(/)

Spring Profiles

Last modified: January 13, 2020

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

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

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

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-start)

1. Overview

In this article, we'll focus on introducing **Profiles in Spring**.

Profiles are a core feature of the framework – allowing us to map our beans to different profiles – for example, *dev*, *test*, *prod*.

We can then activate different profiles in different environments to bootstrap just the beans we need:

Further reading:

Configuring Separate Spring DataSource for Tests (https://www.baeldung.com/spring-testing-separate-data-source)

A quick, practical tutorial on how to configure a separate data source for testing in a Spring application.

Read more (https://www.baeldung.com/spring-testing-separate-data-source) →

Properties with Spring and Spring Boot (https://www.baeldung.com/properties-with-spring)

Tutorial for how to work with properties files and property values in Spring.

Read more (https://www.baeldung.com/properties-with-spring) →

2. Use *@Profile* on a Bean

Let's start simple and look at how we can make a bean belong to a particular profile. Using the *@Profile* annotation – we are mapping the bean to that particular profile; the annotation simply takes the names of one (or multiple) profiles.

Consider a basic scenario – we have a bean that should only be active during development, but not deployed in production. We annotate that bean with a "*dev*" profile, and it will only be present in the container during development – in production, the *dev* simply won't be active:

```
@Component
@Profile("dev")
public class DevDatasourceConfig
```

As a quick sidenote, profile names can also be prefixed with a NOT operator e.g. "*!dev*" to exclude them from a profile.

In the below example, the component is activated only if "*dev*" profile is not active:

```
@Component
@Profile("!dev")
public class DevDatasourceConfig
```

3. Declare Profiles in XML

Profiles can also be configured in XML – the *<beans>* tag has *"profiles"* attribute which takes comma separated values of the applicable profiles:

```
<beans profile="dev">
    <bean id="devDatasourceConfig"
      class="org.baeldung.profiles.DevDatasourceConfig" />
</beans>
```

4. Set Profiles

The next step is to activate and set the profiles so that the respective beans are registered in the container.

This can be done in a variety of ways – which we'll explore in the following sections.

4.1. Programmatically via *WebApplicationInitializer* Interface

In web applications, *WebApplicationInitializer* can be used to configure the *ServletContext* programmatically.

It's also a very handy location to set our active profiles programmatically:

```
@Configuration
public class MyWebApplicationInitializer
  implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {

        servletContext.setInitParameter(
          "spring.profiles.active", "dev");
    }
}
```

4.2. Programmatically via *ConfigurableEnvironment*

You can also set profiles directly on the environment:

```
@Autowired
private ConfigurableEnvironment env;
...
env.setActiveProfiles("someProfile");
```

4.3. Context Parameter in *web.xml*

Similarly, **profiles can be activated in the *web.xml*** of the web application as well, using a context parameter:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/app-config.xml</param-value>
</context-param>
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>dev</param-value>
</context-param>
```

4.4. JVM System Parameter

The profile names can also be passed in via a JVM system parameter. The profile names passed as the parameter will be activated during application start-up:

```
-Dspring.profiles.active=dev
```

4.5. Environment Variable

In a Unix environment, **profiles can also be activated via the environment variable**:

```
export spring_profiles_active=dev
```

4.6. Maven Profile

Spring profiles can also be activated via Maven profiles, by specifying the *spring.profiles.active* configuration property.

In every Maven profile, we can set a *spring.profiles.active* property:

```
<profiles>
    <profile>
        <id>dev</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <spring.profiles.active>dev</spring.profiles.active>
        </properties>
    </profile>
    <profile>
        <id>prod</id>
        <properties>
            <spring.profiles.active>prod</spring.profiles.active>
        </properties>
    </profile>
</profiles>
```

Its value will be used to replace the *@spring.profiles.active@* placeholder in *application.properties*:

```
spring.profiles.active=@spring.profiles.active@
```

Now, we need to enable resource filtering in *pom.xml*:

```
<build>
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
    ...
</build>
```

And append a *-P* parameter to switch which Maven profile will be applied:

```
mvn clean package -Pprod
```

This command will package the application for *prod* profile. It also applies the *spring.profiles.active* value *'prod'* for this application when it is running.

4.7. *@ActiveProfile* in Tests

Tests make it very easy to specify what profiles are active – using the *@ActiveProfile* annotation to enable specific profiles:

```
@ActiveProfiles("dev")
```

To summarize, we looked at multiple ways of activating profiles. Let's now see which one has priority over the other and what happens if you use more than one – from highest to lowest priority:

1. Context parameter in *web.xml*
2. *WebApplicationInitializer*
3. JVM System parameter
4. Environment variable
5. Maven profile

5. The Default Profile

Any bean that does not specify a profile belongs to "*default*" profile.

Spring also provides a way to set the default profile when no other profile is active – by using the "*spring.profiles.default*" property.

6. Get Active Profiles

Spring's active profiles drive the behavior of the *@Profile* annotation for enabling/disabling beans. However, we may also wish to access the list of active profiles programmatically.

We have two ways to do it, **using *Environment* or *spring.active.profile***.

6.1. Using *Environment*

We can access the active profiles from the *Environment* object by injecting it:

```
public class ProfileManager {
    @Autowired
    private Environment environment;

    public void getActiveProfiles() {
        for (String profileName : environment.getActiveProfiles()) {
            System.out.println("Currently active profile - " + profileName);
        }
    }
}
```

6.2. Using *spring.active.profile*

Alternatively, we could access the profiles by injecting the property *spring.profiles.active*:

```
@Value("${spring.profiles.active}")
private String activeProfile;
```

Here, our *activeProfile* variable **will contain the name of the profile that is currently active,** and if there are several, it'll contain their names separated by a comma.

However, we should **consider what would happen if there is no active profile at all**. With our code above, the absence of an active profile would prevent the application context from being created. This would result in an *IllegalArgumentException* owing to the missing placeholder for injecting into the variable.

In order to avoid this, we can **define a default value**:

```
@Value("${spring.profiles.active:}")
private String activeProfile;
```

Now, if no profiles are active, our *activeProfile* will just contain an empty string. And, if we want to access the list of them just like in the previous example, we can do it by splitting (https://www.baeldung.com/java-split-string) the *activeProfile* variable:

```
public class ProfileManager {
    @Value("${spring.profiles.active:}")
    private String activeProfiles;

    public String getActiveProfiles() {
        for (String profileName : activeProfiles.split(",")) {
            System.out.println("Currently active profile - " + profileName);
        }
    }
}
```

7. Example of Using Profiles

Now that the basics are out of the way, let's take a look at a real example.

Consider a scenario where we have to maintain the datasource configuration for both the development and production environments. Let's create a common interface *DatasourceConfig* that needs to be implemented by both data source implementations:

```
public interface DatasourceConfig {
    public void setup();
}
```

Following is the configuration for the development environment:

```
@Component
@Profile("dev")
public class DevDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
        System.out.println("Setting up datasource for DEV environment. ");
    }
}
```

And configuration for the production environment:

```
@Component
@Profile("production")
public class ProductionDatasourceConfig implements DatasourceConfig {
    @Override
    public void setup() {
       System.out.println("Setting up datasource for PRODUCTION environment. ");
    }
}
```

Now let's create a test and inject our DatasourceConfig interface; depending on the active profile, Spring will inject *DevDatasourceConfig* or *ProductionDatasourceConfig* bean:

```
public class SpringProfilesWithMavenPropertiesIntegrationTest {
    @Autowired
    DatasourceConfig datasourceConfig;

    public void setupDatasource() {
        datasourceConfig.setup();
    }
}
```

When the "*dev*" profile is active spring injects *DevDatasourceConfig* object, and on call of *setup()* method following is the output:

```
Setting up datasource for DEV environment.
```

8. Profiles in Spring Boot

Spring Boot supports all the profile configuration outlined so far, with a few additional features.

The initialization parameter *spring.profiles.active*, introduced in section 4, can also be set up as a property in Spring Boot to define currently active profiles. This is a standard property that Spring Boot will pick up automatically:

```
spring.profiles.active=dev
```

To set profiles programmatically, we can also use the *SpringApplication* class:

```
SpringApplication.setAdditionalProfiles("dev");
```

To set profiles using Maven in Spring Boot, we can specify profile names under *spring-boot-maven-plugin* in *pom.xml:*

```
<plugins>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
            <profiles>
                <profile>dev</profile>
            </profiles>
        </configuration>
    </plugin>
    ...
</plugins>
```

And execute the Spring Boot specific Maven goal:

```
mvn spring-boot:run
```

But the most important profiles-related feature that Spring Boot brings is **profile-specific properties files**. These have to be named in the format *applications-{profile}.properties*.

Spring Boot will automatically load the properties in an *application.properties* file for all profiles, and the ones in profile-specific *.properties* files only for the specified profile.

For example, we can configure different data sources for *dev* and *production* profiles by using two files named *application-dev.properties* and *application-production.properties*:

In the *application-production.properties* file, we can set up a *MySql* data source:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/db (mysql://localhost:3306/db)
spring.datasource.username=root
spring.datasource.password=root
```

Then, we can configure the same properties for the *dev* profile in the *application-dev.properties* file, to use an in-memory *H2* database:

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
```

In this way, we can easily provide different configurations for different environments.

9. Conclusion