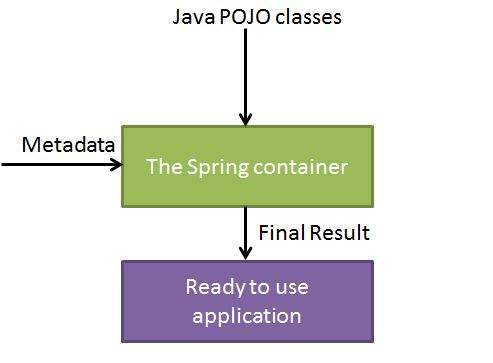
**Spring IoC Containers,** The Spring container is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses dependency injection (DI) to manage the components that make up an application.

*The container gets all the instruction to create, configure and manage the beans from the bean configuration metadata. The configurations can be done through Java code and/or annotations and/or XML files. The following diagram is a simple representation of the Spring container process towards creating a fully configured application.*



Spring provides following two distinct types of containers.

1. BeanFactory container
2. ApplicationContext container

Spring IoC container classes are part of org.springframework.beans and org.springframework.context packages. Spring IoC container provides us different ways to decouple the object dependencies.

BeanFactory is the root interface of Spring IoC container. ApplicationContext is the child interface of BeanFactory interface that provide Spring AOP features, i18n, transaction and publish events etc.

**BeanFactory container,** org.springframework.beans.BeanFactory is the actual representation of the Spring IoC container responsible for containing and managing the Spring beans. The BeanFactory, sources application object, configures them and takes care of assembling the dependencies between objects. One of the most popularly used implementation of BeanFactory is the XMLBeanFactory. XMLBeanFactory allows the representation of objects and their rich dependencies in terms of XML. The XmlBeanFactory takes XML configuration metadata to create the fully configured application

Resource resource = **new** FileSystemResource(Constant.***APPLICATION\_CONTEXT\_XML***);

BeanFactory factory = **new** XmlBeanFactory(resource);

HelloWorld helloWorld = (HelloWorld) factory.getBean("helloWorld");

System.***out***.println(helloWorld.getMessage());

Other ways to create factory are as below:

Resource resource = **new** FileSystemResource(Constant.***APPLICATION\_CONTEXT\_XML***);

BeanFactory factory = **new** XmlBeanFactory(resource);

ClassPathResource resource = **new** ClassPathResource("ContainerOverviewBeans.xml");

BeanFactory factory = **new** XmlBeanFactory(resource);

ApplicationContext context = **new** ClassPathXmlApplicationContext(**new** String[] {"ContainerOverviewBeans.xml"});

BeanFactory factory = context;

HelloWorld helloWorld = (HelloWorld) factory.getBean("helloWorld");

System.***out***.println(helloWorld.getMessage());

Basically that’s all there is. Using getBean(String) you can retrieve instances of your beans; the client-side view of theBeanFactory is surprisingly simple. The BeanFactory interface has only six methods for client code to call:

* boolean containsBean(String): returns true if the BeanFactory contains a bean definition or bean instance that matches the given name
* Object getBean(String): returns an instance of the bean registered under the given name. Depending on how the bean was configured by the BeanFactory configuration, either a singleton and thus shared instance or a newly created bean will be returned. A BeansException will be thrown when either the bean could not be found (in which case it’ll be a NoSuchBeanDefinitionException), or an exception occurred while instantiating and preparing the bean
* Object getBean(String, Class): returns a bean, registered under the given name. The bean returned will be cast to the given Class. If the bean could not be cast, corresponding exceptions will be thrown (BeanNotOfRequiredTypeException). Furthermore, all rules of the getBean(String) method apply
* Class getType(String name): returns the Class of the bean with the given name. If no bean corresponding to the given name could be found, a NoSuchBeanDefinitionException will be thrown
* boolean isSingleton(String): determines whether or not the bean definition or bean instance registered under the given name is a singleton. If no bean corresponding to the given name could be found, aNoSuchBeanDefinitionException will be thrown
* String[] getAliases(String): Return the aliases for the given bean name, if any were defined in the bean definition

**ApplicationContext container,** ApplicationContext is defined by the org.springframework.context.ApplicationContext interface. This is similar to BeanFactory, in that it can load bean definitions, wire them together and dispense beans upon request. In addition, ApplicationContext can also perform more enterprise functionalities like transaction, AOP, resolving text messages from properties files, and push application events to interested listeners.

The *ApplicationContext* container includes all functionality of the *BeanFactory* container, so it is generally recommended over the *BeanFactory*. BeanFactory can still be used for lightweight applications like mobile devices or applet based applications where data volume and speed is significant.

The most commonly used ApplicationContext implementations are:

* **FileSystemXmlApplicationContext**: This container loads the definitions of the beans from an XML file. Here you need to provide the full path of the XML bean configuration file to the constructor.
* **ClassPathXmlApplicationContext** This container loads the definitions of the beans from an XML file. Here you do not need to provide the full path of the XML file but you need to set CLASSPATH properly because this container will look bean configuration XML file in CLASSPATH.
* **WebXmlApplicationContext:** This container loads the XML file with definitions of all beans from within a web application.

A sample code for application context instantiation will look like this.

ApplicationContext applicationContext = **new** FileSystemXmlApplicationContext("beans.xml");

HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorld");

Some of the useful child-interfaces of ApplicationContext are ConfigurableApplicationContextand WebApplicationContext. Spring Framework provides a number of useful ApplicationContext implementation classes that we can use to get the spring context and then the Spring Bean.

Some of the useful ApplicationContext implementations that we use are;

* **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.
* **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.
* **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.
* **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

Usually if you are working on Spring MVC application and your application is configured to use Spring Framework, Spring IoC container gets initialized when application starts and when a bean is requested, the dependencies are injected automatically.

### Spring Bean Configuration

Spring Framework provide three ways to configure beans to be used in the application.

1. **Annotation Based Configuration** – By using @Service or @Component annotations. Scope details can be provided with @Scope annotation.
2. **XML Based Configuration** – By creating Spring Configuration XML file to configure the beans. If you are using Spring MVC framework, the xml based configuration can be loaded automatically by writing some boiler plate code in web.xml file.
3. **Java Based Configuration** – Starting from Spring 3.0, we can configure Spring beans using java programs. Some important annotations used for java based configuration are @Configuration, @ComponentScan @Bean @Import and @DependsOn annotations.

**XML Based Configuration** -

<bean name=*"helloWorld"* class=*"in.spring4buddies.application.ioc.contatiner\_overview.HelloWorld"*

scope=*"singleton"*>

<property name=*"message"* value=*"Hi Spring"* />

</bean>

**Java Based Configuration** =

@Configurable

@ComponentScan(value = "in.spring4buddies.application.ioc.contatiner\_overview")

**public** **class** AnnotationConfig {

@Bean

**public** HelloWorld helloWorld() {

**return** **new** HelloWorld();

}

}

Resource resource = **new** FileSystemResource(Constant.***APPLICATION\_CONTEXT\_XML***);

BeanFactory beanFactory = **new** XmlBeanFactory(resource);

HelloWorld helloWorld = (HelloWorld) beanFactory.getBean("helloWorld");

System.***out***.println("BeanFactory : FileSystemResource == "+ helloWorld.getMessage());

ClassPathResource classPathResource = **new** ClassPathResource("ContainerOverviewBeans.xml");

BeanFactory beanFactory1 = **new** XmlBeanFactory(classPathResource);

HelloWorld helloWorld1 = (HelloWorld) beanFactory1.getBean("helloWorld");

System.***out***.println("BeanFactory : ClassPathResource == "+ helloWorld1.getMessage());

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(**new** String[] {"ContainerOverviewBeans.xml"});

BeanFactory beanFactory2 = applicationContext;

HelloWorld helloWorld2 = (HelloWorld) beanFactory2.getBean("helloWorld");

System.***out***.println("ApplicationContext : ClassPathXmlApplicationContext == "+ helloWorld2.getMessage());

ApplicationContext applicationContext1 = **new** FileSystemXmlApplicationContext(Constant.***APPLICATION\_CONTEXT\_XML***);

HelloWorld helloWorld3 = (HelloWorld) applicationContext1.getBean("helloWorld");

System.***out***.println("ApplicationContext : FileSystemXmlApplicationContext == "+ helloWorld3.getMessage());

AnnotationConfigApplicationContext annotationConfigApplicationContext = **new** AnnotationConfigApplicationContext(AnnotationConfig.**class**);

HelloWorld helloWorld4 = annotationConfigApplicationContext.getBean(HelloWorld.**class**);

System.***out***.println("AnnotationConfigApplicationContext == "+ helloWorld4.getMessage());

annotationConfigApplicationContext.close();

**Composing XML-based configuration metadata**

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(**new** String[] {" *services*.xml", “*messageSource* .xml”, “*themeSource*.xml”});

It can be useful to have bean definitions span multiple XML files. Often each individual XML configuration file represents a logical layer or module in your architecture.

You can use the application context constructor to load bean definitions from all these XML fragments. This constructor takes multiple Resource locations, as was shown in the previous section. Alternatively, use one or more occurrences of the <import/> element to load bean definitions from another file or files. For example:

Beans.xml

<beans>

<import resource=*"services.xml"* />

<import resource=*"resources/messageSource.xml"* />

<import resource=*"/resources/themeSource.xml"* />

<bean id=*"bean1"* class=*"..."* />

<bean id=*"bean2"* class=*"..."* />

</beans>

ApplicationContext applicationContext = **new** ClassPathXmlApplicationContext(**new** String[] {" *beans*.xml"});