S.Jeevan Kumar (Lab 14)

1.What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern and a technique used in software development to manage and handle the dependencies between different components or classes within an application. The primary goal of Dependency Injection is to achieve a separation of concerns and promote modularity, testability, and maintainability in your code.

In a software system, components or classes often rely on other components or services to perform their functions. These dependencies can be database connections, web services, utility classes, or other objects. Without proper management, code can become tightly coupled, making it difficult to change and test individual components in isolation.

Dependency Injection addresses these issues by inverting the control of object creation and management. Instead of a class creating its dependencies, the dependencies are provided from outside the class. Here's how it works:

Injection of Dependencies: Instead of creating instances of dependencies within a class or component, the dependencies are "injected" into the class from external sources. These external sources are typically known as dependency injection containers or frameworks.

Decoupling: By injecting dependencies from the outside, classes are decoupled from the specific implementations of their dependencies. This decoupling allows for easier maintenance and testing, as you can replace or modify dependencies without changing the dependent class.

Testing: Dependency Injection facilitates unit testing. Since you can inject mock or stub dependencies for testing purposes, it becomes easier to isolate and test individual components without relying on the actual implementations of their dependencies.

Configurability: It enables greater configurability and flexibility in your application. You can configure the application's behavior by changing the injected dependencies or by providing different implementations based on specific conditions.

There are several methods to implement Dependency Injection:

Constructor Injection: Dependencies are injected via a class constructor. This is the most common method and ensures that dependencies are available when an instance of the class is created.

Method Injection: Dependencies are injected via methods or setters, allowing for more flexibility when changing dependencies during the object's lifecycle.

Property Injection: Dependencies are injected into public properties of a class. This method is less preferred because it doesn't guarantee that dependencies are available when the object is used.

Dependency Injection is widely used in various programming languages and frameworks, such as Java Spring, .NET Core, and Angular. It helps create maintainable, testable, and modular code, making it an important concept in modern software development.a

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

In Spring Boot (and the wider Spring Framework), the @Autowired annotation is used to automatically inject dependencies into a Spring bean. It's a fundamental part of the dependency injection mechanism in Spring and simplifies the process of wiring together different components in your application.

Here's the main purpose of the @Autowired annotation:

Dependency Injection: By marking a field, constructor, or setter method with @Autowired, you're telling Spring to automatically provide the necessary dependency for that component when it's created. Spring will search for a matching bean (an instance of a class annotated with @Component, @Service, @Repository, etc.) and inject it into the annotated field or method parameter.

Here's a basic example:

java

Copy code

```
@Service

public class MyService {

    private final MyRepository myRepository;



    @Autowired

    public MyService(MyRepository myRepository) {

        this.myRepository = myRepository;

    }



    // ...

}
```

In this example, the MyService class has a dependency on MyRepository, and Spring will automatically inject an instance of MyRepository when creating a MyService bean.

Reducing Boilerplate Code: Using @Autowired helps reduce the boilerplate code required for manual dependency injection. Without it, you'd have to create instances of your dependencies and pass them explicitly into your classes, which can be error-prone and cumbersome, especially in large applications.

Simplifying Configuration: @Autowired works well with Spring's container-managed beans, making it easy to configure and manage your application's components through annotations and XML configurations.

It's important to note that @Autowired can be used in various places, including constructors, fields, and methods, and Spring Boot will automatically identify and inject the appropriate dependencies based on the type and qualifiers (if needed). However, it's essential to ensure that your Spring components are correctly annotated and that the dependencies you're trying to inject are available as beans within the Spring application context.

3. Explain the concept of Qualifiers in Spring Boot.

In Spring Boot, the concept of qualifiers is related to dependency injection, which is a fundamental aspect of the Spring Framework. Qualifiers are used to disambiguate between multiple beans of the same type when they are being injected into a component or a class. This is particularly useful when you have multiple beans of the same type and you want to specify which one should be injected into a particular component.

Here's how qualifiers work in Spring Boot:

Defining Beans: When you create beans in your Spring Boot application, you typically use the @Bean annotation or component scanning to create instances of classes that should be managed by the Spring container. For example:

java

Copy code

```
@Bean
public DataSource dataSource1() {
    // Create and configure a DataSource
}

@Bean
public DataSource dataSource2() {
```

```
    // Create and configure another DataSource
}
```

Injecting Beans: Now, let's say you have a class that needs a DataSource injected, but there are multiple DataSource beans available in the application context:

java

Copy code

```
@Service
public class MyService {
    private final DataSource dataSource;

    @Autowired
    public MyService(@Qualifier("dataSource1") DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

In this example, the @Qualifier annotation is used to specify which DataSource bean should be injected into the MyService class. By providing the qualifier "dataSource1", you are telling Spring to inject the dataSource1 bean.

Using Qualifiers: The @Qualifier annotation should be used in conjunction with the @Autowired annotation (or constructor injection) to specify which bean should be injected. You can use the bean's name (as defined in the @Bean annotation) as the value for the @Qualifier annotation.

By using qualifiers, you can make sure that Spring Boot knows which bean to inject when there are multiple candidates of the same type. This helps in resolving ambiguity and ensures that the correct dependency is injected into your components.

Additionally, if you have components with no qualifiers, Spring Boot will try to match the bean by its name. So, if you have a DataSource bean named "dataSource1", and you simply use @Autowired DataSource dataSource, Spring Boot will try to match it by name without the need for a qualifier.

Overall, qualifiers are a valuable tool for managing bean injection and dealing with scenarios where multiple beans of the same type are available in the Spring Boot application context.

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

In Spring Boot, dependency injection is a fundamental concept that allows you to manage the dependencies between your classes and components. Dependency injection helps achieve loose coupling between classes and promotes the use of interfaces and abstractions, making your application more maintainable and testable. Spring Boot provides several ways to perform dependency injection:

Constructor Injection:

Constructor injection is the most recommended and widely used method for dependency injection in Spring Boot. You simply define constructor(s) in your class that accept the dependencies as parameters, and Spring will automatically inject those dependencies when creating an instance of the class.

java

Copy code

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

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

Setter Injection:

In setter injection, you define setter methods for your dependencies, and Spring injects them through these setters. While not as preferred as constructor injection, setter injection can be useful when you want to inject optional dependencies or when working with legacy code.

java

Copy code

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

    @Autowired
    public void setRepository(MyRepository repository) {
        this.repository = repository;
    }
}
```

Field Injection:

Field injection involves annotating class fields with @Autowired. While this approach is concise, it's generally not recommended because it makes your code tightly coupled to Spring and can make testing more challenging.

java

Copy code

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

Method Injection:

In method injection, you can annotate a method with @Autowired to inject dependencies. This method can be a regular method, not necessarily a constructor or setter.

java

Copy code

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

    @Autowired
    public void injectRepository(MyRepository repository) {
        this.repository = repository;
    }
}
```

Qualifiers:

When multiple beans of the same type exist in the Spring context, you can use the @Qualifier annotation to specify which bean should be injected.

java

Copy code

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

    @Autowired
```

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

Constructor-Based @Autowired:

Starting with Spring 4.3, if your class has only one constructor, you can omit the @Autowired annotation, and Spring will automatically inject the dependencies for you.

java

Copy code

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

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

These are the primary ways to perform dependency injection in Spring Boot. Constructor injection is generally recommended as it provides clear dependencies and makes it easier to reason about your code. However, the choice of injection method depends on your specific requirements and design considerations.

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";

}

}
```

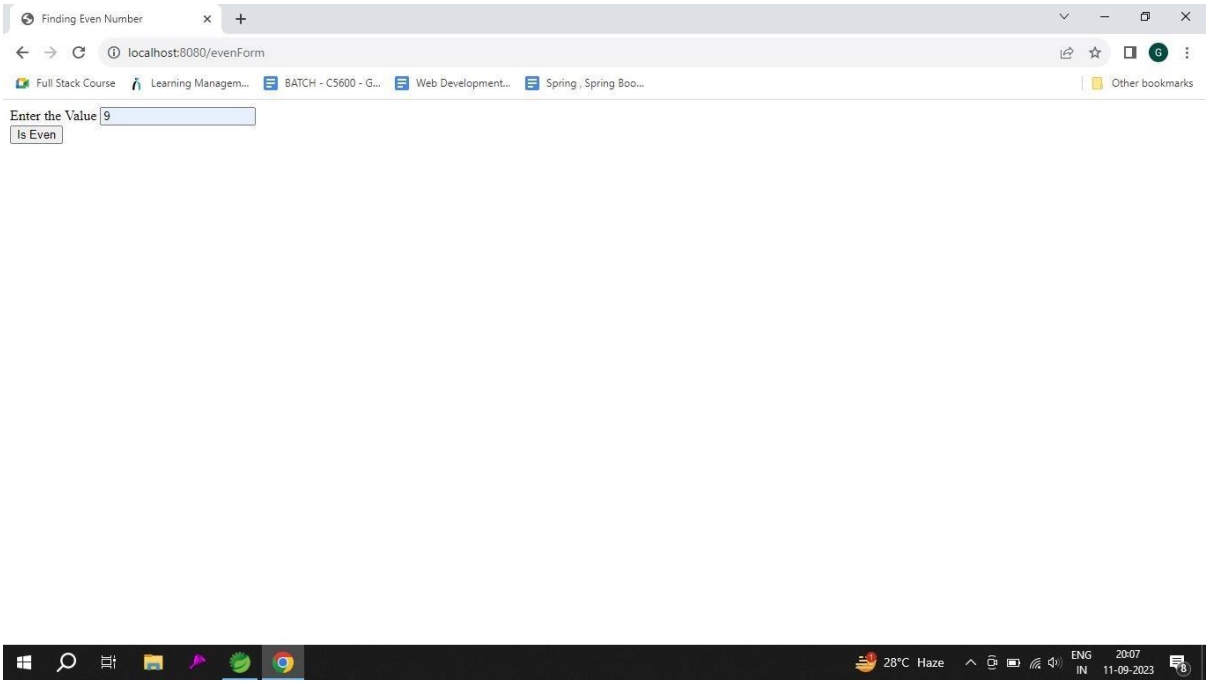
Eventest.html

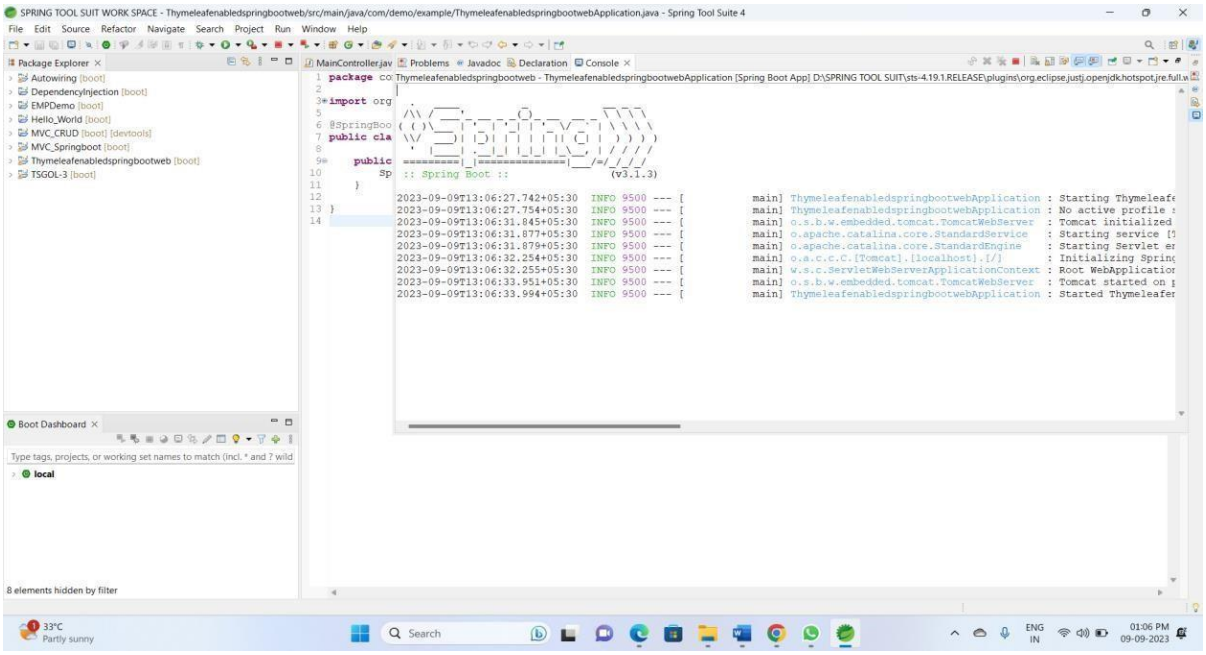
```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8"/>
<title>Finding Even Number</title>
</head>
<body>
<form method="get" action="processEven">
<label>Enter the Value</label>
<input type="text" name="number">
<br/>
<button type="submit">Is Even</button>
</form>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

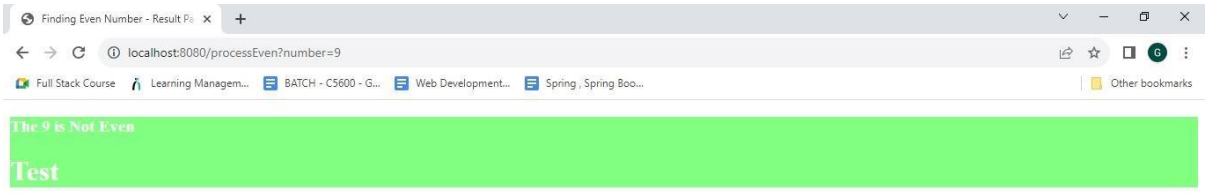
Eventresult.html

```
<!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: rgb(128, 255, 255); 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:




SPRING TOOL SUIT WORK SPACE - Thymeleafenabledspringbootweb/src/main/java/com/demo/example/ThymeleafenabledspringbootwebApplication.java - Spring Tool Suite 4
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Autowiring [boot]
DependencyInjection [boot]
EMPDemo [boot]
Hello_World [boot]
MVC_CRUD [boot] [devtools]
MVC_Springboot [boot]
Thymeleafenabledspringbootweb [boot]
TSGOL-3 [boot]
MainController.jav
Problems
Javadoc
Declaration
Console
Thymeleafenabledspringbootweb - ThymeleafenabledspringbootwebApplication [Spring Boot App] D:\SPRING TOOL SUIT\sts-4.19.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.w
:: Spring Boot :: (v3.1.3)
2023-09-09T13:06:27.742+05:30 INFO 9500 --- [main] ThymeleafenabledspringbootwebApplication : Starting Thymeleafe
2023-09-09T13:06:27.754+05:30 INFO 9500 --- [main] ThymeleafenabledspringbootwebApplication : No active profile
2023-09-09T13:06:31.845+05:30 INFO 9500 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized
2023-09-09T13:06:31.877+05:30 INFO 9500 --- [main] o.apache.catalina.core.StandardService : Starting service [
2023-09-09T13:06:31.879+05:30 INFO 9500 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet e
2023-09-09T13:06:32.254+05:30 INFO 9500 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring
2023-09-09T13:06:32.255+05:30 INFO 9500 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicatio
2023-09-09T13:06:33.951+05:30 INFO 9500 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on
2023-09-09T13:06:33.994+05:30 INFO 9500 --- [main] ThymeleafenabledspringbootwebApplication : Started Thymeleafe
Boot Dashboard
Type tags, projects, or working set names to match (incl. * and ? wild
local
8 elements hidden by filter
33°C Partly sunny
Search
ENG IN
01:06 PM 09-09-2023


Finding Even Number - Result Pa
localhost:8080/processEven?number=9
Full Stack Course
Learning Managem...
BATCH - C5600 - G...
Web Development...
Spring , Spring Boo...
Other bookmarks
The 9 is Not Even
Test


28°C Haze
ENG IN
20:07 11-09-2023