**1. Overview**

Spring Boot made configuring Spring easier with its auto-configuration feature.

In this quick tutorial, we'll explore the annotations from the *org.springframework.boot.autoconfigure* and *org.springframework.boot.autoconfigure.condition* packages.

**2. *@SpringBootApplication***

We use this annotation to **mark the main class of a Spring Boot application**:

@SpringBootApplication

**class** VehicleFactoryApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.run(VehicleFactoryApplication.class, args);

}

}

*@SpringBootApplication* encapsulates ***@Configuration*, *@EnableAutoConfiguration*, and *@ComponentScan*** annotations with their default attributes.

**3. *@EnableAutoConfiguration***

*@EnableAutoConfiguration*, as its name says, enables auto-configuration. It means that **Spring Boot looks for auto-configuration beans** on its classpath and automatically applies them.

Note, that we have to use this annotation with *@Configuration*:

@Configuration

@EnableAutoConfiguration

**class** VehicleFactoryConfig {}

**4. Auto-Configuration Conditions**

Usually, when we write our **custom auto-configurations**, we want Spring to **use them conditionally**. We can achieve this with the annotations in this section.

We can place the annotations in this section on *@Configuration* classes or *@Bean* methods.

In the next sections, we'll only introduce the basic concept behind each condition. For further information, please visit [this article](https://www.baeldung.com/spring-boot-custom-auto-configuration).

**4.1. *@ConditionalOnClass* and *@ConditionalOnMissingClass***

Using these conditions, Spring will only use the marked auto-configuration bean if the class in the annotation's **argument is present/absent**:

@Configuration

@ConditionalOnClass(DataSource.class)

**class** MySQLAutoconfiguration {

//...

}

**4.2. *@ConditionalOnBean* and *@ConditionalOnMissingBean***

We can use these annotations when we want to define conditions based on the **presence or absence of a specific bean**:

@Bean

@ConditionalOnBean(name = "dataSource")

LocalContainerEntityManagerFactoryBean entityManagerFactory() {

// ...

}

**4.3. *@ConditionalOnProperty***

With this annotation, we can make conditions on the **values of properties**:

@Bean

@ConditionalOnProperty(

name = "usemysql",

havingValue = "local"

)

DataSource dataSource() {

// ...

}

**4.4. *@ConditionalOnResource***

We can make Spring to use a definition only when a specific **resource is present**:

@ConditionalOnResource(resources = "classpath:mysql.properties")

Properties additionalProperties() {

// ...

}

**4.5. *@ConditionalOnWebApplication* and *@ConditionalOnNotWebApplication***

With these annotations, we can create conditions based on if the current **application is or isn't a web application**:

@ConditionalOnWebApplication

HealthCheckController healthCheckController() {

// ...

}

**4.6. *@ConditionalExpression***

We can use this annotation in more complex situations. Spring will use the marked definition when the **SpEL expression is evaluated to true**:

@Bean

@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")

DataSource dataSource() {

// ...

}

**4.7. *@Conditional***

For even more complex conditions, we can create a class evaluating the **custom condition**. We tell Spring to use this custom condition with *@Conditional*:

@Conditional(HibernateCondition.class)

Properties additionalProperties() {

//...

}

**5. Conclusion**

In this article, we saw an overview of how can we fine-tune the auto-configuration process and provide conditions for custom auto-configuration beans.