E.V.K.Deepthi (AF0311771)

1. What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern and a software development technique used in object-oriented programming to manage the dependencies between objects or components in a more flexible and maintainable way. The primary goal of Dependency Injection is to decouple classes or modules by removing direct dependencies between them, making the code more modular, testable, and easier to maintain.

In Dependency Injection, the dependencies of a class or module are provided from the outside rather than being created or instantiated within the class itself. This is typically achieved through one of the following methods

Constructor Injection: Dependencies are passed to a class through its constructor. This is the most common form of Dependency Injection.

```
class UserService {

private UserRepository userRepository;

public UserService(UserRepository userRepository) {

this.userRepository = userRepository;

}

}
```

Setter Injection: Dependencies are set using setter methods.

```
class UserService {

private UserRepository userRepository;

public void setUserRepository(UserRepository userRepository) {

this.userRepository = userRepository;

}

}
```

Interface Injection: Dependencies are injected through interfaces that the class implements.

```
interface UserRepositoryProvider {
UserRepository getUserRepository();
}
class UserService implements UserRepositoryProvider {
private UserRepository userRepository;
@Override
public UserRepository getUserRepository() {
return userRepository;
}
}
```

Dependency Injection frameworks and containers, such as Spring (Java), Angular (JavaScript/TypeScript), and .NET Core (C#), provide tools and mechanisms to manage the injection of dependencies automatically. These frameworks help maintain a clean separation of concerns and enable easier unit testing since you can mock or replace dependencies with test doubles during testing.

Advantages of Dependency Injection:

Decoupling: DI reduces the tight coupling between components, making the codebase more modular and easier to maintain.

Testability: It simplifies unit testing because you can easily replace real dependencies with mock objects or stubs for testing purposes.

Reusability: Components with clearly defined dependencies can be reused more easily in different parts of an application or in other applications.

Flexibility: It allows you to change or swap out dependencies without modifying the classes that use them, making it easier to adapt to changing requirements or technologies.

Readability: DI can make the code more self-explanatory as dependencies are explicitly declared in constructors or methods.

Dependency Injection is a powerful technique for improving the maintainability, flexibility, and testability of software systems by managing and injecting dependencies in a controlled manner.

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

In Spring Boot (which is part of the larger Spring Framework), the @Autowired annotation is used for automatic dependency injection. Its primary purpose is to simplify the process of injecting dependencies into Spring-managed beans or components, making it easier to wire together different parts of your application without manually instantiating or configuring dependencies.

Here's how @Autowired works:

Dependency Resolution: When you annotate a field, constructor, setter method, or even a configuration method with @Autowired, Spring will automatically search for a compatible bean (a class marked as a Spring component) to inject into that field or method parameter.

Type Matching: Spring looks for a bean that matches the type of the field or parameter being annotated. If it finds a unique match, it injects that bean. If multiple beans of the same type are available, you can further specify which one to inject using qualifiers or other annotations.

Here's an example of how @Autowired can be used:

```
@Service
public class MyService {
private final MyRepository myRepository;
@Autowired
public MyService(MyRepository myRepository) {
this.myRepository = myRepository;
}
}
```

In this example:

@Service marks the class as a Spring-managed bean.

The constructor of MyService is annotated with @Autowired, indicating that it expects an instance of MyRepository to be injected.

Spring Boot will automatically provide an instance of MyRepository (if it's a Spring bean) when creating a MyService object. This eliminates the need for manual instantiation or configuration.

The @Autowired annotation is a key part of Spring's Inversion of Control (IoC) and Dependency Injection features. It simplifies the configuration and wiring of components in your Spring Boot application, promoting loose coupling between different parts of your application and making it more maintainable and testable. It's important to note that since Spring Framework 4.3, @Autowired is not required for constructor injection if there is only one constructor in the class, as Spring will automatically detect it and inject dependencies. However, it can still be used for clarity and when you have multiple constructors.

3. Explain the concept of Qualifiers in Spring Boot.

In Spring Boot, qualifiers are used to disambiguate between multiple beans of the same type when performing dependency injection. When you have more than one bean of a particular type registered in the Spring context, and you need to specify which one to inject into a particular component, you can use qualifiers to indicate your choice.

Qualifiers are often used in conjunction with the @Autowired annotation to tell Spring which specific bean should be injected when there is more than one candidate bean of the same type.

Here's how qualifiers work:

Bean Registration: First, you register multiple beans of the same type in your Spring application context. For example, you might have multiple implementations of an interface or multiple beans with the same class type.

```
@Component

public class DataSourceA implements DataSource {

}

@Component

public class DataSourceB implements DataSource {

}
```

In this example, we have two beans of the DataSource type.

Qualify the Injection Point: When you need to inject one of these beans into another component, you can use the @Qualifier annotation to specify which bean to inject. You provide the name of the bean you want to inject as a parameter to @Qualifier.

```
@Service

public class MyService {

private final DataSource dataSource;
@Autowired
public MyService(@Qualifier("dataSourceA") DataSource dataSource) {
this.dataSource = dataSource;
}
}
```

In this example:

@Qualifier("dataSourceA") tells Spring to inject the bean named "dataSourceA" of type DataSource into the MyService class. If you want to inject "dataSourceB" instead, you would change the qualifier accordingly.

Matching Bean Name: The name specified in the @Qualifier annotation should match the name of the bean you want to inject. By default, the name of a bean in Spring is the name of the class with a lowercase first letter (e.g., "dataSourceA" for DataSourceA).

Qualifiers are essential when you have multiple beans of the same type, and Spring needs guidance to determine which one to inject. Without qualifiers, Spring may not know which bean to choose and may throw an exception due to ambiguity.

Qualifiers are a useful feature in Spring Boot to help manage complex dependency injection scenarios and ensure that the right beans are wired together in your application.

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

In Spring Boot, there are several ways to perform dependency injection, and you can choose the one that best fits your application's needs and coding preferences. The primary ways to perform dependency injection in Spring Boot include:

Constructor Injection:

This is the most common and recommended way to perform dependency injection in Spring Boot. You declare dependencies as constructor parameters, and Spring automatically provides the required beans when creating instances of your class.

```
@Service
public class MyService {
private final MyRepository myRepository;
@Autowired
public MyService(MyRepository myRepository) {
this.myRepository = myRepository;
}
}
```

Setter Injection:

With setter injection, you provide setter methods for your dependencies, and Spring uses these methods to inject the required beans. This approach allows for optional dependencies or changing dependencies at runtime.

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

Field Injection:

Field injection involves directly annotating the fields that need to be injected with @Autowired. While this approach is concise, it's generally not recommended because it makes testing and mocking dependencies more challenging and can lead to issues with circular dependencies.

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

Method Injection:

You can also inject dependencies into methods using the @Autowired annotation on the method parameters. This is less common but can be useful in specific situations where you want to inject dependencies for a specific method rather than the entire class.

```
@Service
public class MyService {
public void doSomethingWithRepository(@Autowired MyRepository myRepository) {
}
}
```

Interface-Based Injection:

If you have multiple implementations of an interface and need to specify which one to inject, you can use qualifiers or the @Primary annotation to indicate the primary bean to inject. This approach is commonly used when you have multiple implementations, and you want to choose one as the default.

```
@Service
public class MyService {
private final DataSource dataSource;
@Autowired
public MyService(DataSource dataSource) {
this.dataSource = dataSource;
}
}
```

Constructor-Based Injection with Lombok:

If you're using Project Lombok in your Spring Boot project, you can use the @RequiredArgsConstructor annotation to automatically generate constructors with the required dependencies.

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

Choose the dependency injection method that best suits your project's requirements and maintainability goals. Constructor injection is generally recommended because it results in cleaner and more testable code while helping to avoid issues with circular dependencies. However, the choice may also depend on the specific use case and design considerations for your Spring Boot application.

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)

ThymeleafenabledspringbootwebApplication.java

```
package com.demo.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication

public class ThymeleafenabledspringbootwebApplication {


public static void main(String[] args) {

SpringApplication.run(ThymeleafenabledspringbootwebApplication.class, args);

}


}
```

MainController.java

```
package com.demo.example;

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";
}


}
```

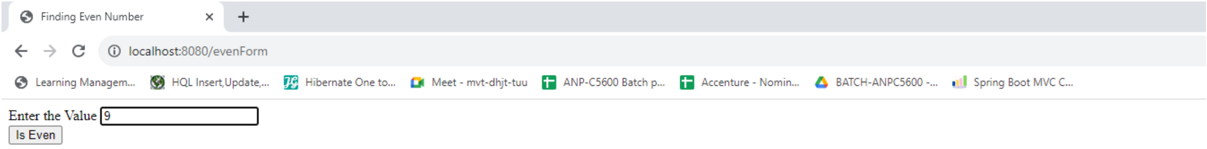
Eventest.html

```
<!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>
```

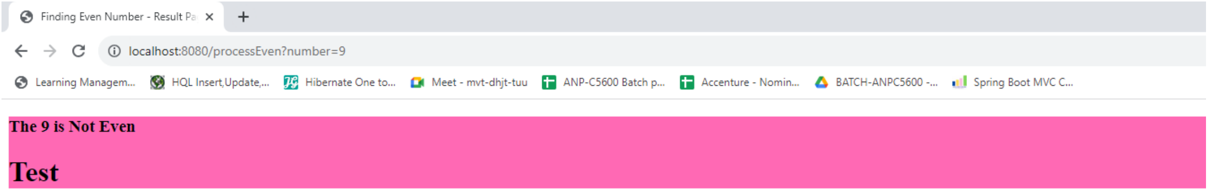
Eventresult.html

```
<!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: hotpink; color: rgb(0, 0, 0)">
<h3>The <span th:text="${number}"></span> is <span
th:text="${result}"></span> </h3>
<h1>Test</h1>
</div>
</body>
</html>
```

OUTPUT:



Enter the Value 9



The 9 is Not Even

Test