HowToDoInJava

Spring Boot Annotations

By: Lokesh Gupta Spring Boot 2

The spring boot annotations are mostly placed in `org.springframework.boot.autoconfigure` and `org.springframework.boot.autoconfigure.condition` packages. Let's learn about some frequently used **spring boot annotations** as well as which work behind the scene.

1. @SpringBootApplication

Spring boot is mostly about auto-configuration. This auto-configuration is done by **component scanning** i.e. finding all classes in classspath for `@Component` annotation. It also involve scanning of `@Configuration` annotation and initialize some extra beans.

`@SpringBootApplication` annotation enable all able things in one step. It enables the three features:

1. `@EnableAutoConfiguration` : enable auto-configuration mechanism
2. `@ComponentScan` : enable @Component scan
3. `@SpringBootConfiguration` : register extra beans in the context

> The java class annotated with `@SpringBootApplication` is the main class of a Spring Boot application and application starts from here.

```
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);
  }

}
```

2. @EnableAutoConfiguration

This annotation enables auto-configuration of the Spring Application Context, attempting to guess and configure beans that we are likely to need based on the presence of predefined classes in classpath.

For example, if we have tomcat-embedded.jar on the classpath, we are likely to want a `TomcatServletWebServerFactory`.

As this annotation is already included via `@SpringBootApplication`, so adding it again on main class has no impact. It is also advised to include this annotation only once via `@SpringBootApplication`.

Auto-configuration classes are regular Spring Configuration beans. They are located using the `SpringFactoriesLoader` mechanism (keyed against this class). Generally auto-configuration beans are `@Conditional` beans (most often using `@ConditionalOnClass` and `@ConditionalOnMissingBean` annotations).

3. @SpringBootConfiguration

It indicates that a class provides Spring Boot application configuration. It can be used as an alternative to the Spring's standard `@Configuration` annotation so that configuration can be found automatically.

Application should only ever include one `@SpringBootConfiguration` and most idiomatic Spring Boot applications will inherit it from `@SpringBootApplication`.

The main difference is both annotations is that `@SpringBootConfiguration` allows configuration to be automatically located. This can be especially useful for unit or integration tests.

4. @ImportAutoConfiguration

It import and apply only the specified auto-configuration classes. The difference between `@ImportAutoConfiguration` and `@EnableAutoConfiguration` is that later attempts to configure beans that are found in the classpath during scanning, whereas `@ImportAutoConfiguration` only runs the configuration classes that we provide in the annotation.

We should use `@ImportAutoConfiguration` when we don't want to enable the default auto-configuration.

```
@ImportAutoConfiguration example

@ComponentScan("path.to.your.controllers")
@ImportAutoConfiguration({WebMvcAutoConfiguration.class
    ,DispatcherServletAutoConfiguration.class
    ,EmbeddedServletContainerAutoConfiguration.class
    ,ServerPropertiesAutoConfiguration.class
    ,HttpMessageConvertersAutoConfiguration.class})
public class App
{
  public static void main(String[] args)
  {
    SpringApplication.run(App.class, args);
  }
}
```

5. @AutoConfigureBefore, @AutoConfigureAfter, @AutoConfigureOrder

We can use the `@AutoConfigureAfter` or `@AutoConfigureBefore` annotations if our configuration needs to be applied in a specific order (before of after).

If we want to order certain auto-configurations that should not have any direct knowledge of each other, we can also use `@AutoConfigureOrder`. That annotation has the same semantic as the regular `@Order` annotation but provides a dedicated order for auto-configuration classes.

```
@AutoConfigureAfter Example

@Configuration
@AutoConfigureAfter(CacheAutoConfiguration.class)
@ConditionalOnBean(CacheManager.class)
@ConditionalOnClass(CacheStatisticsProvider.class)
public class RedissonCacheStatisticsAutoConfiguration
{
  @Bean
  public RedissonCacheStatisticsProvider redissonCacheStatisticsProvider(){
    return new RedissonCacheStatisticsProvider();
  }
}
```

5. Condition Annotations

All auto-configuration classes generally have one or more `@Conditional` annotations. It allow to register a bean only when the condition meets. Following are some useful conditional annotations to use.

5.1. @ConditionalOnBean and @ConditionalOnMissingBean

These annotations let a bean be included based on the presence or absence of specific beans.

It's `value` attribute is used to specify beans **by type** or `by name`. Also the `search` attribute lets us limit the `ApplicationContext` hierarchy that should be considered when searching for beans.

Using these annotations at the class level prevents registration of the `@Configuration` class as a bean if the condition does not match.

In below example, bean `JpaTransactionManager` will only be loaded if a bean of type `JpaTransactionManager` is not already defined in the application context.

```
@Bean
@ConditionalOnMissingBean(type = "JpaTransactionManager")
JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactor
{
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory);
    return transactionManager;
}
```

5.2. @ConditionalOnClass and @ConditionalOnMissingClass

These annotations let configuration classes be included based on the presence or absence of specific classes. Notice that annotation metadata is parsed by using spring ASM module, and even if a class might not be present in runtime – you can still refer to the class in annotation.

We can also use `value` attribute to refer the real class or the `name` attribute to specify the class name by using a String value.

Below configuration will create `EmbeddedAcmeService` only if this class is available in runtime and no other bean with same name is present in application context.

```
@Configuration
@ConditionalOnClass(EmbeddedAcmeService.class)
static class EmbeddedConfiguration
{

  @Bean
  @ConditionalOnMissingBean
  public EmbeddedAcmeService embeddedAcmeService() { ... }

}
```

5.3. @ConditionalOnNotWebApplication and @ConditionalOnWebApplication

These annotations let configuration be included depending on whether the application is a “web application” or not. In Spring, a web application is one which meets at least one of below three requirements:

1. uses a Spring `WebApplicationContext`
2. defines a `session` scope
3. has a `StandardServletEnvironment`

5.4. @ConditionalOnProperty

This annotation lets configuration be included based on the presence and value of a Spring Environment property.

For example, if we have different datasource definitions for different environments, we can use this annotation.

```
@Bean
@ConditionalOnProperty(name = "env", havingValue = "local")
DataSource dataSource()
{
    // ...
}

@Bean
@ConditionalOnProperty(name = "env", havingValue = "prod")
DataSource dataSource()
{
    // ...
}
```

5.5. @ConditionalOnResource

This annotation lets configuration be included only when a specific resource is present in the classpath. Resources can be specified by using the usual Spring conventions.

```
@ConditionalOnResource(resources = "classpath:vendor.properties")
Properties additionalProperties()
{
    // ...
}
```

5.6. @ConditionalOnExpression

This annotation lets configuration be included based on the result of a SpEL expression. Use this annotation when condition to evaluate is complex one and shall be evaluated as one condition.

```
@Bean
@ConditionalOnExpression("${env} && ${havingValue == 'local'}")
DataSource dataSource()
{
    // ...
}
```

5.7. @ConditionalOnCloudPlatform

This annotation lets configuration be included when the specified cloud platform is active.

```
@Configuration
@ConditionalOnCloudPlatform(CloudPlatform.CLOUD_FOUNDRY)
public class CloudConfigurationExample
{
  @Bean
  public MyBean myBean(MyProperties properties)
  {
    return new MyBean(properties.getParam);
  }
}
```

Drop me your questions related to **spring boot annotations** in comments.

Happy Learning !!

Ref: Spring Boot Docs

Recommended Reading:

1. Spring Stereotype Annotations
2. Spring Bean Validation – JSR-303 Annotations
3. Spring annotations
4. Spring Boot Remoting – Spring RMI annotation example
5. Spring retry module example with spring boot
6. Struts 2 Hello World Annotations Example
7. Hibernate Annotations vs XML Mappings
8. Hibernate/JPA Persistence Annotations
9. JAXB Annotations
0. Mockito annotations – @Mock, @Spy, @Captor, @InjectMocks

3 thoughts on “Spring Boot Annotations”

Gani

December 15, 2019 at 6:40 am

Hi
Nice explanation we can easily understandable what bean when to use

Thanks

Reply

vishwas tyagi

November 24, 2019 at 9:05 pm

Could you please elaborate more on the difference between @EnableAutoConfiguration vs @SpringBootConfiguration vs @SpringBootApplication. Also please improve your content. Your writing is much verbose and ambiguous.

Reply

Lokesh Gupta

November 26, 2019 at 12:00 am

Thanks for the feedback.

Reply

Leave a Comment

Name *

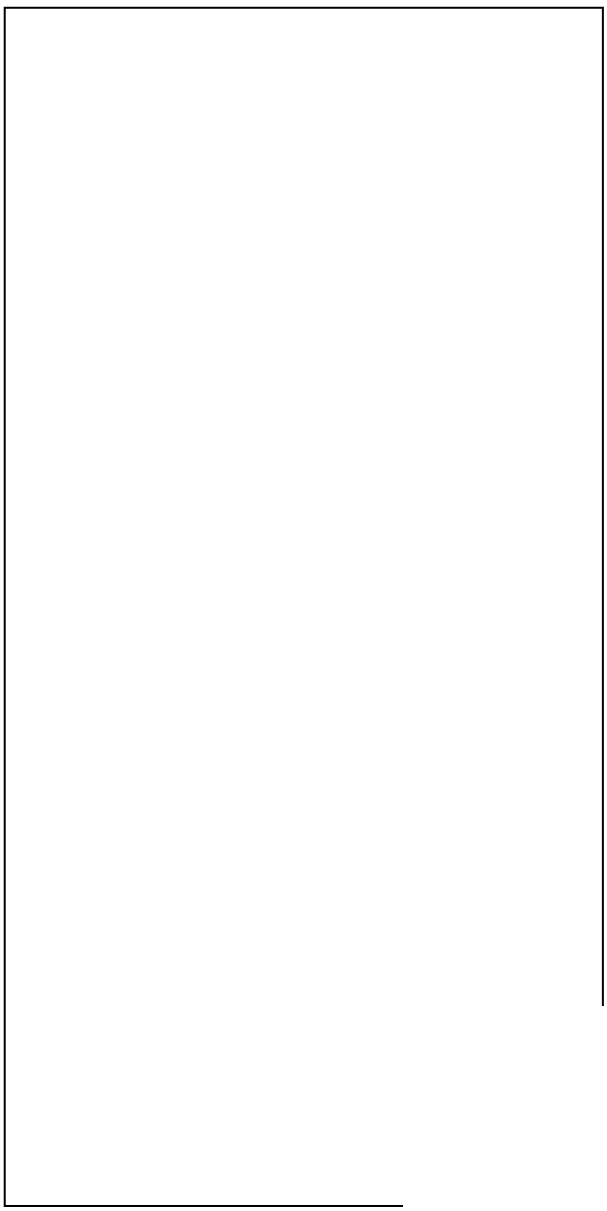
Email *

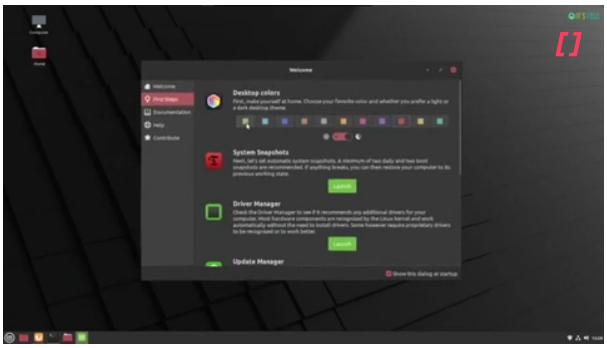
Website

Add me to your newsletter and keep me updated whenever you publish new blog posts

Post Comment

Search ...





HowToDoInJava

A blog about Java and related technologies, the best practices, algorithms, and interview questions.

Meta Links

- About Me
- Contact Us
- Privacy policy
- Advertise
- Guest Posts

Blogs

REST API Tutorial