S.N.R.Likhitha(AF0312909)

1. What is Dependency Injection (DI)?

Dependency Injection:

Dependency Injection (DI) is a software design pattern and a technique used in object-oriented programming to manage the dependencies between different components or classes of an application. It is a fundamental concept in the field of software engineering and is often associated with the principles of Inversion of Control (IoC).

In DI, the main idea is to decouple the components or classes that depend on each other by providing the dependent component with its required dependencies rather than allowing it to create or manage them internally. This approach has several benefits, including:

Decoupling: DI helps decouple the components of an application, reducing the tight coupling between them. This makes the codebase more modular and easier to maintain, test, and extend.

Testability: DI makes it easier to write unit tests for individual components because you can replace real dependencies with mock or stub objects during testing. This allows you to isolate the component being tested and verify its behavior without relying on external systems.

Flexibility: By injecting dependencies from the outside, you can change the behavior of a component by swapping its dependencies without modifying its code. This makes the application more flexible and adaptable to changes.

Reusability: Components with well-defined dependencies can be reused in different parts of the application or even in different projects, promoting code reuse.

DI can be implemented in various ways, including:

- **Constructor Injection:** Dependencies are injected through a class's constructor. This is one of the most common forms of DI and is often used in statically typed languages like Java and C#.
- **Setter Injection**: Dependencies are injected through setter methods or properties of a class. This allows for more flexibility because dependencies can be changed after the object is created.
- **Interface Injection:** Interfaces or abstract classes define methods for setting dependencies, and concrete classes implement these methods to inject dependencies. This is less common than constructor and setter injection.
- **Service Locator:** A central service locator class manages and provides access to dependencies. Components request their dependencies from the service locator when needed.

Dependency Injection is a powerful concept in software design that promotes maintainability, testability, and flexibility in applications by managing dependencies in a controlled and modular manner. It is widely used in modern software development, particularly in

frameworks like Spring (for Java) and Angular (for TypeScript/JavaScript), which provide robust DI mechanisms.

2.What is the purpose of the @Autowired annotation in Spring Boot?

In Spring Boot and the broader Spring Framework, the @Autowired annotation is used to automatically inject dependencies into Spring-managed beans, such as components, services, controllers, or repositories. It's a key part of Spring's dependency injection mechanism, which allows you to declare and manage dependencies between different parts of your application. The purpose of @Autowired is to simplify the process of wiring together components by eliminating the need for manual dependency injection.

The @Autowired annotation can be used to autowire bean on the setter method just like @Required annotation, constructor, a property or methods with arbitrary names and/or multiple arguments. Spring @Autowired annotation is used for automatic dependency injection. Spring framework is built on dependency injection and we inject the class dependencies through spring bean configuration file.

3. Explain the concept of Qualifiers in Spring Boot.

In Spring Boot, qualifiers are used to disambiguate the injection of dependencies when multiple beans of the same type are present in the application context. Qualifiers help the Spring framework to determine which specific bean should be injected when autowiring by type is ambiguous. Qualifiers are typically applied to fields, constructor parameters, or method parameters when using the @Autowired annotation.

Here's how qualifiers work in Spring Boot:

Multiple Beans of the Same Type: When you have multiple beans of the same type in your Spring Boot application context, Spring may not know which one to inject when you use @Autowired without qualifiers. This situation commonly occurs when you have multiple implementations of the same interface or multiple beans of the same class.

Using Qualifiers: To specify which bean should be injected, you can use the @Qualifier annotation along with @Autowired. The @Qualifier annotation allows you to specify the unique name or value of the bean you want to inject.

Example:

```
@Autowired
@Qualifier("myBean")
private MyInterface myBean;
```

In this example, @Qualifier("myBean") indicates that you want to inject the bean with the name "myBean" of the specified type (e.g., MyInterface).

Bean Names as Qualifiers: By default, the bean name is used as the qualifier when you don't specify a custom value in the @Qualifier annotation.

For example:

```
@Autowired
private MyInterface myBean;
```

In this case, Spring will try to find a bean of type MyInterface with the name "myBean" in the application context.

Custom Qualifier Values: You can also create custom qualifier annotations to make your code more readable and maintainable. For example, you can define a custom qualifier annotation.

Example:

```
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface MyCustomQualifier {
}
```

Qualifiers are especially useful when you have complex Spring Boot applications with multiple beans and need to specify precisely which bean should be injected in a particular scenario. They help resolve ambiguity in autowiring and ensure that the correct dependency is injected based on your criteria.

4. What are the different ways to perform Dependency Injection in Spring Boot?

In Spring Boot, dependency injection (DI) is a fundamental concept that allows you to manage and inject dependencies into your application components. Dependency injection helps achieve loose coupling between components and promotes easier testing, scalability, and maintainability of your code. Spring Boot provides various ways to perform dependency injection:

Constructor Injection:

Constructor injection is considered the recommended and most common way to perform DI in Spring Boot.

You can annotate a constructor with @Autowired to indicate that Spring should inject the required dependencies into the constructor parameters.

Here's an example:

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

    @Autowired
    public MyService(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

Setter Injection:

Setter injection involves using setter methods to inject dependencies.

Annotate the setter methods with @Autowired to indicate that Spring should inject the required dependencies.

While it's less common than constructor injection, setter injection can be useful in certain scenarios, such as optional dependencies.

Here's an example:

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

    @Autowired
    public void setMyRepository(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

Field Injection:

Field injection involves annotating class fields directly with @Autowired.

While it's convenient, field injection is generally not recommended because it can lead to tight coupling and is less testable than constructor or setter injection.

Here's an example:

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

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

Method Injection:

Method injection is less common but can be useful when you need to inject dependencies into specific methods.

Annotate a method with @Autowired, and Spring will inject the required dependencies into the method parameters.

Here's an example:

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

    @Autowired
    public void initialize(MyRepository myRepository) {
        this.myRepository = myRepository;
    }
}
```

Qualifiers:

Use the @Qualifier annotation in combination with @Autowired to specify which bean should be injected when multiple beans of the same type are available. Qualifiers help resolve ambiguity.

See the previous answer for more details on using qualifiers.

Primary Annotation:

You can mark one bean as the primary bean when multiple beans of the same type exist. Spring will prefer to inject the primary bean when using @Autowired.

To declare a primary bean, use the @Primary annotation on that bean's class or configuration.

Constructor Binding:

Spring Boot provides the @ConstructorBinding annotation to perform constructor-based dependency injection for configuration properties. This is particularly useful for injecting configuration properties into components.

Example:

```
@ConfigurationProperties(prefix = "myapp")
@ConstructorBinding
public class MyConfigProperties {
    private final String propertyValue;

    public MyConfigProperties(String propertyValue) {
        this.propertyValue = propertyValue;
    }

    public String getPropertyValue() {
        return propertyValue;
    }
}
```

These are the primary ways to perform dependency injection in Spring Boot. The choice of which method to use depends on your specific requirements and best practices, with constructor injection being the recommended approach for most cases.

5. Create a SpringBoot application with MVC using Thymeleaf.

(create a form to read a number and check the given number is even or not).

```
package com.demo.EvenOdd;

import org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class MainController {
	/* http://localhost:8080/evenForm */
	@GetMapping("/evenForm")
	public String evenForm() {
		return "eventest";
	}

	/*http://localhost:8080/processEven */
	@GetMapping("/processEven")
	public String processEven(@RequestParam("number") int number, Model model) {
		model.addAttribute("number", number);
		if (number % 2 == 0) {
			model.addAttribute("result", "Even");
		}else {
			model.addAttribute("result", "Not Even");
		}
		return "eventresult";
	}
```

```
}
```

```
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
```

```
public class EvenOddApplication {
```

```
	public static void main(String[] args) {
```

```
		SpringApplication.run(EvenOddApplication.class, args);
```

```
	}
```

```
		}
```

Eventest.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8"/>
<title>Finding Even Number</title>
</head>
<body>
	<form method="get" action="processEven">
	<label>Enter the Value</label>
	<input type="text" name="number">
	<br/>
	<button type="submit">Is Even</button>
	</form>
</body>
</html>
```

Evenresult.html:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Finding Even Number - Result Page</title>
</head>
<body>
```

```
        <div style="background-color: lightcyan; color: green">
            <h3>The <span th:text="${number}"></span> is <span
th:text="${result}"></span> </h3>
            <h1>Test</h1>
        </div>
</body>
</html>
```

Output: