Injecting Prototype Beans into a Singleton Instance in Spring

In this quick article, we're going to show different approaches of injecting prototype beans into a singleton instance. We'll discuss the use cases and the advantages/disadvantages of each scenario.

By default, Spring beans are singletons. The problem arises when we try to wire beans of different scopes. For example, a prototype bean into a singleton. This is known as the scoped bean injection problem.

To learn more about bean scopes, this write-up is a good place to start **(https://www.baeldung.com/spring-bean-scopes)**

Prototype Bean Injection Problem

In order to describe the problem, let's configure the following beans:

```
@Configuration
public class AppConfig {

   @Bean
   @Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
   public PrototypeBean prototypeBean() {
      return new PrototypeBean();
   }

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

Notice that the first bean has a prototype scope, the other one is a singleton.

Now, let's inject the prototype-scoped bean into the singleton – and then expose if via the getPrototypeBean() method:

```
public class SingletonBean {
    // ..

   @Autowired
   private PrototypeBean prototypeBean;

   public SingletonBean() {
      logger.info("Singleton instance created");
   }

   public PrototypeBean getPrototypeBean() {
      logger.info(String.valueOf(LocalTime.now()));
      return prototypeBean;
   }
}
```

Then, let’s load up the ApplicationContext and get the singleton bean twice:

```
public static void main(String[] args) throws InterruptedException {
   AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

   SingletonBean firstSingleton = context.getBean(SingletonBean.class);
   PrototypeBean firstPrototype = firstSingleton.getPrototypeBean();

   // get singleton bean instance one more time
   SingletonBean secondSingleton = context.getBean(SingletonBean.class);
   PrototypeBean secondPrototype = secondSingleton.getPrototypeBean();

   isTrue(firstPrototype.equals(secondPrototype), "The same instance should be returned");
}
```

Here’s the output from the console:

```
Singleton Bean created
Prototype Bean created
11:06:57.894
// should create another prototype bean instance here
11:06:58.895
```

Both beans were initialized only once, at the startup of the application context.

Injecting ApplicationContext

We can also inject the ApplicationContext directly into a bean.

To achieve this, either use the @Autowire annotation or implement the ApplicationContextAware interface:

```
public class SingletonAppContextBean implements ApplicationContextAware {
    private ApplicationContext applicationContext;

   public PrototypeBean getPrototypeBean() {
      return applicationContext.getBean(PrototypeBean.class);
   }

   @Override
   public void setApplicationContext(ApplicationContext applicationContext)    throws
BeansException {
      this.applicationContext = applicationContext;
   }
}
```

Every time the getPrototypeBean() method is called, a new instance of PrototypeBean will be returned from the ApplicationContext.

However, this approach has serious disadvantages. It contradicts the principle of inversion of control, as we request the dependencies from the container directly.

Also, we fetch the prototype bean from the applicationContext within the SingletonAppcontextBean class. This means coupling the code to the Spring Framework.

Method Injection

Another way to solve the problem is method injection with the **@Lookup** annotation:

```
@Component
public class SingletonLookupBean {

   @Lookup
   public PrototypeBean getPrototypeBean() {
      return null;
   }
}
```

Spring will override the getPrototypeBean() method annotated with @Lookup. It then registers the bean into the application context. Whenever we request the getPrototypeBean() method, it returns a new PrototypeBean instance.

It will use CGLIB to generate the bytecode responsible for fetching the PrototypeBean from the application context.

javax.inject API

The setup along with required dependencies are described in this Spring wiring article.

Here’s the singleton bean:

```
public class SingletonProviderBean {

   @Autowired
   private Provider<PrototypeBean> myPrototypeBeanProvider;

   public PrototypeBean getPrototypeInstance() {
      return myPrototypeBeanProvider.get();
   }
}
```

Add the following dependency in pom.xml

```
<dependency>
       <groupId>javax.inject</groupId>
      <artifactId>javax.inject</artifactId>
      <version>1</version>
</dependency>
```

We use Provider interface to inject the prototype bean. For each getPrototypeInstance() method call, the myPrototypeBeanProvider.get() method returns a new instance of PrototypeBean.

Scoped Proxy

By default, Spring holds a reference to the real object to perform the injection. Here, we create a proxy object to wire the real object with the dependent one.

Each time the method on the proxy object is called, the proxy decides itself whether to create a new instance of the real object or reuse the existing one.

To set up this, we modify the Appconfig class to add a new @Scope annotation:

```
@Scope(
  value = ConfigurableBeanFactory.SCOPE_PROTOTYPE,
  proxyMode = ScopedProxyMode.TARGET_CLASS)
```

By default, Spring uses CGLIB library to directly subclass the objects. To avoid CGLIB usage, we can configure the proxy mode with ScopedProxyMode.INTERFACES, to use the JDK dynamic proxy instead.

ObjectFactory Interface

Spring provides the ObjectFactory<T> interface to produce on demand objects of the given type:

```
public class SingletonObjectFactoryBean {

    @Autowired
    private ObjectFactory<PrototypeBean> prototypeBeanObjectFactory;

    public PrototypeBean getPrototypeInstance() {
        return prototypeBeanObjectFactory.getObject();
    }
}
```

Let's have a look at getPrototypeInstance() method; getObject() returns a brand new instance of PrototypeBean for each request. Here, we have more control over initialization of the prototype.

Also, the ObjectFactory is a part of the framework; this means avoiding additional setup in order to use this option.

Create a Bean at Runtime Using java.util.Function

Another option is to create the prototype bean instances at runtime, which also allows us to add parameters to the instances.

To see an example of this, let's add a name field to our PrototypeBean class:

```
public class PrototypeBean {
    private String name;

    public PrototypeBean(String name) {
        this.name = name;
        logger.info("Prototype instance " + name + " created");
    }

    //...
}
```

Next, we'll inject a bean factory into our singleton bean by making use of the java.util.Function interface:

```
public class SingletonFunctionBean {

    @Autowired
    private Function<String, PrototypeBean> beanFactory;

    public PrototypeBean getPrototypeInstance(String name) {
        PrototypeBean bean = beanFactory.apply(name);
        return bean;
    }

}
```

Finally, we have to define the factory bean, prototype and singleton beans in our configuration:

```
@Configuration
public class AppConfig {

   @Bean
    public Function<String, PrototypeBean> beanFactory() {
        return name -> prototypeBeanWithParam(name);
    }

    @Bean
    @Scope(value = "prototype")
    public PrototypeBean prototypeBeanWithParam(String name) {
       return new PrototypeBean(name);
    }

    @Bean
    public SingletonFunctionBean singletonFunctionBean() {
        return new SingletonFunctionBean();
    }
    //...
}
```

Conclusion

In this short tutorial, we learned several ways to inject the prototype bean into the singleton instance.
As always, the complete code for this tutorial can be found on GitHub project.
(https://github.com/srjainapur/Injecting-Prototype-Beans-into-a-Singleton-Instance-in-Spring)