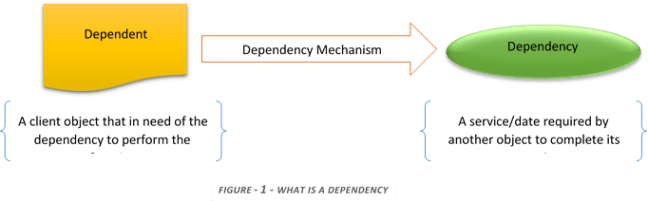
1. [RESPECTABLE]

Proof of Concept to demonstrate an application designed using Dependency Injection design pattern. Expected the architecture and an implementation approach with a sample project

**EXPLANATION**

**Dependency** - If one object (dependent object) requires data and service of another object for its functioning it is called as dependency.



Dependency injection is a Design Pattern that makes a class independent of its dependencies. It achieves so by decoupling the usage of an object from object creation.

**DI uses:**

* Improve code reusability
* Reduces coupling
* Dependencies can be replaced without changing the class that uses them.
* Makes code easier to develop and test

**Spring Framework** is built around the concept of Dependency Injection and IOC (Inversion of Control).

**Types of DI in Spring:**

**Dependency Injection in Spring can be done through *Constructors, Setters* or *Fields*.**

* In Field-Based DI, the dependencies are injected by marking with @**Autowired** annotation, this method is costlier than constructor-based or setter-based injection
* For setter-based DI, the container will call setter methods after invoking a no-argument constructor or no-argument static factory method to instantiate the bean.

**IOC Container:**

The Spring Framework comes with two IOC containers – BeanFactory and ApplicationContext. IOC container is responsible for instantiating, configuring and assembling objects namely **Beans**, and also manages bean lifecycle.

Implementations of ApplicationContext interface are:

1. ClassPathXmlApplicationContext and FileSystemXmlApplicationContext for standalone applications
2. WebApplicationContext for web applications.

**DEMO APPLICATION:**

**GIT Repo for Source Code:**

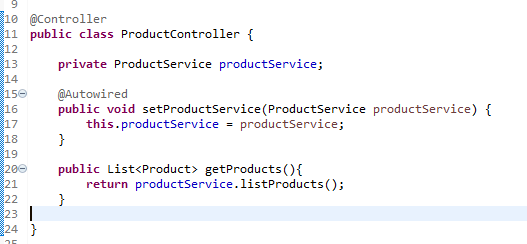
[**https://github.com/tinaash/SpringDI.git**](https://github.com/tinaash/SpringDI.git)

[**git@github.com:tinaash/SpringDI.git**](mailto:git@github.com:tinaash/SpringDI.git)

**Classes used for implementing DI in our Demo application:**

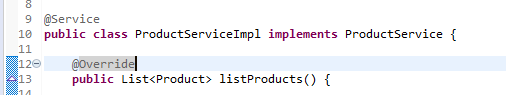
1. **Service** used by Controller - ProductService.java
2. **Controller** (Client) that uses the service - ProductController.java
3. **Interface** used by controller and implemented by the service - ProductServiceImpl.java
4. **Injector** which creates a service instance and injects it into the client - Spring IOC Container

The demo SpringBoot application implemented for the given requirement uses setter based DI in Controller class:



ProductController class is annotated with @Controller annotation. This marks the class as a Spring IOC Managed bean. Without this annotation, Spring will not load this Controller class into the context and it will not inject an instance of the service into this class.

On the setter method, @Autowired annotation is specified. This indicates Spring to inject a Spring managed ProductService bean into this class.



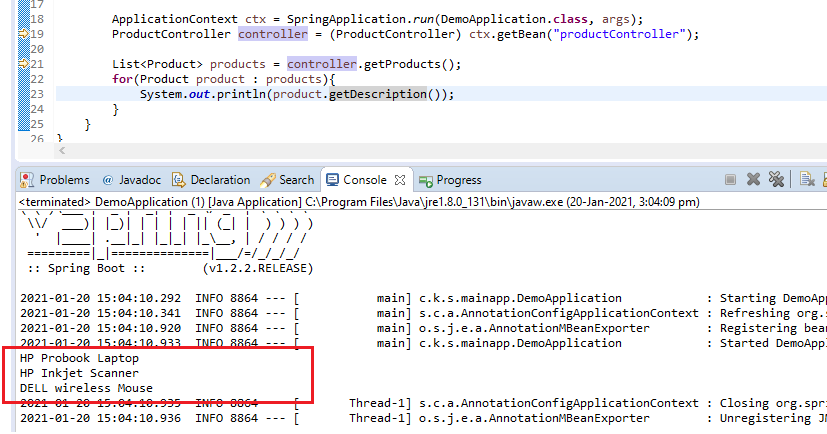
ProductServiceImpl class is annotated with @Service, indicating this is also a Bean class to be managed by IOC container. This ServiceImpl class implements ProduceService Interface and overrides the listProducts method() of the interface.

@ComponentScan annotation in Main class DemoApplication.java will scan the specified package for Spring components or beans. We can inject more dependent objects using autowired annotation to the Controller class later also to expand the functionality. If Spring cannot fulfill a dependency, the application will fail to startup.

**TEST CASES:**

1. **Positive test case – Success Case**

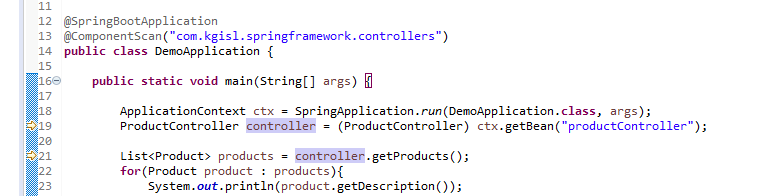
List of Products are printed in console output:

****

1. **Negative Test case –** BeanCreationException thrown during startup.

In @**ComponentScan**, if we mention the base package as com.kgisl.springframework.controllers, the dependency for **ProductService** will not get resolved.

Because the dependency service class **ProductService** is present inside another subpackage com.kgisl.springframework.services. This way we can replicate the BeanCreationException due to unresolved dependency.



**Logs:**

2021-01-20 14:21:19.062 ERROR 16140 --- [ main] o.s.boot.SpringApplication : Application startup failed

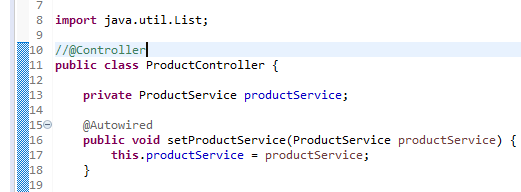
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'productController': Injection of autowired dependencies failed; nested exception is org.springframework.beans.factory.BeanCreationException: Could not autowire method: public void com.kgisl.springframework.controllers.ProductController.setProductService(com.kgisl.springframework.services.ProductService); nested exception is org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type [com.kgisl.springframework.services.ProductService] found for dependency: expected at least 1 bean which qualifies as autowire candidate for this dependency. Dependency annotations: {}

...

at com.kgisl.springframework.mainapp.DemoApplication.main(DemoApplication.java:18)

1. **Negative Test case –** NoSuchBeanDefinitionException thrown during start up

By removing @**Controller** annotation in ProductController class, we can replicate NoSuchBeanDefinitionException as the Controller class no longer qualifies as a Spring Bean



**Logs**

2021-01-20 15:50:59.332 INFO 12668 --- [ main] c.k.s.mainapp.DemoApplication : Started DemoApplication in 0.512 seconds (JVM running for 0.758)

Exception in thread "main" 2021-01-20 15:50:59.340 INFO 12668 --- [ Thread-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@7a1ebcd8: startup date [Wed Jan 20 15:50:58 SGT 2021]; root of context hierarchy

2021-01-20 15:50:59.343 INFO 12668 --- [ Thread-1] o.s.j.e.a.AnnotationMBeanExporter : Unregistering JMX-exposed beans on shutdown

org.springframework.beans.factory.NoSuchBeanDefinitionException: No bean named 'productController' is defined

...

at com.kgisl.springframework.mainapp.DemoApplication.main(DemoApplication.java:19)