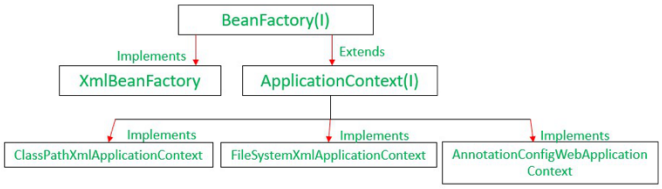
Spring container is the core of Spring Framework. Spring container creates Objects and associates them. Spring Container uses dependency injection to create associations between objects.

Spring comes with 2 different implementations of container.

1. **Bean Factory:** This is the simplest of containers providing basic support for DI. Represented by interface **org.springframework.beans.factory.BeanFactory.**

2. **Application Context:** Application context is a sub interface of BeanFactory interface. BeanFactory offers basic container, application context has enterprise capabilities like
    a. Spring AOP
    b. Message Resource Handling
    c. Application specific contexts like WebApplicationContext etc
    d. Event Propagation to beans.

Represented by **org.springframework.context.ApplicationContext**.



org.springframework.context.ApplicationContext interface represents the Spring IOC container and is responsible for creating and associating the beans. Container achieves this by reading configuration meta-data defined in XML / Annotation / Java code.

Wiring:
The act of creating associations between application objects is called dependency injection (DI) and is commonly known as wiring.

Spring Container is responsible for creating associations between spring components.

Spring offers 3 wiring mechanisms
1. Configuration using XML
2. Configuration in Java
3. Auto bean discovery and auto wiring.

Bean Scoping:
By default, all beans are singletons. Singleton means there will be only one instance of the bean and same bean will be injected, when referenced.

Scopes
1. **Singleton** – Bean will have only one instance per spring container.
2. **Prototype** - A new bean will be created for each injection.
3. **Request** – Bean scope is valid for a HttpRequest. Used only in web capable context such as MVC.

4. **Session** – Bean scope is valid for a HttpSession. Used only in web capable context such as MVC
5. **Global-session** – Valid only for portlet context.

Wiring with Anotations

Autowiring with annotations isn't much different than using the autowired attribute in XML. But it does allow for more fine-trained autowiring, where you can selectively annotate certain properties for autowiring.

```
<?xm! version="1.0"encodingeUTF-8"?>
<beans xmlnsehttp://www.springframeworkoresehema/beans"
xmlns:xsi="http://www.w3.org/2001/XMISchema-instance"
xmlns:context="http://www.springframework.org/sehema/contexts
xsi:schemalocationehttp://wvmspringframework.oreschema/beans
http://ww.springframework.oreschema/beans/spring-beans-3.0.nd
http://www.springframework.oreschema/context
httixthremspringframewortoreschema/context/spring-context-3.0.xsd">

<context:annotation-config/>

</beans>
```

<context:annotation-config> tells Spring that you intend to use annotation-based wiring in Spring.

@Autowired

Annotating with @Autowired on method and instance variable will perform byType autowiring.

Annotating with @Autowired on constructor will perform constructor autowiring.

```
@Autowired
private Instrument instrument

@Autowired
public void heresYourInstrument(Instrument instrument) {
        this.instrument = instrument;
}
```

By default @Autowired has strong contract, which means if the bean is not found then autowiring fails with

NoSuchBeanDefinitionException.

If the developer thinks that bean injection is optional and a null value is acceptable, in that case add required attribute.

```
@Autowired(required=false)
private Instrument instrument;
```

Automatic wiring
This wiring mechanism works without having to mention any configuration in xml and java.

This is a 2-step process
1. Automatic Discovery
2. Auto wiring

Automatic Discovery
Spring automatically discovers beans to be created in the application context.

For automatic discovery to work we must enable support for annotations. Use below to enable support for annotations.

<context: annotation-config/>

The above configuration will add support for annotations, but it doesn't automatically discover the beans.

Use **<context:component-scan/>** for auto discovery.

The **<context:component-scan>** element does everything that **<context:annotation-config>** does, plus it configures Spring to automatically discover beans and declare them for you.

By default, **<context:component-scan>** looks for classes that are annotated with one of a handful of special stereotype annotations:

@Component—A general-purpose stereotype annotation indicating that the class is a Spring component

@Controller— Indicates that the class defines a Spring MVC controller

@Repository— Indicates that the class defines a data repository

@Service—Indicates that the class defines a service.

Filtering component scans:

You can include or exclude component to scan

<context:component-scan base-package="com.springinaction.spring idol"> <context:include-filtertype="assignable" expression="com.springinaction.spri ngidol.Instrument"/> </context.component-scan›
So, instead of relying on annotation-based component scanning, you can ask <context: component - scan> to automatically register all classes that are assignable to Instrument by adding an include filter,
<context:component-scan base•package="coru.springinattion. springidol"> <context:include-filter type="assignable" expression="com.springinaction.spri ngidol.Instrument"/> <context:exclude•filter type- "annotation"

```
expression="com.springinaction.spri
ngidol.SkipIt"/>
</context:component-scan>
```

Just as **<context:include-filter>** can be used- to tell <context : component-scan> what it Should register as beans, you can use <context : exclude-filter> to tell It what not to register. For example, to register all Instrument implementations except for those annotated with a custom @SkipIt annotation

Auto-Wiring

Spring itself will automatically inject the bean dependencies without developer specifying the dependencies.

4 kinds of auto-wiring

byName – Attempts to match all properties of the autowired bean with beans that have the same name as the properties. Properties for which there is no matching beans will remain un-wired.

byType – Attempts to match all properties of the autowired bean with beans whose type is assignable to the properties.

Autowlring using byType works in a similar way to byName, except that instead of considering a property's name, the property's type is examined. When attempting to autowire a property by type, Spring will look for beans whose type is
assignable to the property's type.

But there's a limitation to autowiring by type. What happens if Spring finds more than one bean whose type is assignable to the autowired property? In such a case, Spring isn't going to guess which bean to autowire and will instead throw an exception.
Consequently, you're allowed to have only one bean configured that matches the autowired property.
To overcome ambiguities with autowiring by type, Spring offers two options: you can either identify a primary candidate for autowiring or you can eliminate beans from autowiring candidacy

For example, to establish that the saxophone bean isn't the primary choice when autowiring Instruments:

```
<bean id=”saxophone"
classecom.springinaction.springidol.
Saxophone" />
<bean id="trumphet"
class="contspringinaction.springidol.
Trumphet” primary="false”/>
```

Trumphet extends Saxophone.

The primary attribute is only useful for identifying a preferred autowire candidate. If you'd rather eliminate some beans from consideration when autowiring, then you can set their autowire-candidate attribute to false, as follows:

```
<bean id="saxophone" classe="com.springinaction.springidol.Saxophone" />

<bean id="trumphet" class="com.springinaction.springidol.Trumphet" autowire-candidate="false"/>  This will be ignored for autowiring
```

Constructor – Tires to match up a constructor of the autowired bean with beans whose types are assignable to the constructor arguments.

Autodetect – Attempts to apply constructor autowiring first. If that fails byType will be tired.

@Qualifier

The @Qualifier annotation is used to resolve the autowiring conflict, when there are multiple beans of same type.

The @Qualifier annotation can be used on any class annotated with @Component or on method annotated with @Bean. This annotation can also be applied on constructor arguments or method parameters.

Problem:

```
public interface Vehicle (
//.
}.
@Component
public class Car implements Vehicle {
//..
}
@Component
public class Bike implements Vehicle{
//..
}
@Component
public class VehicleService (
   @Autowired
   private Vehicle vehicle;
//..
}

It will throw below error during spring initialization
```

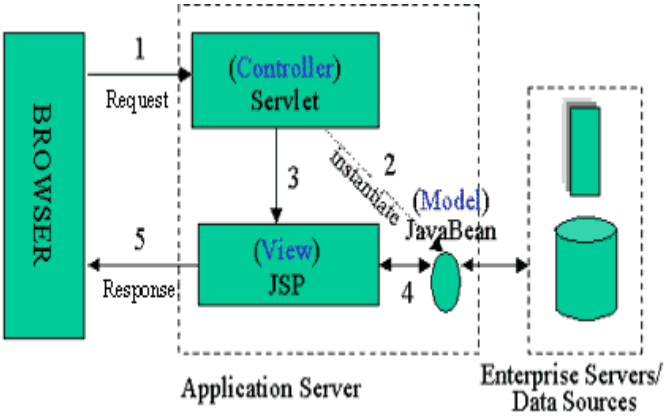
Solution:

Use @Qualifier

```
@Component
@QualifierrcarBean")
public class Car implements Vehicle
{
//...
//...
}
@Component
@Qualifier("bikeBean")
public class Bike implements Vehicle{
//...
//... .
}
@Component
```

```
public class VehicleService {
   @Autowired
   @Qualifier("carBean")
   private Vehicle vehicle;
   //...
}
```

MVC Design Pattern



Purpose of MVC pattern is to decouple Model from the presentation (view) layer. If the application has more than one presentation, we can replace only the view layer and re-use code for controller and model.

Model:

Model is the data and business-logic component. A model can serve multiple views. Model can be a Java bean or EJB.

View:

This is the presentation component also known as user-interface component. View renders the state of the model associated with the view. View component can be implemented using JSP pages.

Controller:

The controller translates user events into actions to be performed by the model. The actions performed by the model include executing business logic and changing the state of the model. Based on the user interactions and the outcome of the model actions, appropriate view is selected by the controller.

Front Controller Design pattern

The front controller design pattern is used to provide a centralized entry point for all requests so that all requests are handled by front controller.

The front controller is responsible for common tasks like

1) Authentication and Authorization
2) Delegating business processing
3) Deciding appropriate view
4) Handling errors.

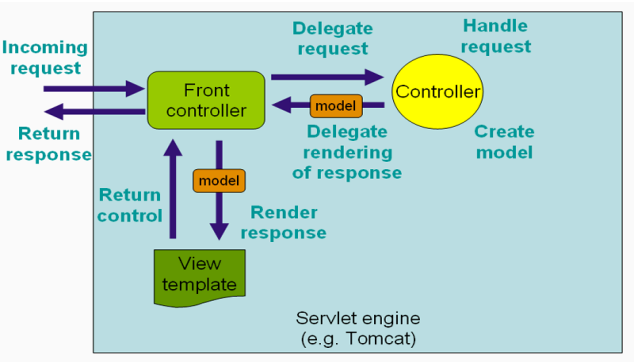
Spring MVC

In Spring MVC DispatcherServlet class acts as a front controller.

The DispatcherServlet is an actual servlet that inherits from HttpServlet class. Dispatcher servlet will be declared in the web.xml of your web application.

```
<web-app>
  <servlet>
      <servlet-name>dispatcher</servlet-name>
      <servlet-class>org.springframework.
                     web.servlet.DispatcherServlet
      </servlet-class>
  </servlet>
  <servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>/*</url-pattern>
  </servlet-mapping>
```

Request flow:



1) All incoming requests are handled by Dispatcher Servlet (Front Controller). Dispatcher Servlet delegates the requests to Spring controllers or handlers which will execute the business logic and creates the Model.

2) Dispatcher servlet identifies the view based on spring controller return type and gives the Model to the view.

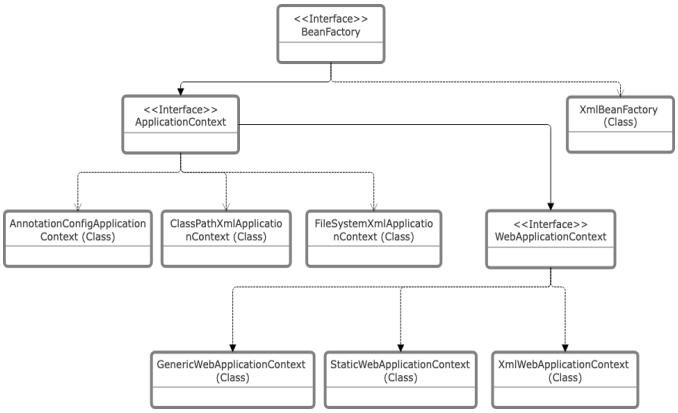
3) The view will be responsible for displaying the state of the Model given by the Dispatcher Servlet.

The core responsibility of a DispatcherServlet is to dispatch the incoming requests to the correct handlers like spring controllers.

Understanding Application Context

The interface **org.springframework.context.ApplicationContext** represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code.

Several implementations of the ApplicationContext interface are supplied out-of-the-box with Spring. In standalone applications it is common to create an instance of ClassPathXmlApplicationContext or FileSystemXmlApplicationContext.



Application Context Types

1) **AnnotationConfigApplicationContext**.
   Loads a Spring application context from one or more Java-Based configuration classes.

```
ApplicationContext context = new
AnnotationConfigApplicationContext(JavaConfig.class);
```

2) **AnnotationConfigWebApplicationContext**.
   Loads a Spring web application context from one or more Java-Based configuration classes.

```
ApplicationContext context = new
AnnotationConfigWebApplicationContext(JavaWebConfig.class;
```

3) **ClassPathXMLApplicationContext**
   Loads a context definition from one or more XML files located in the classpath

```
ApplicationContext context = new
ClassPathXMLApplicationContext("spingConfig.xml");
```

4) **FileSystemXMLApplicationContext**.
   Loads a context definition from one or more XML files located in the filesystem.

```
ApplicationContext context = new
FileSystemXMLApplicationContext("c:/spingConfig.xml");
```

5) **XMLWebApplicationContext**.
   Loads a context definition from one or more XML files contained in the web application.

```
ApplicationContext context = new
XMLWebApplicationContext("/WEB-INF/config/spingConfig.xml");
```

In Spring web application has web application contexts.

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always look up the WebApplicationContext if you need access to it.

For spring web application there can be multiple application contexts.

1) Root web application context.
2) Web application context for each dispatcher servlet defined in the application.

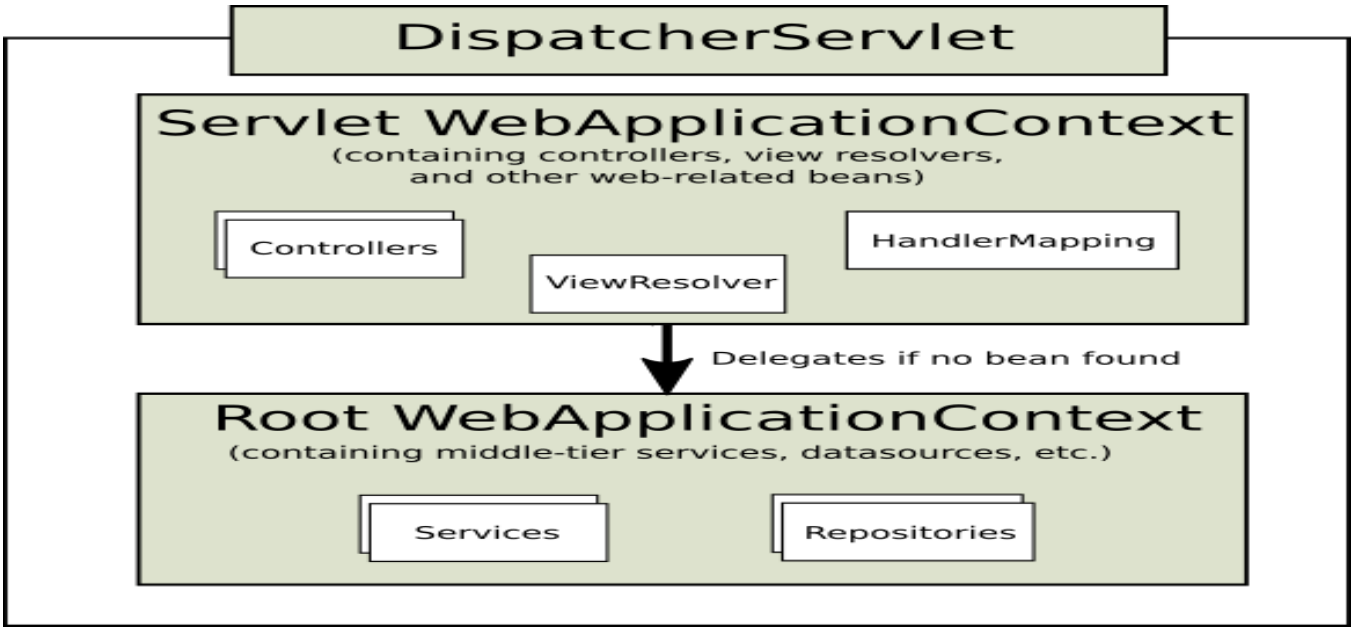
In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in the

Root WebApplication Context. These inherited beans can be overridden in the servlet-specific scope, and you can define new scope-specific beans local to a given Servlet instance.

Upon initialization of a DispatcherServlet, Spring MVC looks for a file named [servlet-name]-servlet.xml in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

Root web application context	**Web application context (Dispatcher-servlet)**
There is only one root web application context per web application. The root web application context is created by ContextLoaderListener which is declared in web.xml <web-app> <**listener**> <**listener-class**>org.springframework.web.context.ContextLoader Listener</**listener-class**> </**listener**> <**context-param**> <param-name>**contextConfigLocation**</param-name> <param-value>/WEB-INF/root-application-context.xml</param-value> </**context-param**> </web-app> The listener inspects the contextConfigLocation parameter. If the parameter does not exist, the listener uses /WEB-	There can be multiple web application. Contexts one per dispatcher servlet defined in the application. <**servlet**> <servlet-name>dispatcher</servlet-name> <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class> <load-on-startup>1</load-on-startup> </**servlet**> <**servlet-mapping**> <servlet-name>dispatcher</servlet-name> <url-pattern>/*</url-pattern> </**servlet-mapping>** </servlet>. With the above Servlet configuration in place, you will need to have a file called **/WEB-INF/dispatcher-servlet.xml** in your application; this file will contain all of your Spring Web MVC-specific components (beans). All beans defined in golfing-servlet.xml will be created in the dispatcher servlet specific webapplicationcontext. We can define the xml file name using init-param like below. <servlet> <servlet-name>dispatcher</servlet-name>

INF/applicationContext.xml as a default. When the parameter does exist, the listener separates the String by using predefined delimiters (comma, semicolon and whitespace) and uses the values as locations where application contexts will be searched.	<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class> <init-param> <param-name>contextConfigLocation</param-name> <param-value>/WEB-INF/dispatcher-servlet.xml</param-value> </init-param> <load-on-startup>1</load-on-startup> </servlet> <servlet-mapping> <servlet-name>dispatcher</servlet-name> <url-pattern>/</url-pattern> </servlet-mapping> </servlet>



Single root web application context in Spring MVC:

If you need only one root web application context and all servlet application contexts simply inherits the root context.

This can be configured by setting an empty **contextConfigLocation** servlet init parameter, as shown below:

```
<web-app>
   <context-param>
      <param-name>contextConfigLocation</param-name>
```

```
      <param-value>/WEB-INF/root-application-context.xml</param-value>
   </context-param>
   <servlet>
      <servlet-name>dispatcher</servlet-name>
      <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <init-param>
         <param-name>contextConfigLocation</param-name>
         <param-value></param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
      <servlet-name>dispatcher</servlet-name>
      <url-pattern>/*</url-pattern>
   </servlet-mapping>
   <listener>
      <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
   </listener></web-app>
</web-app>
```