### Application Properties and YAML Configuration in Spring Boot

In Spring Boot, configuration settings are often stored in files such as \*\*`application.properties`\*\* or \*\*`application.yml`\*\*. These files allow you to externalize configuration, making it easier to manage and change settings without modifying the source code. The configuration settings can include things like server port, database connection details, logging levels, and custom application properties.

Both \*\*`application.properties`\*\* and \*\*`application.yml`\*\* are supported by Spring Boot, and they serve the same purpose. The choice between them is a matter of preference.

---

### 1. \*\*application.properties\*\* File

The \*\*`application.properties`\*\* file is the simplest format for configuration, where each property is represented as a key-value pair.

#### Example of `application.properties`:

```properties

# Server configuration

server.port=8081

# Database configuration

spring.datasource.url=jdbc:mysql://localhost:3306/mydb

spring.datasource.username=root

spring.datasource.password=pass123

# Hibernate settings

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

# Custom property

app.title=My Spring Boot Application

```

- \*\*Key-Value Format\*\*: Properties are written in the format `key=value`.

- \*\*Overriding Defaults\*\*: You can override default settings (e.g., `server.port` changes the default port from `8080` to `8081`).

- \*\*Custom Properties\*\*: You can define custom properties like `app.title`, which you can access in your Spring code.

---

### 2. \*\*application.yml\*\* File

The \*\*`application.yml`\*\* file offers a more structured, hierarchical way to represent configuration, which can be easier to read and manage for complex configurations.

#### Example of `application.yml`:

```yaml

server:

port: 8081

spring:

datasource:

url: jdbc:mysql://localhost:3306/mydb

username: root

password: pass123

jpa:

show-sql: true

hibernate:

ddl-auto: update

app:

title: My Spring Boot Application

```

- \*\*YAML Format\*\*: YAML uses indentation to represent nested properties, making it more readable for complex configurations.

- \*\*Hierarchical Structure\*\*: Configuration properties are represented in a hierarchy, which can be cleaner and easier to manage compared to the flat structure of `application.properties`.

---

### Accessing Properties in Code

You can access the configuration properties in Spring-managed components using the \*\*`@Value`\*\* annotation or by using \*\*`@ConfigurationProperties`\*\*.

#### Using `@Value` to Inject a Single Property

```java

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Component;

@Component

public class MyService {

@Value("${app.title}")

private String appTitle;

public void printTitle() {

System.out.println("Application Title: " + appTitle);

}

}

```

- \*\*`@Value("${property.name}")`\*\*: Injects the value of a property directly into a field.

- In this example, the `app.title` property is injected into the `appTitle` field.

#### Using `@ConfigurationProperties` for Grouped Properties

For more complex configurations, you can use the `@ConfigurationProperties` annotation, which allows you to map hierarchical properties directly into a Java object.

1. \*\*Define the Configuration Class\*\*:

```java

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.stereotype.Component;

@Component

@ConfigurationProperties(prefix = "app")

public class AppProperties {

private String title;

// Getter and Setter

public String getTitle() {

return title;

}

public void setTitle(String title) {

this.title = title;

}

}

```

2. \*\*Use the Configuration Class\*\*:

```java

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

@Service

public class MyService {

private final AppProperties appProperties;

@Autowired

public MyService(AppProperties appProperties) {

this.appProperties = appProperties;

}

public void printTitle() {

System.out.println("Application Title: " + appProperties.getTitle());

}

}

```

- \*\*`@ConfigurationProperties(prefix = "app")`\*\*: Maps all properties starting with `app.` (e.g., `app.title`) to the `AppProperties` class.

- This approach is more scalable and maintainable when dealing with multiple related properties.

To use `@ConfigurationProperties`, you might need to enable it by adding the `@EnableConfigurationProperties` annotation in your main application class:

```java

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.context.properties.EnableConfigurationProperties;

@SpringBootApplication

@EnableConfigurationProperties(AppProperties.class)

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

```

---

### Profile-Specific Properties

Spring Boot supports \*\*profiles\*\*, which allow you to define environment-specific configurations (e.g., for \*\*development\*\*, \*\*production\*\*, or \*\*testing\*\* environments). You can use the `application.properties` or `application.yml` files to create profile-specific configurations.

#### Defining Profile-Specific Files

You can create profile-specific files using the naming convention `application-{profile}.properties` or `application-{profile}.yml`. For example:

- \*\*application-dev.properties\*\*: Configuration for the development environment.

- \*\*application-prod.properties\*\*: Configuration for the production environment.

#### Example:

\*\*application-dev.properties\*\*:

```properties

server.port=8081

spring.datasource.url=jdbc:mysql://localhost:3306/devdb

```

\*\*application-prod.properties\*\*:

```properties

server.port=8080

spring.datasource.url=jdbc:mysql://localhost:3306/proddb

```

#### Activating a Profile

To activate a specific profile, you can set the \*\*`spring.profiles.active`\*\* property in the \*\*`application.properties`\*\* file or pass it as a command-line argument when running the application:

```properties

spring.profiles.active=dev

```

Or via the command line:

```bash

$ java -jar myapp.jar --spring.profiles.active=prod

```

When the `prod` profile is active, Spring Boot will automatically load \*\*`application-prod.properties`\*\* or \*\*`application-prod.yml`\*\*.

---

### Using Profiles in YAML

In \*\*`application.yml`\*\*, you can define profile-specific configurations inline:

```yaml

server:

port: 8080

spring:

datasource:

url: jdbc:mysql://localhost:3306/defaultdb

---

spring:

profiles: dev

datasource:

url: jdbc:mysql://localhost:3306/devdb

---

spring:

profiles: prod

datasource:

url: jdbc:mysql://localhost:3306/proddb

```

In this example:

- The default configuration will use `defaultdb`.

- When the `dev` profile is active, the configuration will use `devdb`.

- When the `prod` profile is active, the configuration will use `proddb`.

---

### Property Hierarchies and Overriding

Spring Boot allows you to override properties in various ways, including:

1. \*\*Application Properties or YAML\*\*:

- The properties defined in `application.properties` or `application.yml` are loaded first and can be overridden by other sources.

2. \*\*Command-Line Arguments\*\*:

- You can override properties by passing them as command-line arguments:

```bash

$ java -jar myapp.jar --server.port=9090

```

3. \*\*Environment Variables\*\*:

- Environment variables can override properties as well:

```bash

export SERVER\_PORT=9090

```

4. \*\*Profile-Specific Properties\*\*:

- Profile-specific properties (e.g., `application-dev.properties`) override the default configuration.

---

### Benefits of Using Externalized Configuration

1. \*\*Separation of Concerns\*\*:

- Configuration is kept separate from the source code, making it easier to manage and change without needing to recompile the application.

2. \*\*Environment-Specific Configurations\*\*:

- You can define different configurations for different environments (e.g., development, production) without changing the source code.

3. \*\*Dynamic Configuration\*\*:

- Properties can be changed at runtime or before deployment, giving you flexibility to adapt the application’s behavior based on the environment.

4. \*\*Readability\*\*:

- Using \*\*YAML\*\* files can make configurations more readable and maintainable for complex hierarchical configurations.

---

### Conclusion

Spring Boot’s support for \*\*`application.properties`\*\* and \*\*`application.yml`\*\* files provides a flexible way to configure your applications. Both formats allow you to define key-value pairs, manage profiles, and externalize settings, but \*\*YAML\*\* offers a more structured and readable format for complex configurations. Additionally, Spring Boot’s configuration system allows you to override properties in various ways, making it adaptable to different environments and requirements.