3          // ...
4      }
```

@Repository, @Service, @Configuration, and @Controller are all meta-annotations of (i.e., themselves annotated with) @Component. E.g.,

```
1 @Component
2 public @interface Service {}
```

Spring also automatically picks them up during the component scanning process.



#### 5.3.4 @Repository

```
1      @Repository
2      class VehicleRepository {
3          // ...
4      }
```

#### 5.3.5 @Service

```
1      @Service
2      public class VehicleService {
3          // ...
4      }
```

#### 5.3.6 @Controller

```
1      @Controller
2      public class VehicleController {
3          // ...
4      }
```

#### 5.3.7 @Configuration

Configuration classes can contain bean definition methods annotated with @Bean.

```
1      @Configuration
2      class VehicleFactoryConfig {
3
4          @Bean
5          Engine engine() {
6              return new Engine();
7          }
8
9      }
```

## 5.4 Spring Boot Annotations

### 5.4.1 @SpringBootApplication

This is a combination of three annotations:

```
1      @Configuration
2      @EnableAutoConfiguration
3      @ComponentScan
```

## 6 Aware Interfaces

Indicates that the bean is eligible to be notified by the Spring container through the callback methods. A typical use case for `BeanNameAware` could be acquiring the bean name for logging or wiring purposes. For the `BeanFactoryAware` it could be the ability to use a spring bean from legacy code. In most cases, we should avoid using any of the Aware interfaces, unless we need them. Implementing these interfaces will couple the code to the Spring framework.

### 6.1 BeanNameAware

Makes the object aware of the bean name defined in the container.

```
1      public class MyBeanName implements BeanNameAware {
2          @Override
3          public void setBeanName(String beanName) {
4              System.out.println(beanName);
5          }
6      }
7      @Configuration
8      public class Config {
9          @Bean(name = "myCustomBeanName")
10         public MyBeanName getMyBeanName() {
11             return new MyBeanName();
12         }
13     }
14     AnnotationConfigApplicationContext context
15     = new AnnotationConfigApplicationContext(Config.class);
16     MyBeanName myBeanName = context.getBean(MyBeanName.class);
```

## 6.2 BeanFactoryAware

Provides access to the BeanFactory which created the object.

```
1      public class MyBeanFactory implements BeanFactoryAware
2      {
3          private BeanFactory beanFactory;
4          @Override
5          public void setBeanFactory(BeanFactory
6              beanFactory) throws BeansException {
7              this.beanFactory = beanFactory;
8          }
9          public void getMyBeanName() {
10             MyBeanName myBeanName =
11                 beanFactory.getBean(MyBeanName.class);
12             System.out.println(beanFactory.isSingleton("myCustomBeanName"));
13         }
14     }
15     MyBeanFactory myBeanFactory =
16         context.getBean(MyBeanFactory.class);
17     myBeanFactory.getMyBeanName();}
```

## 6.3 ApplicationContextAware

```
1      public class ApplicationContextAwareImpl implements
2      ApplicationContextAware {
3          @Override
4          public void
5              setApplicationContext(ApplicationContext
6                  applicationContext) throws BeansException {
7              User user = (User)
8                  applicationContext.getBean("user");
9              System.out.println("User Id: " +
10                  user.getUserId() + " User Name : " +
11                  user.getName());}}
```

## 7 Bean Naming

### 7.1 Default Bean Naming

#### 7.1.1 Class-level

For an annotation used at the class level, Spring uses the class name and converts the first letter to lowercase. The same default naming strategy is applicable for all class-level annotations that are used to create a Spring bean, such as `@Component`, `@Service`, and `@Controller`.

```
1      @Service
2      public class LoggingService { // bean name =
3                                  loggingService
4      }
```

#### 7.1.2 Method-level

When we use the `@Bean` annotation on a method, Spring uses the method name as a bean name.

```
1      @Configuration
2      public class AuditConfiguration {
3          @Bean
4          public AuditService audit() {
5              return new AuditService();
6          }
7      }
```

### 7.2 Custom naming

```
1      @Component("myBean")
2      public class MyCustomComponent {
3      }
```

Similar to `@Component("myBean")`, we can specify the name using other annotations such as `@Service("myService")`, `@Controller("myController")`, and `@Bean("myCustomBean")`.

## 7.3 Naming Beans With @Bean and @Qualifier

### 7.3.1 @Bean With Value

The @Bean annotation is applied at the method level, and by default, Spring uses the method name as a bean name. We can override this using the @Bean annotation.

```
1      @Configuration
2      public class MyConfiguration {
3          @Bean("beanComponent")
4          public MyCustomComponent myComponent() {
5              return new MyCustomComponent();
6          }
7      }
```

### 7.3.2 @Qualifier With Value

We can also use the @Qualifier annotation to name the bean.

```
1      @Component
2      @Qualifier("cat")
3      public class Cat implements Animal {
4          @Override
5          public String name() {
6              return "Cat";
7          }
8      }
9      @Component
10     @Qualifier("dog")
11     public class Dog implements Animal {
12         @Override
13         public String name() {
14             return "Dog";
15         }
16     }
17     @Service
18     public class PetShow {
19         private final Animal dog;
20         private final Animal cat;
21
22         public PetShow (@Qualifier("dog") Animal dog,
23                         @Qualifier("cat") Animal cat) {
24             this.dog = dog;
25             this.cat = cat;
26         }
27     }
```

```

25         }
26         public Animal getDog() {
27             return dog;
28         }
29         public Animal getCat() {
30             return cat;
31         }
32     }

```

## 8 Spring Expression Language vs. Property Evaluation

Expressions in `@Value` annotations are of two types:

- Expressions starting with `$`. Such expressions reference a property name in the application's environment. These expressions are evaluated by the `PropertySourcePlaceholderConfigurer` `BeanFactoryPostProcessor` prior to bean creation and can only be used in `@Value` annotations.
- Expressions starting with `#`. These expressions are parsed by a SpEL expression parser, and are evaluated by a SpEL expression instance.

In some cases, both can be used. For example, property values by default are Strings, but may be converted to primitives implicitly. So, both of these work:

```

1     @Value("${daily.limit}")
2     int limit;
3
4     @Value("#{environment['daily.limit']}")
5     int limit;

```

But if computations are to be performed, or object types are required, SpEL has to be used:

```

1     // NO
2     @Value("${daily.limit} * 2")
3
4     // instead, do
5     @Value("#{new Integer(environment['daily.limit']) * 2}")

```

To provide defaults, use a colon with property evaluation, and `?:` in SpEL.

```
1      @Value("${daily.limit}: 1000")
2      int limit;
3
4      @Value("#{environment['daily.limit']} ?: 1000")
5      int limit;
```

In addition to application-defined beans, SpEL can make use of beans implicitly provided by Spring, namely `environment`, `systemProperties`, and `systemEnvironment`.