Spring Framework

Framework: It allows you to create the complex applications in a simple way, because it takes care of all the common features every application needs so that developers can concentrate on the new functionalities of the project, Framework forces developers to implement the application that is easy to test & maintain, some of the common features the framework provides are:-

1. Exception handling
2. Type conversion
3. Design patterns - related to object creation, initializing the object
4. Internationalization - adapting the applications to specific region or the language
5. Structuring the application - Configurations, Coding, testing and so on

Spring Framework: It is an application framework which can be used to develop any kind of applications like desktop, mobile, web, enterprise, cloud based

It has provided many modules for different types of applications, each modules are independent from other modules so that you can use the ones you need

ex: If you want to connect to the database you can use spring-jdbc module, spring-orm module, spring-data module and so on.

Some of the important modules are:

1. Spring Core: Provides all the basic features or necessities for the application - design patterns, type conversion, exception handling, object creation & initialization, internationalization
2. Spring MVC: For web application & rest based applications
3. Spring Boot: To easily setup the spring related features
4. Spring Cloud: For cloud based applications like microservices
5. Spring Security: Token based Authentication & Authorization (OAuth2.0)
6. Spring AOP: For Cross Cutting Concerns

Spring Core: It is one of the core module in spring that takes care of creating & maintaining the object using Dependency Injection feature, apart from that it provides many design patterns to easily develop the application

Dependency Injection: It is a design pattern where object is supplied to another dependent object ex: Service object supplying to controller, dao object supplying to service, transaction object supplying to service and so on.

Spring IoC: IoC stands for Inversion of Control, it is a container that maintains the objects, since the process of object creation and initializing is reverse it is called as Inversion of Control

Note: Dependency Injection is another way of initializing the object i.e., framework takes care of initializing the object or initializing the dependencies

There are 2 types of dependency injection

1. Setter Injection: Framework supplies the dependencies through setter methods
2. Constructor Injection: Framework supplies the dependencies through Constructor arguments

Benefits of Dependency Injection

* Loosely coupled code
* Flexible to the changes in the implementation

Note: You must use interface based approach when you want to get the full advantage of dependency injection

Usual way of initializing the dependencies i.e., dao of Service that needs instance of EmployeeDao

1. EmployeeDao dao = new EmployeeJdbcImpl();
2. EmployeeDao dao = ObjectFactory.getDaoInstance();

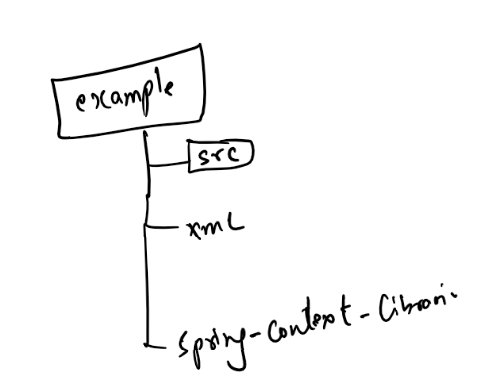
Both the references are going to be initialized by the 2nd statement is loosely coupled compare to the first statement, in the 2nd statement if the implementation object needs to be changed then we only change the factory pattern class no the service layer code

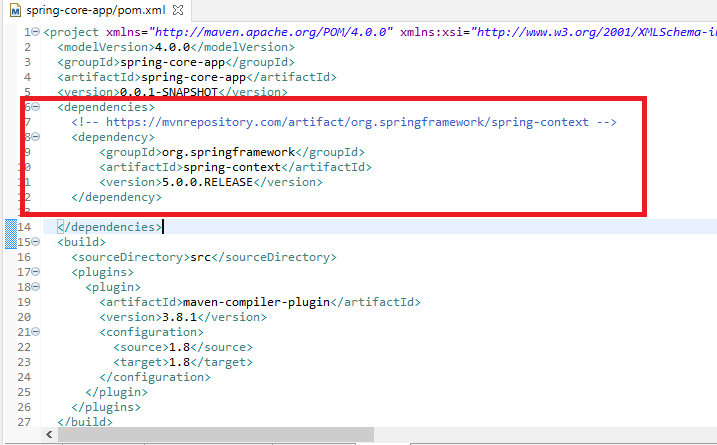
Dependency Injection way of initializing the dependencies

1. EmployeeService(EmployeeDao dao) { this.dao = dao; }
2. setDao(EmployeeDao dao) { this.dao = dao; }

Both the code are called by Framework by supplying the instance of dao, here the framework creates the instance of EmployeeDao & EmployeeService & supplies the EmployeeDao instance to the EmployeeService

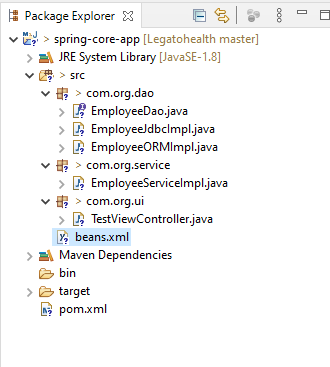
Both the statements are loosely coupled & flexible to the changes as you never create object of either EmployeeService or EmployeeDao because framework takes care of object creation.



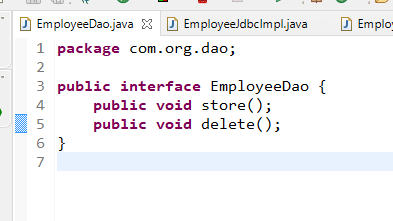


Setter Injection: Object initialization happens via setter method

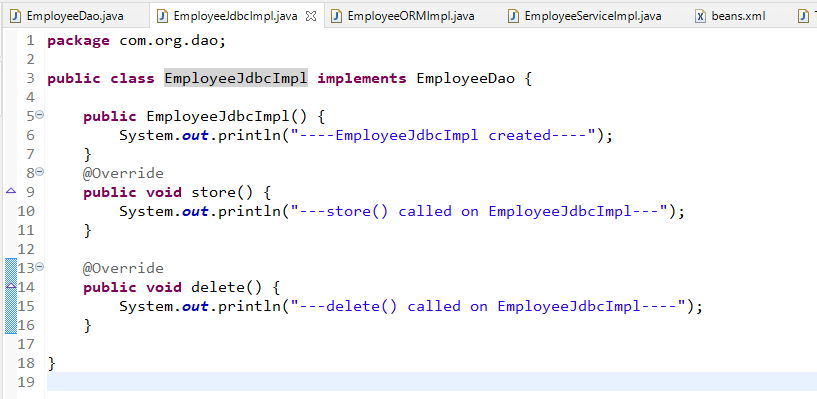
Project Structure:



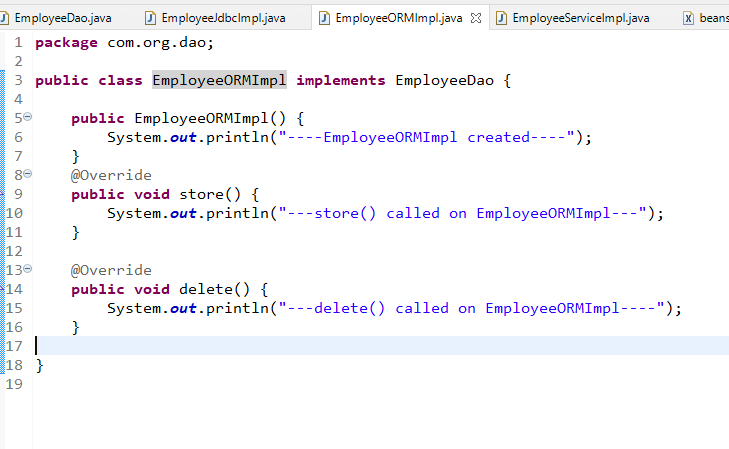
EmployeeDao.java



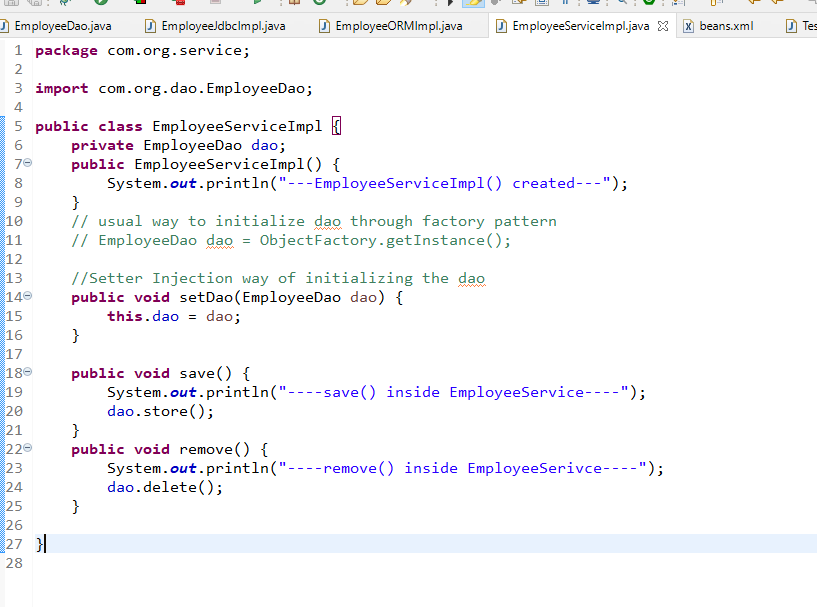
EmployeeJdbcImpl.java



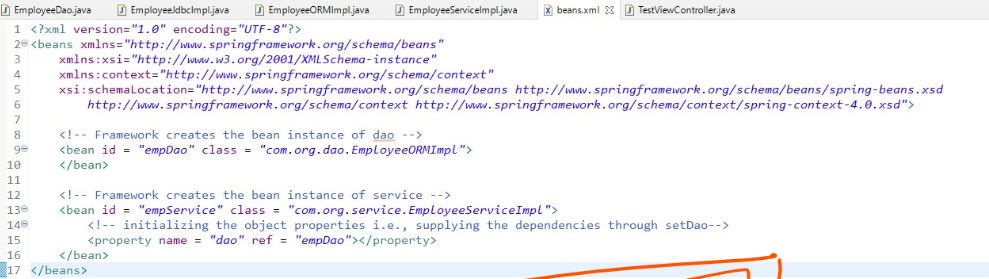
EmployeeORMImpl.java

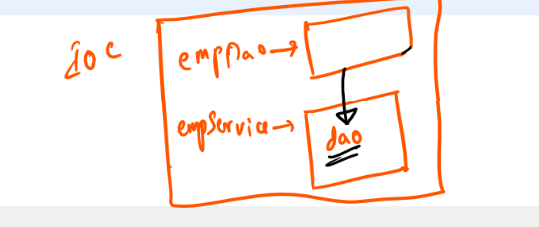


EmployeeServiceImpl.java

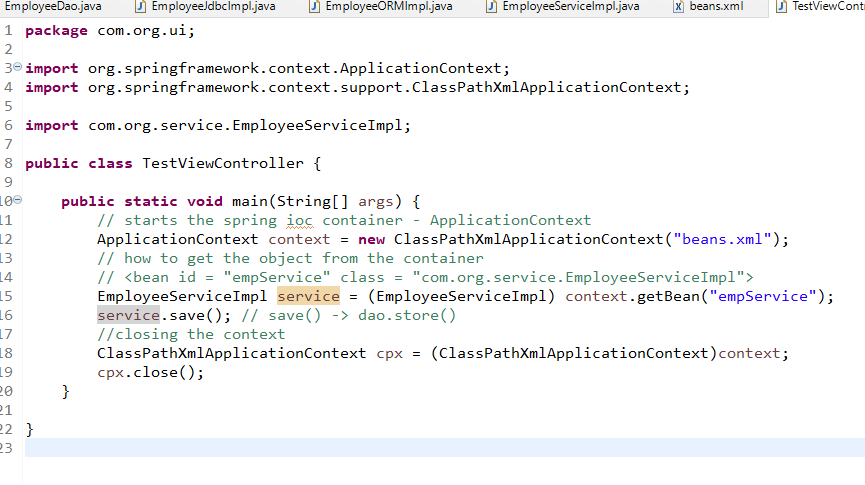


beans.xml

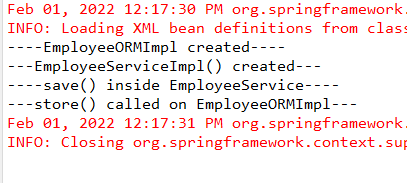




TestViewController.java



Output:

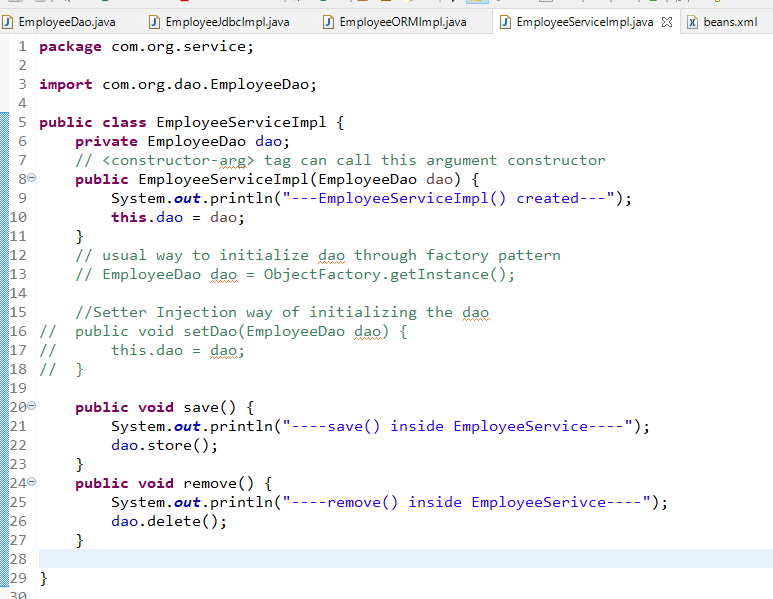


The above code is supplying the EmployeeDao implementation instance using setter injection i.e., it is calling setDao() method & passing the object that implements EmployeeDao, but another way of supplying the dependency is via constructor argument.

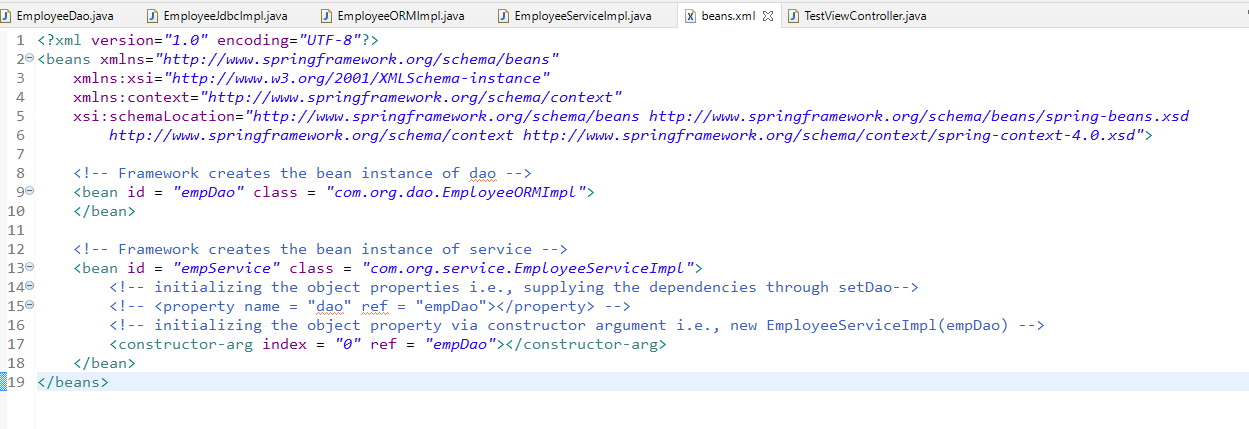
Constructor Injection: Object initialization happens via constructor argument

Sometimes you may have a class where the properties must be initialized only through constructors and after the object is created nobody needs to modify its property in that case you there wouldn’t be setter methods for that property, in such cases you can initialize only by constructors

Change the EmployeeServiceImpl.java

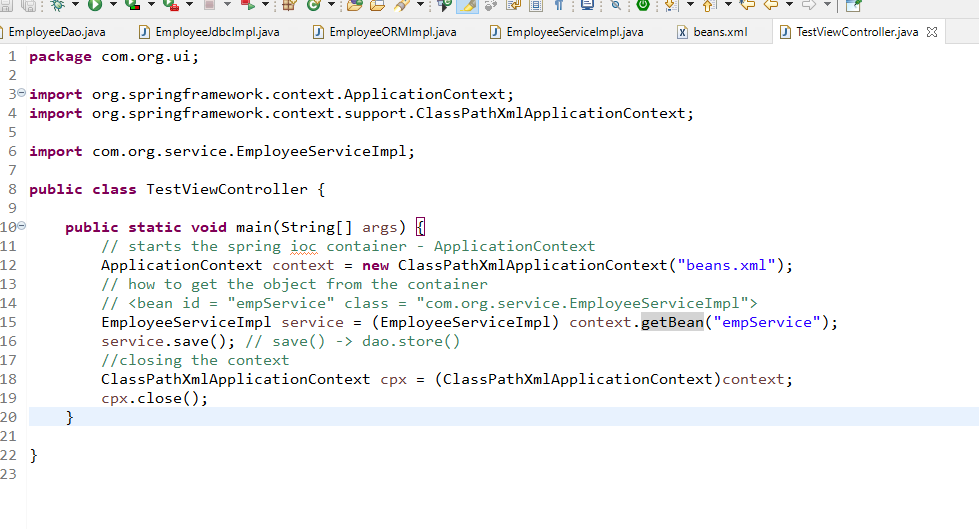


beans.xml



The above configuration supplies the dao to the service using argument constructor,

TestViewController.java

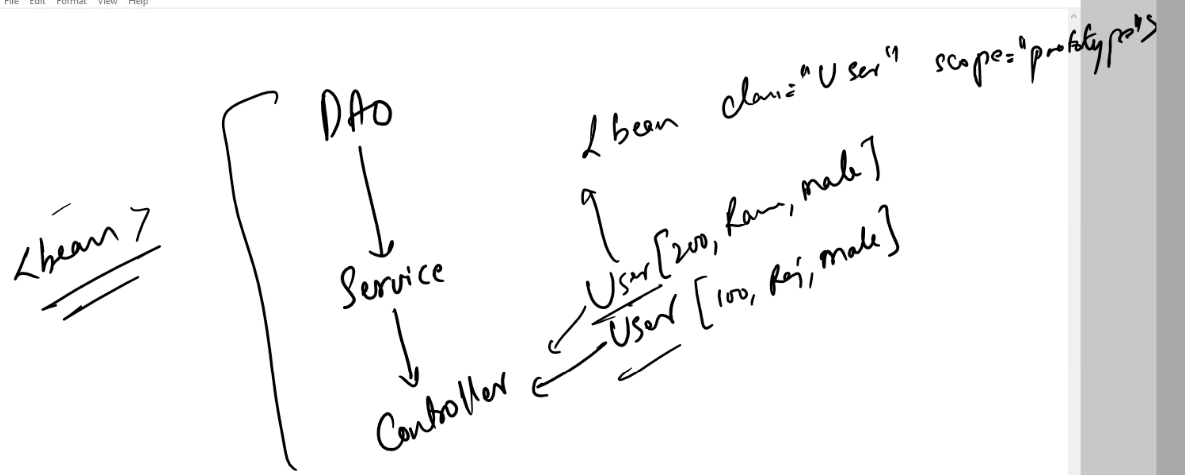


Scopes of bean

1. singleton: By default every bean is singleton per <bean> configuration
2. prototype: This allows multiple instance of the bean to be created per <bean> configuration.

<bean id = ‘xyz’ class = ‘com.org.Xyz’> : Xyz is single ton

<bean id = ‘abc’ class = ‘com.org.Abc’ scope = ‘prototype’>: Abc can have multiple instances



Note: If you omit scope attribute in the <bean> tag then they are considered singleton.

Java based configuration:

Here you can configure the beans through annotations, spring has provided some annotations that are scanned to configure the classes, since most of the applications uses layered architecture spring has introduced annotations that can be used for the classes that are part of particular layer, like model, controller & service layers.

Some of the annotations are:

@Component

@Service

@Repository

@Controller

@RestController

@Configuration

@Bean

@Autowired

@Qualifier

Annotations like @Component, @Service, @Controller, @Repository, @RestController, @Configuration creates instance in the IOC, i.e., similar to <bean>

@Component: It is to instantiate the normal java class like Utility classes, instead of configuring in the xml like <bean> tag for the class you can use @Component

@Controller: It is to instantiate the class that acts like controller

@Service & @Repository: These are specific to service & dao layers, both the annotations instantiates the class

@RestController: This is to instantiate the class used for REST services.

@Autowired: This injects the object to the dependency, it automatically checks the type of the objects & initializes the object

@Service  
class Abc { }

@Controller  
class Xyz {  
 @Autowired  
 Abc a;  
}

Here both Abc & Xyz instances will be created in the container, & Abc object is assigned to the Xyz.a property because @Autowired.

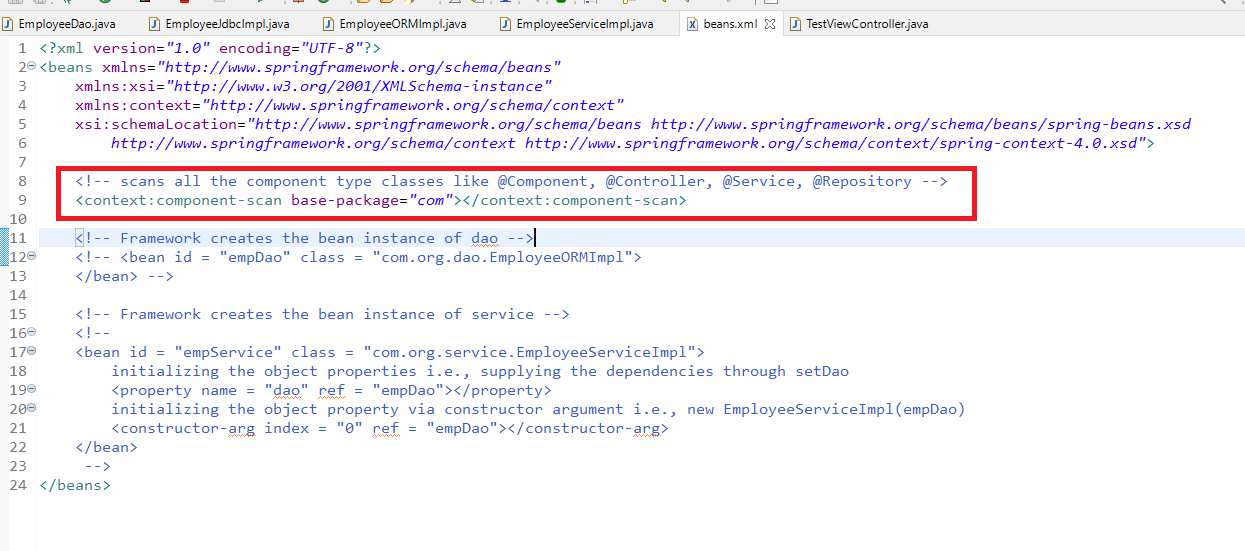
The advantage is you don’t have to write <bean> in the XML now onwards.

The configuration you keep in the XML is to scan all these classes belonging to some packages.

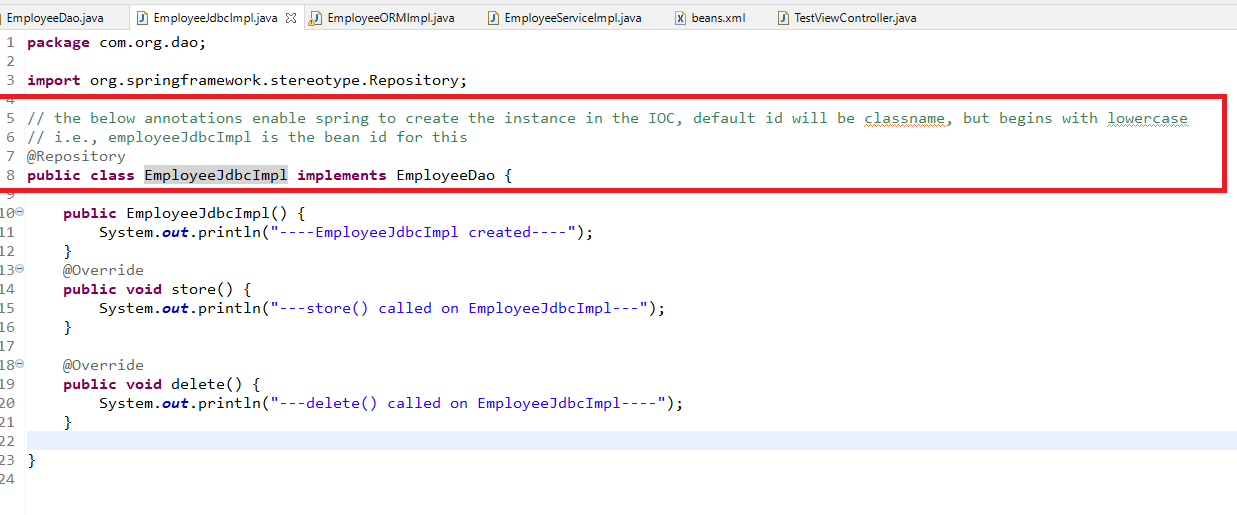
*<component-scan base-package = ‘com’>*

The above tag scans all the classes inside com folder & their sub-folders, then instantiates those classes accordingly based on the annotations like @Component, @Service, @Repository.

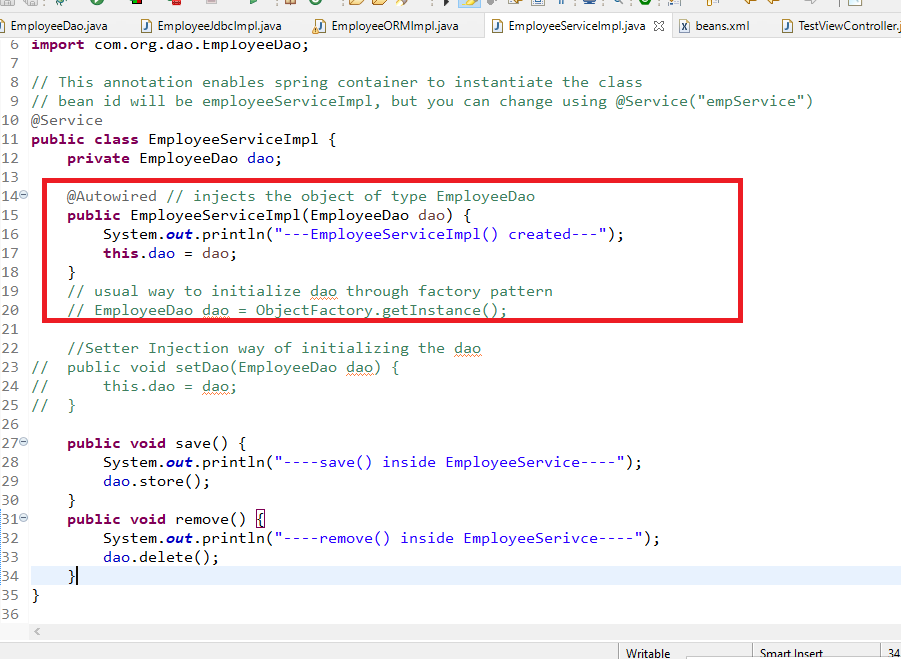
beans.xml



Add @Repository to EmployeeJdbcImpl.java

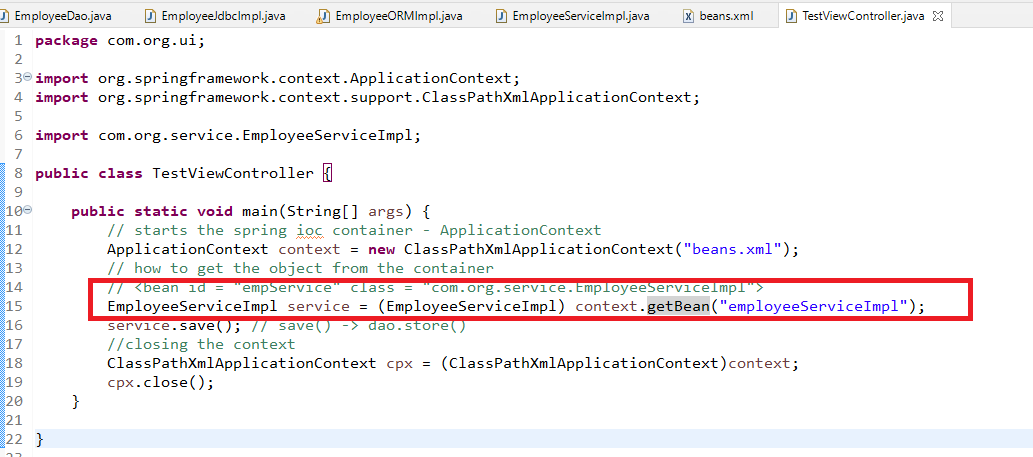


Using @Service in the EmployeeServiceImpl

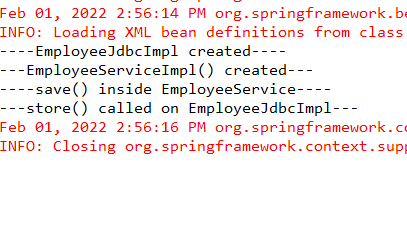


@Autwired injects the object based on the type, it can be used on top of the constructor, setter method or on the variable

TestViewController.java



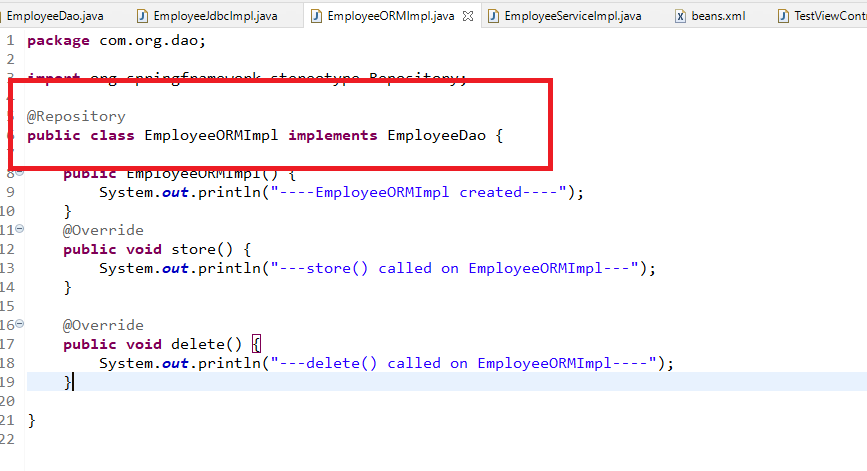
Output:



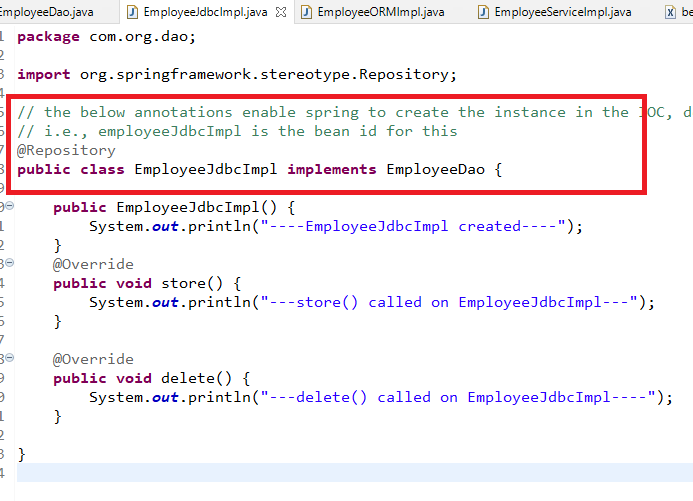
The above code works fine as long as there is only one instance of EmployeDao, but if there are more than one instance of EmployeeDao i.e., both EmployeeJdbcImpl and EmployeeORMImpl then you will get exception, because framework looks to supply the object of EmployeeDao but it doesn’t know which object to supply as there are two objects, if it was only one type, then there wouldn’t by any exception.

To solve this you can use @Qualifier(name = ‘beanName’)

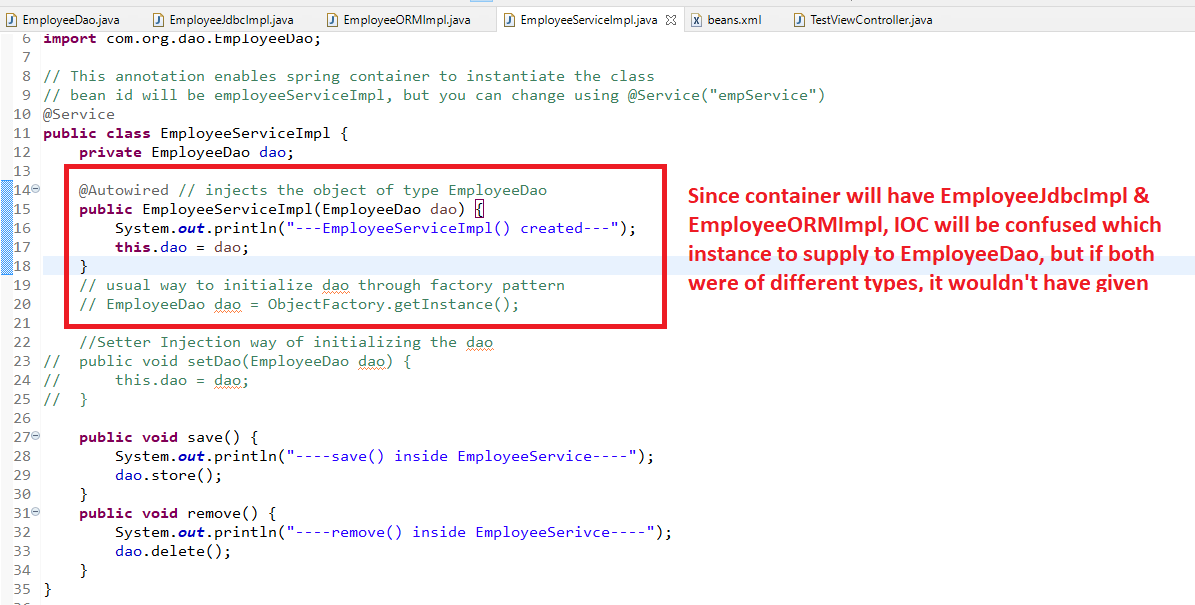
EmployeeORMImpl.java

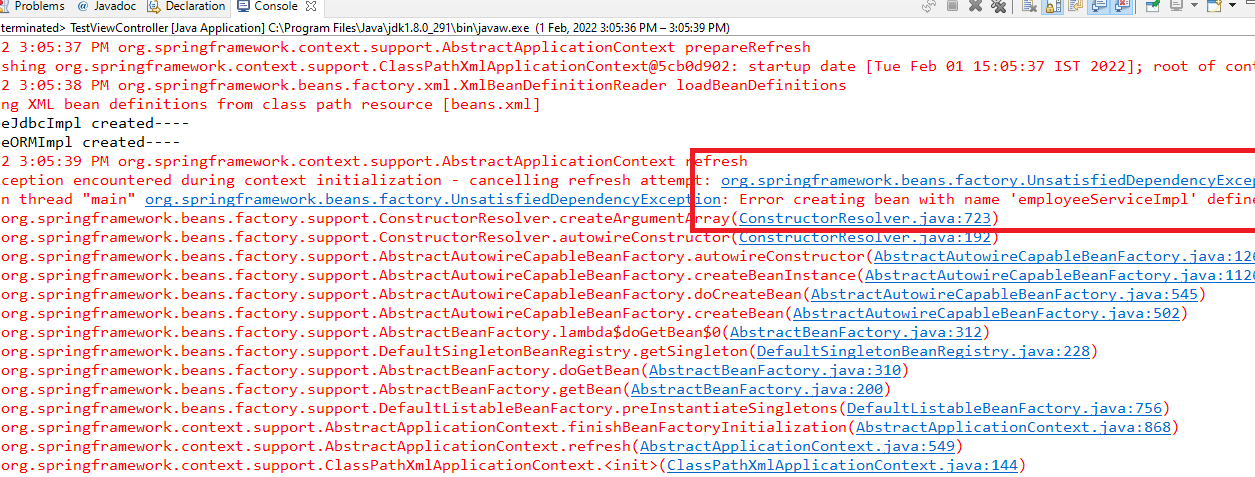


The @Repository is instantiating EmployeDao type, but we have another class of EmployeeDao which is also created in the IOC

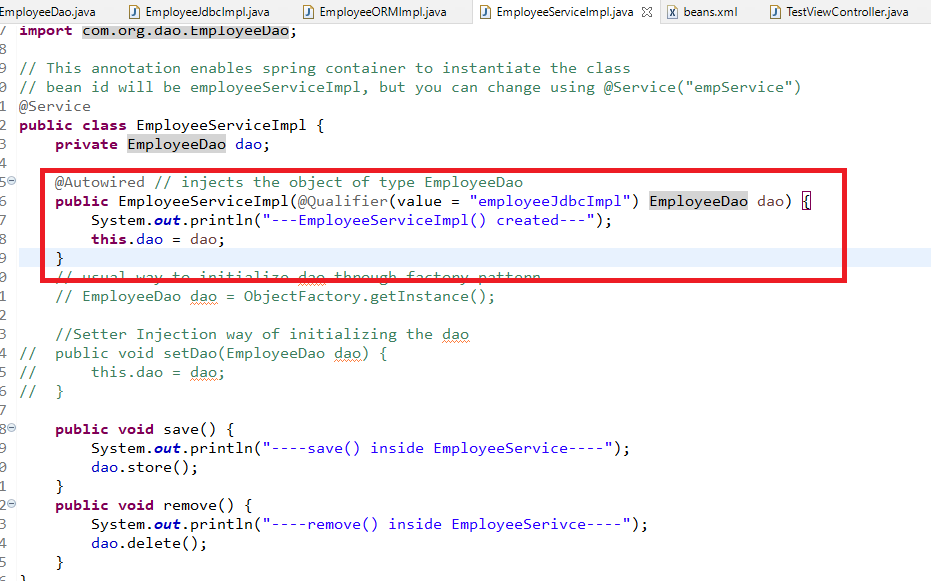


Note: Since both the classes fall into EmployeeDao type, then @Autwored on EmployeeDao will give error

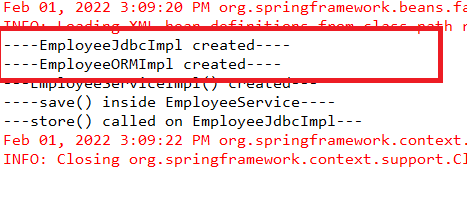




To avoid this you can use @Qualifier on EmployeeServiceImpl or @Primary on the implementation class.



Output:



Though the IOC creates EmployeeJdbcImpl & EmployeeORMImpl, it injects EmployeeJdbcImpl to the EmployeeService.dao because of @Qualifier.

Another way to solve this is without using @Qualifier i.e., using @Primary on top the class to give preference to it when there are multiple candidates of same type



Note: If you use @Qualifer in the EmployeeServiceImpl, then it override @Primary, preference will be given to @Qualifer in that case.