7. Dependency Injection in Spring boot | With Advantages and Disadvantages

What is Dependency Injections ?

Let's see the problem first.

```
public class User {

    Order order = new Order();

    public User(){
        System.out.println("initializing User");
    }


}
```

```
public class Order{
    public Order(){
        System.out.println("initializing Order");
    }
}
```

Problems:

- User is dependent on Order so that's Why User creating new instance of Order.

1. Both User and Order class are tightly coupled.

Suppose, Order object creation logic get's changed [let's say in future Object becomes an interface and it has many concrete class], then User class has to be changed too.

```
public interface Order{

}
```

```
publi class onlineOrder implements Order{

}
```

```
publi class offlineOrder implements Order{
```

```
}
```

```
public class User {

    //Order order = new Order();
    Order order = new onlineOrder();

    public User(){
        System.out.println("initializing User");
    }


}
```

2. It breaks Dependency Inversion rule of S.O.L.I.D principles.

This principle says DO NOT depend on concrete implementation, rather depends on abstraction.

Breaks Dependency Inversion Principle [DIP]

```
public class User {

    //Order order = new Order();
    Order order = new onlineOrder();

    public User(){
        System.out.println("initializing User");
    }
}
```

Follows Dependency Inversion

```
@Component
public class User {

    @Autowired
    Order order;

    public User(){
        System.out.println("initializing User");
    }
}
```

Now in Spring Boot how to achieve Dependency Inversion Principle.

Through Dependency Injection.

What is Dependency Injection.

- Using Dependency Injection. we can makes our class independent of it's dependencies.
- It helps to remove the dependency on concrete implementation and inject the dependencies from external sources.

```
@Component
public class User {

   @Autowired
   Order order;
}
```

```
@Component
public class User Order{
}
```

@Autowired

- First look for a bean of the require type.

If bean found, Spring will inject it.

If bean not found then create it and inject it.

Different ways of injections and which one is better ?

- Field Injection
- Setter Injection
- Constructor Injection :VVI ***Industry Uses Constructor Injection.

Field Injection

- Dependency is set into fields of the class directly.
- Spring uses reflection, it iterates over the fields and resolve the dependency.
- if any field has annotations of @Autowired Spring need to resolve the dependency.
- So, Spring called it's default constructor.
- After creation of bean it inject into that field.

```
@Component
public class User {

    //field injection
    @Autowired
    Order order;

    public User(){
        System.out.println("initializing User");
    }
}
```

```
@Component
@Lazy
public class Order{
    public Order(){
        System.out.println("initializing Order");
    }
}
```

Advantage

- Very simple and easy to use.

Dis-Advantage

- Can not be used with Immutable fields.

```
@Component
public class User {

    //field injection
    //compile time error
    @Autowired
    public final Order order;

    public User(){
        System.out.println("initializing User");
    }
}
```

In spring for Field injection it uses **REFLECTION** and during reflection it ignores the **final** keyword.

Bean will be created and Object instance ultimately injected to that **order** field.

you will not seen null.

It's break the **LAW OF IMMUTABILITY.**

Reflection Breaks Many Principles

- **Chances Of NPE [Null Pointer Exception].**

```
@Component
public class User {

    //field injection
    @Autowired
    public Order order;

    public User(){
        System.out.println("initializing User");
    }

    public void process(){
       order.process();
    }

}
```

```
//In Main method
User userObj = new User();
userObj.process();

//Exception in thread "main" java.langNullPointerException Create breakpoint
```

- **During Unit Testing, setting MOCK dependency to this field becomes difficult.**

```
@Component
public class User {

    //field injection
    @Autowired
    public Order order;

    public User(){
        System.out.println("initializing User");
    }

    public void process(){
       order.process();
    }
```

```
}
```

```
class UserTest{
   private Order orderMockObj;
   private User user;

   @BeforeEach
   pubic void setup(){
     this.orderMockObj = Mockito.mock(Order.clas);
     this.user = new User();
   }
}
```

We cannot mock **orderMockObj** because we don't have constructor present in User class that can help while creating mock object.

Spring internally called default constructor while iterating fields which is annotated with **Autowired** Mocking cannot control spring.

How to set this MOCK, we have @InjectMock annotation internally it uses reflection to create field object.

```
class UserTest{
   @Mock
   private Order orderMockObj;

   @InjectMocks
   private User user;

   @BeforeEach
   pubic void setup(){
     MockitoAnnotations.initMocks(this);
   }
}
```

Setter Injections

- Dependency is set into the fields using the setter method.
- We have to annotate the method using @Autowired.

```
@Component
@Lazy
public class Order{
    public Order(){
        System.out.println("initializing Order");
    }
}
```

```
@Component
public class User {

    //setter injection

    public Order order;

    public User(){
        System.out.println("initializing User");
    }

    //Using Setter Method for injectiong the Order field
    //You can pass onlineOrder,offlineOrderObj and Mock obj to just call this function.
    @Autowired
    public void setOrderDependency(Order order){
       this.order=order;
    }

}
```

Advantage:

- Dependency can be changed any time after the object the creation [As cannot be marked as final]
- Ease of testing, as we can pass mock object in the dependency easily.

Dis-Advantage:

- Field Can not be marked as final. [we can not make it immutable].

```
@Component
public class User {

    //setter injection
   //Compile time error
    public final Order order;
```

```
    public User(){
        System.out.println("initializing User");
    }


    @Autowired
    public void setOrderDependency(Order order){
       //compile time error
       //spring will able initialize it by using reflection whick breaks the
immutability LAW.
       this.order=order;
    }

}
```

- Difficult to read and maintained, as per standard, object should be initialized during object creation, so this might create code readability issue.

Constructor Injection

- Dependency get resolved at the time of initialization of the Object itself.
- It's recommended to use.
- it's not like after User creation it will be resolved, it's resolved at the time object initialization.

```
@Component
@Lazy
public class Order{
    public Order(){
        System.out.println("initializing Order");
    }
}
```

```
@Component
public class User {

    //Constructor injection
    public Order order;

    @Autowired
    public User(Order order){
       this.order = order;
       System.out.println("User Initialized");
```

```
    }

}
output:
//initializing Order
//User Initialized
```

When only one constructor is present then using @Autowired on constructor is not mandatory.

from Spring Version 4.3

When More than 1 constructor is present then using @Autowired on constructor is mandatory.

```
@Component
@Lazy
public class Order{
    public Order(){
        System.out.println("initializing Order");
    }
}
```

```
@Component
@Lazy
public class Invoice{
    public Order(){
        System.out.println("initializing Invoice");
    }
}
```

```
@Component
public class User {


    public Order order;
    public Invoice invoice;

    public User(Order order){
       this.order = order;
       System.out.println("User Initialized");
    }
```

```
    public User(Invoice invoice){
       this.order = order;
       System.out.println("User Initialized");
    }

}
output:
//Caused by: org.springframework.beans.BeansInstantiationException create breakpoint
failed to instantiate
```

Here you need to define or annotate On of the constructor with @Autowired otherwise Spring get's confused and throw exception

```
@Component
public class User {
    public Order order;
    public Invoice invoice;

    public User(Order order){
       this.order = order;
       System.out.println("User Initialized");
    }

    @Autowired
    public User(Invoice invoice){
       this.invoice= invoice;
       System.out.println("User Initialized");
    }

}
output:
initializing Invoice
User Initialized
```

Why Constructor Injection is Recommended [Advantage]

- All mandatory dependencies are created at the time of initialization itself. Makes 100% sure that our object is fully initialized with mandatory dependencies.

  Avoid NPE during runtime.

  Unnecessary null check can be avoided too.

- We can create immutable object using Constructor injection.

```
@Component
public class User {

    //Constructor injection
    public final Order order;

    @Autowired
    public User(Order order){
       this.order = order;
       System.out.println("User Initialized");
    }

}
output:
//initializing Order
//User Initialized
```

```
@Component
public class User {

    //Field injection
    //Compile time error
    @Autowired
    public final Order order;

    public User(){
       System.out.println("User Initialized");
    }

}
```

- Fail Fast: If there is any missing dependency, it will fail during compilation itself, rather than failing during runtime.

Here what happens spring create object of User and call default constructor and @Autowired is missing order object will not injected by Order instance it will initialize with null.

If Someone try to access order Object within User class it will throw compile time error.

```
@Component
public class User {


    public Order order;
```

```
    public User(){
       System.out.println("User Initialized");
    }

    @PostConstruct
    public void init(){
      System.out.println(order==null);
    }

}
```

Here we are missing @Autowired on constructor.

During Constructor Injection we have only on constructor so, spring not get confused and object will be created and injected to order field.

If more than one constructor spring will fail and get's confused. and at this point of we have to annotate with @Autowired on constructor.

It will fail fast if Order Bean is missing.

Means Order class is not annotate with @Component or @Configuration and Custom bean is not created so spring will failed to create bean of order object.

While Constructor injection Bean not found corresponding to Order so, **that will be FAIL FAST.**

```
@Component
public class User {


    public Order order;


    public User(Order order){
       this.order = order;
       System.out.println("User Initialized");
    }

    @PostConstruct
    public void init(){
      System.out.println(order==null);
    }
}
output:
initializing Invoice
User Initialized
```

- **Unit Testing is easy**

```
@Component
public class User {


    public Order order;

    @Autowired
    public User(Order order){
       this.order = order;
       System.out.println("User Initialized");
    }


    public void process(){
      order.process();
    }
}
output:
initializing Invoice
User Initialized
```

```
Here you don't need Mock reflection for creating or mocking object's using annotation
like @Mock,@InjectMocks
```

```
class UserTest{
   private Order orderMockObj;
   private User user;

   @BeforeEach
   pubic void setup(){
     this.orderMockObj = Mockito.mock(Order.class);
     this.user = new User();
   }
}
```

[Dis-Advantages]:

1. If we 20 fields then constructor parameter have 20 parameter which is not readable.
2. This good too because you need to refactor your code.

Common Issues when dealing with Dependency Injection.

1. **CIRCULAR DEPENDENCY.**

- Invoice is dependent on order and order dependent on Invoice.
- they are both using @Autowired.

```
@Component
public class Invoice{

    @Autowired
    Order order

    public Invoice(){
        System.out.println("initializing Invoice");
    }
}
```

```
@Component
public class Order{
    @Autoired
    Invoice invoice;
    public Order(){
        System.out.println("initializing Order");
    }
}
```

SOLUTION:

1. First and foremost, can we refactor the code and remove this cycle dependency.

- For example, common code in which both are dependent, can be taken out to separate class. this way we can break the circular dependency.
- This is recommended one.

2. Using @Lazy on @Autowired annotation.

- Spring will create proxy bean instead of creating the bean instance immediately during application startup.

@Lazy on Field Injection

```
@Component
@Lazy
```

```
public class Order{

    public Order(){
        System.out.println("initializing Order");
    }
}
```

```
@Component
public class Invoice{

    //@Lazy
    @Autowired
    public Order order

    public Invoice(){
        System.out.println("initializing Invoice");
    }
}
```

Output:

initializing Invoice

initializing order

Next Example Of @Lazy

```
@Component
@Lazy
public class Order{

    public Order(){
        System.out.println("initializing Order");
    }
}
```

@Lazy on @Autowired filed of order

```
@Component
public class Invoice{

    @Lazy
    @Autowired
    public Order order

    public Invoice(){
        System.out.println("initializing Invoice");
```

```
    }
}
```

Output:

initializing Invoice

So, Here what happens spring find @Component with Order class but it also annotated with @Lazy so bean will not created for order class.

next spring find Invoice class with annotation with **@Component** so bean will be created after that spring acquires Order Field with @Autowired annotation spring try create bean for this field but again it acquires **@Lazy** annotation on **@Autowired** so spring not creating bean for the same and put some **proxy**

Now We can use @Lazy to resolve circular dependency

```
@Component
public class Order{

    @Autoired
    Invoice invoice;

    public Order(){
        System.out.println("initializing Order");
    }
}
```

```
@Component
public class Invoice{

    @Lazy
    @Autowired
    public Order order

    public Invoice(){
        System.out.println("initializing Invoice");
    }
}
```

output

initializing Invoice

@Lazy On Setter Injection

```
@Component
public class Order{
```

```
    Invoice invoice;

    public Order(){
        System.out.println("initializing Order");
    }

    @Autowired
    public void setInvoice(Invoice invoice){
      this.invoice=invoice;
    }

}
```

```
@Component
public class Invoice{


    public Order order

    public Invoice(){
        System.out.println("initializing Invoice");
    }

    @Lazy
    @Autowired
    public void setOrder(Order order){
      this.order=order;
    }
}
```

3. **Using @PostConstruct**

- Not recommended way

```
@Component
public class Order{

    @Autowired
    Invoice invoice;

    public Order(){
        System.out.println("initializing Order");
    }

    @postConstruct
```

```
    public void intialize(){
      invoice.setOrder(this);
    }

}
```

```
@Component
public class Invoice{



    public Order order

    public Invoice(){
        System.out.println("initializing Invoice");
    }

    public void setOrder(Order order){
      this.order=order;
    }
}
```

So, firstly order get's initialized by spring and then spring encounters @Autowired field then Invoice is also get's initialized.

After Bean construction @postConstruct method **intialize()** called from Order class and it will call the **setOrder()** method of Invoice class and it will initialize the Order field of initalize class.

UNSATISFIED DEPENDENCY

Problem.

```
@Component
public class User {
    @Autowired
    public Order order;
    public User(){
       System.out.println("User Initialized");
    }
}
```

```
public interface Order{
}
```

```
public class onlineOrder implements Order{
}
```

```
public class offlineOrder implements Order{
}
```

- Application Failed To Start
- Unsatisfied Dependency Exception
- Not able to construct bean because spring get's confused whether Online Order to put or Offline Order.

Solution

1. **@Primary annotation**

@Primary annotation tells spring gives first priority. to this concrete class.

```
public interface Order{
}
```

```
@Primary
@Component
public class onlineOrder implements Order{
  public onlineOrder(){
     System.out.println("Online Order initialized");
  }
}
```

```
@Component
public class offlineOrder implements Order{
  public offlineOrder (){
     System.out.println("offlineOrder initialized");
  }
}
```

```
@Component
public class User {
    @Autowired
    public Order order;//Onlineorder inint
    public User(){
       System.out.println("User Initialized");
    }
```

```
}
ouput
offlineOrder initialized
Online Order initialized
User Initialized
```

2. @Qualifier annotation

```
@Qualifier("onlineOrderName")
@Component
public class onlineOrder implements Order{
  public onlineOrder(){
     System.out.println("Online Order initialized");
  }
}
```

```
@Qualifier("offlineOrderName")
@Component
public class offlineOrder implements Order{
  public onlineOrder(){
     System.out.println("offlineOrder  initialized");
  }
}
```

```
@Component
public class User {
    @Qualifier("offlineOrderName")
    @Autowired
    public Order order;//offlineOrder inint
    public User(){
       System.out.println("User Initialized");
    }
}
ouput
offlineOrder initialized
Online Order initialized
User Initialized
```