Table of Contents

Learning Outcome **3**
Introduction **4**
1. Command Line Properties: **4**
2. Properties File **5**
3. YAML File **6**
4. Externalized Properties **8**
5. @Value Annotation **9**
6. Spring Boot Active Profile **10**
7. @SpringBootApplication **11**
8. Spring Boot Annotations **12**
@Controller 12
@RestController 13
@Service 13
@Component 14
@Autowired 14
@Configuration 15
@Bean 15
@Profile 16
9. Dependency Management **16**
Maven Dependency 17
Dependency Scopes 17
Gradle Dependency 18
Setting up Gradle 18
Creating a Gradle Project 18
Managing Dependencies using Gradle 19
Building and Running the Application 20

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

UNIT 2: Properties, Annotations and Dependencies

Objectives:

In this Unit you will learn to –

- Explain the concept of command line properties in Spring Boot and how to pass command line arguments to a Spring Boot application
- Create and read properties files and YAML files in Spring Boot and understand the differences between them
- Set and use active profiles in Spring Boot and understand the benefits of using them
- Use the @SpringBootApplication annotation to configure a Spring Boot application and understand its different components
- Understand the different Spring Boot annotations (e.g. @Controller, @RestController, @Service, @Component, etc.) and their roles and use cases
- Manage dependencies in a Spring Boot application using Maven and Gradle and understand the benefits of using each of them for dependency management.

Learning Outcome:

Understand Spring Boot's properties, annotations, and dependencies.

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

2. Properties, Annotations and Dependencies

Introduction

Spring Boot is a powerful framework that provides an easy and efficient way to develop enterprise-grade applications. In this unit, we will be discussing properties, annotations, and dependencies in Spring Boot. We will be covering topics such as command-line properties, properties files, YAML files, externalized properties, @Value annotation, Spring Boot active profile, @SpringBootApplication, Spring Boot annotations, and dependency management using Maven and Gradle.

1. Command Line Properties:

Command line properties are a way of providing external configuration to a Spring Boot application. In Spring Boot, command line properties are passed as arguments to the application at runtime. These properties can be used to configure various aspects of the application, such as database connection settings, logging levels, and other application settings.

To pass command line properties to a Spring Boot application, you can use the -- (double hyphen) notation followed by the property name and value. For example, to set the logging level to **DEBUG**, you can pass the following argument to the application:

```
java -jar myapp.jar --logging.level.root=DEBUG
```

In this example, **logging.level.root** is the property name and **DEBUG** is the property value.

Command line properties can also be used to override values that are defined in other configuration files, such as properties files or YAML files. This provides a convenient way to change application behavior without having to modify the application's code.

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

Command line properties are a powerful feature of Spring Boot that allows for easy configuration of applications at runtime.

2. Properties File

In Spring Boot, properties files are used to store configuration properties that can be used throughout the application. These files typically have a ".properties" extension and are stored in the "src/main/resources" directory of the project.

Properties files can contain key-value pairs, where each key represents a property name and each value represents the value for that property. For example, a properties file may contain the following entries:

```
database.url=jdbc:mysql://localhost:3306/mydb
database.username=root
database.password=secret
```

To read properties from a properties file in a Spring Boot application, you can use the **@ConfigurationProperties** annotation. This annotation can be applied to a class or a method, and is used to bind properties to fields or method parameters.

For example, let's say we have a properties file named "application.properties" that contains the following entries:

```
myapp.title=My Spring Boot App
myapp.version=1.0.0
```

We can create a Java class to represent these properties as follows:

```
@ConfigurationProperties(prefix = "myapp")
public class MyAppProperties {
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    private String title;
    private String version;

    // getters and setters
}
```

The **prefix** attribute in the **@ConfigurationProperties** annotation specifies the common prefix for all properties that should be bound to this class.

To use this class in our application, we can simply autowire it into any component or service:

```
@Service
public class MyService {

    private MyAppProperties appProperties;

    public void printProperties() {
        System.out.println("Title: " + appProperties.getTitle());
        System.out.println("Version: " + appProperties.getVersion());
    }
}
```

When we run the application, Spring Boot will automatically read the values from the "application.properties" file and bind them to the **MyAppProperties** class. We can then use the **appProperties** object to access these values in our code.

3. YAML File

YAML (short for "YAML Ain't Markup Language") is a human-readable data serialization format that is commonly used for configuration files. In Spring Boot, YAML files can be used instead of properties files to configure an application.

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

Here are some concepts to understand when working with YAML files in Spring Boot:

- Structure: YAML files use indentation to structure data, instead of curly braces ({}) used in JSON or properties files. This makes YAML files easier to read and write than some other formats.
- Key-value pairs: Just like properties files, YAML files use key-value pairs to store configuration data.
- Comments: YAML files can contain comments, which start with the "#" character and are ignored by the parser.
- Lists: YAML files can represent lists using hyphens (-), similar to how you would represent a list in JSON or an array in other programming languages.
- Nesting: YAML files can nest data structures within each other using indentation, which can help keep related configuration options together.

To create a YAML file in a Spring Boot project, you can create a new file with a .yml or .yaml extension and use the YAML syntax to define your configuration options. For example, here is a simple YAML file that sets the server port and a message to display:

```
server:
  port: 8080
myapp:
  message: Hello, world!
```

In this example, the **server** key contains a nested **port** key that sets the port number for the server to run on. The **myapp** key contains a **message** key that sets a custom message to display on the home page of the application.

To read values from a YAML file in a Spring Boot application, you can use the **@ConfigurationProperties** annotation on a class that has fields that match the keys in the YAML file. For example, to read the values from the YAML file shown above, you could create a class like this:

```
@ConfigurationProperties("myapp")
public class MyAppProperties {
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

In this example, the **@ConfigurationProperties("myapp")** annotation tells Spring Boot to map the **myapp** keys from the YAML file to the fields in the **MyAppProperties** class. The **message** field corresponds to the **myapp.message** key in the YAML file.

4. Externalized Properties

Externalized properties in Spring Boot refer to the practice of configuring the application properties outside of the packaged application, so that they can be easily modified without the need to repackage and redeploy the application. This is useful for configuring properties that may vary between different environments, such as development, testing, and production environments.

To externalize properties in Spring Boot, we can use either properties files or YAML files. By default, Spring Boot looks for an application.properties file in the classpath and loads the properties from it. Alternatively, we can use an application.yml file to define the properties in YAML format.

To externalize the properties in Spring Boot, we can use the following methods:

- Using command line arguments: We can pass the properties as command line arguments when starting the Spring Boot application, using the --name=value format.

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

- Using environment variables: We can set the properties as environment variables, and Spring Boot will automatically read them when the application starts.
- Using system properties: We can also set the properties as system properties by using the -D option when starting the application.

The benefits of externalizing properties in Spring Boot include:

- Flexibility: It allows us to easily change the configuration properties for different environments without modifying the application code.
- Security: Sensitive information, such as database passwords, can be externalized and stored securely, rather than hard-coded in the application code.
- Simplified deployment: Externalizing properties makes it easier to deploy the application to different environments, as it eliminates the need to modify the application code for each environment.

Overall, externalizing properties in Spring Boot is a best practice for managing configuration properties in an application, and it can greatly simplify the deployment process.

5. @Value Annotation

The @Value annotation is a powerful feature in Spring Boot that allows developers to inject values into Spring components, such as classes and methods. This annotation can be used to inject values from various sources, including properties files, command line arguments, and system properties.

To use the @Value annotation, simply add it to the field or method where you want to inject the value, and provide the key or expression for the value you want to inject. For example:

```
@Component

public class MyComponent {
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    @Value("${my.property}")
    private String myProperty;

    public void doSomething() {
        // use the value of myProperty here
    }
}
```

In this example, the @Value annotation is used to inject the value of the my.property key from a properties file into the myProperty field of the MyComponent class.

One of the limitations of the @Value annotation is that it can only be used to inject simple values, such as strings, integers, and booleans. If you need to inject more complex values, such as lists or maps, you may need to use a different approach, such as the @ConfigurationProperties annotation.

6. Spring Boot Active Profile

Active profiles are an essential feature of Spring Boot that allows developers to configure different aspects of the application based on the current environment or context. In other words, Spring Boot applications can be designed to run in multiple environments, such as development, testing, staging, and production, with different configurations, such as database connection settings, logging levels, and security settings.

The active profile concept allows developers to define different sets of configurations for each environment and select the appropriate configuration based on the environment in which the application is running. By default, Spring Boot applications have a default profile, but developers can create and activate other profiles as well.

To set an active profile in Spring Boot, developers can use the spring.profiles.active property. They can set this property in various ways, such as through a command-line argument, environment variable, or configuration file.

Using active profiles in Spring Boot offers many benefits, such as:

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

1. **Separation of concerns:** Active profiles help separate configuration concerns between different environments, reducing the risk of issues occurring during deployment or testing.
2. **Simplification of configuration:** Active profiles can help simplify configuration by allowing developers to define different sets of configuration for each environment instead of a single monolithic configuration.
3. **Improved security:** By defining separate configurations for different environments, developers can ensure that sensitive data and credentials are not exposed to unauthorized users.
4. **Increased flexibility:** Active profiles provide greater flexibility to developers, allowing them to deploy and test the application in different environments without the need for significant code changes.

Understanding and using active profiles is a critical aspect of building and deploying Spring Boot applications in various environments.

7. @SpringBootApplication

The @SpringBootApplication annotation is a meta-annotation that is used to configure a Spring Boot application. It is a combination of three annotations: @Configuration, @EnableAutoConfiguration, and @ComponentScan.

- **@Configuration:** This annotation indicates that the class is a configuration class that defines beans that should be added to the Spring application context.
- **@EnableAutoConfiguration:** This annotation enables Spring Boot's auto-configuration feature, which automatically configures the application based on the dependencies present in the classpath.
- **@ComponentScan:** This annotation scans the application for Spring components and configuration classes.

By using the **@SpringBootApplication** annotation, you can quickly configure a Spring Boot application with sensible defaults and start building your application.

Here's an example:

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
@SpringBootApplication
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

In this example, we are marking the MyApplication class as the main class of our Spring Boot application. The SpringApplication.run method is used to start the application.

8. Spring Boot Annotations

Annotations are an important part of Spring Boot development as they provide a way to easily configure and manage components within your application. In this study material, we will go over some of the most common Spring Boot annotations and their use cases.

@Controller

The @Controller annotation is used to indicate that a class is a controller in the Spring MVC framework. Controllers are responsible for processing incoming requests and returning a response to the client. This annotation can be used in conjunction with @RequestMapping to map incoming requests to specific methods in the controller.

Example:

```
@Controller
public class MyController {
    @RequestMapping("/hello"
    public String helloWorld() {
        return "Hello World!";
    }
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
}
```

@RestController

The @RestController annotation is a combination of @Controller and @ResponseBody. It is used to indicate that a class is a controller and that the return value of its methods should be serialized directly to the HTTP response body. This is useful when building RESTful APIs.

Example:

```
@RestController
public class MyRestController {
    @RequestMapping("/hello"
    public String helloWorld() {
        return "Hello World!";
    }
}
```

@Service

The @Service annotation is used to indicate that a class is a service in the Spring framework. Services are used to encapsulate business logic and are typically used in conjunction with repositories to perform CRUD operations.

Example:

```
@Service
public class MyService {
    @Autowired
    private MyRepository myRepository;

    public List<MyObject> findAll() {
        return myRepository.findAll();
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    }
}
```

@Component

The @Component annotation is used to indicate that a class is a generic Spring-managed component. This annotation can be used in conjunction with @Autowired to inject dependencies into your component.

Example:

```
@Component
public class MyComponent {
    @Autowired
    private MyService myService;

    public void doSomething() {
        List<MyObject> objects = myService.findAll();
        // do something with objects
    }
}
```

@Autowired

The @Autowired annotation is used to inject dependencies into your Spring-managed components. This annotation can be used in conjunction with @Component, @Service, and @Repository.

Example:

```
@Component
public class MyComponent {
    @Autowired
    private MyService myService;
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    public void doSomething() {
        List<MyObject> objects = myService.findAll();
        // do something with objects
    }
}
```

@Configuration

The @Configuration annotation is used to indicate that a class contains Spring configuration information. This annotation is typically used in conjunction with @Bean to define beans that can be injected into other components.

Example:

```
@Configuration
public class MyConfiguration {
    @Bean
    public MyObject myObject() {
        return new MyObject();
    }
}
```

@Bean

The @Bean annotation is used to indicate that a method produces a bean to be managed by the Spring container. This annotation is typically used in conjunction with @Configuration.

Example:

```
@Configuration
public class MyConfiguration {
    @Bean
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    public MyObject myObject() {
        return new MyObject();
    }
}
```

@Profile

The @Profile annotation is used to define a specific profile that a component belongs to. Profiles allow you to define different configurations for different environments (e.g. development, production, etc.).

Example:

```
@Component
@Profile("dev")
public class MyComponent {
    // this component will only be used in the "dev" profile
}
```

9. Dependency Management

In software development, dependencies refer to external libraries or modules that an application requires to function correctly. Dependency management is the process of handling and organizing these external libraries or modules. It ensures that the application has the necessary dependencies and versions to function correctly.

Dependency management is an essential aspect of software development. It involves managing the libraries or modules used by an application to function. In Spring Boot, dependency management is handled by build tools like Maven and Gradle. Maven is a popular build tool for managing dependencies in Spring Boot applications.

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

Maven Dependency

Maven is a popular build tool for managing dependencies in Spring Boot applications. It is an open-source project management and comprehension tool that is widely used in Java-based projects. Maven helps to automate the building process, manage dependencies, and generate documentation for the project. Maven uses an XML file called the "pom.xml" to configure and manage dependencies.

Adding Dependencies using Maven:

In Maven, dependencies can be added to a project by adding the dependency tag to the "pom.xml" file. The dependency tag contains the group id, artifact id, and version number of the dependency. For example, to add the Spring Web dependency to a Spring Boot application, the following dependency tag can be added to the "pom.xml" file:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.6.3</version>
</dependency>
```

In the above example, the dependency tag contains the group id "org.springframework.boot", the artifact id "spring-boot-starter-web", and the version "2.6.3".

Dependency Scopes

In Maven, dependencies can have different scopes. The scope determines where the dependency is available in the project. The following are the different scopes in Maven:

- Compile: This scope is the default scope. The dependency is available in all

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

phases of the project.

- Provided: The dependency is provided by the container (e.g., Tomcat) during runtime, and it is not included in the packaged application.
- Runtime: The dependency is required during runtime, but it is not required for compilation.
- Test: The dependency is only required for testing purposes and is not included in the packaged application.
- System: The dependency is available in the system path and is not included in the repository.

Gradle Dependency

Gradle is a popular build automation tool used for managing dependencies in Spring Boot applications. In this section, we will discuss how to manage dependencies in a Spring Boot application using Gradle.

Setting up Gradle

Before we can use Gradle to manage dependencies, we need to install it on our system. Gradle can be downloaded from the official Gradle website. Once installed, we can verify the installation by running the following command in the terminal:

```
gradle -v
```

This command will display the version of Gradle installed on our system.

Creating a Gradle Project

We can create a new Gradle project using the Spring Initializr. The Spring Initializr is a web-based tool that generates a project with the required dependencies and configuration files. To create a new Gradle project using the Spring Initializr, we need to follow these steps:

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

1. Open the Spring Initializr website.
2. Select Gradle as the build system.
3. Select the required dependencies for the project.
4. Click on the Generate button to download the project as a zip file.
5. Extract the contents of the zip file to a new directory.

Managing Dependencies using Gradle

Gradle uses the build.gradle file to manage the dependencies required by a Spring Boot application. The build.gradle file contains the project configuration and the list of dependencies required by the application.

Here is an example of a build.gradle file that includes the required dependencies for a Spring Boot application:

```
plugins {
    id 'org.springframework.boot' version '2.6.2'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
    mavenCentral()
}

dependencies {
```

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.

```
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
    useJUnitPlatform()
}
```

In this file, we can see that the required dependencies for the Spring Boot application are listed under the dependencies section. We can add or remove dependencies as required by our application.

Building and Running the Application

To build and run the Spring Boot application using Gradle, we need to run the following commands in the terminal:

```
./gradlew build
./gradlew bootRun
```

The first command builds the application, and the second command starts the application.

Conclusion

In this unit, we covered properties, annotations, and dependencies in Spring Boot. We discussed command-line properties, properties files, YAML files, externalized properties, @Value annotation, Spring Boot active profile, @SpringBootApplication, Spring Boot annotations, and dependency management using Maven and Gradle. By understanding these concepts, you can build powerful and efficient Spring Boot applications that are easily configurable and maintainable.

Proprietary content. All rights reserved. Unauthorized use or distribution prohibited.