Since its release, Spring Boot has been a huge success: it boosts developers productivity with its convention over configuration philosophy. However, sometimes, it just feels too magical. I have always been an opponent of autowiring for this exact same reason. And when something doesn’t work, it’s hard to get back on track.

This is the reason why I wanted to dig deeper into Spring Boot starter mechanism - to understand every nook and cranny. This post is the first part and will focus on analyzing how it works. The second part will be a case study on creating a starter.

**spring.factories**

At the root of every Spring Boot starter lies the META-INF/spring.factories file. Let’s check the content of this file in the spring-boot-autoconfigure.jar. Here’s an excerpt of it:

...

# Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\

org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\

org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\

org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\

org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\

org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\

org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\

...

Now let’s have a look at their content. For example, here’s the JpaRepositoriesAutoConfiguration class:

@Configuration

@ConditionalOnBean**(**DataSource**.**class**)**

@ConditionalOnClass**(**JpaRepository**.**class**)**

@ConditionalOnMissingBean**({** JpaRepositoryFactoryBean**.**class**,** JpaRepositoryConfigExtension**.**class **})**

@ConditionalOnProperty**(**prefix **=** "spring.data.jpa.repositories"**,** name **=** "enabled"**,**

havingValue **=** "true"**,** matchIfMissing **=** **true)**

@Import**(**JpaRepositoriesAutoConfigureRegistrar**.**class**)**

@AutoConfigureAfter**(**HibernateJpaAutoConfiguration**.**class**)**

**public** **class** **JpaRepositoriesAutoConfiguration** **{}**

There are a couple of interesting things to note:

1. It’s a standard Spring @Configuration class
2. The class contains no “real” code but imports another configuration - JpaRepositoriesAutoConfigureRegistrar, which contains the “real” code
3. There are a couple of @ConditionalOnXXX annotations used
4. There seem to be a order dependency management of some sort with @AutoConfigureAfter

Points 1 and 2 are self-explanatory, point 4 is rather straightforward so let’s focus on point 3.

**@Conditional annotations**

If you didn’t start to work with Spring yesterday, you might know about the @Profile annotation. Profiles are a way to mark a bean-returning method as being *optional*. When a profile is activated, the relevant profile-annotated method is called and the returning bean contributed to the bean factory.

Some time ago, @Profile looked like that:

@Retention**(**RetentionPolicy**.**RUNTIME**)**

@Target**(**ElementType**.**TYPE**)**

**public** @interface Profile **{**

String**[]** **value();**

**}**

Interestingly enough, @Profile has been rewritten to use the new @Conditional annotation:

@Retention**(**RetentionPolicy**.**RUNTIME**)**

@Target**({**ElementType**.**TYPE**,** ElementType**.**METHOD**})**

@Documented

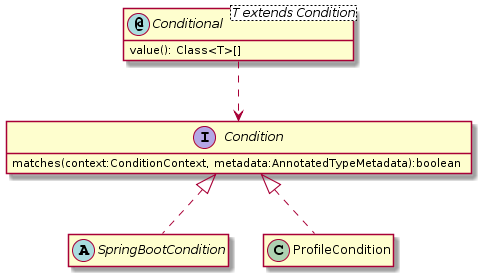
@Conditional**(**ProfileCondition**.**class**)**

**public** @interface Profile **{**

String**[]** **value();**

**}**

Basically, a @Conditional annotation just points to a Condition. In turn, a condition is a functional interface with a single method that returns a boolean: if true, the @Conditional-annotated method is executed by Spring and its returning object added to the context as a bean.



There are a lot of conditions available out-of-the-box with Spring Boot:

| **Condition** | **Description** |
| --- | --- |
| OnBeanCondition | Checks if a bean is in the Spring factory |
| OnClassCondition | Checks if a class is on the classpath |
| OnExpressionCondition | Evalutates a SPeL expression |
| OnJavaCondition | Checks the version of Java |
| OnJndiCondition | Checks if a JNDI branch exists |
| OnPropertyCondition | Checks if a property exists |
| OnResourceCondition | Checks if a resource exists |
| OnWebApplicationCondition | Checks if a WebApplicationContext exists |

Those can be combined together with boolean conditions:

| **Condition** | **Description** |
| --- | --- |
| AllNestedConditions | AND operator |
| AnyNestedConditions | OR operator |
| NoneNestedCondition | NOT operator |

Dedicated @Conditional annotations point to those annotations. For example, @ConditionalOnMissingBean points to the OnBeanCondition class.