What is Dependency Injection (DI)?

In Spring Boot, Dependency Injection (DI) is a fundamental concept used for managing and injecting dependencies into your application components, such as beans, services, controllers, and repositories. It is an essential part of the Spring framework and is used to achieve Inversion of Control (IoC) and facilitate the creation and management of Spring beans.

Here's how Dependency Injection works in the context of Spring Boot:

1. **IoC Container**: Spring Boot provides an IoC container, often referred to as the Spring container. This container is responsible for managing the lifecycle and dependencies of Spring beans.
2. **Bean Definitions**: In Spring Boot, you define beans by using annotations like **@Component**, **@Service**, **@Repository**, or **@Controller**. These annotations mark classes as Spring-managed beans.
3. **Automatic Bean Scanning**: Spring Boot's IoC container automatically scans your application's classpath for classes marked with these annotations. When it finds such classes, it creates instances of them and manages their lifecycle.
4. **Dependency Injection**: Spring Boot uses Dependency Injection to inject dependencies into beans. This can be done through constructor injection, setter injection, or field injection, depending on your preference and coding style.

Benefits of Dependency Injection in Spring Boot:

1.Modularity: Dependencies are clearly defined and can be easily swapped or extended without modifying the dependent classes.

2.Testability: It's easy to write unit tests for Spring Boot components because you can provide mock or stub implementations of dependencies during testing.

3.Maintainability: Dependency Injection promotes loosely coupled components, making it easier to maintain and extend your application.

4.Reusability: Components can be reused in different parts of your application or in other applications with minimal changes.

5.Configuration Flexibility: You can configure dependencies externally (e.g., in configuration files) without modifying the code.

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

In Spring Boot, the @Autowired annotation is used for automatic dependency injection. Its purpose is to inform the Spring framework that a particular class field, constructor, or method parameter should be automatically populated with an instance of a Spring-managed bean of the same type or a compatible type.

Here are the key purposes and benefits of using the @Autowired annotation in Spring Boot:

1. Dependency Injection: The primary purpose of @Autowired is to inject dependencies into Spring components (such as beans, services, controllers, or repositories). When you

annotate a field, constructor, or method parameter with @Autowired, Spring Boot automatically resolves and injects the required dependencies.

2. Reduces Boilerplate Code: It reduces the need for boilerplate code related to manually looking up or instantiating dependencies. Spring takes care of wiring components together, simplifying your codebase.
3. Promotes Loose Coupling: By using @Autowired, you specify the interface or abstract type of the dependency you need rather than referencing a specific implementation. This promotes loose coupling, making your code more flexible and maintainable.
4. Testability: It greatly facilitates unit testing. During testing, you can provide mock or stub implementations for dependencies, allowing you to isolate the code under test and verify its behavior.

Explain the concept of Qualifiers in Spring Boot.

In Spring Boot, qualifiers are a way to disambiguate and specify which bean should be injected when there are multiple beans of the same type available in the Spring application context. Qualifiers are used in conjunction with the @Autowired annotation or constructor injection to specify which bean should be injected when there is more than one bean of the same type.

Here's how qualifiers work in Spring Boot:

1. Multiple Beans of the Same Type: In some cases, you may have multiple beans of the same type registered in the Spring application context. For example, you might have multiple implementations of an interface or multiple beans of the same class.
2. Ambiguity: When you attempt to inject such a bean using @Autowired or constructor injection, Spring Boot might not be able to determine which bean should be injected, and this results in ambiguity.

Qualifiers: Qualifiers are used to provide additional information to Spring so that it can resolve this ambiguity. You can add the @Qualifier annotation along with @Autowired or constructor injection to specify the name or value of the qualifier that should be used to identify the desired bean.

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 and inject dependencies into your application components. There are several ways to perform dependency injection in Spring Boot:

Constructor Injection:

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

You declare dependencies as parameters in the constructor of a class.

Spring Boot automatically resolves and injects the required dependencies when creating an instance of the class.

Example Structure:

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

Setter Injection:

In this approach, you define setter methods for your dependencies.

Spring Boot will use these setter methods to inject dependencies after creating an instance of the class.

Example Structure:

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

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

Field Injection:

While field injection is possible, it's generally discouraged because it tightly couples your classes to the Spring framework and makes testing more difficult.

You can annotate fields with @Autowired to indicate that Spring should inject the dependencies directly into the fields.

Example Structure:

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

Method Injection:

You can also use method injection by annotating methods with @Autowired.

Spring Boot will call these methods to inject dependencies after creating an instance of the class.

Example Structure:

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

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

Interface-Based Injection:

You can create interfaces for your dependencies and then use constructor injection with those interfaces.

This allows for greater flexibility and easier testing through the use of mock implementations.

Example Structure:

```
public interface MyRepository {
    // ...
}
@Repository
public class MyRepositoryImpl implements MyRepository {
 // ...
}
@Service
public class MyService {
    private final MyRepository repository;

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

    // ...
}
```

Qualifiers:

When you have multiple beans of the same type, you can use the @Qualifier annotation in conjunction with the injection annotations (@Autowired, @Inject) to specify which bean should be injected.

Example Structure:

```
@Service
public class MyService {
    private final MyRepository mysqlRepository;
    @Autowired
    public MyService(@Qualifier("mysqlRepository") MyRepository repository) {
        this.mysqlRepository = repository;
```

```
    }


    // ...
}
```

Constructor-Based Injection with Lombok:

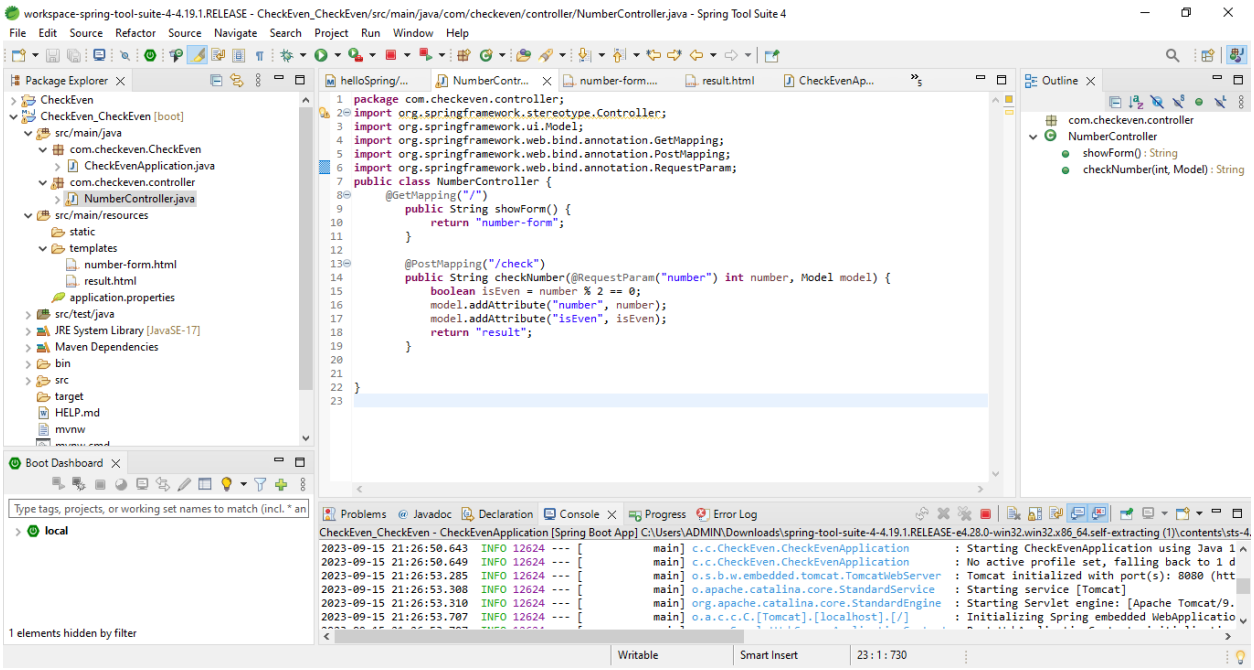
If you're using Project Lombok, you can simplify constructor injection using the @RequiredArgsConstructor annotation.

Example Structure:

```
@Service
@RequiredArgsConstructor
public class MyService {
    private final MyRepository repository;
    // ...
}
}
```

Create a SpringBoot application with MVC using Thymeleaf.

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



Web page