Spring - Injecting a Prototype Bean into a Singleton Bean Problem
If same scoped beans are wired together there's no problem. For example a singleton bean A injected into another singleton bean B. But if the bean A has the narrower scope say prototype scope then there's a problem.

To understand the problem let's see an example. We are going to have two beans MyPrototypeBean, scoped as prototype and MySingletonBean, scoped as singleton. We will inject the prototype bean into the singleton bean. We will also access MySingletonBean via method call context#getBean(MySingletonBean.class) multiple times. We are expecting (per prototype specifications) that a new prototype bean will be created and be injected into MySingletonBean every time.

Example Link

On running AppConfig main method, we are expecting MySingletonBean to print two different time, with approximately one sec difference. But here's a actual output:

```
Hi, the time is 2016-01-02T16:01:55.477
Hi, the time is 2016-01-02T16:01:55.477
```

That means there's only one instance of the prototype bean within the singleton bean. Well, a prototype bean should not behave that way. There should be a new instance every time.

The problem is: spring container only creates the singleton bean MySingletonBean once, and thus only gets one opportunity to inject the dependencies into it. The container cannot provide MySingletonBean with a new instance of MyPrototypeBean every time one is needed.

There are four solutions to that problem

1. Inject ApplicationContext bean into MySingletonBean to get instance of MyPrototypeBean, whenever we need it. .
2. Lookup method injection. This approach involves in dynamic bytecode generation.
3. Using Scoped Proxy.
4. Using JSR 330 Provider<T> injection by Spring

1. Spring - Making Bean ApplicationContext aware

We can inject spring ApplicationContext as a bean into any bean. By doing so, we are effectively making our bean ApplicationContext aware. Our bean then programmatically can retrieve other beans by calling ApplicationContext.getBean()or retrieve any resources by calling applicationContext#getResource()

Using this approach, we can solve the problem of injecting a prototype bean into singleton bean as mentioned in the last topic.

We can inject ApplicationContext as bean using @Autowire annotation or by implementing spring ApplicationContextAware interface.

In the following example, we are going to modify our last example to demonstrate the recommended @Autowire approach

Example Link

Output:

```
Hi, the time is 2016-01-02T18:11:11.541
Hi, the time is 2016-01-02T18:11:12.588
```

Disadvantage of Injecting ApplicationContext in the Beans
Injecting ApplicationContext, couples the application code to Spring API. Secondly, it does not follow the Inversion of Control principle as we are not letting Spring to inject the dependencies but we are asking the container to give us the dependencies. In some situation this capability might be useful, e.g. some bean taking care of some sort of integration to some other framework etc. However, in general we should avoid this approach for fixing narrower scoped bean injection problem

Example
https://github.com/srjainapur/Injecting-Prototype-Into-Singleton-Bean/tree/master/BeanScopeApplicationContextApproach

2. Spring - Lookup method Injection using @Lookup

Spring lookup method injection is the process of dynamically overriding a registered bean method.

The bean method should be annotated with @Lookup.

Spring returns the lookup result matched by the method's return type.

```
@Component
public class MySingletonBean {

    public void showMessage(){
        MyPrototypeBean bean = getPrototypeBean();
        //each time getPrototypeBean() call
        //will return new instance
    }

    @Lookup
    public MyPrototypeBean getPrototypeBean(){
        //spring will override this method
        return null;
    }
}
```

In above example the method getPrototypeBean is returning null. That doesn't matter, because this method will actually be overridden by spring dynamically. Spring uses CGLIB library to do so.

The dynamically generated code will look for the target bean in the application context. Something like this:

```
    public MyPrototypeBean getPrototypeBean(){
```

```
    return applicationContext.getBean(MyPrototypeBean.class);
  }
  ...
```

This is the one of the solution for a prototype bean being injected in singleton bean, as described in the previous topics.

For dynamic code generation to work, we have to follow these conditions on the bean class :

1. The bean class cannot be final.
2. The method annotated with @Lookup, cannot be private , static or final
3. The factory approach of JavaConfig doesn't work i.e. a factory method annotated with @Bean and returning a manually created instance of the bean doesn't work. Since the container is not in charge of creating the instance, therefore it cannot create a runtime-generated subclass on the fly. So we have to use component scanning approach as described here

Example Link
https://github.com/srjainapur/Injecting-Prototype-Into-Singleton-Bean/tree/master/PrototypeScopeInSingleton

3.1. Spring - Injecting a Bean as a class based Proxy Object

This is another way to solve injecting a narrower scoped bean into wider scoped bean problem.

We need to inject a proxy object that exposes the same public interface as the original scoped object.

Spring uses CGLIB to create the proxy object.

The proxy object delegates method calls to the real object.

Each access of underlying prototype object causes a new object to be created.

Configuring the Scoped Proxy for Prototype bean

```
@Configuration
public class AppConfig {

   @Bean
   @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
         proxyMode = ScopedProxyMode.TARGET_CLASS)
   public MyPrototypeBean prototypeBean () {
      return new MyPrototypeBean();
   }

   @Bean
   public MySingletonBean singletonBean () {
      return new MySingletonBean();
   }
}
```

In above examples **proxyMode=ScopedProxyMode.TARGET_CLASS** causes an AOP proxy to be

injected at the target injection point. The default proxyMode is ScopedProxyMode.NO.

Another possible value, ScopedProxyMode.INTERFACES creates JDK dynamic proxy (instead of class based CGLIB proxy) which can only take the target bean's interface types.
Autowiring The prototype bean into Singleton bean

```
public class MySingletonBean {

   @Autowired
   private MyPrototypeBean prototypeBean;

   public void showMessage(){
      System.out.println("Hi, the time is "+prototypeBean.getDateTime());
   }
}
```

Client code

```
 AnnotationConfigApplicationContext context =
                    new AnnotationConfigApplicationContext(AppConfig.class);
   MySingletonBean bean = context.getBean(MySingletonBean.class);
   bean.showMessage();
   Thread.sleep(1000);

   bean = context.getBean(MySingletonBean.class);
   bean.showMessage();
```

Output:

```
Hi, the time is 2016-03-05T13:40:06.003
Hi, the time is 2016-03-05T13:40:07.060
```

Different time above shows that a new prototype bean instance is created each time.

Example

3.2. Spring - Injecting a Bean as a JDK interface based Proxy Object

As alternative to our last example we can specify proxyMode as ScopedProxyMode.INTERFACES for @Scope annotation.

This mode needs to be specified if we are autowiring interface rather than a concrete class.

Spring injects JDK interface based proxy rather than CGLIB class based proxy.

This example is a modification of our previous example. We are going to create an interface for our prototype bean so that it can be injected into wider scoped singleton bean using JDK interface based proxy mode.

Configuring the Scoped Proxy for Prototype bean

```
@Configuration
public class AppConfig {

   @Bean
   @Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
                  proxyMode = ScopedProxyMode.INTERFACES)
   public MyPrototypeBean prototypeBean () {
      return new MyPrototypeBean();
   }

   @Bean
   public MySingletonBean singletonBean () {
      return new MySingletonBean();
   }
 }
```

IPrototype Interface

```
public interface IPrototype {
   String getDateTime();
}
```

Prototype Bean implementation

```
public class MyPrototypeBean implements IPrototype {
   private String dateTimeString = LocalDateTime.now().toString();

   @Override
   public String getDateTime() {
      return dateTimeString;
   }
}
```

Autowiring the prototype bean into Singleton bean

```
public class MySingletonBean {

   @Autowired
   private IPrototype prototypeBean;

   public void showMessage(){
      System.out.println("Hi, the time is "+prototypeBean.getDateTime());
   }
}
```

Client code

```
AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppCon-
```

```
fig.class);
MySingletonBean bean = context.getBean(MySingletonBean.class);
bean.showMessage();

Thread.sleep(1000);
bean = context.getBean(MySingletonBean.class);
bean.showMessage();
```

Output:
Hi, the time is 2016-03-05T14:33:37.399
Hi, the time is 2016-03-05T14:33:38.454

Example
https://github.com/srjainapur/Injecting-Prototype-Into-Singleton-Bean/tree/master/BeanScopeApplicationContextApproach

4. Spring - Using JSR 330 Provider to Inject Narrower Scoped Bean
Spring 3.0, started support for JSR-330 standard annotations (Dependency Injection). That means now we can use @javax.inject.Inject instead of @Autowire along with other standard annotations and interfaces.

To solve narrowing scoped bean injection problem, we now have a standard option, i.e. using javax.inject.Provider<T> interface. We can inject or autowire the Provider interface having component type parameter as our prototype bean:

```
@Autowired
private Provider<MyPrototypeBean> myPrototypeBeanProvider;
```

According to the specification, Provider.get() will always return the new instance of type T, in our case MyPrototypeBean.

We won't need any other extra configuration in @Configuration class. We just have to return instance of MyPrototypeBean bean from one of the factory method, annotated with @Bean.

Having this interface injected has other advantages as well e.g. lazy retrieval of an instance etc.

The good thing is, this approach doesn't involve any runtime byte code generation but it's just like Spring's ObjectFactory implementation. ObjectFactory is Spring specific alternative to Provider interface. In the following example with can replace Provider with ObjectFactory without any other change.

Example
https://github.com/srjainapur/Injecting-Prototype-Into-Singleton-Bean/tree/master/BeanScope-ProviderInterfaceApproach