https://springframework.guru/spring-framework-annotations/

Annotations in java

The @SpringBootApplication Annotation

Spring needs to know which packages to scan for annotated components in order to add them to the IoC container. In a Spring Boot project, we typically set the main application class with the @SpringBootApplicationannotation. Under the hood, @SpringBootApplication is a composition of the @Configuration, @ComponentScan, and @EnableAutoConfiguration annotations. With this default setting, Spring Boot will auto scan for components in the current package (containing the @SpringBoot main class) and its sub packages.

To know more about these annotations go through my [Spring Framework Annotations](https://springframework.guru/spring-framework-annotations/) post.

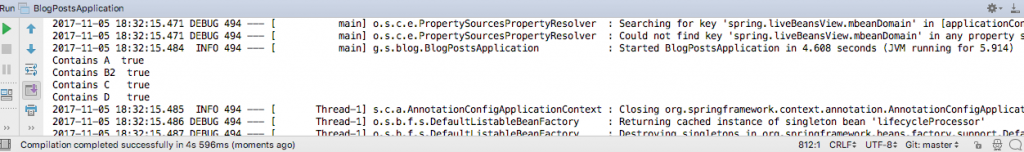
Note: It is recommended that you locate your main application class in a root package above the component classes of the application.

The code to create the main class and access components is this.

BlogPostsApplication.java



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22** | **package guru.springframework.blog;**    **import org.springframework.boot.SpringApplication;**  **import org.springframework.boot.autoconfigure.SpringBootApplication;**  **import org.springframework.context.ApplicationContext;**      **@SpringBootApplication**  **public class BlogPostsApplication {**    **public static void main(String[] args) {**  **ApplicationContext context = SpringApplication.run(BlogPostsApplication.class,args);**  **System.out.println("Contains A  "+context.**  **containsBeanDefinition("demoBeanA"));**  **System.out.println("Contains B2  " + context.**  **containsBeanDefinition("demoBeanB2"));**  **System.out.println("Contains C   " + context.**  **containsBeanDefinition("demoBeanC"));**      **}**  **}** |

The output of running the main class is this.  
[](https://springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplication.java_.png)

As you can notice, all the classes in the sub packages of the main class BlogPostsApplication are auto scanned by Spring.

@ComponentScan – Identifying Base Packages

The @ComponentScan annotation is used with the @Configuration annotation to tell Spring the packages to scan for annotated components. @ComponentScan also used to specify base packages and base package classes using thebasePackageClasses or basePackages attributes of @ComponentScan.

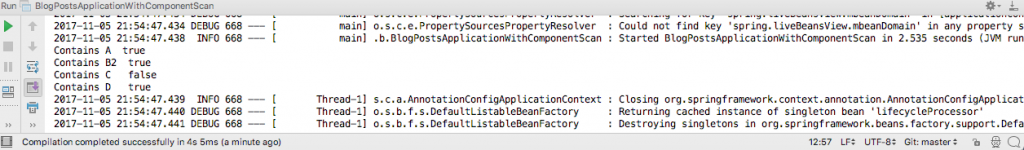
The basePackageClasses attribute is a type-safe alternative to basePackages. When you specify basePackageClasses, Spring will scan the package (and subpackages) of the classes you specify.

A Java class annotated with @ComponentScan with the basePackageClassesattribute is this.

BlogPostsApplicationWithComponentScan.java



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27** | **package guru.springframework.blog;**  **import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;**  **import org.springframework.boot.SpringApplication;**  **import org.springframework.context.ApplicationContext;**  **import org.springframework.context.annotation.ComponentScan;**  **import org.springframework.context.annotation.Configuration;**    **@Configuration**  **@ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",**  **"guru.springframework.blog.componentscan.example.demopackageD",**  **"guru.springframework.blog.componentscan.example.demopackageE"},**  **basePackageClasses = DemoBeanB1.class)**  **public class BlogPostsApplicationWithComponentScan {**  **public static void main(String[] args) {**  **ApplicationContext context = SpringApplication.**  **run(BlogPostsApplicationWithComponentScan.class,args);**  **System.out.println("Contains A  "+context.**  **containsBeanDefinition("demoBeanA"));**  **System.out.println("Contains B2  " + context.**  **containsBeanDefinition("demoBeanB2"));**  **System.out.println("Contains C   " + context.**  **containsBeanDefinition("demoBeanC"));**  **System.out.println("Contains D   " + context.**  **containsBeanDefinition("demoBeanD"));**    **}**  **}** |

The output on running the main class is this.  
[](https://springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationWithComponentScan.java_.png)

The @ComponentScan annotation uses the basePackages attribute to specify three packages (and subpackages) that will be scanned by Spring. The annotation also uses the basePackageClasses attribute to declare theDemoBeanB1 class whose package Spring Boot should scan.

As demoBeanC is in a different package, Spring did not find it during component scanning.

Component Scanning Filters

You can configure component scanning by using different type filters that Spring provides.

By using filters, you can further narrow the set of candidate components from everything in basePackages to everything in the base packages that matches the given filter or filters.

Filters can be of two types: include and exclude filters. As their names suggest, include filters specify which types are eligible for component scanning, while exclude filters specify which types are not.

You can use the include and/or exclude filters with or without the default filter. To disable the default filter, set the useDefaultFilters element of the @ComponentScan annotation to false.

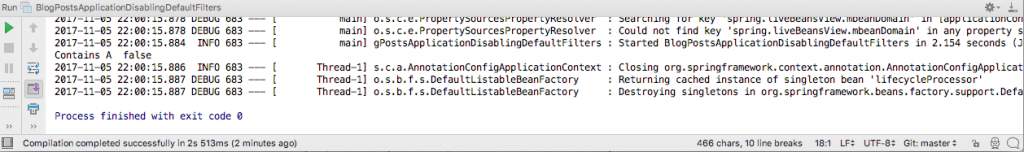
The code to disable the default filter is this.

BlogPostsApplicationDisablingDefaultFilters.java



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17** | **package guru.springframework.blog;**    **import org.springframework.boot.SpringApplication;**  **import org.springframework.context.ApplicationContext;**  **import org.springframework.context.annotation.ComponentScan;**  **import org.springframework.context.annotation.Configuration;**    **@Configuration**  **@ComponentScan(value = "guru.springframework.blog.componentscan.example.demopackageA",**  **useDefaultFilters = false)**  **public class BlogPostsApplicationDisablingDefaultFilters {**  **public static void main(String[] args) {**  **ApplicationContext context = SpringApplication.**  **run(BlogPostsApplicationDisablingDefaultFilters.class,args);**  **System.out.println("Contains A  " + context.containsBean("demoBeanA"));**  **}**  **}** |

In the preceding code, the value member defines the specificguru.springframework.blog.componentscan.example.demopackageA package to scan, while the useDefaultFiltersmember disables the default filter.

The output on running the main class is this.  
[](https://springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationDisablingDefaultFilters.java_.png)

As you can notice, the class DemoBeanA in the package demopackageA is unavailable when the useDefaultFilterselement of the @ComponentScan annotation is set to false.

Component Scanning Filter Types

Spring provides the FilterType enumeration for the type filters that may be used in conjunction with@ComponentScan.

The available FilterType values are:

* FilterType.ANNOTATION: Include or exclude those classes with a stereotype annotation
* FilterType.ASPECTJ: Include or exclude classes using an AspectJ type pattern expression
* FilterType.ASSIGNABLE\_TYPE: Include or exclude classes that extend or implement this class or interface
* FilterType.REGEX: Include or exclude classes using a regular expression
* FilterType.CUSTOM: Include or exclude classes using a custom implementation of theorg.springframework.core.type.TypeFilter interface

Include Filters

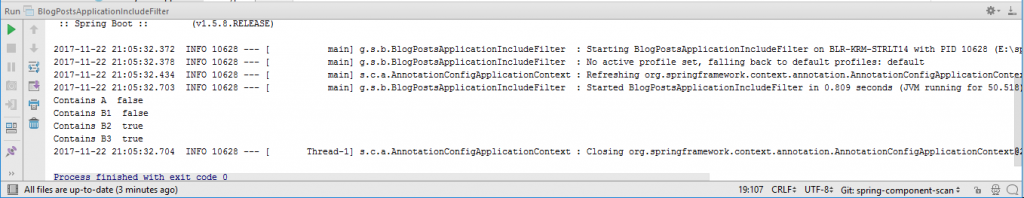
With include filters, you can include certain classes to be scanned by Spring. To include assignable type, use the includeFilters element of the @ComponentScan annotation with FilterType. ASSIGNABLE\_TYPE. Using this filter, you can instruct Spring to scan for classes that extends or implements the class or interface you specify.

The code to use the includeFilters element of @ComponentScan is this.

BlogPostsApplicationIncludeFilter.java



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24** | **package guru.springframework.blog;**    **import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;**  **import org.springframework.boot.SpringApplication;**  **import org.springframework.context.ApplicationContext;**  **import org.springframework.context.annotation.ComponentScan;**  **import org.springframework.context.annotation.Configuration;**  **import org.springframework.context.annotation.FilterType;**    **@Configuration**  **@ComponentScan(basePackages = {"guru.springframework.blog.componentscan.example.demopackageA",**  **"guru.springframework.blog.componentscan.example.demopackageB"},**  **includeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE, value = DemoBeanB2.class),**  **useDefaultFilters = false)**  **public class BlogPostsApplicationIncludeFilter {**  **public static void main(String[] args) {**  **ApplicationContext context = SpringApplication.**  **run(BlogPostsApplicationIncludeFilter.class,args);**  **System.out.println("Contains A  " + context.containsBean("demoBeanA"));**  **System.out.println("Contains B1  " + context.containsBean("demoBeanB1"));**  **System.out.println("Contains B2  " + context.containsBean("demoBeanB2"));**  **System.out.println("Contains B3  " + context.containsBean("demoBeanB3"));**  **}**  **}** |

The output on running the main class is this.  
[](https://springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationIncludeFilter.java_.png)

As shown in the preceding figure, Spring detected and used the demoBean3 component that extendsdemoBean2.

[](https://bit.ly/2yhpu6x)Learn Spring Framework 5 with my Spring Framework 5: Beginner to Guru course!

Include Filters using Regex

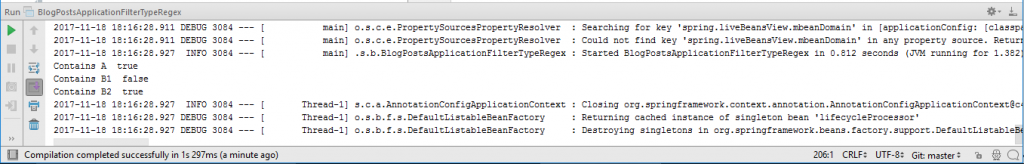
You can use regular expressions to filter out components to be scanned by Spring. Use theincludeFiltersnested annotation @ComponentScan.Filter type FilterType.REGEXto set a pattern.

The code to use an exclude filter based on regular expression is this.

BlogPostsApplicationFilterTypeRegex.java



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20** | **package guru.springframework.blog;**    **import org.springframework.boot.SpringApplication;**  **import org.springframework.context.ApplicationContext;**  **import org.springframework.context.annotation.ComponentScan;**  **import org.springframework.context.annotation.Configuration;**  **import org.springframework.context.annotation.FilterType;**    **@Configuration**  **@ComponentScan(useDefaultFilters = false, includeFilters = @ComponentScan.Filter**  **(type = FilterType.REGEX, pattern = ".\*[A2]"))**  **public class BlogPostsApplicationFilterTypeRegex {**  **public static void main(String[] args) {**  **ApplicationContext context = SpringApplication.**  **run(BlogPostsApplicationFilterTypeRegex.class,args);**  **System.out.println("Contains A  " + context.containsBean("demoBeanA"));**  **System.out.println("Contains B1  " + context.containsBean("demoBeanB1"));**  **System.out.println("Contains B2  " + context.containsBean("demoBeanB2"));**  **}**  **}** |

The output of the following code snippet is this.  
[](https://springframework.guru/wp-content/uploads/2017/11/Output-of-BlogPostsApplicationFilterTypeRegex.java_.png)  
As shown in the preceding figure, the classes whose names end with A, or 2 are detected by Spring.

Exclude Filters

The @ComponentScan annotation enables you to exclude those classes that you do not want to scan.

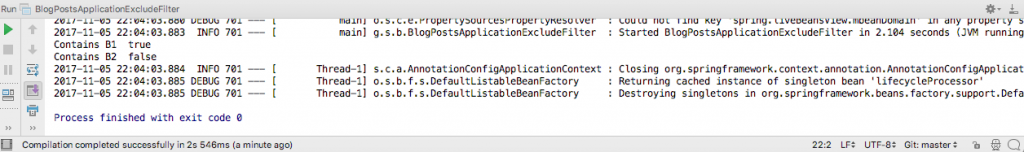
The code to use an exclude filter is this.

BlogPostsApplicationExcludeFilter.java



|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22** | **package guru.springframework.blog;**    **import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB1;**  **import guru.springframework.blog.componentscan.example.demopackageB.DemoBeanB2;**  **import org.springframework.boot.SpringApplication;**  **import org.springframework.context.ApplicationContext;**  **import org.springframework.context.annotation.ComponentScan;**  **import org.springframework.context.annotation.Configuration;**  **import org.springframework.context.annotation.FilterType;**    **@Configuration**  **@ComponentScan(basePackageClasses = {DemoBeanB1.class},**  **excludeFilters = @ComponentScan.Filter(type = FilterType.ASSIGNABLE\_TYPE,**  **value = DemoBeanB2.class))**  **public class BlogPostsApplicationExcludeFilter {**  **public static void main(String[] args) {**  **ApplicationContext context = SpringApplication.**  **run(BlogPostsApplicationExcludeFilter.class,args);**  **System.out.println("Contains B1  " + context.containsBean("demoBeanB1"));**  **System.out.println("Contains B2  " + context.containsBean("demoBeanB2"));**  **}**  **}** |

In this code, the nested annotation @ComponentScan.Filter is used to specify the filter type asFilterType.ASSIGNABLE\_TYPE and the base class that should be excluded from scanning.

The output is this.  
[](https://springframework.guru/wp-content/uploads/2017/11/Output-of-using-the-FilterType.ASSIGNABLE_TYPE-type.png)

As you can see, the class DemoBeanB2 has been excluded from being scanned.

Summary

When using Spring Boot, most of the time the default auto scanning will work for your project. You only need to ensure that your @SpringBoot main class is at the base package of your package hierarchy. Spring Boot will automatically perform a component scan in the package of the Spring Boot main class and below.

One related annotation that I didn’t mention in this post is @EntityScan is more about JPA entity scanning rather than component scanning. The @EntityScan annotation, unlike @ComponentScan does not create beans. It only identifies which classes should be used by a specific persistence context.

# Spring Boot @Qualifier annotation

In this tutorial, we present the Spring Boot @Qualifier annotation. The @Qualifier annotation is used to differentiate beans of the same type. It can also be used to annotate other custom annotations that can then be used as qualifiers.

[Tweet](https://twitter.com/share)

Spring is a popular Java application framework and Spring Boot is an evolution of Spring which helps create stand-alone, production-grade Spring based applications with minimal effort.

The following three applications are command line Spring Boot applications.

## Differentiating Person beans

In our application, we have two beans of Person type: Student and Manager. We use the @Qualifier annotation to distinguish between them.

$ tree

.

├── pom.xml

└── src

├── main

│ └── java

│ └── com

│ └── zetcode

│ ├── Application.java

│ ├── bean

│ │ ├── Manager.java

│ │ ├── Person.java

│ │ └── Student.java

│ └── MyRunner.java

└── test

└── java

This is the project structure of the Spring Boot application.

pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.zetcode</groupId>

<artifactId>SpringBootQualifier</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

</properties>

<parent>

<groupId>org.sringframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

This is the Maven build file. The spring-boot-starter is the core starter that includes auto-configuration support, logging, and YAML. The application is packaged into a JAR file.

Person.java

package com.zetcode.bean;

public interface Person {

public String info();

}

We have an interface that defines the Person type.

Student.java

package com.zetcode.bean;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

@Qualifier("student")

public class Student implements Person {

@Override

public String info() {

return "Student";

}

}

Student inherits from Person. @Component is a basic Spring annotation that allows Student to be detected by Spring containter. The @Qualifier("student") uniquely identifies this bean with the "student" string.

Manager.java

package com.zetcode.bean;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.stereotype.Component;

@Component

@Qualifier("manager")

public class Manager implements Person {

@Override

public String info() {

return "Manager";

}

}

We have another bean called Manager. This bean is also identified with the @Qualifier("manager") annotation.

MyRunner.java

package com.zetcode;

import com.zetcode.bean.Person;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

@Component

public class MyRunner implements CommandLineRunner {

@Autowired

@Qualifier("student")

Person p1;

@Autowired

@Qualifier("manager")

Person p2;

@Override

public void run(String... args) throws Exception {

System.out.println(p1.info());

System.out.println(p2.info());

}

}

The CommandLineRunner interface indicates that a bean should run when it is contained within a SpringApplication. It can be used to create command line applications in Spring Boot.

@Component

public class MyRunner implements CommandLineRunner {

The CommandLineRunner is also a Spring bean and is decorated with the @Component annotation; it is auto-detected by Spring.

@Autowired

@Qualifier("student")

Person p1;

We inject a Person bean into the p1 field. The @Qualifier("student") specifies that it is a Student bean.

@Autowired

@Qualifier("manager")

Person p2;

Likewise, we inject the Manager bean into the p2 field.

Application.java

package com.zetcode;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

The Application sets up the Spring Boot application. The @SpringBootApplication annotation enables auto-configuration and component scanning.

## Using factory to create beans

In the second application, we use a factory class to generate beans. The pom.xml, Person.java, Application.java, MyRunner.java remain unchanged.

$ tree

.

├── pom.xml

└── src

├── main

│ ├── java

│ │ └── com

│ │ └── zetcode

│ │ ├── Application.java

│ │ ├── bean

│ │ │ ├── Manager.java

│ │ │ ├── Person.java

│ │ │ └── Student.java

│ │ ├── conf

│ │ │ └── PersonFactory.java

│ │ └── MyRunner.java

│ └── resources

└── test

└── java

This is the project structure.

Manager.java

package com.zetcode.bean;

public class Manager implements Person {

@Override

public String info() {

return "Manager";

}

}

The annotations are removed from the Manager class.

Student.java

package com.zetcode.bean;

public class Student implements Person {

@Override

public String info() {

return "Student";

}

}

Likewise, there are no annotations for the Student class.

PersonFactory.java

package com.zetcode.conf;

import com.zetcode.bean.Manager;

import com.zetcode.bean.Person;

import com.zetcode.bean.Student;

import org.springframework.beans.factory.annotation.Qualifier;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

@Configuration

public class PersonFactory {

@Bean

@Qualifier("student")

public Person createStudent() {

return new Student();

}

@Bean

@Qualifier("manager")

public Person createManager() {

return new Manager();

}

}

In the previous example, the beans were auto-detected by Spring. Here, the PersonFactory creates two beans with the help of the @Bean annotation.

@Bean

@Qualifier("student")

public Person createStudent() {

return new Student();

}

The @Bean annotation marks methods that define beans. The @Qualifier("student") tells which implementation of the Person to create.

## Creating custom @Qualifier annotation

To reduce code, we can create custom @Qualifier annotations.

tree

.

├── pom.xml

└── src

├── main

│ ├── java

│ │ └── com

│ │ └── zetcode

│ │ ├── Application.java

│ │ ├── bean

│ │ │ ├── Manager.java

│ │ │ ├── Person.java

│ │ │ └── Student.java

│ │ ├── conf

│ │ │ └── PersonFactory.java

│ │ ├── MyRunner.java

│ │ └── qualifier

│ │ └── PersonQ.java

│ └── resources

└── test

└── java

This is the project structure; we list all files except for pom.xml, which is listed in the first application.

Person.java

package com.zetcode.bean;

public interface Person {

public String info();

}

This is the Person type.

Manager.java

package com.zetcode.bean;

import org.springframework.stereotype.Component;

@Component

public class Manager implements Person {

@Override

public String info() {

return "Manager";

}

}

The Manager class is decorated with @Component annotation; it will be auto-detected by Spring.

Student.java

package com.zetcode.bean;

import org.springframework.stereotype.Component;

@Component

public class Student implements Person {

@Override

public String info() {

return "Student";

}

}

The same applies for the Student bean.

PersonQ.java

package com.zetcode.qualifier;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

import org.springframework.beans.factory.annotation.Qualifier;

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

@Qualifier

public @interface PersonQ {

String value();

}

Here we define a new @PersonQ qualifier.

@Target({ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})

The @Targer annotation tells where the annotation can be applied. In our case, it can be applied to fields, methods, and parameters.

@Retention(RetentionPolicy.RUNTIME)

The @Retention annotation specifies how the marked annotation is stored. With RetentionPolicy.RUNTIME the marked annotation is retained by the JVM so it can be used by the runtime environment.

public @interface PersonQ {

The @interface keyword is used to declare a new annotation type.

PersonFactory.java

package com.zetcode.conf;

import com.zetcode.bean.Manager;

import com.zetcode.bean.Person;

import com.zetcode.bean.Student;

import com.zetcode.qualifier.PersonQ;

import org.springframework.context.annotation.Configuration;

@Configuration

public class PersonFactory {

@PersonQ("student")

public Person createStudent() {

return new Student();

}

@PersonQ("manager")

public Person createManager() {

return new Manager();

}

}

In the PersonFactory we use the @PersonQ to identify what kind of beans are created.

MyRunner.java

package com.zetcode;

import com.zetcode.bean.Person;

import com.zetcode.qualifier.PersonQ;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.stereotype.Component;

@Component

public class MyRunner implements CommandLineRunner {

@Autowired

@PersonQ("student")

Person p1;

@Autowired

@PersonQ("manager")

Person p2;

@Override

public void run(String... args) throws Exception {

System.out.println(p1.info());

System.out.println(p2.info());

}

}

In the MyRunner, we inject beans with @Autowired and @PersonQ annotations.

Application.java

package com.zetcode;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}