Neelisetti.HariPrasad(LAB-14)

5.Create a SpringBoot application with MVC using Thymeleaf.(create a form to read a number and check the givennumber is even or not).

```
package com.example.demo;


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


@SpringBootApplication
public class ThymeleafSpringbootApplication {

	public static void main(String[] args) {
		SpringApplication.run(ThymeleafSpringbootApplication.class, args);
	}
}


package com.example.demo;
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

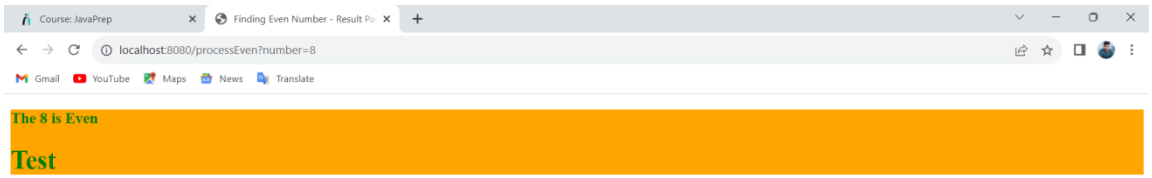
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8"/>
<title>Finding Even Number</title>
</head>
```

```
<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>
```

Evenresut.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: orange; color: green">
<h3>The <span th:text="${number}"></span> is <span
th:text="${result}"></span> </h3>
<h1>Test</h1>
</div>
</body>
</html>
```

Output:



1.What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern in which an object does not create its own dependencies, but instead receives them from an external source. This allows the object to be more easily tested and maintained, as the dependencies can be easily swapped out.
DI is often used in conjunction with a framework or library that provides the necessary dependencies. This can make it easier to develop applications, as the developer does not need to worry about creating and managing the dependencies themselves.

There are two main types of DI:

Constructor injection: The dependencies are passed to the object through its constructor. This is the most common type of DI.

Setter injection: The dependencies are passed to the object through setter methods.This is less common than constructor injection, but it can be useful in some cases
.
DI can be a powerful tool for improving the test ability, maintainability, and extensibility of your applications. Dependency Injection (DI) is a design pattern in which an object does not create its own dependencies, but instead receives them from an external source.

This is in contrast to the traditional approach, where an object would create its own dependencies, which can lead to tight coupling between objects and make it difficult to test and maintain code.

With DI, the dependencies of an object are provided to it through its constructor or through setter methods. This allows the object to be more easily tested and maintained, as the dependencies can be easily swapped out.

DI can also help to improve the extensibility of code, as new dependencies can be easily added without having to rewrite the code that uses them.

There are two main types of DI:

Constructor injection: The dependencies are passed to the object through its constructor. This is the most common type of DI.

Setter injection: The dependencies are passed to the object through setter methods.This is less common than constructor injection, but it can be useful in some cases.

DI is a powerful tool that can help to improve the testability, maintainability, and extensibility of your code. It is a good practice to use DI whenever possible.

Dependency Injection (DI) is a design pattern in which an object does not create its own dependencies, but instead receives them from an external source.

This is in contrast to the traditional approach, where an object would create its own dependencies, which can lead to tight coupling between objects and make it difficult to test and maintain code.

With DI, the dependencies of an object are provided to it through its constructor or through setter methods. This allows the object to be more easily tested and maintained, as the dependencies can be easily swapped out.

DI can also help to improve the extensibility of code, as new dependencies can be easily added without having to rewrite the code that uses them.
There are two main types of DI:

Constructor injection: The dependencies are passed to the object through its constructor. This is the most common type of DI.

Setter injection: The dependencies are passed to the object through setter methods.This is less common than constructor injection, but it can be useful in some cases.

DI is a powerful tool that can help to improve the testability, maintainability, and extensibility of your code. It is a good practice to use DI whenever possible.
Here is a simple example of DI in Java:

EXAMPLE:

```
public class Car {
    private Engine engine;
    private Wheels wheels;
    public Car(Engine engine, Wheels wheels) {
        this.engine = engine;
        this.wheels = wheels;
    }
    public void drive() {
        engine.start();
        wheels.turn();
    }
}
```

In this example, the **`Car`** class does not create its own dependencies (the `Engine` and **`Wheels` objects),** but instead receives them from an external source (the **`Car`** constructor). This makes the **`Car`** class easier to test and maintain, as the dependencies can be easily swapped out.
DI is a powerful tool that can help to improve the quality of your code. It is a good practice to use DI whenever possible.

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

The `**@Autowired**` annotation in Spring Boot is used to wire dependencies into a bean.

It tells Spring Boot to automatically wire the bean with the appropriate dependency, based on the type of the dependency.

This can be used to wire in other beans, or to wire in values from properties files or environment variables.

For example, the following code shows how to use the `**@Autowired**` annotation to wire in a `DataSource` bean:

```
@Component
public class MyBean {
    @Autowired
    private DataSource dataSource;
    // ...
}
```

In this example, the `**@Autowired**` annotation tells Spring Boot to wire the `dataSource` field with a `DataSource` bean. Spring Boot will automatically find the appropriate `DataSource` bean to wire in, based on the type of the `dataSource` field.

The `**@Autowired**` annotation can also be used to wire in values from properties files or environment variables.
For example, the following code shows how to use the `**@Autowired**` annotation to wire in a value from a properties file:

```
@Component
public class MyBean {
    @Autowired
    @Value("${my.property}")
    private String myProperty;
    // ...
}
```

In this example, the `**@Autowired**` annotation tells Spring Boot to wire the `myProperty` field with the value of the `my.property` property from the properties file. Spring Boot will automatically find the value of the `my.property` property and wire it into the `myProperty` field.
The `**@Autowired**` annotation is a powerful tool that can be used to simplify the wiring of dependencies in Spring Boot applications. It can be used to wire in other beans, or to wire in values from properties files or environment variables.

`@Autowired` annotation is used for automatic dependency injection. It allows you to automatically wire (or inject) dependencies into your Spring beans. By using `@Autowired`, you don't need to manually create instances of dependencies, as

Spring will handle it for you. It helps to reduce boilerplate code and makes your code more maintainable.
@Autowired` annotation is used to automatically wire dependencies. It works by searching for a bean of the required type and injecting it into the annotated field, constructor, or method parameter.
It simplifies the process of managing dependencies and promotes loose coupling in your application. With `@Autowired`, you can focus on writing business logic

without worrying about creating and managing dependencies manually. It's a handy feature that enhances the flexibility and maintainability of your Spring Boot application!

`@Autowired` annotation in Spring Boot, it's important to note that it works based on type matching.
If there are multiple beans of the same type, Spring may throw an exception. To resolve this, you can use the `@Qualifier` annotation along with `@Autowired` to specify the exact bean you want to inject.
This allows for more fine-grained control over the dependency injection process. It's a handy technique to ensure the correct dependencies are injected into your Spring beans.

2. Explain the concept of Qualifiers in Spring Boot

Qualifiers in Spring Boot are used to differentiate between multiple beans of the same type.
By default, Spring Boot will autowire beans by type.
This means that if there are two beans of the same type in the application context, Spring Boot will not know which one to autowire.

Qualifiers can be used to resolve this ambiguity.
To use a qualifier, simply add the @Qualifier annotation to the property or field that you want to autowire.

The value of the @Qualifier annotation should be the name of the bean that you want to autowire.

Qualifiers can also be used to autowire beans by name.
To do this, simply add the @Autowired annotation to the property or field that you want to autowire and specify the name of the bean in the value attribute.

Qualifiers are a powerful tool that can be used to improve the readability and maintainability of your Spring Boot applications.
In Spring Boot, qualifiers are used to resolve ambiguity when there are multiple beans of the same type. They provide a way to specify which bean should be injected when there are multiple candidates.

To use qualifiers, you can combine the `@Autowired` annotation with the `@Qualifier` annotation. The `@Qualifier` annotation allows you to specify the name or value of the desired bean. This way, Spring knows exactly which bean to inject.

For example, let's say you have multiple implementations of an interface called `UserService`.
To specify which implementation to inject, you can define a qualifier annotation, such as `@Admin`, and annotate the desired implementation with `@Qualifier("admin")`.
Then, in your code, you can use `@Autowired` along with `@Qualifier("admin")` to inject the specific implementation.

By using qualifiers, you can have more control over the dependency injection process and ensure that the correct beans are injected in your Spring Boot application.
It's a powerful feature that helps resolve ambiguity and maintain the desired behavior of your application.

Qualifiers in Spring Boot are used to differentiate between multiple beans of the same type.

By default, Spring Boot will autowire beans by type.
This means that if there are two beans of the same type in the application context, Spring Boot will not know which one to autowire.

Qualifiers can be used to resolve this ambiguity.
To use a qualifier, simply add the @Qualifier annotation to the property or field that you want to autowire.

The value of the @Qualifier annotation should be the name of the bean that you want to autowire.
Qualifiers can also be used to autowire beans by name.

To do this, simply add the @Autowired annotation to the property or field that you want to autowire and specify the name of the bean in the value attribute.
Qualifiers are a powerful tool that can be used to improve the readability and maintainability of your Spring Boot applications.

WITH EXAMPLE PROGRAM
Here is an example of how to use qualifiers in Spring Boot:

```
public class MyService {
    @Autowired
    @Qualifier("myBean1")
    private MyBean bean1;
    @Autowired
    @Qualifier("myBean2")
    private MyBean bean2;
    // ...
}
```

In this example, the @Autowired annotation is used to autowire the two beans of type MyBean.
The @Qualifier annotation is used to specify the name of the bean that should be autowired.

In this case, the bean1 property will be autowired with the bean named "myBean1" and the bean2 property will be autowired with the bean named "myBean2".

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

There are three ways to perform Dependency Injection (DI) in Spring Boot:
* Constructor Injection
* Setter Injection
* Field Injection

- **Constructor Injection**

Constructor injection is the most preferred way of performing DI in Spring Boot.
In constructor injection, the dependencies are passed as arguments to the constructor of the dependent class.
This allows the dependent class to have its dependencies injected at the time of its creation.
Here is an example of constructor injection:

```
public class MyService {
    private MyRepository repository;
    public MyService(MyRepository repository) {
        this.repository = repository;
    }

}
```

In this example, the MyService class has a dependency on the MyRepository interface. The MyRepository dependency is injected into the MyService class through its constructor.

- **Setter Injection**

Setter injection is another way of performing DI in Spring Boot.
In setter injection, the dependencies are set on the properties of the dependent class using setter methods.
This allows the dependent class to have its dependencies injected after it has been created.
Here is an example of setter injection:

```
public class MyService  {
private MyRepository repository;
    public void setRepository(MyRepository repository) {
        this.repository = repository;
    }

}
```

In this example, the MyService class has a dependency on the MyRepository interface. The MyRepository dependency is injected into the MyService class through its setRepository() method.

- **Field Injection**

Field injection is the least preferred way of performing DI in Spring Boot.
In field injection, the dependencies are injected directly into the fields of the dependent class.
This allows the dependent class to have its dependencies injected at the time of its creation.
Here is an example of field injection:

```
public class MyService {
    @Autowired
    private MyRepository repository;

}
```

In this example, the MyService class has a dependency on the MyRepository interface. The MyRepository dependency is injected into the MyService class through its repository field.
Which DI method should you use?
Constructor injection is the most preferred way of performing DI in Spring Boot.
This is because constructor injection allows the dependencies to be injected into the dependent class at the time of its creation.