**SPRINGBOOT**

Spring Boot is an open-source Java-based framework used to create stand-alone, production-grade Spring-based applications with minimal configuration. It simplifies the development process by providing a set of conventions, auto-configuration, and pre-built components that accelerate the creation of Spring applications.

1. **Standalone**: Spring Boot applications are self-contained and do not require any external application server, as they embed Tomcat, Jetty, or Undertow servlet containers. This simplifies deployment and makes it easy to run Spring applications as executable JAR files.
2. **Convention over Configuration**: Spring Boot follows the principle of "convention over configuration," reducing boilerplate code and configuration. It provides sensible defaults and auto-configuration based on classpath dependencies, minimizing manual configuration.
3. **Auto-Configuration**: Spring Boot automatically configures the application based on its dependencies and environment. It scans the classpath and configures beans, components, and other settings, reducing the need for explicit configuration.
4. **Spring Ecosystem Integration**: Spring Boot seamlessly integrates with other Spring projects and libraries, such as Spring MVC, Spring Data, Spring Security, and Spring Cloud. This allows developers to leverage the full power of the Spring ecosystem in their applications.
5. **Embedded Servers**: Spring Boot supports embedded servlet containers like Tomcat, Jetty, and Undertow, eliminating the need for external application servers. Developers can run Spring Boot applications as standalone executables, making deployment and scaling easier.
6. **Actuator**: Spring Boot Actuator provides built-in endpoints for monitoring and managing the application at runtime. It exposes metrics, health checks, environment information, and other operational endpoints via HTTP or JMX, facilitating monitoring and management of the application in production.
7. **DevTools**: Spring Boot DevTools provide a set of development-time features, including automatic application restarts, live reload of static resources, and remote debugging. These features enhance developer productivity during the development and testing phases.
8. **Spring Initializr**: Spring Initializr is a web-based tool that generates Spring Boot project skeletons with customizable configurations and dependencies. Developers can use Initializr to quickly bootstrap Spring Boot projects with the required dependencies and settings.

Overall, Spring Boot simplifies the development, configuration, and deployment of Spring-based applications, enabling developers to focus on building business logic rather than dealing with infrastructure and boilerplate code. It has become a popular choice for developing microservices, web applications, RESTful APIs, and other types of Spring-based applications in the Java ecosystem.

**SPRINGBOOT PROFILES**

Profiles are the group of configuration properties. These configuration properties belong to a specific environment. If all the configuration properties belong to development environment, we can call it development profile. Similarly, based on the specific configuration properties it holds, we can call other profiles as test profile, prod profile etc. Spring profiles provide a way to separate out parts of our application configuration and make it be available only in the specific environment/(s).

Programmatically, a profile is a property file with extension ‘.properties’ similar to ‘application.properties’ file. In spring Boot, apart from ‘.properties’, a profile can also have ‘.yml’ extension.

## What are the usages of Profiles in Spring Based Application?

Typically, Profiles control two things in a spring based application: configuration properties, and beans.

1) Configuration Properties: Profile controls which configuration properties should be active.

2) Beans: Profile controls which beans should be loaded into the Spring Container. It offers a provision to register beans by condition.

## What is the Default Profile?

Spring Boot comes with a property file, named ‘application.properties’ file by default. Therefore, this file is the default profile. Similarly, ‘application.yml’ file will also be the default profile. The default profile is always active. Spring Boot loads all properties from the default profile first.

If a property is defined in the default profile, but not in the prod profile, the property value will be populated from the default profile. This is very handy for defining default values that are valid across all profiles. Hence, we should keep all properties in default profile (application. properties) which are common across all profiles.

## How to create a Spring Profile?

Instead of changing multiple entries in the default profile each time, we will create a separate dedicated file for each environment. Further, we can activate the required profile by changing only a single entry in the default profile.

[Spring Profile](https://docs.spring.io/spring-boot/docs/1.2.0.M1/reference/html/boot-features-profiles.html) contains the configuration properties of a particular environment. Therefore, we need to know how many environments we have in place. Suppose we have three environments such as: development, test, and production. In this case, we need to create three profiles.

Spring Boot comes with a property file, named ‘application.properties’ file by default. Let’s create three more property files at the same location where ‘application.properties’ file resides as below.

1) application-dev.properties

2) application-test.properties

3) application-prod.properties

Please note the naming convention: application-<environment>.properties

Now, let’s insert some configuration properties in these files.

## How to Activate a Particular Profile?

Although, we have multiple ways to make a particular profile active. Let’s discuss some of them one by one.

### **Approach#1:** By setting ‘spring.profile.active’ in application.properties

The ‘application.properties’ file will be the boss among all the properties files. Here we will specify which profile is active by setting the value of property ‘spring.profiles.active’. For example, below ‘application.properties’ file represents that currently ‘development (dev)’ profile is active.

**application.properties:**

**spring.application.name = Spring Profiles**

**spring.profiles.active = dev**

**app.info = This is the Primary Application Property file**

The value of  ‘spring.profiles.active’ property tells Spring which profile to use. Here we have set the dev profile as active.

This is the most commonly used approach of making a particular profile active.

### **Approach#2:** By setting in JVM System Parameter

We can also pass the profile name in a JVM System parameter as below. The specified profile will be activated during application startup.

**-Dspring.profiles.active=dev**

### **Approach#3:** By setting context parameter in web.xml

If you have web.xml in a web application, spring active profile can be set by specifying a context parameter as shown below:

**<context-param>**

**<param-name>spring.profiles.active</param-name>**

**<param-value>dev</param-value>**

**</context-param>**

### Approach#4: By implementing **WebApplicationInitializer Interface**

Programmatically, in web applications, we can configure the ServletContext by implementing WebApplicationInitializer interface. Here, we can also set our active profiles programmatically as shown below:

**@Configuration**

**public class MyWebApplicationInitializer implements WebApplicationInitializer {**

**@Override**

**public void onStartup(ServletContext servletContext) throws ServletException {**

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

**}**

**}**

### **Approach#5:** By setting active profiles in pom.xml

Furthermore, we can activate spring profile via Maven profiles by specifying the ‘spring.profile.active’ property as shown below.

**<profiles>**

**<profile>**

**<id>dev</id>**

**<activation> <activeByDefault>true</activeByDefault> </activation>**

**<properties> <spring.profiles.active>dev</spring.profiles.active> </properties>**

**</profile>**

**<profile>**

**<id>test</id>**

**<properties> <spring.profiles.active>test</spring.profiles.active> </properties>**

**</profile>**

**</profiles>**

### **Priority on Using Multiple Approaches**

Suppose we have used multiple approaches to set active profile in the same project, then Spring will prioritize approaches in below order.

1) By Setting a context parameter in web.xml

2) By Implementing WebApplicationInitializer Interface

3) By Setting JVM System Parameter

4) By Setting active profiles on pom.xml

## How To Use @Profile Annotation?

By using @Profile annotation, we can make a bean belong to a particular profile. The @Profile annotation simply takes the names of one or multiple profiles. In simple words, @Profile annotation associates a bean to a particular profile. For example, let’s consider a scenario: We need a bean that should only be active during development, but not production. It will only be present in the container during development. In production, it won’t be active. Below code demonstrates the concept:

**@Profile("dev")**

**@Component**

**public class MyBeanConfig{...}**

Profiles in Spring Boot also offers to work with YAML(.yml) files. As an alternative to application.properties file, we can create application.yml file as a default profile. Like application.properties file, when application.yml file exists in the project, Spring Container will load it automatically on application startup. Further, similar to .properties file, we can create multiple profiles following the naming convention (application-<environment>.yml).

**application.yml**

**spring:**

**profiles:**

**active: dev**

**application:**

**name: This is default profile**

**app:**

**name: This is the Primary Application yml file**

**SPRING LOGGING**

Logging in Spring refers to the process of recording information about the execution flow, errors, and other events within a Spring application. Proper logging is essential for debugging, monitoring, and auditing purposes. Spring provides support for various logging frameworks, allowing developers to choose the one that best fits their needs. Here's a brief overview of logging in Spring:

1. **Commons Logging**: Historically, Spring Framework used Apache Commons Logging (JCL) as its default logging abstraction. It provides a simple logging API and allows developers to choose the underlying logging implementation at runtime by including the corresponding logging framework (such as Log4J, JDK Logging, or SLF4J) in the classpath.
2. **SLF4J (Simple Logging Facade for Java)**: Spring has gradually shifted towards SLF4J as its preferred logging facade. SLF4J provides a more modern and flexible logging API compared to Commons Logging. It serves as a simple facade for various logging frameworks like Log4j, Logback, JDK Logging, etc., allowing developers to switch between implementations easily.
3. **Log4j**: Log4j is a widely used logging framework in the Java ecosystem. It offers features like logging levels, appenders (for directing log output), and configuration through properties or XML files. Spring applications can integrate with Log4j by including its dependencies and configuring it accordingly.
4. **Logback**: Logback is another logging framework created by the creator of Log4j. It offers similar features to Log4j but with some improvements in performance and functionality. Logback is the successor to Log4j and is gaining popularity. Spring applications can integrate with Logback in a similar way to Log4j.
5. **Java Util Logging (JUL)**: Java's built-in logging framework, also known as JUL, is another option for logging in Spring applications. While it lacks some advanced features found in other logging frameworks, it's simple to use and requires no additional dependencies since it's part of the Java SE platform.

In summary, Spring provides support for various logging frameworks, allowing developers to choose the one that best fits their requirements and preferences. SLF4J is increasingly becoming the preferred choice due to its flexibility and ease of integration with other logging implementations. Developers can configure logging levels, formats, and destinations according to the needs of their applications.

**EXAMPLE CODE:**

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/api")

public class MyController {

private static final Logger logger = LoggerFactory.getLogger(MyController.class);

@GetMapping("/hello")

public String hello() {

logger.debug("Debug message");

logger.info("Info message");

logger.warn("Warning message");

logger.error("Error message");

return "Hello, world!";

}

}

This is a simple Spring Boot REST controller (MyController) that handles HTTP GET requests to the /api/hello endpoint. It logs messages at different levels (debug, info, warn, error) using SLF4J.

Let's break down the code:

1. **Imports**:
   * org.slf4j.Logger: Interface provided by SLF4J for logging.
   * org.slf4j.LoggerFactory: Factory class for obtaining SLF4J logger instances.
   * org.springframework.web.bind.annotation.GetMapping: Annotation for mapping HTTP GET requests onto specific handler methods.
   * org.springframework.web.bind.annotation.RequestMapping: Annotation for mapping web requests onto specific handler classes and/or methods.
   * org.springframework.web.bind.annotation.RestController: Annotation that indicates the controller provides RESTful responses.
2. **Class Declaration**:
   * @RestController: Indicates that this class serves as a REST controller.
   * @RequestMapping("/api"): Specifies that all endpoints in this controller are relative to /api.
3. **Logger Declaration**:
   * private static final Logger logger = LoggerFactory.getLogger(MyController.class); Defines a logger instance named logger for the MyController class. The logger is obtained from LoggerFactory with the class name MyController passed as an argument.
4. **Controller Method**:
   * @GetMapping("/hello"): Maps HTTP GET requests for /api/hello to the hello() method.
   * public String hello() { ... }: This method returns a simple "Hello, world!" message.
   * Inside the method, different log messages are generated using the logger instance:
     + logger.debug("Debug message"): Logs a debug-level message.
     + logger.info("Info message"): Logs an info-level message.
     + logger.warn("Warning message"): Logs a warning-level message.
     + logger.error("Error message"): Logs an error-level message.

Overall, this controller demonstrates how to handle HTTP requests and perform logging in a Spring Boot application using SLF4J.

**Configure logging:**

<!-- logback-spring.xml -->

<configuration>

<appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>

</encoder>

</appender>

<root level="INFO">

<appender-ref ref="CONSOLE"/>

</root>

</configuration>

This logback-spring.xml configuration file is used to configure logging for a Spring Boot application using Logback as the logging framework. Let's break down the configuration:

1. **<configuration>**:
   * This is the root element of the Logback configuration file.
2. **<appender>**:
   * The <appender> element defines an appender named "CONSOLE" of type ch.qos.logback.core.ConsoleAppender. This appender is responsible for printing log messages to the console.
3. **<encoder>**:
   * Inside the <appender>, there's an <encoder> element. The <encoder> specifies how log messages should be formatted before being output by the appender.
   * The %d{HH:mm:ss.SSS} specifies the format for the date and time of the log message.
   * [%thread] specifies the thread that generated the log message.
   * %-5level specifies the log level with a minimum width of 5 characters.
   * %logger{36} specifies the logger name with a maximum width of 36 characters.
   * %msg%n specifies the log message followed by a newline character.
4. **<root>**:
   * The <root> element configures the root logger, which serves as the parent logger for all other loggers in the application.
   * level="INFO" sets the logging level for the root logger to INFO. This means that log messages with a level of INFO or higher (INFO, WARN, ERROR) will be logged.
   * <appender-ref ref="CONSOLE"/> specifies that the "CONSOLE" appender should be used to handle log messages generated by the root logger.

Overall, this Logback configuration sets up logging to output log messages to the console in a specific format (%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n) and configures the root logger to log messages with a level of INFO or higher. You can customize this configuration further based on your application's logging requirements.

**OUTPUT:**

FORMAT:

HH:mm:ss.SSS [thread] LEVEL logger\_name – message

Where:

* HH:mm:ss.SSS represents the timestamp of the log message in hours, minutes, seconds, and milliseconds.
* [thread] represents the thread that generated the log message.
* LEVEL represents the log level (e.g., DEBUG, INFO, WARN, ERROR).
* logger\_name represents the name of the logger (in this case, MyController).
* message represents the actual log message.

OUTPUT:

13:16:50.085 [main] DEBUG MyController - Debug message

13:16:50.087 [main] INFO MyController - Info message

13:16:50.089 [main] WARN MyController - Warning message

13:16:50.090 [main] ERROR MyController - Error message

Each log message corresponds to a different log level (**DEBUG**, **INFO**, **WARN**, **ERROR**) as specified in the controller's **hello()** method. The messages are formatted according to the pattern specified in the Logback configuration file (**logback-spring.xml**), and they're output to the console because of the appender configuration.