(/)

A Guide to Spring Boot Configuration Metadata

Last modified: September 5, 2019

by Dionis Prifti (https://www.baeldung.com/author/dionis-prifti/)

Spring Boot (https://www.baeldung.com/category/spring/spring-boot/)

I just announced the new *Learn Spring* course, focused on the fundamentals of

1. Overview

When writing a Spring Boot application, it's helpful to map configuration properties onto Java beans (https://www.baeldung.com/configuration-properties-in-spring-boot). What's the best way to document these properties, though?

In this tutorial, we'll explore the Spring Boot Configuration Processor (https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html#configuration-metadata-annotation-processor) and the associated JSON metadata files (https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html#configuration-metadata-format) that document each property's meaning, constraints, and so on.

2. Configuration Metadata

Most of the applications we work on as developers must be configurable to some extent. However, usually, we don't really understand what a configuration parameter does, if it has a default value, if it's deprecated, and at times, we don't even know the property exists.

To help us out, Spring Boot generates configuration metadata in a JSON file, which gives us useful information on how to use the properties. So, **the configuration metadata is a descriptive file which contains the necessary information for interaction with the configuration properties.**

The really nice thing about this file is that **IDEs can read it, too**, giving us autocomplete of Spring properties, as well as other configuration hints.

X

3. Dependencies

In order to generate this configuration metadata, we'll use the configuration processor from the *spring-boot-configuration-processor* dependency (https://search.maven.org/search?q=spring-boot-configuration-processor).

So, let's go ahead and add the dependency as *optional*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <version>2.1.7.RELEASE</version>
    <optional>true</optional>
</dependency>
```

This dependency will provide us with a Java annotation processor invoked when we build our project. We'll talk in detail about this later on.

It's a best practice to add a dependency as *optional* in Maven in order to prevent *@ConfigurationProperties* from being applied to other modules that our project uses.

Sé de
en ex
últim

X

To see the processor in action, let's imagine we have a few properties that we need to include in our Spring Boot application via a Java bean:

```
@Configuration
@ConfigurationProperties(prefix = "database")
public class DatabaseProperties {

    public static class Server {

        private String ip;
        private int port;

        // standard getters and setters
    }

    private String username;
    private String password;
    private Server server;

    // standard getters and setters
}
```

To do this, we'd use the *@ConfigurationProperties* (https://www.baeldung.com/configuration-properties-in-spring-boot) annotation. **The configuration processor scans for classes and methods with this annotation** to access the configuration parameters and generate configuration metadata.

Let's add a couple of these properties into a properties file. In this case, we'll call it *databaseproperties-test.properties*:

```
#Simple Properties
database.username=baeldung
database.password=password
```

And, just to be sure, we'll also add a test to make sure that we are all lined up:

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = AnnotationProcessorApplication.class)
@TestPropertySource("classpath:databaseproperties-test.properties")
public class DatabasePropertiesIntegrationTest {

    @Autowired
    private DatabaseProperties databaseProperties;

    @Test
    public void whenSimplePropertyQueriedThenReturnsPropertyValue()
      throws Exception {
        Assert.assertEquals("Incorrectly bound Username property",
          "baeldung", databaseProperties.getUsername());
        Assert.assertEquals("Incorrectly bound Password property",
          "password", databaseProperties.getPassword());
    }

}
```

We've also added the nested properties *database.server.id* and *database.server.port* via the inner class *Server*. **We should add the inner class *Server* as well as a field *server* with its own getter and setter.**

In our test, let's do a quick check to make sure we can set and read successfully nested properties as well:

Sé de
en ex
últim

```
      "127.0.0.1", databaseProperties.getServer().getIp());
    Assert.assertEquals("Incorrectly bound Server Port nested property",
      3306, databaseProperties.getServer().getPort());
}
```

Okay, now we're ready to use the processor.

5. Generating Configuration Metadata

We mentioned earlier that the configuration processor generates a file – it does this uses annotation processing.

So, after compiling our project, we'll see a **file called *spring-configuration-metadata.json* inside *target/classes/META-INF***:

```
{
  "groups": [
    {
      "name": "database",
      "type": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties"
    },
    {
      "name": "database.server",
      "type": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties$Server",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties",
      "sourceMethod": "getServer()"
    }
  ],
  "properties": [
    {
      "name": "database.password",
      "type": "java.lang.String",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties"
    },
    {
      "name": "database.server.ip",
      "type": "java.lang.String",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties$Server"
    },
    {
      "name": "database.server.port",
      "type": "java.lang.Integer",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties$Server",
      "defaultValue": 0
```

```
    }
  ],
  "hints": []
}
```

Next, let's see how changing annotations on our Java beans affect the metadata.

5.1. Additional Information on Configuration Metadata

First, let's add JavaDoc comments on *Server*.

Second, let's give a default value to the *database.server.port* field and finally add the *@Min* and *@Max* annotations:

```
public static class Server {

    /**
     * The IP of the database server
     */
    private String ip;

    /**
     * The Port of the database server.
     * The Default value is 443.
     * The allowed values are in the range 400-4000.
     */
    @Min(400)
    @Max(800)
    private int port = 443;

    // standard getters and setters
}
```

If we check the *spring-configuration-metadata.json* file now, we'll see this extra information reflected:

```
{
  "groups": [
    {
      "name": "database",
      "type": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties"
    },
    {
      "name": "database.server",
      "type": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties$Server",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties",
      "sourceMethod": "getServer()"
    }
  ],
  "properties": [
    {
      "name": "database.password",
      "type": "java.lang.String",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties"
    },
    {
      "name": "database.server.ip",
      "type": "java.lang.String",
      "description": "The IP of the database server",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties$Server"
    },
    {
      "name": "database.server.port",
      "type": "java.lang.Integer",
      "description": "The Port of the database server. The Default value is 443.
        The allowed values are in the range 400-4000",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties$Server",
      "defaultValue": 443
    },
    {
      "name": "database.username",
      "type": "java.lang.String",
      "sourceType": "com.baeldung.autoconfiguration.annotationprocessor.DatabaseProperties"
    }
  ],
  "hints": []
}
```

We can check the differences with the *database.server.ip* and *database.server.port* fields. Indeed, the extra information is quite helpful. As a result, it's much easier for developers and IDEs to understand what each property does.

We should also make sure we trigger the build to get the updated file. In Eclipse, if we check the *Build Automatically* option, each save action will trigger a build. In IntelliJ, we should trigger the build manually.

5.2. Understanding the Metadata Format

Let's have a closer look at the JSON metadata file and discuss its components.

Groups are higher-level items used to group other properties, without specifying a value itself. In our example, we have the *database* group, which is also the prefix of the configuration properties. We also have a *server* group, which we created via an inner class and groups *ip* and *port* properties.

Properties are configuration items for which we can specify a value. These properties are set in *.properties* or *.yml* files and can have extra information, like default values and validations, as we saw in the example above.

Hints are additional information to help the user set the property value. For example, if we have a set of allowed value for a property, we can provide a description of what each of them does. The IDE will provide auto-competition help for these hints.

Each component on the configuration metadata has its own attributes (https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html) to explain in finer details the configuration properties.

6. Conclusion

In this article, we looked at the Spring Boot Configuration Processor and its ability to create configuration metadata. Using this metadata makes it a lot easier to interact with our configuration parameters.

We gave an example of a generated configuration metadata and explained in details its format and components.

We also saw how helpful the autocomplete support on our IDE can be.

As always, all of the code snippets mentioned in this article can be found on our GitHub repository (https://github.com/eugenp/tutorials/tree/master/spring-boot-autoconfiguration).

X

Enter your email address

>> Get the eBook

Leave a Reply

Start the discussion...

Subscribe

ezoic (https://www.ezoic.com/what-is-ezoic/) report this ad

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)