*History*

*Java EE is a specification that specifies services that an App server must support*

*Web server*

*Transaction management*

*security management*

*and so on*

*Also the API for calling these services is a standard so that when u change your app server from JBoss to Weblogic*

*then ur code wouldn't change.*

*Java EE is a library of different app libraries that app server must support*

*Example libraries are*

*RMI-IIOP + CORBA for platform independence*

*JDBC*

*JNDI*

*JSP/Servlets*

*JMS*

*JTA -> Transactions*

*In early 2000's they came up with an idea to integrate all these services*

*and they came up with EJB which is Enterprise Java Bean*

*EJB is official standard defined by JSR process*

*When a EJB class is deployed on app server then*

*Automatically all methods would become*

*1. Remote*

*2. Transactions (rollback if there is exception)*

*3. Secured*

*Then any Client be it Java, Visual Basic etc can call methods on this EJB class*

*Drawbacks:*

*---------*

*1. Would only run on App server*

*2. If u just need transaction service then all other services would still be added*

*3. It is complicated and adds a lot of overhead*

*Spring became an alternative to EJB where u would only add whatever u need.*

*What is Inversion of Control?*

*Instead of giving control to individual classes to create the instances for dependencies, Spring takes the control of providing the dependent objects, this is called Inversion of Control.*

*What are the major features in different version of Spring?*

*Spring 2.5 -> Annotation driven config*

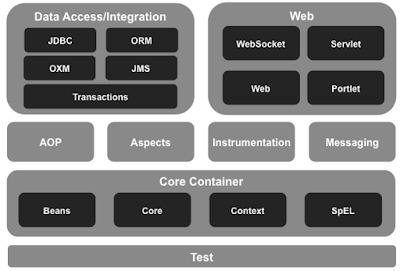
*Spring 3.0 -> Java 5 features added like Generics, variable args etc*

*Spring 4.0 -> fully supports Java 8 features like functional programming Lambda, Streams*

*What are the new features in Spring 4.0?*

1. *Spring-websocket module provides support for WebSocket-based communication in web apps.*
2. *@RestController annotation is introduced for Spring MVC Rest API*
3. *@JmsListener annotations to easily register JMS listener endpoints.*
4. *Jackson’s @JsonView is supported directly on @ResponseBody and @ResponseEntity controller methods which help in converting from object to JSON directly.*
5. *Spring 4.1 Supports JCache annotations using Spring’s existing cache configuration.*

*What are the different modules in Spring?*



*The Core package provides*

* *IoC*
* *Dependency Injection*

*The Context package provides*

This package builds on the beans package to add support for message sources and for the Observer design pattern, and the ability for application objects to obtain resources using a consistent API.

*Contains the following interfaces*

1. *ApplicationContext is central interface to provide configuration for an app*
2. *ApplicationContextAware is the interface that needs to be implemented by any object that needs to be notified of ApplicationContext that it runs in.*
3. *ApplicationEventPublisher is interface that encapsulates event publication functionality*
4. *ApplicationEventPublisherAware is interface that needs to be implemented by any object that wishes to be notified of the ApplicationEventPublisher(ApplicationContext) that it runs in.*
5. *ApplicationListener<E extends ApplicatonEvent> is interface to be implemented by application listeners.*

*BeanFactory, an implementation of the factory pattern, helps us decouple configuration and injection of dependencies from program logic.  
The DAO package provides an abstraction over JDBC to simplify writing code that interacts with database. Declarative transaction management is an addition feature provided by DAO package.  
The ORM package provides integration for Spring with most popular JPA implementations (Hibernate etc) and Query Mapping (iBatis).  
Spring AOP package provides a basic AOP implementation featuring definition of interceptors and pointcuts. Cross cutting concerns like security and transaction management can be implemented using Spring AOP.*

*Spring Web package provides*

* *multipart file-upload functionality*
* *Integration with Struts and other MVC Frameworks*

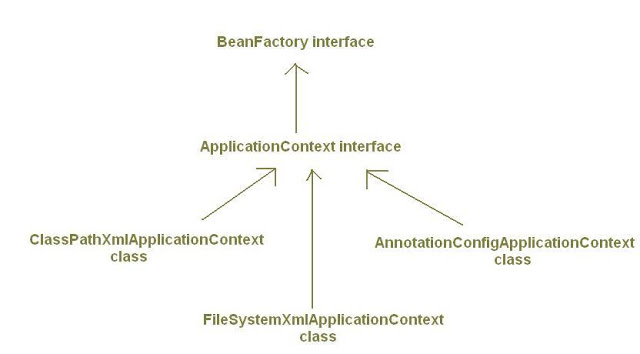
*Spring's MVC package provides a clean implementation of the MVC model for web applications.*

*What is Application Context and it’s implementations?*

*ApplicationContex*t, like *[BeanFactory](http://myjavadiary.blogspot.in/2012/08/what-is-bean-factory-and-xmlbeanfactory.html),* is also used to represent Spring Container. It is built upon *BeanFactory* interface.

*BeanFactory* provides basic functionality while *ApplicationContext* provides advance features to our spring applications which make them enterprise level applications, like i18n, event publishing, JNDI access, EJB integration, Remoting etc.

Interface ApplicationContext extends BeanFactory

[](https://4.bp.blogspot.com/-szB6l8UWlhA/UC-SrBHNI-I/AAAAAAAAAGQ/nxq7H8s-bcs/s1600/applicationcontext_7_1.JPG)

*ApplicationContext* is always preferred over *BeanFactory* and is suitable for J2EE Applications. *ApplicationContext* includes all functionality of the BeanFactory. SUPERSET

**ApplicationContext and Singleton beans:**

While using *BeanFactory*, beans get instantiated when they get requested*first time,* like in *getBean("bean\_id")* method, not when object of *BeanFactory* itself gets created. This is known as *lazy-instantiation*.

But while using *ApplicationContext, singleton beans* does not get created *lazily*. By default, *ApplicationContex*t immediately instantiates the singleton beans and wire/set its properties as it's object itself gets created. So *ApplicationContext* loads singleton beans *eagerly (pre-instantiated*).

*Example:*

ApplicationContext acObj=new ClassPathXmlApplicationContext("beanconfig.xml");

MyBean beanObj=(MyBean)acObj.getBean("mybean");

beanObj.someMethod();

In above example we have created an *ApplicationContext* object *acObj* using one if its implementations *ClassPathXmlApplicationContext* which loads the configuration from source *beanconfig.xml* under classpath.

If we had used *BeanFactory* here,*mybean* would get instantiated when the method *getBean(mybean)* would  be called.

But here with *ApplicationContext*, instantiation of bean with id *mybean* does not get delayed until *getBean()* method is called. If scope of the bean *mybean* is declared in configuration file as *singleton*, it will be immediately instantiated when we create *ApplicationContext* object*acObj*. So when*getBean()* would be called, *mybean* would already have got loaded and its dependencies set.

We can change this default behavior so that *ApplicationContext* does not load singleton beans eagerly by using  **lazy-init**attribute as:

<bean id="mybean" class="x.y.z..MyBean" **lazy-init=**"**true**"/>

A lazy-initialized bean tells the Spring to create a bean instance when it is first requested, rather than at the time of creation of ApplicationContext object.

**Implementations of ApplicationContext:**

There are many implementations of *ApplicationContext* interface. Important ones are:

**1)  *ClassPathXmlApplicationContext:***

It loads bean definitions from XML files located in the classpath.

*Example1:*

ApplicationContext context = new ClassPathXmlApplicationContext("myconfig.xml");

*Example2:*  Loading configuration from multiple files under classpath.

ApplicationContext context = new ClassPathXmlApplicationContext(

newString[]{"servicesconfig.xml","daoconfig.xml"});

**2) *FileSystemXmlApplicationContext:***

            It loads bean definitions from XML files in the file system.

*Example:*

ApplicationContext context = new FileSystemXmlApplicationContext("c:/myconfig.xml");

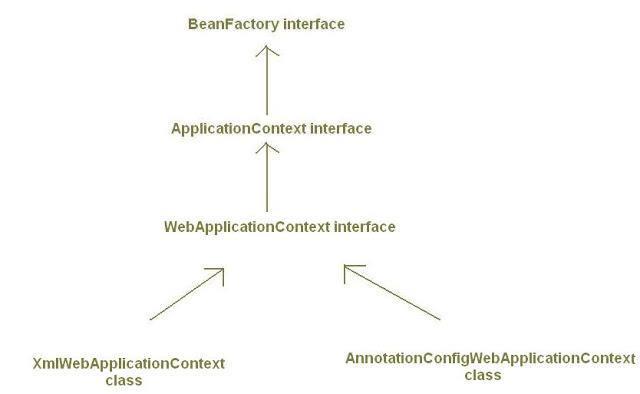
**3)*XmlWebApplicationContext:***

*XmlWebApplicationContext* is used to represent Spring Container for web applications. By default Spring creates object of *XmlWebApplicationContext* class to represent application context/spring container for web applications.

It loads bean definitions from an XML file contained within a web application. By default it loads the configuration from file *"/WEB-INF/applicationContext.xml".*

If we want to load bean definitions from more than one xml files we can specify their locations in  *contextConfigLocation*  parameterof *ContextLoaderListener* or *DispatcherServlet* in web.xml.[Read more about this here.](http://myjavadiary.blogspot.in/2012/08/webappicationcontext-and_5211.html)

*XmlWebApplicationContext* is an implementation of *WebApplicationContext* interface which in turn extends *ApplicationContext*interface.

[](https://1.bp.blogspot.com/--Ejm4KKBCk8/UC-Sr9zanoI/AAAAAAAAAGY/57DP2Z13L1A/s1600/applicationcontext_8.JPG)

***4)AnnotationConfigApplicationContext:***

*AnnotationConfigApplicationContext* class is used when we are using *Java-based configuration* for the bean definitions*instead of Xml files*.

Inabove *ApplicationContext* implementations (*ClassPathXmlApplicationContext, FileSystemXmlApplicationContext*) we have supplied bean configuration from xml configuration files. *AnnotationConfigApplicationContext* class is used to create Spring container which takes bean definitions from java classes annotated with @Configuration, instead of xml files.

It is introduced in Spring 3.0.

*Example*:

public static void main(String[]args){

/\* Creating Spring IoC Container Without XML configuration file\*/

ApplicationContext context= new AnnotationConfigApplicationContext(MyConfig.class);  
  
MyBean beanObj = context.getBean(MyBean.class);  
  
beanObj.someMethod();

}

In above code, *AnnotationConfigApplicationContext* is accepting*MyConfig* class as input. Here we are obtaining bean definitions from a java class named *MyConfig* annotated with @Configuration, instead of a Xml file. *MyConfig* class is described as:

@Configuration

public class MyConfig{

            @Bean

            public MyBean myBeanId(){

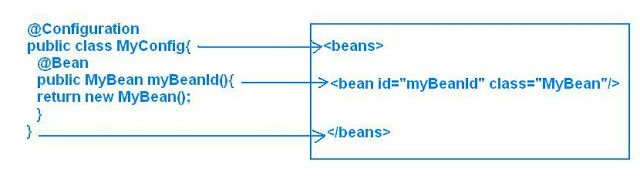
            return new MyBean();

            }

}

By giving @Configuration annotation we are treating *Myconfig* class as  <beans></beans>  tag of xml file.

By giving @Bean annotation we are treating myBean() method as  <bean id="..." class="..."/>

[](https://1.bp.blogspot.com/-_80gXx0KxOo/UC_BXzbWZ0I/AAAAAAAAAGw/ZYOgrNCDYg8/s1600/applicationcontext_9_1.JPG)

Name of myBeanId() method will be treated as *bean id*.

Both @Configuration and @Bean are also introduced in Spring 3.0.

***5)AnnotationConfigWebApplicationContext:***

Like *XmlWebApplicationContext* is web counterpart for the *ClassPathXmlApplicationContext* and *FileSystemXmlApplicationContext* and is used to create application context for web  applications, similarly, *AnnotationConfigWebApplicationContext* is web counterpart for *AnnotationConfigApplicationContext.*

*AnnotationConfigWebApplicationContext* is used to create application context for web applications by using java classes as input for bean definitions instead of xml files.

By default Spring use *XmlWebApplicationContext* (an implementation of *WebApplicationContext*) for creating spring container in web applications. But we can change this default value to *AnnotationConfigWebApplicationContext* by changing the value of *contextClass* parameter of *ContextLoaderListener* or *DispatcherServlet* in web.xml as shown below:

*For ContextLoaderListener:*

<web-app>

<context-param>

<param-name>**contextClass**</param-name>

<param-value>

org.springframework.web.context.support.**AnnotationConfigWebApplicationContext**

</param-value>

</context-param>

<context-param>

<param-name>contextConfigLocation</param-name>

<!--MyConfig must be annotated with @Configuration-->

<param-value> MyConfig</param-value>

</context-param>

<listener>

<listener-class> org.springframework.web.context.ContextLoaderListener </listener-class>

</listener>

</web-app>

*For* *DispatcherServlet:*

<web-app>

<servlet>

<servlet-name>mydispatcher</servlet-name>

<servlet-class > org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>**contextClass**</param-name>

<param-value>

 org.springframework.web.context.support.**AnnotationConfigWebApplicationContext**

</param-value>

</init-param>

<init-param>

<param-name>contextConfigLocation</param-name>

<!--MyConfig must be class annotated with @Configuration-->

<param-value> MyConfig </param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>mydispatcher</servlet-name>

<url-pattern>\*.htm</url-pattern>

</servlet-mapping>

</web-app>

What are the ways of creating Beans?

1. Using @Bean annotation
   1. Need to specify all the methods which return a bean in @Configuration class
   2. Default id of the bean is methodName
2. Using @Component annotation
   1. Need to specify @ComponentScan(basePackages=””) in @Configuration class
   2. Default id of the bean is de-capitalized class name

**Connection pools**

There is the DataSource interface in Java

Various implementations for this interface

1. DBCP Apache

2. C3PO

3. Proxool

4. Tomcat connection pool

So now we can use any of these implementations for getting connection pool

We need to set the properties for the selected implementation

The properties are

1. driverClass

2. url

3. userName

4. password

If we use DBCP then we need commons-dbcp.jar, commons-pool.jar

The DataSource implementing class is BasicDataSource

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="...">

.

.

.

</bean>