@Order in Spring

Last modified: February 21, 2019

by baeldung

Spring · Spring DI

1. Overview

In this tutorial, we're going to learn about Spring's *@Order* annotation. **The *@Order* annotation defines the sorting order of an annotated component or bean.**

It has an optional value argument which determines the order of the component; the default value is *Ordered.LOWEST_PRECEDENCE*. This marks that the component has the lowest priority among all other ordered components.

Similarly, the value *Ordered.HIGHEST_PRECEDENCE* can be used for overriding the highest priority among components.

2. When to Use *@Order*

Before Spring 4.0, the *@Order* annotation was used only for the AspectJ execution order. It means the highest order advice will run first.

Since Spring 4.0, it supports the ordering of injected components to a collection. As a result, Spring will inject the auto-wired beans of the same type based on their order value.

Let's explore it with a quick example.

3. How to Use *@Order*

First of all, let's set up our project with the relevant interface and classes.

3.1. Interface Creation

Let's create the *Rating* interface that determines the rating of a product:

```
public interface Rating {
    int getRating();
}
```

3.2. Components Creation

Finally, let's create three components that define the ratings of some products:

```
@Component
@Order(1)
public class Excellent implements Rating {

    @Override
    public int getRating() {
        return 1;
    }
}

@Component
@Order(2)
public class Good implements Rating {

    @Override
    public int getRating() {
        return 2;
    }
}

@Component
@Order(Ordered.LOWEST_PRECEDENCE)
public class Average implements Rating {

    @Override
    public int getRating() {
        return 3;
    }
}
```

Note that the *Average* class has the lowest priority because of its overridden value.

4. Testing Our Example

Up until now, we've created all the required components and the interface to test the *@Order* annotation. Now, let's test it to confirm that it works as expected:

```
public class RatingRetrieverUnitTest {

    @Autowired
    private List<Rating> ratings;

    @Test
    public void givenOrder_whenInjected_thenByOrderValue() {
        assertThat(ratings.get(0).getRating(), is(equalTo(1)));
        assertThat(ratings.get(1).getRating(), is(equalTo(2)));
        assertThat(ratings.get(2).getRating(), is(equalTo(3)));
    }
}
```

5. Conclusion

We've learned about the *@Order* annotation in this quick article. We can find the application of *@Order* in various use cases – where the ordering of the auto-wired components matter. One example is the Spring's request filters.

Due to its influence on injection precedence, it may seem like it might influence the singleton startup order also. But in contrast, the dependency relationships and *@DependsOn* declarations determine the singleton startup order.

All examples mentioned in this tutorial can be found over on Github.

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE

Learning to build your API **with Spring**?

Enter your email address

>> Get the eBook

Comments are closed on this article!

Baeldung

CATEGORIES

SPRING
REST
JAVA
SECURITY
PERSISTENCE
JACKSON
HTTP CLIENT-SIDE
KOTLIN

SERIES

JAVA "BACK TO BASICS" TUTORIAL
JACKSON JSON TUTORIAL
HTTPCLIENT 4 TUTORIAL
REST WITH SPRING TUTORIAL
SPRING PERSISTENCE TUTORIAL
SECURITY WITH SPRING

ABOUT

ABOUT BAELDUNG
THE COURSES
CONSULTING WORK
META BAELDUNG
THE FULL ARCHIVE
WRITE FOR BAELDUNG
EDITORS
OUR PARTNERS
ADVERTISE ON BAELDUNG

TERMS OF SERVICE PRIVACY POLICY COMPANY INFO CONTACT