SpringMVC Thymeleaf Assessment

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 different components or objects in a system. It is primarily employed in the context of inversion of control (IoC) containers and frameworks to facilitate loose coupling and improve the maintainability, testability, and flexibility of software systems.

In a typical software application, objects or components often rely on other objects or services to perform their tasks. These dependencies can include things like database connections, file access, network communication, or other custom services. Managing these dependencies manually can lead to tightly coupled code, making it hard to change or test individual components in isolation.

Dependency Injection addresses this issue by allowing dependencies to be injected into an object rather than the object creating or managing them itself. Here's how it works:

Dependency: A dependency is an object or service that another object needs to perform its function. For example, a UserService object might depend on a UserRepository to retrieve user data from a database.

Dependency Provider: Instead of creating the dependency within the dependent object, the dependency is provided to the dependent object from the outside. This is typically done through constructor injection, method injection, or property injection.

Constructor Injection: Dependencies are passed to the dependent object through its constructor. This is the most common and preferred way of implementing DI. Method Injection: Dependencies are provided to specific methods of the dependent object when needed.
Property Injection: Dependencies are set as properties of the dependent object. **IoC Container**: In larger applications, an IoC container (Inversion of Control container)

is often used to manage the creation and lifetime of objects and their dependencies. The container maintains a registry of object types and their associated dependencies, and it can automatically resolve and inject these dependencies when creating objects.

Benefits of Dependency Injection:

Loose Coupling: DI promotes loose coupling between components, making it easier to change or replace dependencies without affecting the dependent objects.

Testability: By injecting mock or test implementations of dependencies, it becomes easier to unit test individual components in isolation.

Flexibility: Components can be configured and wired together differently at runtime, making it easier to adapt the system to changing requirements. Reusability: Components can be reused in different contexts or applications because their dependencies can be easily adjusted.

Dependency Injection is a powerful technique for managing dependencies in software applications, promoting modularity, maintainability, and testability while reducing the tight coupling between components.

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

The @Autowired annotation in Spring Boot is used to automatically inject dependencies into a bean. This annotation is used on the field, setter method, or constructor of the bean that needs to be injected. The Spring Boot container will automatically find the corresponding bean and inject it into the dependency. This annotation makes it easy to develop Spring Boot applications because it eliminates the need to manually configure the dependencies.

Here are some of the benefits of using @Autowired annotation in Spring Boot:
* It makes the code more readable and maintainable.

* It reduces the amount of code that needs to be written.

* It makes it easier to test the application.

* It makes the application more modular and reusable.

The @Autowired annotation provides more fine-grained control over where and how autowiring should be accomplished. 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.

@Autowired on Setter Methods

You can use @Autowired annotation on setter methods to get rid of the <property> element in XML configuration file. When Spring finds an @Autowired annotation used with setter methods, it tries to perform byType autowiring on the method.

@Autowired on Properties

You can use @Autowired annotation on properties to get rid of the setter methods. When you will pass values of autowired properties using <property> Spring will automatically assign those properties with the passed values or references.

@Autowired on Constructors

You can apply @Autowired to constructors as well. A constructor @Autowired annotation indicates that the constructor should be autowired when creating the bean, even if no <constructor-arg> elements are used while configuring the bean in XML file.

@Autowired with (required = false) option

By default, the @Autowired annotation implies the dependency is required similar to @Required annotation, however, you can turn off the default behavior by using (required=false) option with @Autowired.

3. Explain the concept of Qualifiers in Spring Boot.

Spring boot qualifier is used when we need to create more than one bean for the same type and need to wired-only one bean. In this situation, we are using autowired and qualifier annotation to remove confusion which bean we need to wired in our spring boot project. Basically, it is showing how to differentiate bean of same type which is qualifier.

As we know that autowired annotation is mostly used in injecting the spring boot dependency.

By default, the annotation of autowired will resolve its dependencies. This annotation is working well if we have only one bean of the same type.

Spring boot framework is throwing an exception when we have used the same type of bean one or more times.

Spring boot qualifier annotation is used to distinguish the references of bean. For selecting the correct bean from code we are using qualifier annotation in spring boot project.

Spring boot qualifier annotation bean is also used to eliminate the issue which was injected by beans.

We are using bean in one class to eliminate the issue of which beans to be injected from project.

The below example shows how to solve the issue of bean injection in our project are as follows.

Example:

Public class qualifier

{

@Autowired

@Qualifier ("SpringBootQualifier")

Private SBQualifier squalifier
}

- In the above example, we can see that we have used qualifier annotation with the implementation name as SpringBootQualifier.

- In the above example we have avoided the ambiguity when spring boot will find multiple beans of a single type.

- To use same bean in single code we need to use @component annotation to obtain same result in program.

- Below example shows how to use two qualifier annotation in single code are as follows.

Example:

@Component

@Qualifier ("SpringBootQualifier")

Public class SpringBootQualifier implements SBQualifier

{

….

}

@Component

@Qualifier ("SBQualifier")

Public class sbQualifier implements SBQualifier

{

….

}

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

In Spring Boot, there are several ways to perform Dependency Injection, which is a fundamental feature of the Spring Framework that allows you to manage and inject dependencies into your components. Here are the different ways to achieve Dependency Injection in Spring Boot:

Constructor Injection:

This is the most recommended and widely used method for dependency injection in Spring Boot.

You can inject dependencies by defining constructor(s) in your Spring components and using the @Autowired annotation on the constructor parameter(s).

Example: @Service public class

MyService { private final MyRepository

repository;

@Autowired public MyService(MyRepository

repository) { this.repository = repository;

}

}

Setter Injection:

You can also perform dependency injection by defining setter methods in your components and annotating them with @Autowired.

Example:
@Service public class MyService {

private MyRepository repository;

@Autowired public void setRepository(MyRepository

repository) { this.repository = repository;

}

}

Field Injection:

While field injection is less recommended due to reduced testability and encapsulation, you can inject dependencies directly into fields using the @Autowired annotation.

Example: @Service public class

MyService { @Autowired private

MyRepository repository;

}

Qualifier Annotation:

When you have multiple beans of the same type, you can use the @Qualifier annotation along with @Autowired to specify which bean to inject.

Example:

@Service

public class MyService { private final

MyRepository myRepository;

@Autowired

public MyService(@Qualifier("myRepositoryImpl") MyRepository myRepository) { this.myRepository =

myRepository;

}

}

Constructor-Based @Autowired (Spring 4.3+):

Starting from Spring 4.3, you can omit the @Autowired annotation on the constructor if you have only one constructor. Spring will automatically inject dependencies into the constructor.

Example: @Service public class MyService {

private final MyRepository repository; public

MyService(MyRepository repository) {

this.repository = repository;

}

}

Qualifiers with Constructor Injection:

You can combine constructor injection with qualifiers to specify the bean to inject.
Example: @Service public class

MyService { private final MyRepository

repository;

```
  @Autowired
    public MyService(@Qualifier("myRepositoryImpl") MyRepository repository)
{
        this.repository = repository;
    }
}
```

@Resource Annotation:

Instead of using @Autowired, you can use the @Resource annotation to inject dependencies by name.

Example: @Service

public class MyService {

```
    @Resource(name = "myRepositoryImpl")
    private MyRepository repository;
```

}

Choose the appropriate method based on your application's requirements and best practices. Constructor injection is generally recommended as it ensures that the dependencies are available when the object is created, leading to better testability and reducing the chances of null pointers.

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)

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="ISO-8859-1">
<title>Finding Even Number - Result Page</title>
<body>
<div style="background-color: rgb(0, 0, 64); color: rgb(255, 255, 255)">
                    <h3>The <span th:text="${number}"></span> is <span
th:text="${result}"></span> </h3>
                  <h1>Test</h1>
          </div>
</body>
</html>
```

MainController.java :

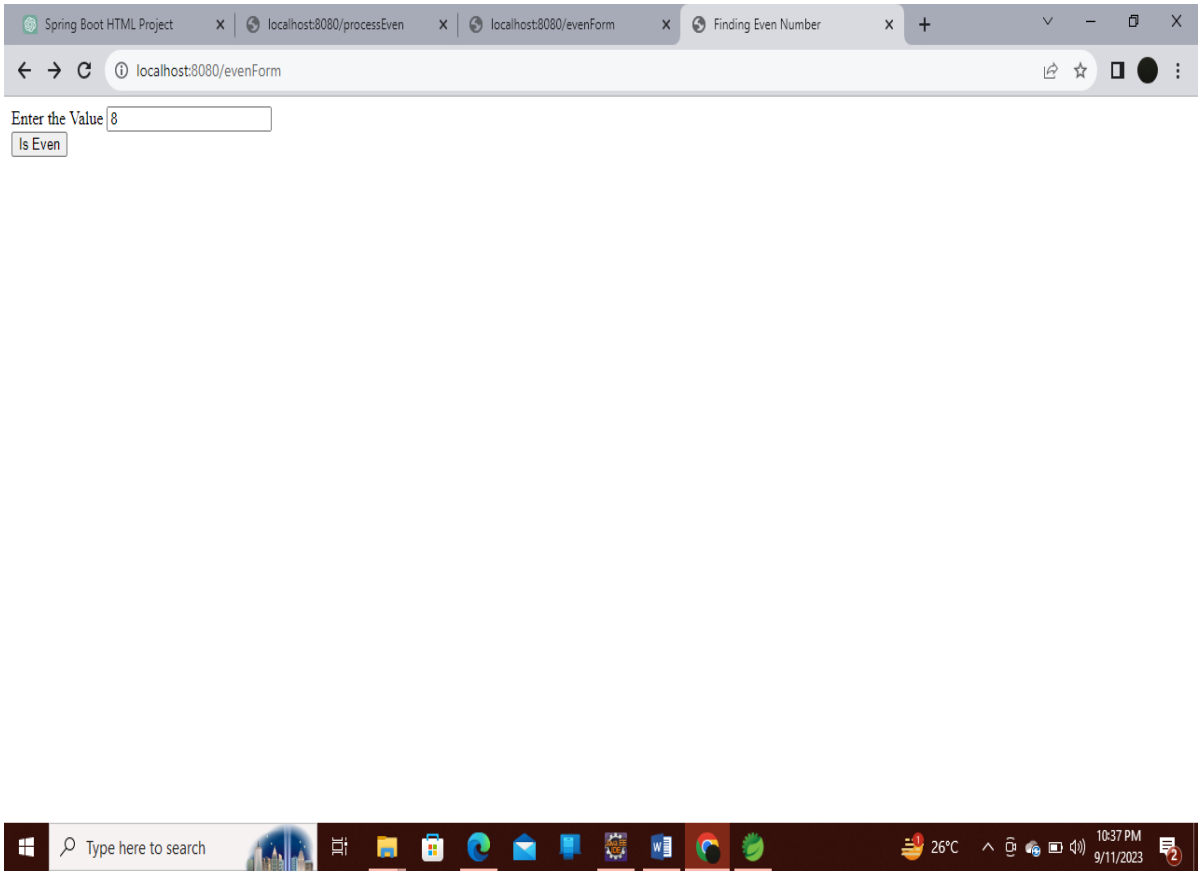
```
package com.example.demo; import
org.springframework.stereotype.Controller; import
org.springframework.ui.Model; import
org.springframework.web.bind.annotation.GetMapping; import
org.springframework.web.bind.annotation.RequestParam; import
org.springframework.web.bind.annotation.ResponseBody;
@Controller public class
MainController {

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

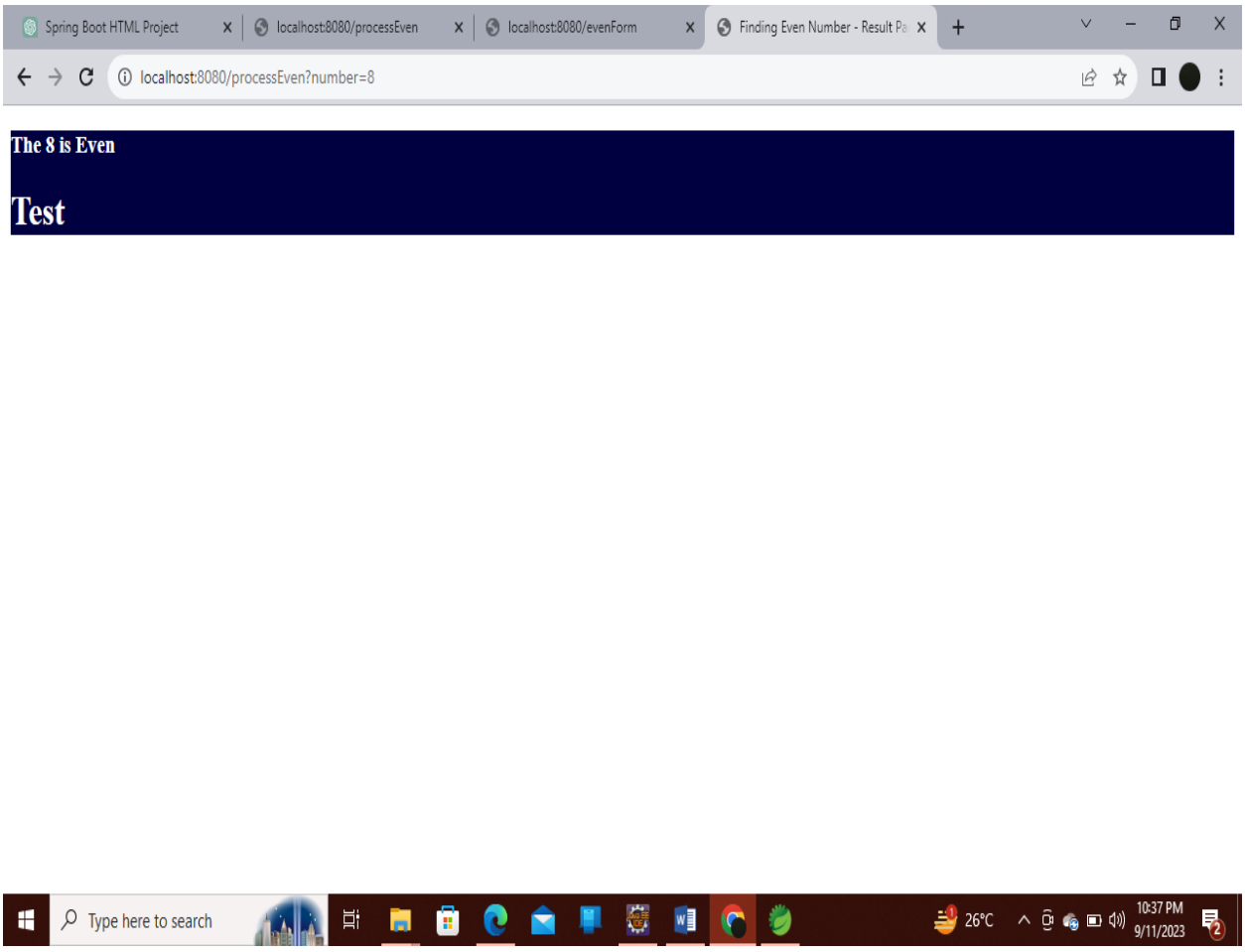
```
	/*http://localhost:8080/processEven */ @GetMapping("/processEven")
	public String processEven(@RequestParam("number") int number, Model
model) {
		model.addAttribute("number", number) ;
		if (number % 2 == 0) {
			model.addAttribute("result", "Even") ;
		}else { model.addAttribute("result", "Not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

}
```

SpringBoot.Application.java :

```
package com.example.demo; import
org.springframework.boot.SpringApplication; import
org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication public class
SpringBootFormApplication { public static
void main(String[] args) {
SpringApplication.run(SpringBootFormAppl
ication.class, args) ;
	}
}
```

Output:


Spring Boot HTML Project
localhost:8080/processEven
localhost:8080/evenForm
Finding Even Number
localhost:8080/evenForm
Enter the Value 8
Is Even
Type here to search
26°C
10:37 PM
9/11/2023


Spring Boot HTML Project
localhost:8080/processEven
localhost:8080/evenForm
Finding Even Number - Result Pa
localhost:8080/processEven?number=8
The 8 is Even
Test
Type here to search
26°C
10:37 PM
9/11/2023