(/)

Properties with Spring and Spring Boot

Last modified: January 27, 2020

by Eugen Paraschiv (https://www.baeldung.com/author/eugen/)

Spring (https://www.baeldung.com/category/spring/) +

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

Spring Core Basics (https://www.baeldung.com/tag/spring-core-basics/)

1. Overview

This tutorial will show how to set up and use **Properties in Spring** – via Java configuration and *@PropertySource* or via XML and *<property-placeholder>*, **as well how properties work in Spring Boot.**

2. Register a Properties File via Java Annotations

Spring 3.1 also introduces **the new *@PropertySource* annotation**, as a convenient mechanism for adding property sources to the environment. This annotation is to be used in conjunction with Java-based configuration and the *@Configuration* annotation:

```
@Configuration
@PropertySource("classpath:foo.properties")
public class PropertiesWithJavaConfig {
    //...
}
```

One other very useful way of registering a new properties file is using a placeholder to allow you to **dynamically select the right file at runtime**; for example:

```
@PropertySource({
  "classpath:persistence-${envTarget:mysql}.properties"
})
...
```

2.1. Defining Multiple Property Locations

The *@PropertySource* annotation is repeatable according to Java 8 conventions (https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html). Therefore, if we're using Java 8 or higher, we can use this annotation to define multiple property locations:

```
@PropertySource("classpath:foo.properties")
@PropertySource("classpath:bar.properties")
public class PropertiesWithJavaConfig {
    //...
}
```

Of course, we can also use the *@PropertySources* annotation and specify an array of *@PropertySource*. This works in any supported Java version, not just in Java 8 or higher:

```
@PropertySources({
    @PropertySource("classpath:foo.properties"),
    @PropertySource("classpath:bar.properties")
})
public
    //
}
```

In either case, it's worth noting that **in the event of a property name collision, the last source read takes precedence**.

3. Register a Properties File in XML

In XML, new properties files can be made accessible to Spring via **the *<context:property-placeholder … >* namespace element**:

```
<context:property-placeholder location="classpath:foo.properties" />
```

The *foo.properties* file should be placed under */src/main/resources* so that it will be available on the classpath at runtime.

In case **multiple *<property-placeholder>* elements** are present in the Spring context, there are a few recommended best practices:

- the *order* attribute needs to be specified to fix the order in which these are processed by Spring
- all property placeholders minus the last one (highest *order*) should have ***ignore-unresolvable="true"*** to allow the resolution mechanism to pass to others in the context without throwing an exception

3.1. Register Multiple Properties Files in XML

In the previous section, we saw how to define multiple properties files using annotations in Java 8 or later. Similarly, we can **define multiple properties files using XML configuration**:

```
<context:property-placeholder location="classpath:foo.properties, classpath:bar.properties"/>
```

And, as before, **in the event of a property name collision, the last source read takes precedence**.

4. Using / Injecting Properties

Injecting a property with the *@Value* annotation (https://www.baeldung.com/spring-value-annotation) is straightforward:

```
@Value( "${jdbc.url}" )
private String jdbcUrl;
```

A default value of the property can also be specified:

```
@Value( "${jdbc.url:aDefaultUrl}" )
private String jdbcUrl;
```

Using properties in Spring XML configuration:

```
<bean
  <pro
</bean
```

Both the older *PropertyPlaceholderConfigurer* and the new *PropertySourcesPlaceholderConfigurer* added in Spring 3.1 **resolve ${…} placeholders** within bean definition property values and *@Value* annotations.

Finally – we can **obtain the value of a property using the *Environment* API:**

```
@Autowired
private Environment env;
...
dataSource.setUrl(env.getProperty("jdbc.url"));
```

A very important caveat here is that **using *<property-placeholder>* will not expose the properties to the Spring Environment** – this means that retrieving the value like this will not work – it will return *null*:

```
env.getProperty("key.something")
```

5. Properties with Spring Boot

Before we go into more advanced configuration options for properties, let's spend some time looking at **the new properties support in Spring Boot**.

Generally speaking, this new support involves **less configuration** compared to standard Spring, which is, of course, one of the main goals of Boot.

5.1. *application.properties* – the Default Property File

Boot applies its typical convention over configuration approach to property files. This means that we can simply put an "*application.properties*" file in our "*src/main/resources*" directory, and it will be auto-detected. We can then inject any loaded properties from it as normal.

So, by using this default file, we don't have to explicitly register a *PropertySource*, or even provide a path to a property file.

We can also configure a different file at runtime if we need to, using an environment property:

```
java -jar app.jar --spring.config.location=classpath:/another-location.properties
```

5.2. Environment Specific Properties File

If we need to target different environments, there's a built-in mechanism for that in Boot.

We can simply define an *"application-environment.properties"* file in the *"src/main/resources"* directory – and then set a Spring profile with the same environment name.

For example, if we define a "staging" environment, that means we'll have to define a *staging* profile and then *application-staging.properties*.

This env file will be loaded and **will take precedence over the default property file**. Note that the default file will still be loaded, it's just that when there is a property collision the environment-specific property file takes precedence.

5.3. Test

We might al t.

Spring Boot handles this for us by looking in our "*src/test/resources*" directory during a test run. Again, default properties will still be injectable as normal but will be overridden by these if there is a collision.

5.4. The *@TestPropertySource* Annotation

If we need more granular control over test properties, then we can use the *@TestPropertySource (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/context/TestPropertySource.html)* annotation.

This allows us to set test properties for a specific test context, taking precedence over the default property sources:

```
@RunWith(SpringRunner.class)
@TestPropertySource("/foo.properties")
public class FilePropertyInjectionUnitTest {

    @Value("${foo}")
    private String foo;

    @Test
    public void whenFilePropertyProvided_thenProperlyInjected() {
        assertThat(foo).isEqualTo("bar");
    }
}
```

If we don't want to use a file, we can specify names and values directly:

```
@RunWith(SpringRunner.class)
@TestPropertySource(properties = {"foo=bar"})
public class PropertyInjectionUnitTest {

    @Value("${foo}")
    private String foo;

    @Test
    public void whenPropertyProvided_thenProperlyInjected() {
        assertThat(foo).isEqualTo("bar");
    }
}
```

We can also achieve a similar effect using the *properties* argument of the *@SpringBootTest (https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/test/context/SpringBootTest.html)* annotation:

```
@RunWith(SpringRunner.class)
@SpringBootTest(properties = {"foo=bar"}, classes = SpringBootPropertiesTestApplication.class)
public class SpringBootPropertyInjectionIntegrationTest {

    @Value("${foo}")
    private String foo;

    @Test
    public void whenSpringBootPropertyProvided_thenProperlyInjected() {
        assertThat(foo).isEqualTo("bar");
    }
}
```

5.5. Hierarchical Properties

If we have properties that are grouped together, we can make use of the *@ConfigurationProperties (https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/properties/ConfigurationProperties.html)* annotation which will map these property hierarchies into Java objects graphs.

Let's take some properties used to configure a database connection:

```
database.url=jdbc:postgresql:/localhost:5432/instance
database.username=foo
database.password=bar
```

And then let's use the annotation to map them to a database object:

```
@ConfigurationProperties(prefix = "database")
public class Database {
    String url;
    String username;
    String password;

    // standard getters and setters
}
```

Spring Boot applies it's convention over configuration approach again, automatically mapping between property names and their corresponding fields. All that we need to supply is the property prefix.

If you want to dig deeper into configuration properties, have a look at the in-depth article (/configuration-properties-in-spring-boot).

5.6. Alternative – YAML Files

YAML files are also supported.

All the same naming rules apply for test specific, environment-specific, and default property files. The only difference is the file extension, and a dependency on the SnakeYAML (https://bitbucket.org/asomov/snakeyaml) library being on your classpath.

YAML is particularly good for hierarchical property storage; the following property file:

```
database.url=jdbc:postgresql:/localhost:5432/instance
database.username=foo
database.password=bar
secret: foo
```

Is synonymous to the following YAML file:

```
database:
  url: jdbc:postgresql:/localhost:5432/instance
  username: foo
  password: bar
secret: foo
```

It's also worth mentioning that YAML files do not support the *@PropertySource* annotation, so if the use of the annotation was required it would constrain us to using a properties file.

5.7. Properties from Command Line Arguments

As opposed to using files, properties can be passed directly on the command line:

```
java -jar app.jar --property="value"
```

You can also do this via system properties, which are provided before the *-jar* command rather than after it:

```
java -Dproperty.name="value" -jar app.jar
```

5.8. Properties from Environment Variables

Spring Boot will also detect environment variables, treating them as properties:

```
export name=value
java -jar app.jar
```

5.9 Randomization of Property Values

If we don't want determinist property values, *RandomValuePropertySource (https://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/context/config/RandomValuePropertySource.html)* can be used to randomize the values of properties:

```
random.number=${random.int}
random.long=${random.long}
random.uuid=${random.uuid}
```

5.10. Additional Types of Property Sources

Spring Boot supports a multitude of property sources, implementing a well-thought ordering to allow sensible overriding. It's worth consulting the official documentation (https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html), which goes further than the scope of this article.

6. Configuration using Raw Beans in Spring 3.0 – the *PropertyPlaceholderConfigurer*

Besides the convenient methods of getting properties into Spring – annotations and the XML namespace – the property configuration bean can also be defined and registered **manually**. Working with the *PropertyPlaceholderConfigurer* gives us full control over the configuration, with the downside of being more verbose and most of the time, unnecessary.

6.1. Java configuration

```
@Bean
public static PropertyPlaceholderConfigurer properties() {
    PropertyPlaceholderConfigurer ppc
      = new PropertyPlaceholderConfigurer();
    Resource[] resources = new ClassPathResource[]
      { new ClassPathResource( "foo.properties" ) };
    ppc.setLocations( resources );
    ppc.setIgnoreUnresolvablePlaceholders( true );
    return ppc;
}
```

6.2. XML configuration

```
<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:foo.properties</value>
        </list>
    </property>
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
</bean>
```

7. Configuration using Raw Beans in Spring 3.1 – the *PropertySourcesPlaceholderConfigurer*

Similarly, in Spring 3.1, the new *PropertySourcesPlaceholderConfigurer* can also be configured manually:

7.1. Java configuration

```
@Bean
public static PropertySourcesPlaceholderConfigurer properties(){
    PropertySourcesPlaceholderConfigurer pspc
      = new PropertySourcesPlaceholderConfigurer();
    Resource[] resources = new ClassPathResource[ ]
      { new ClassPathResource( "foo.properties" ) };
    pspc.setLocations( resources );
    pspc.setIgnoreUnresolvablePlaceholders( true );
    return pspc;
}
```

7.2. XML configuration

```
<bean
  class="org.springframework.context.support.PropertySourcesPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>classpath:foo.properties</value>
        </list>
    </property>
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
</bean>
```

8. Properties in Parent-Child Contexts

This question keeps coming up again and again – what happens when your **web application has a parent and a child context**? The parent context may have some common core functionality and beans, and then one (or multiple) child contexts, maybe containing servlet specific beans.

In that case, what's the best way to define properties files and include them in these contexts? What's more – how to best retrieve these properties from Spring? Here is a simple breakdown.

8.1. If the Properties File Is Defined in XML With <p*roperty-placeholder*>

If the file is **defined in the Parent context**:

- *@Value* works in **Child context**: NO
- *@Value* works in **Parent context**: YES

If the file is **defined in the Child context**:

- *@Value* works in **Child context**: YES
- *@Value* works in **Parent context**: NO

Finally, as we discussed before, *<property-placeholder>* does not expose the properties to the environment, so:

- *environment.getProperty* works in **either context**: NO

8.2. If the Properties File Is Defined in Java With *@PropertySource*

If the file is **defined in the Parent context**:

- *@Value* works in **Child context**: YES
- *@Value* works in **Parent context**: YES
- *environment.getProperty* in **Child context**: YES
- *environment.getProperty* in **Parent context**: YES

If the file is **defined in the Child context**:

- *@Value*
- *@Value*
- *environ*