Configuration Metadata

Spring configuration metadata needs to be created to tell Spring container how to initiate, configure, wire and assemble the application specific objects.

Since Spring's first release in 2002 to the latest release, spring has provided three ways of configurations:

XML-based Configuration : All configurations are in one or multiple XML files. This is the most verbose way of configuration. Huge projects require tedious amount of XML which is difficult to manage.

Annotation-based configuration : Spring 2.5 introduces annotation-based configuration. We still have to write XML files but just to indicate "component-scan" on the packages of annotated classes.

Java-based configuration (JavaConfig): Starting with Spring 3.0, a pure-Java means of configuring container was provided. We don't need any XML with this method of configuration. JavaConfig provides a truly object-oriented mechanism for dependency injection, meaning we can take full advantage of reusability, inheritance and polymorphism in the configuration code. Application developer has complete control over instantiation and dependency injection here.

We will mainly focus only on JavaConfig. Learning only one method is good enough to understand key concepts and features of Spring container.

Regardless of what method we use, we mainly have to use configuration metadata at three places:

Beans : The objects managed by Spring Container. They are registered to the Spring container by the use of some metadata.

Injection Points : The places where dependencies have to be injected. The Injection Points typically are fields/setters/constructors in a Spring bean class. Spring framework

populates/inserts the injection points with the required instances of other beans. That happens during the bean loading time.

The Configuration : This can be a Java class annotated with @Configuration or it can be XML if we are using old way of configuration. This is where we wire the injection points with dependencies.

Java-based configuration: Using @Configuration

@Configuration

This is a class level annotation. The class annotated with this annotation may consist of methods annotated with @Bean. Spring container invokes such methods to get the object instances, so that the container can register them as beans.

```
@Configuration
public class MyAppConfig {
  @Bean
  public SomeBean someBean() {
 // instantiate, configure and return bean instance ...
  return new SomeBeanImpl();
  }
}
```

The equivalent XML configuration will look like this:

```
<bean name="someBean" class="spring.example.SomeBeanImpl"/>
```

@Configuration classes are in fact nothing but spring managed factories to create and register bean instances.

Bootstrapping Spring Container

In Java-based spring configuration, the container can be bootstrapped using either AnnotationConfigApplicationContext or for web application: AnnotationConfigWebApplicationContext.

```
new AnnotationConfigApplicationContext(MyAppConfig.class);
```

We can also specify the qualified package name containing @Configuration annotated classes:

```
new AnnotationConfigApplicationContext("piseth.java.school.spring");
```

By using above overloaded variant, we can have multiple configuration classes in a single package

@Configuration Classes are Subclassed by CGLIB

All @Configuration classes are subclassed at startup-time with CGLIB. In the subclass, the child method checks the container first for any cached (scoped) beans before it calls the parent method and creates a new instance.

CGLIB proxying is the means by which invoking methods or fields within @Bean methods in @Configuration classes creates bean metadata references to collaborating objects; such methods are not invoked with normal Java semantics but rather go through the container in order to provide the usual lifecycle management and proxying of Spring beans even when referring to other beans via programmatic calls to @Bean methods.

That's why all methods will return the same instance at multiple calls (if they are singleton scoped which is the default scope).

There has to be @Configuration annotation, otherwise this runtime manipulation won't be done.

@ComponentScan

So far we saw JavaConfig using factory methods (annotated with @Bean) to provide bean implementation. Using this approach, we have to create the bean implementation instances ourselves. Instead of this factory approach, we can have spring to scan the provided packages and create all implementation automatically and then inject the dependencies. We can do that by using @ComponentScan along with @Configuration. We still don't have to use any XML.

Examples

Using @Configuration annotation

```
package com.piseth.java.school;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;


@Configuration

public class ConfigExample {

    private int counter;

    @Bean

    public Greeter greeterBean() {

        return new Greeter();
```

```
    }

    public static void main(String... strings) {
        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(ConfigExample.class);
        Greeter greeter = context.getBean(Greeter.class);
        greeter.sayHi("Piseth");
    }

    public static class Greeter {
        public void sayHi(String name) {
            System.out.println("Hi there, " + name);
        }
    }
}
```

Output

Hi there, Piseth

Demo for CGLIB manipulation of @Configuration classes

@Configuration classes are subclassed by CGLIB, poxying the @Bean methods to control their lifecycles. The bean instances created by these proxy methods are cached. That's why all bean methods will return the same instance if called multiple times.

```
@Configuration
public class DemoCGLIB{
  private int counter;

  @Bean
  public String something(){
      System.out.println("method invoked");
      return String.valueOf(++counter);
  }

      public static void main(String... strings) {
          AnnotationConfigApplicationContext context =
                  new AnnotationConfigApplicationContext(DemoCGLIB.class);
          DemoCGLIB bean = context.getBean(DemoCGLIB.class);
          System.out.println(bean.something());
          System.out.println(bean.something());
      }
}
```

Output:

```
method invoked
1
1
```

Let's remove @Bean annotation from something() method and see the difference:

```
@Configuration
public class WithoutCGLIB{
  private int counter;


  public String something(){
      System.out.println("method invoked");
      return String.valueOf(++counter);
  }
      public static void main(String... strings){
          AnnotationConfigApplicationContext context =
                  new AnnotationConfigApplicationContext(WithoutCGLIB.class);
          WithoutCGLIB bean = context.getBean(WithoutCGLIB.class);
          System.out.println(bean.something());
          System.out.println(bean.something());
      }
}
```

Output

```
method invoked
1
method invoked
2
```