Spring and Annotations

Introduction

Annotations in Spring are metadata markers that provide additional information or instructions to the Spring framework. They are applied to classes, methods, fields, or method parameters and help configure various aspects of the application's behaviour, such as dependency injection, aspect-oriented programming, transaction management, and more.

Annotations in Spring make it possible to declaratively define the application's configuration and behaviour, reducing the need for explicit XML configuration files. Let's explore the different annotations in Spring:

A. @Component Annotation

The @Component annotation is a feature provided by several Java frameworks, such as Spring Framework, that allows you to declare a class as a component or bean. This annotation marks a class as a candidate for auto-detection and auto-configuration when the application context is created.

When annotating a class with @Component, it becomes a Spring bean registered in the application context. The Spring container then manages the lifecycle and dependencies of these beans.

Here are some key points about the @Component annotation:

- **Auto-detection**: When using component scanning in Spring, the framework automatically detects classes annotated with @Component and registers them as beans in the application context.

- **Dependency injection**: Components annotated with @Component can be injected into other classes using dependency injection mechanisms provided by the framework, such as @Autowired.

- **Customization**: The @Component annotation is a generic annotation used as a base annotation for more specific stereotypes like @Service, @Repository, or @Controller. These specialised annotations help semantically categorise the components and provide additional functionality or behaviour.

- **Configuration**: By default, Spring treats classes annotated with @Component as singleton beans. However, you can customise the bean's scope by using other annotations like @Scope or @RequestScope to change the instantiation and lifecycle behaviour.

- **Component scanning**: To enable component scanning and make Spring automatically detect and register the @Component annotated classes, you need to configure it in the Spring configuration file or use the appropriate annotations, such as @ComponentScan, on a configuration class.

- **XML-based configuration**: If you use XML-based configuration instead of annotations, you can still define a bean using the <bean> element and specify the class as a component using the class attribute.

For Example,

```
@Component("javaInstructor")
public class JavaInstructor implements Instructor {

      String name;
      String age;

      @Override
      public void setInstructorDetails(String name, String age) {
            this.name = name;
            this.age = age;
      }

      @Override
      public String takeClass() {
            return "Hi my name is "+this.name+" and I will be your
java" + "instructor";
      }
}
```

In the example above, The @Component annotation is used to mark the JavaInstructor class as a Spring component. The optional parameter "javaInstructor" specifies the name or identifier for the bean in the Spring container. If no name is specified by default, the bean name is derived from the class name with the initial letter converted to lowercase.

Overall, the @Component annotation provides a convenient way to mark classes as Spring components or beans, enabling automatic dependency injection and lifecycle management by the Spring container.

B. @Autowire Annotation

The @Autowired annotation is a feature of the Spring Framework in Java that allows automatic dependency injection. When you annotate a field, constructor, or setter method with @Autowired, Spring will automatically resolve and inject the appropriate dependency at runtime.

Here are some key points about the @Autowire annotation:

- **Dependency injection**: You can annotate a field directly with @Autowired to indicate that Spring should inject a dependency into that field.

- **Dependency resolution**: When you use @Autowired, Spring will analyse the type of dependency and search for a matching bean in the application context. It will be injected if there's exactly one bean of that type. If multiple beans are of the same type, you can use additional annotations like @Qualifier to specify the bean to be injected.

- **Optional dependencies**: By default, @Autowired requires a dependency in the application context. However, you can make a dependency optional by using the required attribute of @Autowired and setting it to false. The dependency will be set to null if a matching bean is not found.

- **Qualifiers**: In scenarios with multiple beans of the same type, you can use the @Qualifier annotation in conjunction with @Autowired to specify the exact bean to be injected. The @Qualifier annotation can be applied to a field, constructor parameter, or setter method parameter.

- **Primary bean**: If multiple beans of the same type are available and you want to specify a primary bean that should be injected by default, you can use the @Primary annotation on that bean. When using @Autowired, the primary bean will be selected if no specific qualifier or name is provided.

- Here is an example of DI using @Autowired annotation,

```
@Component
public class PaidCourse implements Course{

      String courseName;

      @Autowired
      UserList userList;

      @Override
      public void setCourseDetails(String courseName) {
            this.courseName = courseName;
```

```
        }

        @Override
        public UserList getUserList() {
                return this.userList;
        }
}
```

Overall, the @Autowired annotation simplifies the process of injecting dependencies by allowing Spring to wire beans together automatically. It promotes loose coupling and improves the maintainability and flexibility of your code.

C. @Qualifier Annotation

The @Qualifier annotation is used with the @Autowired annotation in the Spring Framework to specify the exact bean to inject when multiple beans of the same type are available.

Here are some key points about the @Qualifier annotation:

- **Multiple bean candidates**: In some cases, there may be multiple beans of the same type may be available in the application context. For example, if you have multiple implementations of an interface, Spring may not be able to determine which bean to inject automatically.

- **Qualifying a specific bean**: Using the @Qualifier annotation, you can specify the name or identifier of the bean that will be injected. The annotation allows you to disambiguate between multiple beans and provide clarity to Spring about which bean should be used.

- **Bean qualifier values**: The @Qualifier annotation can be applied to a field, constructor, or setter method parameter. You can provide a qualifier value as an argument to the @Qualifier annotation to indicate the specific bean to be injected. For example,

```
@Component
public class PaidCourse implements Course{

        String courseName;

        @Autowired
        @Qualifier("javaInstructor")
        Instructor courseInstructor;

        @Autowired
```

```
      UserList userList;

      // additional methods
}
```

In the example above, the @Qualifier("javaInstructor") annotation is used to specify the bean with the name *"javaInstructor"* to be injected. It helps resolve any ambiguity when multiple beans of the Instructor type are available.

In the example below, an interface Salary with two implementations FixedSalary and, VariableSalary, and Salary, is injected in the Employee class. We only have to inject FixedSalary. We use @Qualifier("fixedSalary"), i.e. qualifier annotation with the bean name to inject in string format.

```
// Salary interface
public interface Salary {
    void getSalary();
}

// FixedSalary class
public class FixedSalary implements Salary {
    //...
}

// VariableSalary class
public class VariableSalary implements Salary {
    //...
}

// Employee class
public class Employee {
    private String name;

    @Autowired
    @Qualifier("fixedSalary")
    Salary salary;
}
```

The @Qualifier annotation is a powerful tool for fine-grained control over dependency injection in Spring when dealing with multiple beans of the same type. It lets you specify the bean to inject, promoting clarity and resolving ambiguities in the dependency resolution process.

D. Dependency injection in the Main class,

The main method in the main class of a Spring Boot application, is static. Therefore, that dependency cannot be used in the main method. To use that dependency, we use the CommandLineRunner interface. The CommandLineRunner interface is commonly used in Spring Boot to execute code after the application context is initialised and the main method is invoked.

After the application context is initialised and the main method is invoked, the run method of CommandLineRunner will be automatically executed by Spring Boot. This provides an opportunity to perform additional actions or invoke methods that depend on the injected dependencies.

Let's take an example of a vaccine application where we want to inject a User dependency in our application using annotation. Here we are going to inject the User directly into our application rather than fetching from context.

```
@SpringBootApplication
public class VaccineWithAnnotationApplication implements CommandLineRunner {

    @Autowired
    @Qualifier("Father")
    User user;

    public static void main(String[] args) {
        // application logic
    }
    @Override
    public void run(String... args) throws Exception {
        System.out.println("Please choose your vaccine type:");
        System.out.println("1. Covaxin\n2. Covishield\n3Moderna");
        int vaccineChoice = scanner.nextInt();
        scanner.nextLine();
        String myVaccine;
            switch (vaccineChoice) {
                case 1 -> myVaccine = "Covaxin";
                case 2 -> myVaccine = "Covishield";
                case 3 -> myVaccine = "Moderna";
                    default -> {
                        System.out.println("Invalid choice.");
                        return;
                    }
                }

    }
    this.user.setVaccine(myVaccine);
}
```

This is useful when we want to insert specific data in the database when the application is ready, integrate some external system or schedule some tasks at application startup etc.

E. @Scope Annotation

The @Scope annotation is used in the Spring Framework to specify the scope or lifecycle of a bean. It allows you to define how bean instances should be created, managed, and destroyed within the Spring container.

Here are some key points about the @Scope annotation:

- **Singleton scope (default):** By default, Spring beans have a singleton scope, meaning that only one instance of the bean is created and shared across the entire application context. The @Scope annotation can be used to explicitly define the singleton scope, although it is not necessary as it is the default behaviour. For example:

- **Prototype scope:** The @Scope annotation can specify a bean's prototype scope. In this case, a new instance of the bean is created whenever it is requested from the container. Each instance is independent of others and has its state. For example,

```
@Component
@Scope("prototype")
public class SimpleUser implements User {

      String name;
      String age;
      String location;
      String collegeName;

      @Override
      public String getUserDetails() {
            return this.name + " age:" + this.age;
      }

      @Override
      public void setUserDetails(String name, String age, String
location, String collegeName) {
            this.name = name;
            this.age = age;
            this.collegeName = collegeName;
            this.location = location;
      }
}
```

In the example above, @Scope("prototype") is used to specify the scope of bean of class SimpleUser should be a prototype.

The @Scope annotation allows you to control the lifecycle and availability of beans within the Spring container. By specifying different scopes, you can define whether beans should be singletons, prototypes, or tied to specific contexts like requests or sessions. It provides flexibility in managing the lifecycle and state of beans based on your application's requirements.

F. @PostConstruct Annotation

The @PostConstruct annotation is used in the Java EE and Spring frameworks to annotate a method that needs to be executed after dependency injection and bean initialisation. It is often a callback method for a bean's initialisation tasks.

Here's a description of the @PostConstruct annotation and its common uses:

- **Initialization callback:** By annotating a method with @PostConstruct, you indicate to the framework that this method should be invoked after the bean has been constructed, its dependencies have been injected, and before it is put into service.

- **Method requirements:** The method annotated with @PostConstruct should not have any arguments and must be non-private. It can have any return type, although void is commonly used since the method is typically used for initialisation rather than returning a value.

- **Order of execution:** When a bean is instantiated and all its dependencies are injected, the method annotated with @PostConstruct is automatically called by the container or framework. This ensures that the initialisation tasks defined in the method are executed at the appropriate time.

- **Initialization tasks:** The method annotated with @PostConstruct can be used to perform various initialisation tasks, such as setting up resources, establishing connections, initialising variables, or any other operations required to prepare the bean for use.

- **Multiple methods:** Annotating multiple methods with @PostConstruct within a single bean is possible. In such cases, the order of execution is not guaranteed. If you need specific ordering, you can use the @Order annotation or implement the org.springframework.core.Ordered interface.

  For example,

```
@Component("javaInstructor")
public class JavaInstructor implements Instructor {

      String name;
      String age;

      @PostConstruct
      public void init() {
            System.out.println("Java instructor bean created");
        }
}
```

The @PostConstruct annotation provides a convenient way to define initialisation behaviour for beans in Java EE and Spring applications. It allows you to centralise and manage the initialisation tasks of the bean, ensuring they are executed appropriately in the bean's lifecycle.

G. @Predestroy Annotation

The @PreDestroy annotation is used in Java EE and Spring applications to annotate a method that should be called before a managed bean is destroyed or removed from service. It is commonly used as a callback method to perform cleanup or release resources held by a bean.

Here's a description of the @PreDestroy annotation and its common uses:

- Destruction callback: By annotating a method with @PreDestroy, you indicate to the container or framework that this method should be invoked before the bean is destroyed or removed from service. It lets you perform necessary cleanup tasks or release the bean's resources.

- Method requirements: The method annotated with @PreDestroy should not have any arguments and must be non-private. It can have any return type, although void is commonly used since the method is typically used for cleanup rather than returning a value.

- Order of execution: When a managed bean is being destroyed, the method annotated with @PreDestroy is automatically called by the container or framework. This ensures that the cleanup tasks defined in the method are executed before the bean is removed from the service.

- Resource cleanup: The @PreDestroy method is often used to release resources such as open files, database connections, network connections, or other system resources held by the bean. It allows you to gracefully shut down or clean up resources before the bean is destroyed.

- Bean lifecycle management: The @PreDestroy annotation is part of the broader bean lifecycle management mechanism provided by Java EE and Spring. It complements the @PostConstruct annotation, which is used for initialisation tasks. Together, these annotations provide a way to perform actions at specific points in the lifecycle of a managed bean.

For example,

```
@Component("javaInstructor")
public class JavaInstructor implements Instructor {

      String name;
      String age;

      @PreDestroy
      public void cleanup() {
            System.out.println("Java instructor bean about to be
destroyed");
      }
}
```

The @PreDestroy annotation provides a convenient way to define cleanup behaviour for managed beans in Java EE and Spring applications. It allows you to centralise and manage resource cleanup for the bean, ensuring it is performed before it is destroyed or removed from service.

H. @SpringBootApplication Annotation

The @SpringBootApplication annotation is a convenience annotation provided by the Spring Boot framework. It is used to annotate the main class of a Spring Boot application to enable various auto-configuration and component scanning features.

Here are some key points about the @SpringBootApplication annotation:

- **Combination of annotations:** The @SpringBootApplication annotation combines three commonly used annotations in Spring Boot applications: @Configuration, @EnableAutoConfiguration, and @ComponentScan. By using @SpringBootApplication, you can apply these three annotations with a single annotation.

- **@Configuration:** The @Configuration annotation indicates that the class should be treated as a source of bean definitions. It lets you define beans and their configurations using methods annotated with @Bean.

- **@EnableAutoConfiguration:** The @EnableAutoConfiguration annotation triggers Spring Boot's auto-configuration mechanism. It automatically configures the application based on the classpath's dependencies and the content of the classpath. Spring Boot analyses the classpath, identifies the required libraries, and automatically configures various components, such as database connectivity, web servers, and messaging systems.

- **@ComponentScan:** The @ComponentScan annotation enables component scanning in the specified base packages. It instructs Spring to scan and detect components, such as @Component, @Controller, @Service, and @Repository, to register them as beans in the application context.

- **Main application class:** The @SpringBootApplication annotation is typically used to annotate the main class of a Spring Boot application. This class serves as the entry point for the application and contains the main() method. By annotating this class, you enable Spring Boot features and ensure the application's necessary configurations and components are set up.

  For example,

```
@SpringBootApplication
public class CodingNinjasAppApplication {

      public static void main(String[] args) {

            System.out.println("Welcome   to   the   coding   ninjas
application");
      ApplicationContext                 context                 =
SpringApplication.run(CodingNinjasAppApplication.class);

}
```

The @SpringBootApplication annotation simplifies the configuration and setup of a Spring Boot application by combining multiple annotations into one. It enables auto-configuration and component scanning, making it easier to develop production-ready Spring Boot applications with minimal configuration.

I. YAML

YAML (*Yet Another Markup Language*) is a human-readable data serialisation format. It is commonly used for configuration files, data exchange, and defining structured data concisely and easily readable. YAML files use indentation and simple syntax to represent data structures, making the language more popular among developers and system administrators.

Here are some key features and characteristics of YAML:

- **Structure:** YAML uses indentation and nesting to represent data structures. It employs a hierarchical structure with indentation levels to define relationships between elements.

- **Readability:** YAML is designed to be human-friendly and easy to read. It uses plain text and avoids excessive punctuation, making it more natural and intuitive for users.

- **Data Types:** YAML supports various data types, including scalars (strings, numbers, booleans), sequences (arrays or lists), and mappings (key-value pairs or dictionaries). It provides a flexible and expressive way to represent structured data.

- **Comments:** YAML allows adding comments to the files using the '#' symbol. Comments provide additional explanations or annotations to the configuration and are ignored during parsing.

- **Inclusion and References:** YAML supports including and referencing other YAML files or parts of the same file. This feature enables code reuse and modularisation and simplifies complex configurations.

- **Serialisation and Deserialization:** YAML can be easily converted to and from programming language objects through serialisation and deserialisation processes. This allows developers to work with YAML data in their preferred programming language.

- **Portability:** YAML is language-agnostic and widely supported by various programming languages and frameworks. It provides a consistent format for data exchange and configuration format across different platforms and systems.

```yaml
# Example YAML Configuration File

server:
  port: 8080
  host: localhost

database:
  driver: com.mysql.jdbc.Driver
  url: jdbc:mysql://localhost:3306/mydatabase
  username: myusername
  password: mypassword

logging:
  level: INFO
  file: logs/application.log
```

In the above example, the YAML file defines a server configuration with the port and host, a database configuration with driver, URL, username, and password, and a logging configuration with log level and log file path.

YAML's simplicity, readability, and flexibility make it popular for configuration files, data representation, and interchanging data between different systems.

J. Some extra annotations

- **@Service:** The @Service annotation marks a class as a service component. It is typically used to encapsulate business logic and perform operations related to the application's domain. Services are typically injected into other components, such as controllers, using the @Autowired annotation.

- **@Controller:** The @Controller annotation marks a class as a controller component in the MVC pattern. It handles incoming HTTP requests, performs required processing, and returns the appropriate response. Controllers typically contain methods annotated with @RequestMapping or other specialised request mapping annotations.

- **@Repository:** The @Repository annotation marks a class as a repository component. It is typically used for database access, data retrieval, and storage operations. Repositories often work with entities and provide an abstraction layer for data persistence. Repositories are typically injected into other components using the @Autowired annotation.

- **@Bean:** The @Bean annotation declares a method as a bean definition method in a configuration class. It tells Spring to manage the object's lifecycle returned by the annotated method and make it available for dependency injection. Beans can be configured with additional annotations such as @Scope or @Qualifier for customisation.

- **@RequestMapping:** The @RequestMapping annotation maps an HTTP request to a method in a controller class. It defines the URL pattern and the HTTP method(s) the method should handle. Additional annotations like @GetMapping, @PostMapping, etc., can be used for specific HTTP methods.

Conclusion

In conclusion, annotations play a crucial role in the Spring Boot framework, providing a declarative way to configure and manage various aspects of the application. Here's a summary of the key points regarding annotations in Spring Boot:

- Configuration Annotations: Spring Boot provides a range of annotations such as @EnableAutoConfiguration, @ComponentScan, and @ConfigurationProperties that simplify the configuration process and enable auto-configuration based on classpath scanning.

- Dependency Injection Annotations: Annotations like @Autowired and @Qualifier are used for dependency injection, allowing the framework to wire dependencies automatically and resolve ambiguities when multiple beans of the same type are available.

- Stereotype Annotations: Stereotype annotations such as @Component, @Service, @Repository, and @Controller are used to mark classes as Spring-managed components. These annotations help with component scanning and auto-detection of beans.

- Lifecycle Annotations: Annotations like @PostConstruct and @PreDestroy define lifecycle callback methods in managed beans. They allow you to execute specific tasks after bean initialisation or before destruction.

By utilising these annotations, developers can significantly reduce the amount of boilerplate code and configuration, resulting in more concise and readable code. Spring Boot's annotation-driven approach promotes convention over configuration and simplifies the development of robust and scalable applications.

Instructor Codes

- Coding Ninjas Application

References

1. Official Documentation
2. @Component Annotation
3. @Scope Annotation
4. Request and Session Scopes
5. @Qualifier Annotation
6. Spring lifecycle Annotation
7. @SpringBootApplication
8. YAML
9. More Annotations
10. @Data, @AllArgsConstructor ,@NoArgsConstructor
11. Create your custom annotations