Spring Boot 3 Autoconfiguration

Author: Hong Le Nguyen Update: **05.2024**

1 - How Spring Boot autoconfiguration works

Autoconfiguration enable

You annotate the application entry point class with @SpringBootApplication, equivalent to declaring the @Configuration, @ComponentScan, and @EnableAutoConfiguration annotations.

The @EnableAutoConfiguration annotation enables the autoconfiguration of Spring ApplicationContext by scanning the classpath components, detecting auto-configuration classes and registering the beans that match various conditions.

Autoconfiguration process

Spring Boot reads *org.springframework.boot.autoconfigure.AutoConfiguration.imports files* from all jars in the classpath, gathering a list of auto-configuration classes. Each auto-configuration class can have multiple conditional annotations.

If conditions are met, Spring Boot executes the auto-configuration class, resulting in the creation of beans and other configurations.

2 - An (custom) autoconfiguration class

Condition annotations

Usually auto-configuration classes use @ConditionalOnClass and @ConditionalOnMissingBean annotations. This ensures that auto-configuration only applies when relevant classes are found and when you have not declared your own @Configuration.

@ConditionalOnClass({TwitterFactory.class, Twitter.class}) to specify that this autoconfiguration should take place only when the TwitterFactory.class and Twitter.class are present.

@ConditionalOnMissingBean on bean definition methods to consider this bean definition only if the TwitterFactory bean or Twitter bean is not already defined explicitly.

Locating auto-configuration candidates

Spring Boot checks for the presence of a src/main/resources/META-INF/spring/ *org.springframework.boot.autoconfigure.AutoConfiguration.imports file* within your published jar.

ConfigurationProperties enable

The example annotated @EnableConfigurationProperties(TwitterProperties.class) to enable support for ConfigurationProperties and injected the TwitterProperties bean.

TwitterAutoConfiguration class

which contains the bean definitions that will be automatically configured based on some criteria.

```
@Configuration
@ConditionalOnClass({TwitterFactory.class, Twitter.class})
@EnableConfigurationProperties(TwitterProperties.class)
public class TwitterAutoConfiguration {

        private final TwitterProperties properties;

        @Bean
        @ConditionalOnMissingBean
        public TwitterFactory twitterFactory(){ ... }

        @Bean
        @ConditionalOnMissingBean
        public Twitter twitter(TwitterFactory twitterFactory){
                return twitterFactory.getInstance();
        }
}

@ConfigurationProperties(prefix = "twitter4j")
public class TwitterProperties {
        private String accessToken;
        ...
 }
```

When should I create and use custom auto-configurations?

Create custom auto-configurations if Spring Boot does not auto-configure a bean that is used in multiple projects in your organization and needs to be configured based on certain conditions. For simpler scenarios, @Configuration class within your application code would be the way to go.

3 - Overriding auto-configuration

Overriding bean auto-configuration

All you need to do to override Spring Boot auto-configuration is to write ***explicit configuration***. Spring Boot will see your configuration, step back, and let your configuration take precedence.

The auto-configuration uses Spring's conditional support (@ConditionalOnMissingBean annotation) to make runtime decisions to whether or not bean definitions should be used or ignored.

Spring Boot loads *application-level configuration before considering auto-configuration classes*.

Therefore, if you've already configured a TwitterFactory bean, then there will be a bean of type TwitterFactory by the time that auto-configuration takes place, and the auto-configured TwitterFactory bean will be ignored.

Overriding configuration properties

The beans that are automatically configured by Spring Boot offer properties for fine-tuning. When you need to adjust the settings, you can specify these properties via environment variables, Java system properties, JNDI , command-line arguments, or property files.

There are, in fact, several ways to set properties for a Spring Boot application. Spring Boot will draw properties from several property sources, including the following:

1 Command-line arguments
2 JNDI attributes from java:comp/env
3 JVM system properties
4 Operating system environment variables
5 Randomly generated values for properties prefixed with random.* (referenced when setting other properties, such as `${random.long})
6 An application.properties or application.yml file outside of the application
7 An application.properties or application.yml file packaged inside of the application
8 Property sources specified by @PropertySource
9 Default properties

This list is in order of precedence. That is, any property set from a source higher in the list will override the same property set on a source lower in the list. Command-line arguments, for instance, override properties from any other property source.