1.What is Dependency Injection (DI)?

Dependency Injection (DI) is a design pattern and a technique used in software engineering to achieve the principle of **Inversion of Control (IoC)**. It is primarily used in object-oriented programming to manage the dependencies between classes and promote decoupling of components within an application.

In Dependency Injection:

1. **Dependencies**: A dependency is an object that another object relies upon. For example, if you have a **Car** object, it may depend on an **Engine** object, a **Transmission** object, and other components.
2. **Injection**: Instead of a class creating its own dependencies or knowing how to obtain them, dependencies are "injected" or provided to the class from an external source, typically through constructor injection, method injection, or property injection.
3. **Inversion of Control (IoC)**: With DI, control over the creation and management of dependencies is inverted. Instead of a class controlling the instantiation of its dependencies, the responsibility is shifted to an external component, typically a DI container or framework.

Here's a breakdown of DI concepts:

- **Constructor Injection**: Dependencies are passed to a class through its constructor. This is the most common form of DI and helps ensure that an object is in a valid state when it's created.

```
public class Car {
   private Engine engine;
   public Car(Engine engine) {
      this.engine = engine;
   }
}
```

4.Method Injection: Dependencies are provided to a class through a method call. This is less common than constructor injection but can be useful in certain scenarios.

```
public class Car {
   public void setEngine(Engine engine) {
      // Set the engine dependency
   }
}
```

5.Property Injection: Dependencies are set directly on public properties of a class. While this approach is simple, it can make it harder to ensure that an object is in a valid state.

```
public class Car {
    public Engine engine;


    public void start() {
        // Use the engine dependency
    }
}
```

The benefits of Dependency Injection include:

- **Decoupling**: Classes become less tightly coupled because they don't need to know how to create their dependencies. This makes code more modular and easier to maintain.
- **Testability**: It becomes easier to write unit tests for classes when you can easily substitute real dependencies with mock objects or test doubles.
- **Flexibility**: You can change the behavior of a component by providing different implementations of its dependencies without modifying the component itself. This promotes flexibility and extensibility.
- **Reusability**: Components with well-defined dependencies can be reused more easily in different parts of an application or even in different projects.

DI is often used in conjunction with a Dependency Injection Container or Framework (e.g., Spring, Guice, Dagger), which automates the process of managing and providing dependencies to classes. These containers configure and wire up the dependencies based on configuration files or annotations, further simplifying the development process.

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

The @Autowired annotation in Spring Boot is used to automatically inject (or wire) a dependency into a Spring bean. It's one of the most commonly used annotations for achieving dependency injection in Spring-based applications. The primary purpose of @Autowired is to simplify and automate the process of wiring dependencies, making it easier to manage the relationships between Spring beans.

Here's how @Autowired works and its main purposes:

1. **Automatic Dependency Injection:** When you annotate a field, constructor, or method with @Autowired, Spring Boot automatically resolves and injects the appropriate dependency at runtime. You don't need to manually instantiate or look up dependencies; Spring takes care of it for you.
2. **Eliminates Boilerplate Code:** It eliminates the need for writing boilerplate code for dependency injection, such as creating constructors or setter methods to inject dependencies manually.

3. **Type-Based Injection:** @Autowired performs type-based injection, which means it looks for a bean of the same type (or a subtype) and injects it. If there is exactly one matching bean, it's injected. If there are multiple candidates, Spring may throw an exception unless you specify additional criteria or use qualifiers.
4. **Constructor Injection:** You can use @Autowired on a constructor to indicate constructor-based injection. This is considered a recommended approach because it ensures that an object is fully initialized when it's created.

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

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

5.Field Injection: You can use **@Autowired** on fields to inject dependencies directly into fields. While convenient, it's worth noting that field injection is less testable than constructor injection.

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

6.Method Injection: You can use **@Autowired** on setter methods to indicate method-based injection. This is less common but is sometimes used when you need to inject dependencies dynamically after bean creation.

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

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

7.Qualifiers and Names: If you have multiple beans of the same type and want to specify which one to inject, you can use qualifiers or **@Qualifier** annotations.

```
@Service
public class MyService {
    private final MyRepository primaryRepository;
    private final MyRepository secondaryRepository;

    @Autowired
    public MyService(
        @Qualifier("primaryRepository") MyRepository primaryRepository,
        @Qualifier("secondaryRepository") MyRepository secondaryRepository
    ) {
        this.primaryRepository = primaryRepository;
        this.secondaryRepository = secondaryRepository;
    }
}
```

@Autowired simplifies the process of injecting dependencies in Spring Boot applications, reducing the need for manual configuration and boilerplate code. It promotes clean and maintainable code by managing the relationships between Spring beans automatically. However, it's essential to understand how it works and when to use it appropriately to avoid common pitfalls, such as circular dependencies and ambiguity in injection.

3. Explain the concept of Qualifiers in Spring Boot.

In Spring Boot, qualifiers are used to specify which bean should be injected when there are multiple beans of the same type in the application context. When you have multiple beans of the same type and want to disambiguate and specify which one should be injected into a particular dependency, you can use qualifiers to provide additional information to the Spring container.

Here's a more detailed explanation of the concept of qualifiers in Spring Boot:

1. **Problem of Ambiguity:** Consider a scenario where you have multiple beans of the same type. Without qualifiers, Spring may not know which bean to inject, leading to ambiguity and potentially causing errors.

```
@Component
public class DataSourceConfig {
   // Define multiple data source beans of the same type here
}
```

2.Using Qualifiers: Qualifiers are annotations or strings that you can attach to a dependency to indicate which specific bean should be injected. They provide a hint to the Spring container about which bean to choose among multiple candidates of the same type.

```
@Service
public class MyService {
   private final MyRepository primaryRepository;
   private final MyRepository secondaryRepository;

   @Autowired
   public MyService(
      @Qualifier("primaryRepository") MyRepository primaryRepository,
      @Qualifier("secondaryRepository") MyRepository secondaryRepository
   ) {
      this.primaryRepository = primaryRepository;
      this.secondaryRepository = secondaryRepository;
   }
}
```

3.Using @Qualifier Annotation: The **@Qualifier** annotation is a commonly used qualifier in Spring Boot. You can apply it to a constructor parameter, field, or method parameter to specify the bean name or value that should be used as a qualifier.

```
@Autowired
public MyService(
   @Qualifier("primaryRepository") MyRepository primaryRepository,
   @Qualifier("secondaryRepository") MyRepository secondaryRepository
) {
    this.primaryRepository = primaryRepository;
    this.secondaryRepository = secondaryRepository;
}
```

4.Bean Names as Qualifiers: By default, Spring Boot uses the bean names as qualifiers. If you define your beans with specific names and use those names as qualifiers, Spring will use the names to determine which bean to inject.

```
@Component("primaryRepository")
public class PrimaryRepositoryImpl implements MyRepository {
    // ...
}

@Component("secondaryRepository")
public class SecondaryRepositoryImpl implements MyRepository {
    // ...
}
```

5.Custom Qualifiers: You can also define custom qualifiers by creating your own annotations and using them as qualifiers. This is useful when you want to provide more meaningful and context-specific qualifiers in your code.

```
@Qualifier("customDataSource")
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE, ElementType.ANNOTATION_TYPE })
public @interface CustomDataSource {
}
@Autowired
public MyService(@CustomDataSource MyRepository customRepository) {
    this.customRepository = customRepository;
}
```

6.Primary Bean: Additionally, you can mark one of the beans as the primary bean using the **@Primary** annotation. When you have multiple beans of the same type and one of them is marked as primary, Spring will automatically inject the primary bean if a qualifier is not specified.

```
@Component
```

```
@Primary

public class PrimaryRepositoryImpl implements MyRepository {

   // ...

}
```

Qualifiers in Spring Boot are used to specify which bean should be injected when there are multiple beans of the same type. They help resolve ambiguity in dependency injection and provide more fine-grained control over which bean should be used in a particular context. You can use the **@Qualifier** annotation provided by Spring or create custom qualifiers to suit your application's needs.

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

Spring Boot provides several ways to perform dependency injection, allowing you to inject dependencies into your Spring beans. Dependency injection is a core concept in Spring Boot, and it helps manage the relationships between components in your application. Here are the different ways to perform dependency injection in Spring Boot:

1. **Constructor Injection:** This is the most recommended and widely used method for dependency injection in Spring Boot. You inject dependencies by defining a constructor in your bean and annotating it with @Autowired. Spring automatically provides the required dependencies when creating the bean.

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

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

2.Field Injection: In field injection, you inject dependencies directly into class fields. While it's convenient, it's considered less preferable compared to constructor injection because it can make your code less testable and harder to identify dependencies

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

3.Setter Injection: This method involves creating setter methods for your dependencies and annotating them with @Autowired. Spring will call the setter methods to inject the dependencies after the bean is created.

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

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

4.Method Injection: Similar to setter injection, method injection allows you to inject dependencies using methods other than constructors. You can annotate any method with **@Autowired** to indicate that it should be used for dependency injection.

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

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

5.Interface-Based Injection: You can use the @Autowired annotation on interfaces. Spring Boot will search for a bean that implements the interface and inject it. This is useful when you have multiple implementations of an interface and need to specify which one to inject.

```
public interface PaymentGateway {
    void processPayment();
}

@Service
public class CreditCardPayment implements PaymentGateway {
    // ...
}

@Service
public class PayPalPayment implements PaymentGateway {
    // ...
}

@Service
public class PaymentService {
    private final PaymentGateway paymentGateway;

    @Autowired
    public PaymentService(PaymentGateway paymentGateway) {
        this.paymentGateway = paymentGateway;
    }
}
```

6.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 based on a qualifier value or name. This is useful for disambiguating dependencies.

```
@Service

public class MyService {

 private final MyRepository primaryRepository;

private final MyRepository secondaryRepository;

@Autowired

public MyService(

 @Qualifier("primaryRepository") MyRepository primaryRepository,

@Qualifier("secondaryRepository") MyRepository secondaryRepository

) {

this.primaryRepository = primaryRepository;

this.secondaryRepository = secondaryRepository;

            }
        }
```

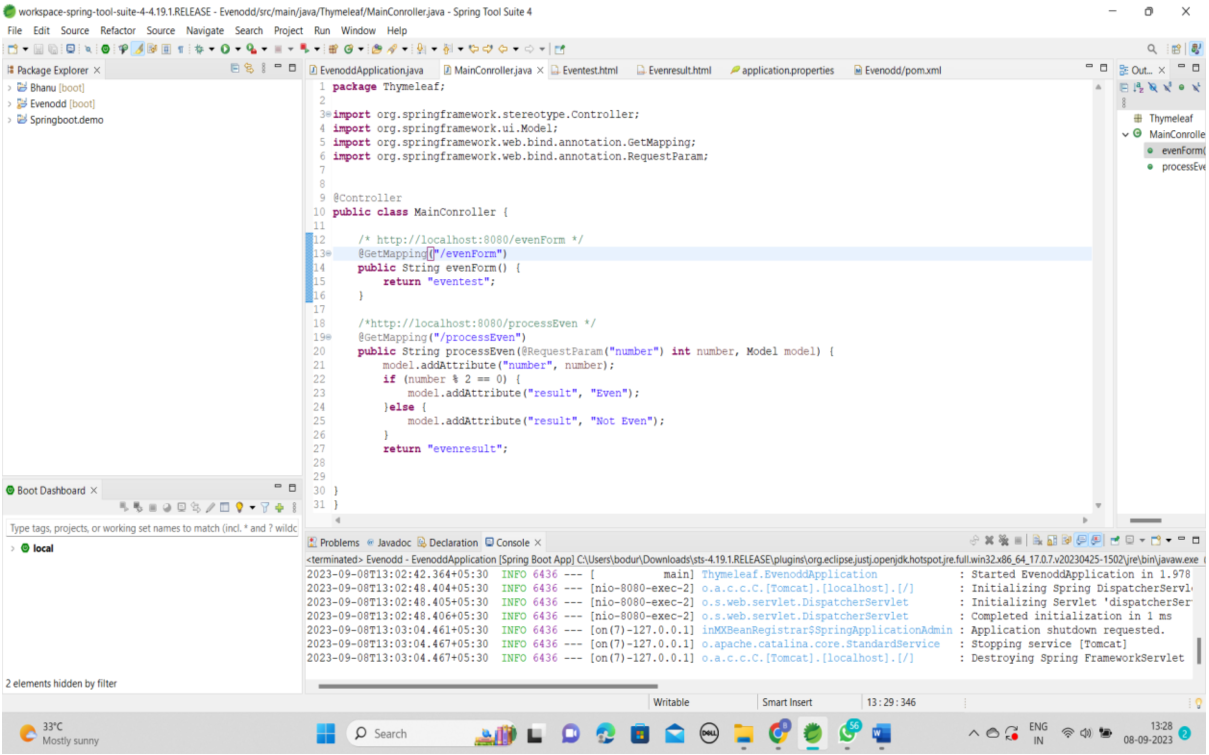
7.Primary Bean: You can mark one of the beans of a particular type as the primary bean using the @Primary annotation. When there are multiple candidates for injection, the primary bean is chosen by default unless a qualifier is specified.

```
        @Component
        @Primary
        public class PrimaryRepositoryImpl implements MyRepository {
           // ...
        }
```

These are the primary ways to perform dependency injection in Spring Boot. Constructor injection is generally considered the best practice because it promotes immutability and helps avoid circular dependencies. However, the choice of injection method may depend on your specific use case and design preferences.

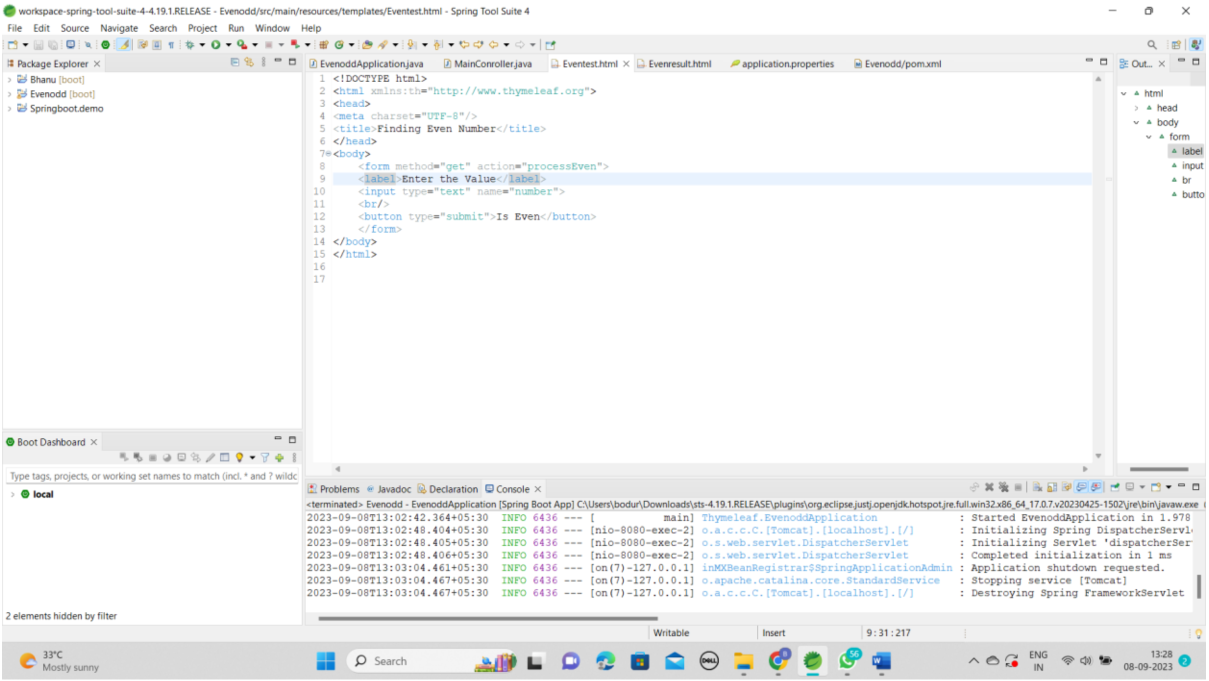
5. Create a SpringBoot application with MVC using Thymeleaf.

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



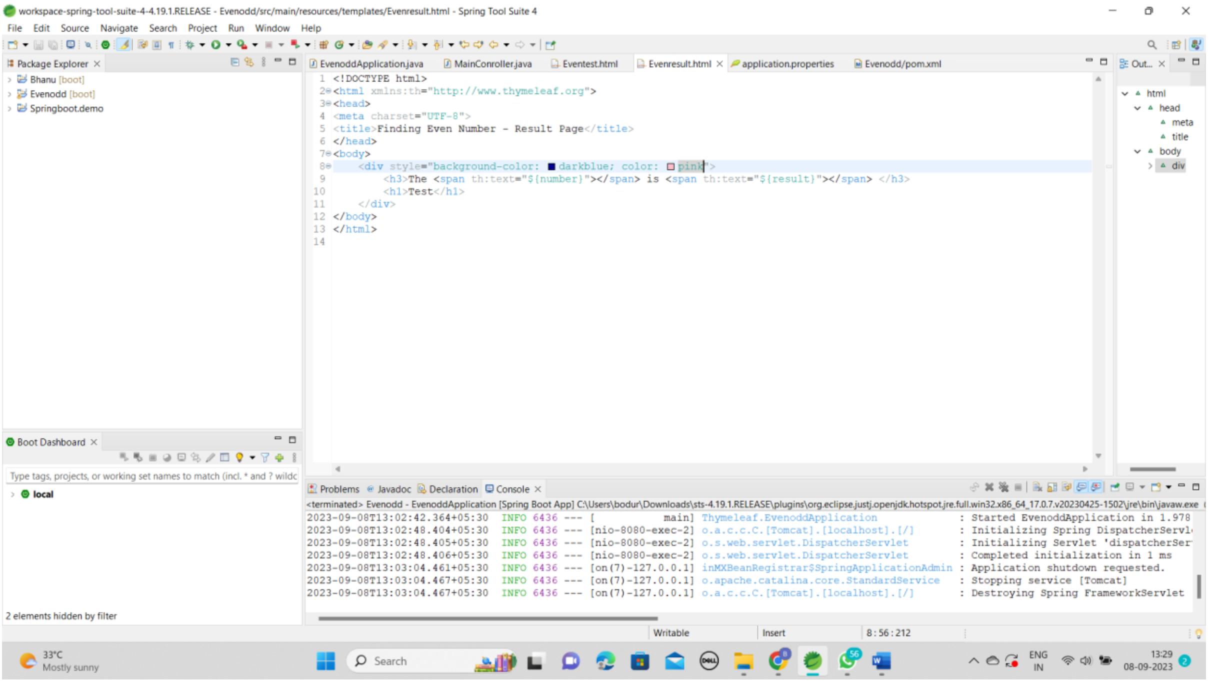
```java
package Thymeleaf;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;


@Controller
public class MainConroller {

    /* 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";


}
}
```

```
<terminated> Evenodd - EvenoddApplication [Spring Boot App] C:\Users\bodur\Downloads\sts-4.19.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe
2023-09-08T13:02:42.364+05:30  INFO 6436 --- [           main] Thymeleaf.EvenoddApplication             : Started EvenoddApplication in 1.978
2023-09-08T13:02:48.404+05:30  INFO 6436 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServl
2023-09-08T13:02:48.405+05:30  INFO 6436 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherSer
2023-09-08T13:02:48.406+05:30  INFO 6436 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
2023-09-08T13:03:04.461+05:30  INFO 6436 --- [on(7)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.
2023-09-08T13:03:04.467+05:30  INFO 6436 --- [on(7)-127.0.0.1] o.apache.catalina.core.StandardService   : Stopping service [Tomcat]
2023-09-08T13:03:04.467+05:30  INFO 6436 --- [on(7)-127.0.0.1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Destroying Spring FrameworkServlet
```



```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>
```

```
<terminated> Evenodd - EvenoddApplication [Spring Boot App] C:\Users\bodur\Downloads\sts-4.19.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe
2023-09-08T13:02:42.364+05:30  INFO 6436 --- [           main] Thymeleaf.EvenoddApplication             : Started EvenoddApplication in 1.978
2023-09-08T13:02:48.404+05:30  INFO 6436 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServl
2023-09-08T13:02:48.405+05:30  INFO 6436 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherSer
2023-09-08T13:02:48.406+05:30  INFO 6436 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
2023-09-08T13:03:04.461+05:30  INFO 6436 --- [on(7)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.
2023-09-08T13:03:04.467+05:30  INFO 6436 --- [on(7)-127.0.0.1] o.apache.catalina.core.StandardService   : Stopping service [Tomcat]
2023-09-08T13:03:04.467+05:30  INFO 6436 --- [on(7)-127.0.0.1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Destroying Spring FrameworkServlet
```

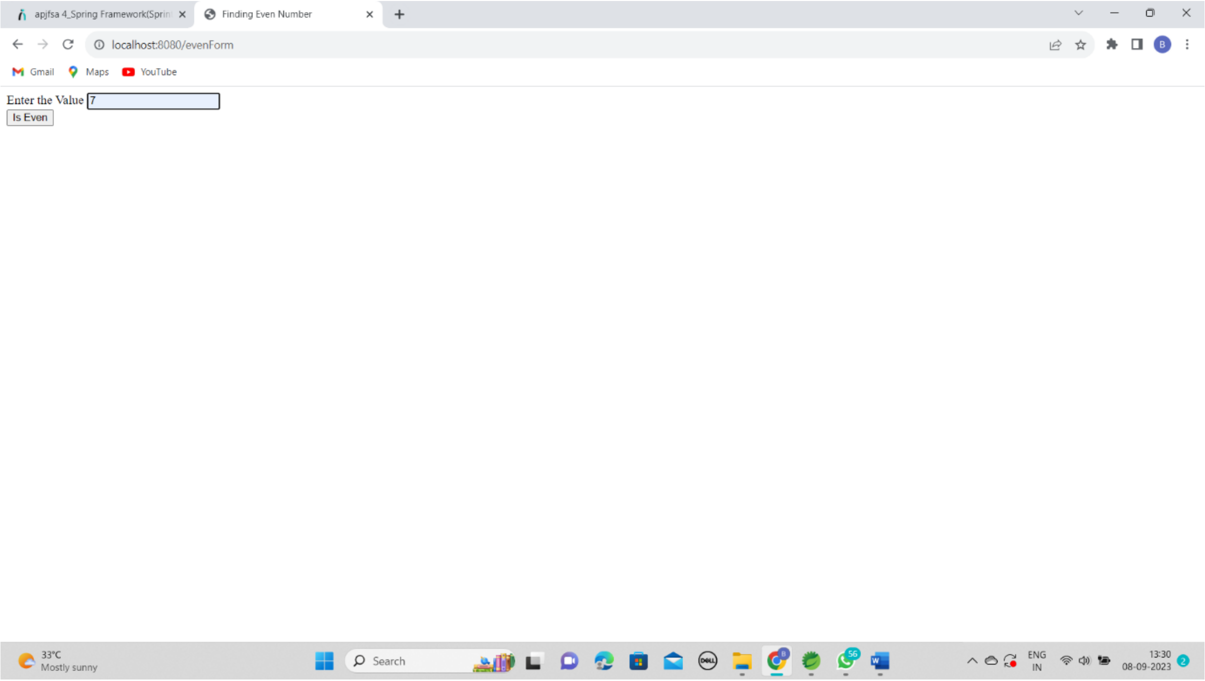


workspace-spring-tool-suite-4-4.19.1.RELEASE - Evenodd/src/main/resources/templates/Evenresult.html - Spring Tool Suite 4

```
<!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: darkblue; color: pink">
        <h3>The <span th:text="${number}"></span> is <span th:text="${result}"></span> </h3>
        <h1>Test</h1>
    </div>
</body>
</html>
```

```
<terminated> Evenodd - EvenoddApplication [Spring Boot App] C:\Users\bodur\Downloads\sts-4.19.1.RELEASE\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.7.v20230425-1502\jre\bin\javaw.exe
2023-09-08T13:02:42.364+05:30  INFO 6436 --- [           main] Thymeleaf.EvenoddApplication             : Started EvenoddApplication in 1.978
2023-09-08T13:02:48.404+05:30  INFO 6436 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServl
2023-09-08T13:02:48.405+05:30  INFO 6436 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherSer
2023-09-08T13:02:48.406+05:30  INFO 6436 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet        : Completed initialization in 1 ms
2023-09-08T13:03:04.461+05:30  INFO 6436 --- [on(7)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.
2023-09-08T13:03:04.467+05:30  INFO 6436 --- [on(7)-127.0.0.1] o.apache.catalina.core.StandardService   : Stopping service [Tomcat]
2023-09-08T13:03:04.467+05:30  INFO 6436 --- [on(7)-127.0.0.1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Destroying Spring FrameworkServlet
```

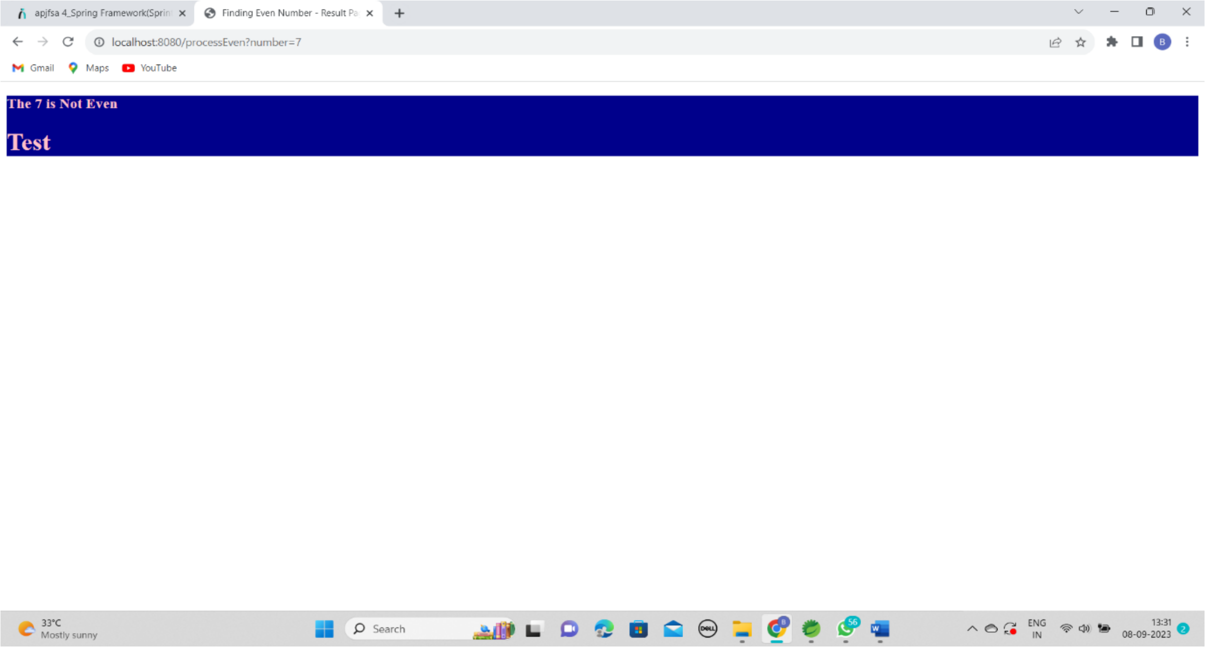
Output:



localhost:8080/evenForm

Enter the Value 7

Is Even



apjfsa 4_Spring Framework(Sprin
Finding Even Number - Result Pa
localhost:8080/processEven?number=7
Gmail
Maps
YouTube
33°C
Mostly sunny
Search
ENG
IN
13:31
08-09-2023

The 7 is Not Even

Test