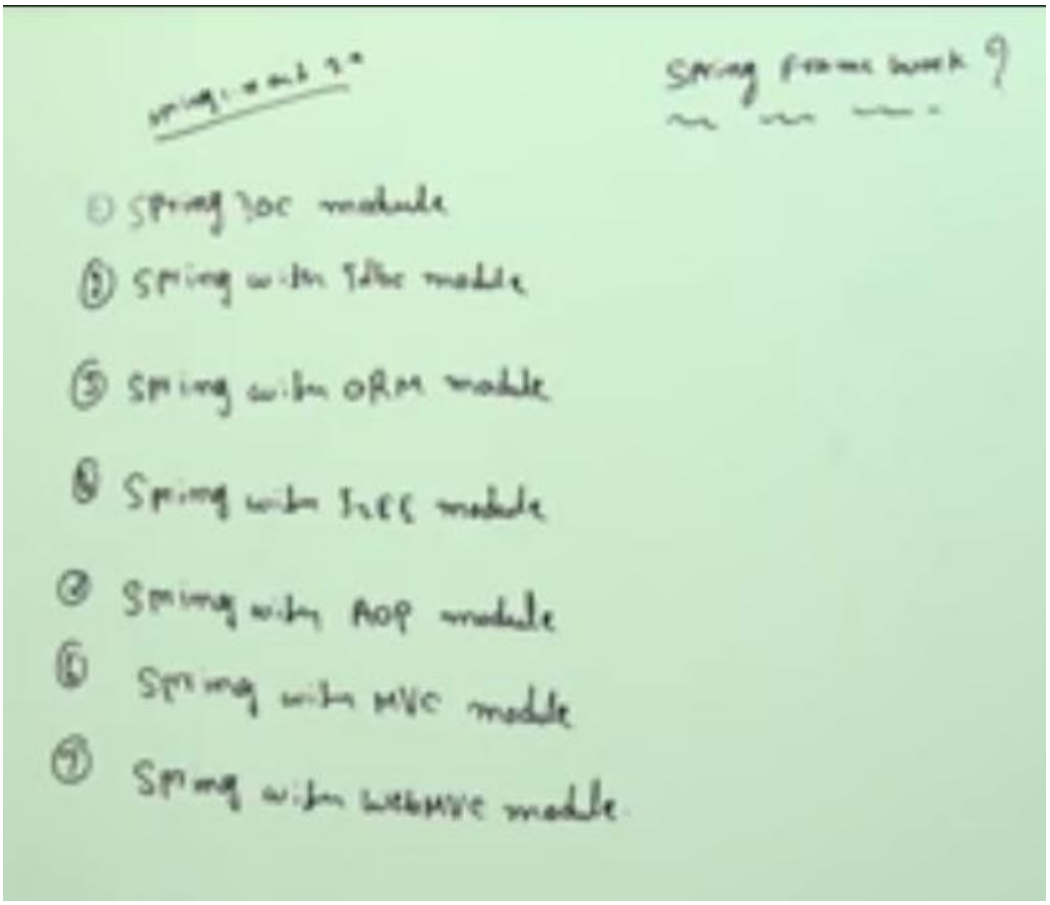
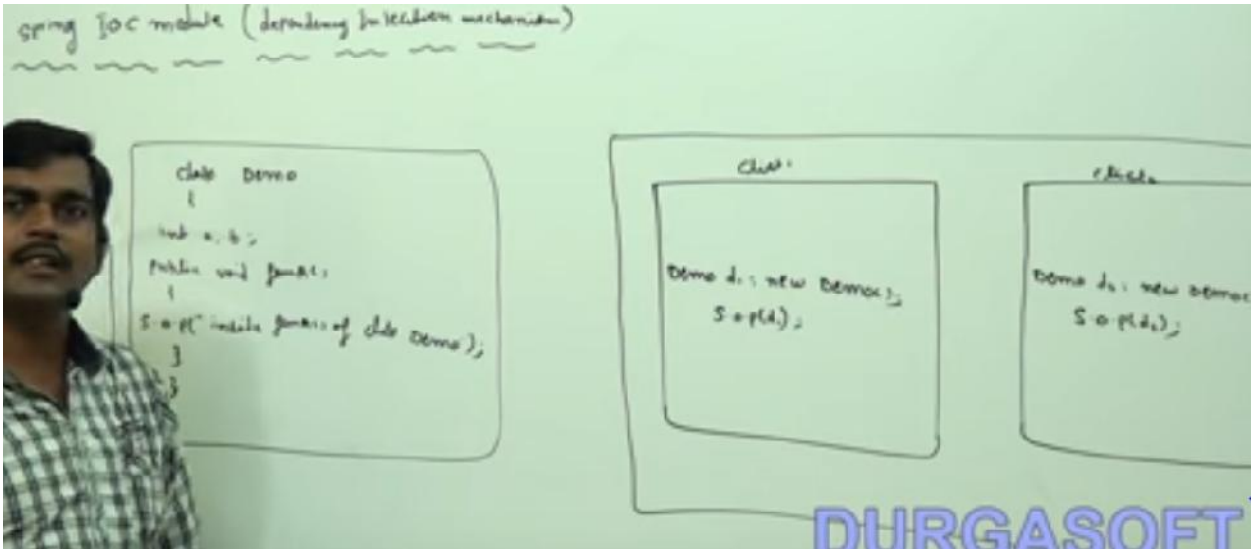


Spring IOC or Dependency Injection Mechanism:

As a developer we need to only concentrate on writing the business logic.

Object creation should be handled by the Spring Framework.



EJBs, (Heavy weight and tightly coupled.) bcz they

Require application server to run.

Spring, (Light weight and loosely coupled.)

Does not have any dependency on application server. Simply JDK and spring jar files are enough to run.

By using interface implementation model and java runtime polymorphism, we can achieve loose coupling between two layers.

- To get your class capabilities of servlet or servlet methods, we need to implement the servlet interface or need to extend GenericServlet abstract class or HttpServlet.
- Spring encourages association to achieve capabilities of certain class to a given class, rather than extending or implementing it.
- Encourages association instead of inheritance.
- If you use a class without implementing or extending other class or interface we can call that class is a pojo class.
- Spring is implemented using two principles
    1. Association(HAS-A)
    2. Runtime Polymorphism
- IOC use XML inputs and pass the runtime inputs to POJO classes.
- i.e. HAS-A relationship inputs we can pass from our IOC container.
- IOC has two containers:
    1. Core container.(BeanFactory)(I)
    2. J2EE container.(ApplicationContext)(I)
- In MVC we have web container(build on top of IOC container)(WebApplicationContext)(I)

- A web container like Tomcat has the capabilities to read the web.xml file and then it will try to initialize the load on start up servlets. Create the servlet objects.

- Then it will call the init(), method of servlet life cycle. Thus it has the capability to manage the life cycle of the servlet.

IOC Container:

- Read XML
- Create instances of pojo classes.
- Manage Life cycle of pojo classes.
- Can pass dynamic arguments to pojo classes.(Dependency Injection)

IOC container Implementations:

- BeanFactory(I)
    1. XMLBeanFactory(C)

- ApplicationContext(I)-> ConfigurableApplicationContext(I)
    1. ClassPathXMLApplicationContext.
- WebApplicationContext(I)
    1. WebApplicationContextUtil(Factory class)

BeanFactory vs ApplicationContext:

- BeanFactory creates object on demand i.e. it create object when the getBean method is called. (Lazy container)
- ApplicationContext creates object when we load the spring config XML file provided the scope is singleton.(Eager or early container)
- For scope is prototype the instantiation behavior is same for BeanFactory and ApplicationContext.
- Spring can access your private constructor.
- IOC container can create instances of classes with private constructor also.
- Class.forName("classNameString").newInstance()

IOC Container Supports:

1. POJO Instantiation
2. Life cycle management
3. Dependency Injection.

Dependency Injection:

1. Setter injection
2. Constructer injection

Types of data we can inject:

1. Primitive
2. Secondary
3. Primitive array
4. Secondary Array
5. Collections

-Property tag will call the setter method (setter injection.)

Secondary Property injection:

1. Pass by reference.
2. Pass by object.(Inner bean approach)(created in between property or in between constructor-arg)
3.

Dependency-checking:

- To make setter DI mandatory.
- Dependency-check(none, simple(compulsory to pass primitive values), objects)
- To make some properties mandatory use @Required annotation.(in setter methods)
- For this we need to create object of RequiredAnnotaionBeanPostProcessor.

Depends on concept:

- Without engine car won't work.

P: and c: namespaces:

Auto wiring and Bean Lifecycle management:

- To do automatic dependency injection.
- We can only inject secondary types with auto wiring.
- Autowire=("byType", "byName", "constructor", "no")
- If multiple bean definitions found (autowire-candiadate ="false")
- For autodetect if both constructor and setter is present, if default constructor is present then autowire bytype(setter) or else by parameterized constructor.
- byname, bean id should match with dependency parameter name.
- Annotation @Autowire(byType)→default
- To solve ambiguity @Qualifier("id")

Spring can also use auto scanning to create object of the beans automatically.

Stereotype annotations: container will do the DI and also can create objects.

1. @Controller→MVC controller
2. @Repository→DAO package classes
3. @Service→Business classes
4. @Component →Non MVC classes

Static Dependency Injection:

MethodInvokingFactoryBean(C), by using this class we need to pass

- Argument(object[] argument)
- StaticMethod(String staticMethod)

```
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod" value="foo.bar.Class.setTheProperty"/>
    <property name="arguments">
        <list>
            <ref bean="theProperty"/>
        </list>
   </property>
</bean>
```

OR

Solution

To fix it, create a “none static setter” to assign the injected value for the static variable. For example :

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component
public class GlobalValue {

    public static String DATABASE;

    @Value("${mongodb.db}")
    public void setDatabase(String db) {
        DATABASE = db;
    }

}
```

```
@Component
public class Car {

    private static Engine engine;

    public void startEngine() {
        System.out.println("Engine started " + engine.makeNoise());
    }


    public static void setEngine(Engine engine) {
        Car.engine = engine;
    }
```

```java
@Component
public class Engine {

    public String makeNoise() {
        return "Ghreee Ghree...";
    }

}
```

```xml
<context:component-scan base-package="beans" />
<context:annotation-config />

<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod" value="beans.Car.setEngine" />
    <property name="arguments">
            <list>
                <ref bean="engine"/>
            </list>
    </property>
</bean>
```

```java
public class Client {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("resources/Spring.xml");
        Car car = (Car)context.getBean(Car.class);
        car.startEngine();
    }

}
```

Singleton or Factory classes:

- Singleton example in JRE (java.util.ResourceBundle) (used for internationalization).
- Calendar class. (Calendar c = Calendar.getInstance())
- Use of singleton: if your class contains static resources, multiple instances unnecessarily will consume heap space.
- So make that class singleton.
- Factory-method

Difference between singleton and factory classes:

- Factory method inside the singleton class returns same class object.
- Factory class instances return other class object.
- Factory class can contain static methods, which gives object or can also contain instance methods.
- Singleton classes should only contain static factory methods, which returns instance of same class.
- <bean id=”s” factory-bean=”sf” factory-method=”openSession”/>
- <bean id=”sf” class=”SessionFactory”/>

- The main aim of factory classes is, they will make client independent (e.g. DriverManager factory class gives us connection object in JDBC irrespective of any database.)
- They will always return interface references to make us independent.
- They will hide the object creation logic.

Instance factory and static factory:

- Hibernate session factory is an instance factory.
- Logger factory and drivermanager are example of static factories.
- http://www.javainterviewpoint.com/static-factory-method-instance-factory-method/

Static Factory:

```
public class ManagerFactory {

    public static Manager getManagerByName(String name) {
        switch (name) {
        case "SeniorManager":
            Manager seniorManager = new Manager("SeniorManager", 55, "tom@gmail.com", 8390061207L);
            return seniorManager;

        case "ProjectManager":
            Manager projectManager = new Manager("ProjectManager", 55, "jerry@gmail.com", 9390061207L);
            return projectManager;

        default:
            Manager manager = new Manager();
            return manager;
        }
    }

}
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" x
xsi:schemaLocation="http://www.springframework.org/sche
    http://www.springframework.org/schema/context http:

<context:component-scan base-package="beans" />
<context:annotation-config />

<bean id="managerFactory" class="beans.ManagerFactory"
      factory-method="getManagerByName">
  <constructor-arg value="SeniorManager"/>
</bean>

/beans>
```

FactoryBean(I) from spring to implement factory pattern:

- Available methods, getObject(): Object, getObjectType(): Class, isSingleton(): Boolean

Note: In Class.forName(), forName(), is a static factory method in class Class.

Bean Lifecycle:

- We can manage lifecycle of beans by using **configurableApplicationContext.**
- Servlet container (Tomcat) manages servlet lifecycle.
- Three approaches in spring to implement lifecycle method,
  1. Programmatic
     - Pojo, implement **initializingBean(I):afterpropertySet** and **DisposableBean:destroy()**
  2. XML File Approach
     - <bean id="c" class="" init-method="" destroy-method=""/>
  3. @Annotation approach
     - @postConstruct , @preDestroy

Destroy method in spring forcefully calls the garbage collector to delete all singleton objects.

Lookup Method DI:

A method not having any implementation is called lookup method.

E.g. methods inside interface and methods we are going to override.

- Spring can provide runtime implementations to the lookup methods.

http://www.javarticles.com/2015/05/spring-lookup-method-example.html

Method Replacers:

Can implement a replacer method to an existing method without extending the class.

To do Autowiring:

Spring: @Autowired, @Qualifier (to reduce ambiguity)

JDK Annotations: @Resource (will search byname first), @Inject(both are equivalent to autowiring)

Stereotype Annotations:

Spring: @Component, @Controller, @Service, @Repository

J2EE: @Named

Exporting properties data to a class:

Properties DI using expressions.

- To load property into IOC
  1. propertyplaceholderConfigurer(setLocation(String location))

IOC stores these data into the context scope.

I18N

To provide language support.

L10N

To provide localization.

To provide business support and validation support.

Event Handling:

Use configurable application context to do the event handling.

IOC(start, stop, close, refersh)

Spring MVC

- form backup support for the presentation part.(Form data we can hold in bean classes.)(can achieve this by spring tags.)
- controllers support.(multiaction, multiform)
- validation support.
- I18N

- Intercepters
- View resolvers.
- Exception handling

Front controller classes is given by the framework people.

Front Controller

- Jsp(jsp-Model1)
- Servlet(jsp-Model 2) |struct 1X| Spring MVC| JSF|
- Filters(jsp-Model 3) |structs 2X|
- TagSupport(jsp-Model 4)

For Spring MVC the front controller class name is **Dispatcher Servlet**.

Front controller DP:

- Map multiple actions to a single controller.

User controllers:
Programatic Approach

- Contrller(I): handleRequest()
- AbstractController(AC): handleRequestInternal()
- AstractCommandController
- SimpleFormController
- AbstractWizardFormController
- MultiActionController

View Controllers:

- Parameterizable viewController(setViewName, getViewName)
- UrlFileNameView Controller

Throwaway controller:

- Throwaway cntroller

Sterotype Annotation:
@Controller

InternalResourceViewResolver

Handler Classes in MVC:

- BeanNameUrlHandlerMapping

- SimpleUrlHandlerMapping(to map pattern to bean Id)
- ControllerClassNameHandlerMapping(to map pattern to controller classname)
- CommomsPathMapHandlerMapping

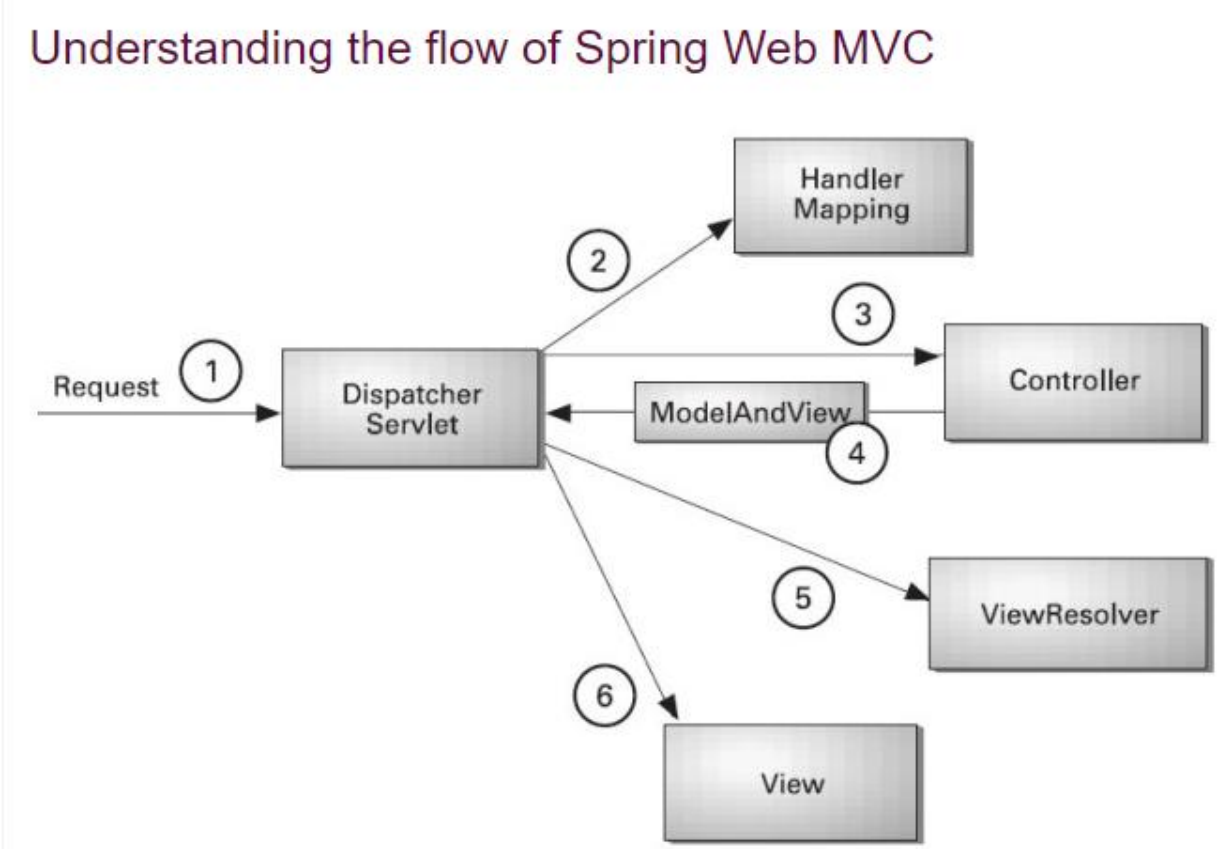
View Resolvers:

- InternalResourceViewResolver
- TilesViewResolver

Before going to the specific controller the request goes to the handlerMapping classes.

HandlerMapping will return the controller name to the dispatcher servlet.

Spring configuration file we can use default or can be configured by init or context param.



Data Access Object:

Spring-jdbc
Spring-ORM
Interface implementation model we must follow.
Any implementation object we can inject to the interface.

@ModelAttribute (“exercise”) Exercise exercise:
Used to get or post data.

- Binds a model to jsp pages.
- Can map in form:form tag using commandname(“exercise”)

Q. What is the difference between “forward:addMoreMinute.html” and “redirect:addMoreMinute.html” ?

Ans: forward works on the same request. Redirect closes the current request and creates the new one.

Tag Libs:

1. Spring.tld
2. Spring-form.tld

Read html text from property file:

ResourceBundleMessageSource.

Intercepters:

1. Used as a part of tag libs.
2. Intercepting data comes from jsp pages and going to the controller.
3. Can have the ability to pre handle or post handle the web requests.
4. Callback methods used to override or change values
5. Commonly used for Locale Changing.
6. We need sessionLocaleResolver and and localeChangeIntercepter.

Session:

@SessionAttribute, above class

Validation:

JSR-303: java standard validation not spring standard.

AutoWring:

@Component

@Autowired

Spring MVC 4:

1. MV* patterns
2. Controllers become lighter weight and Rest based controllers added to remove response body things.
3. Java configuration, servlet 3.0 specification
4. MVC is still a sound pattern
5. MVP, MVVM, MV*
    - JavaScript Frameworks(e.g. Angular, Backbone)
    - Mobile Applications
    - Responsive pages
    - RESTful Services

ModelView-ViewModel

1. Originated from .Net
2. RESTfull Backend
3. Rich JavaScript Frontend
4. Layers cleanly separated

XML Configuartion is reduced.

New @RestController annotation.

@EnableWebMvc

--Convenience annotations similar to namespaces

Convention over configuration.

Controller Responsibilities:

1. Interpret user input and transform that input to a model.
2. Provide access to business logic.
3. Determines view based off of logic
4. Interprets exceptions from the business logic/ service tier

Controller Annotations:

- @Controller
- @RestController (New to Spring 4)
- @Configuration (determines a class which is used to configuring our application context i.e. Signifies a configuration class)
- @EnableWebMvc (Enables our Java configuration, Boot strap our application)
- @ComponentScan(Specifies scan location for controllers)

Enable Web MVC

- Convenience annotation for WebMvcConfigurationSupport

- Only used for Java configuration for spring MVC web apps

Component Scan

@ComponentScan(basePackages = “com.abinash”)

Without web.xml:

- Still need a mapping somewhere
- WebApplicationInitializer(Servlet 3.0 hooks)
- Builds ApplicationContext

Spring Boot:

Spring boot makes it easy to create stand-alone production grade spring based applications that you can just run.

Spring is lightweight dependency injection, aspect oriented container and Framework.

Now it’s a complete Application Framework. It does lot more along with dependency injection now.

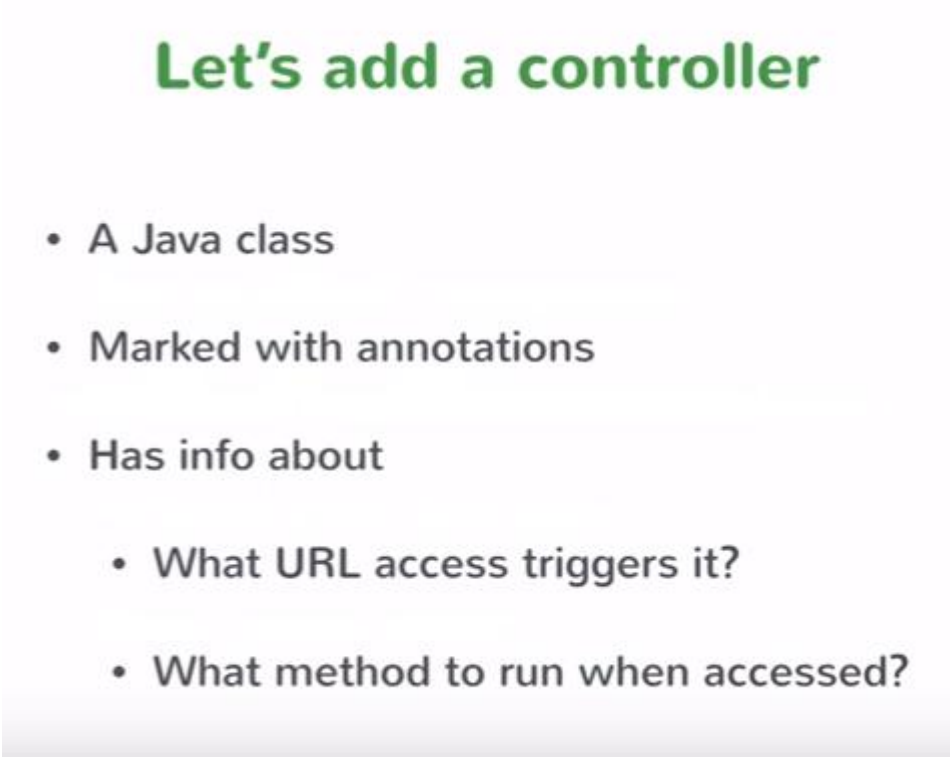
Spring boot features:

1. Convention over configuration.
2. Stand alone.
3. Production ready

```java
package io.javabrains.springbootstrater;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CourseApiApp {

    public static void main(String[] args) {
        SpringApplication.run(CourseApiApp.class, args);
    }

}
```

Starting Spring Boot

- Sets up default configuration
- Starts Spring application context
- Performs class path scan
- Starts Tomcat server



- A Controller is basically a java pojo class with @Controller annotation.
  Pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instanc
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.javabrains.springbootquickstart</groupId>
    <artifactId>course-api</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name> Java Brains Course API</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.6.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
    </parent>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>

    <properties>
        <java-version>1.8</java-version>
    </properties>
</project>
```

Embedded Tomcat Server

- Convenience
- Servlet container config is now application config
- Standalone application
- Useful for microservices architecture

If we use @RestController, returned object automatically converted into JSON.

@Service

Denotes that a pojo class is a service so that the component scanner identifies it.

Default scope is singleton.

Ways to create a spring boot app.

Start.spring.io

Application.properties:

Search for common application properties.

Spring Data JPA:

A separate project to work with the ORM tools in a better way.

Spring boot Actuator:

To know the health of the application.

Spring Transaction Management:

Declarative and Programmatic Transaction Management.

1. Support for most of the transaction APIs such as JDBC, Hibernate, JPA, JDO, JTA etc. All we need to do is use proper transaction manager implementation class. For example `org.springframework.jdbc.datasource.DriverManagerDataSource` for JDBC transaction management and `org.springframework.orm.hibernate3.HibernateTransactionManager` if we are using hibernate as ORM tool.
2. @Transactional annotation can be applied over methods as well as whole class. If you want all your methods to have transaction management features, you should annotate your class with this annotation

In Spring boot:

Inside application class(main method)

@EnableTransactionManagement

==========

WebConfig.java

```java
package springmvc_example.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = "springmvc_example")
public class WebConfig extends WebMvcConfigurerAdapter{

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");
    }

    @Bean
    public InternalResourceViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setViewClass(JstlView.class);
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

WebConfig.java WebInitializer.java

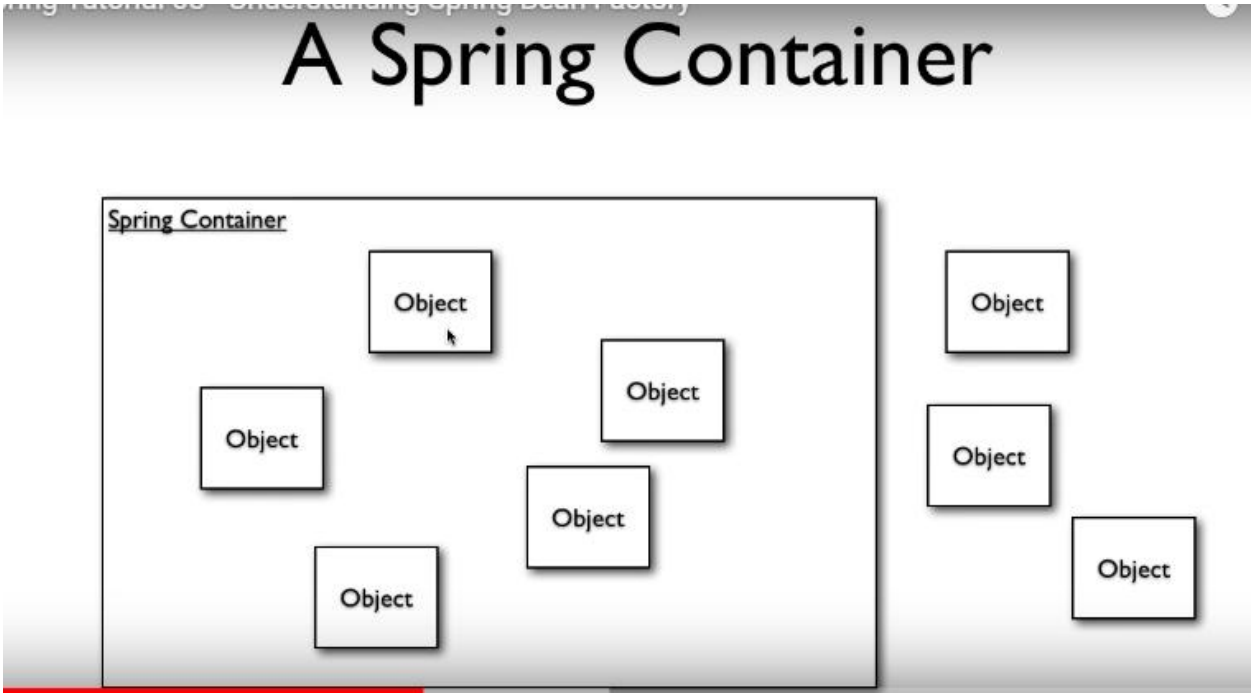
```java
package springmvc_example.config;

import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class WebInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[]{ WebConfig.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return null;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

}
```

WebConfig.java | WebInitializer.java | config.properties

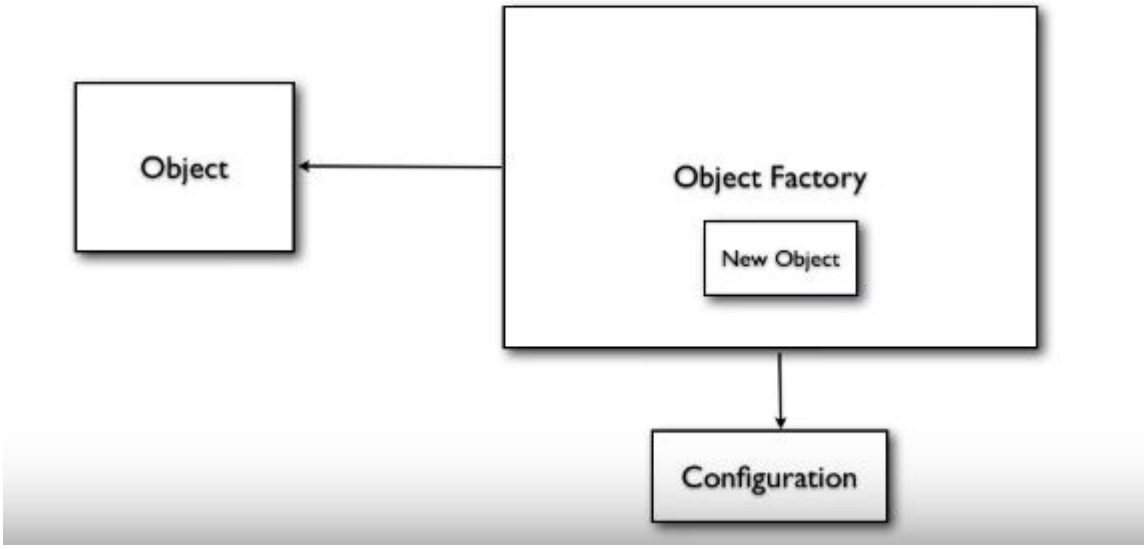
```
jdbc.driverClassName = com.mysql.jdbc.Driver
jdbc.url = jdbc:mysql://localhost:3306/springmvc
jdbc.username = root
jdbc.password = root
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.show_sql = true
hibernate.format_sql = false
```

```
    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
        sessionFactory.setDataSource(dataSource());
        sessionFactory.setPackagesToScan(new String[] { "springmvc_example.model" });
        sessionFactory.setHibernateProperties(hibernateProperties());
        return sessionFactory;
     }

    @Bean
    public DataSource dataSource() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource();
        dataSource.setDriverClassName(environment.getRequiredProperty("jdbc.driverClassName"));
        dataSource.setUrl(environment.getRequiredProperty("jdbc.url"));
        dataSource.setUsername(environment.getRequiredProperty("jdbc.username"));
        dataSource.setPassword(environment.getRequiredProperty("jdbc.password"));
        return dataSource;
    }

    private Properties hibernateProperties() {
        Properties properties = new Properties();
        properties.put("hibernate.dialect", environment.getRequiredProperty("hibernate.dialect"));
        properties.put("hibernate.show_sql", environment.getRequiredProperty("hibernate.show_sql"));
        properties.put("hibernate.format_sql", environment.getRequiredProperty("hibernate.format_sql"));
        return properties;
    }

    @Bean
    @Autowired
    public HibernateTransactionManager transactionManager(SessionFactory s) {
       HibernateTransactionManager txManager = new HibernateTransactionManager();
       txManager.setSessionFactory(s);
       return txManager;
    }

}
```

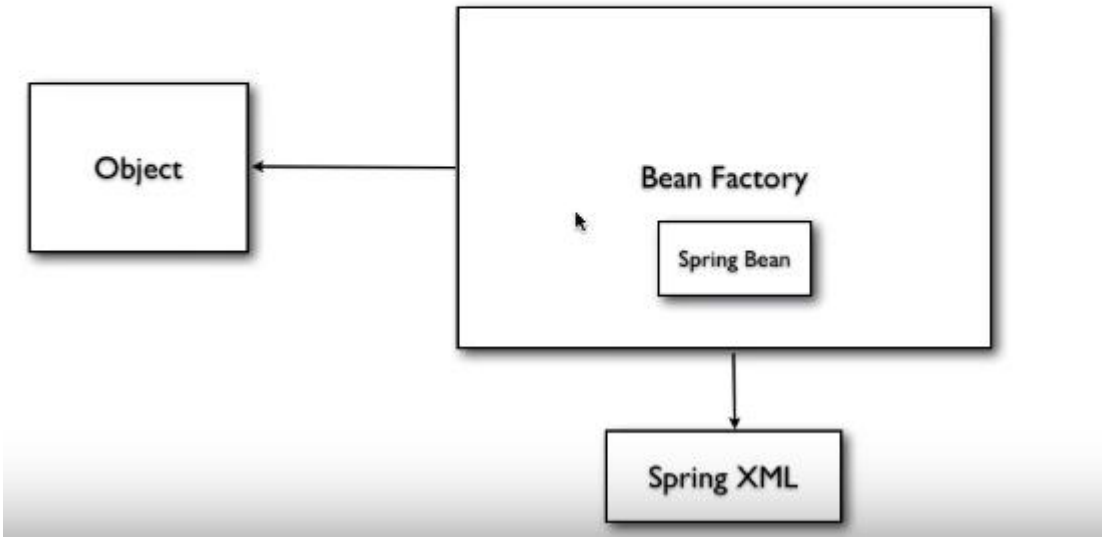
Spring From JavaBrains:



Factory Pattern



Spring Bean Factory



*DrawingApp.java | Triangle.java | *spring.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">


<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle" />

</beans>
```

```java
package org.koushik.javabrains;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class DrawingApp {

    /**
     * @param args
     */
    public static void main(String[] args) {
        //Triangle triangle = new Triangle();
        BeanFactory factory = new XmlBeanFactory(new FileSystemResource("spring.xml"));
        Triangle triangle = (Triangle) factory.getBean("triangle");
        triangle.draw();


    }

}
```

Using Type:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle">
        <constructor-arg type="java.lang.String" value="20" />
    </bean>

</beans>
```

Using Index:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle">
        <constructor-arg index = "0" value="Equilateral" />
        <constructor-arg index = "1" value="20" />
    </bean>

</beans>
```

Ref:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle">
        <property name="pointA" ref="zeroPoint" />
        <property name="pointB" ref="point2" />
        <property name="pointC" ref="point3" />
    </bean>


    <bean id="zeroPoint" class="org.koushik.javabrains.Point">
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>

    <bean id="point2" class="org.koushik.javabrains.Point">
        <property name="x" value="-20" />
        <property name="y" value="0" />
    </bean>

    <bean id="point3" class="org.koushik.javabrains.Point">
        <property name="x" value="20" />
        <property name="y" value="0" />
    </bean>
</beans>
```

InnerBeans:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle">
        <property name="pointA" ref="zeroPoint" />
        <property name="pointB">
            <bean class="org.koushik.javabrains.Point">
                <property name="x" value="-20" />
                <property name="y" value="0" />
            </bean>
        </property>
        <property name="pointC">
            <bean id="point3" class="org.koushik.javabrains.Point">
                <property name="x" value="20" />
                <property name="y" value="0" />
            </bean>
        </property>
    </bean>


    <bean id="zeroPoint" class="org.koushik.javabrains.Point">
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>
```

Alias:

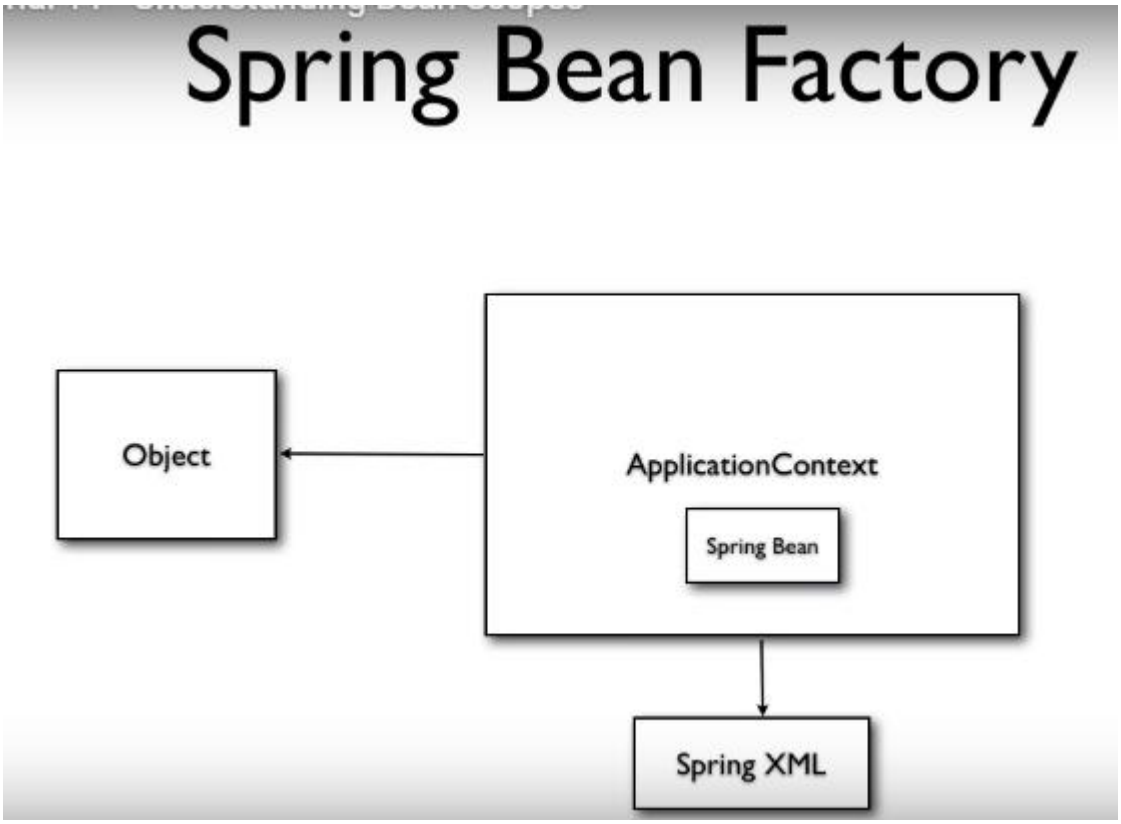
```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle" name="triangle-name">
        <property name="pointA" ref="zeroPoint" />
        <property name="pointB">
            <bean class="org.koushik.javabrains.Point">
                <property name="x" value="-20" />
                <property name="y" value="0" />
            </bean>
        </property>
        <property name="pointC">
            <bean id="point3" class="org.koushik.javabrains.Point">
                <property name="x" value="20" />
                <property name="y" value="0" />
            </bean>
        </property>
    </bean>


    <bean id="zeroPoint" class="org.koushik.javabrains.Point">
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>

    <alias name="triangle" alias="triangle-alias"/>
```

Collections:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle">
        <property name="points">
        <list>
            <ref bean="zeroPoint" />
            <ref bean="point2" />
            <ref bean="point3" />
        </list>

        </property>
    </bean>


    <bean id="zeroPoint" class="org.koushik.javabrains.Point">
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>

    <bean id="point2" class="org.koushik.javabrains.Point">
        <property name="x" value="-20" />
        <property name="y" value="0" />
    </bean>

    <bean id="point3" class="org.koushik.javabrains.Point">
        <property name="x" value="20" />
        <property name="y" value="0" />
```

Autowiring:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle" autowire="byName">

    </bean>


    <bean id="pointA" class="org.koushik.javabrains.Point">
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>

    <bean id="pointB" class="org.koushik.javabrains.Point">
        <property name="x" value="-20" />
        <property name="y" value="0" />
    </bean>

    <bean id="pointC" class="org.koushik.javabrains.Point">
        <property name="x" value="20" />
        <property name="y" value="0" />
    </bean>
</beans>
```



ApplicationContext creates the beans when it loads the spring.xml file provided the scopes are singleton

Basic Bean Scopes

- Singleton - Only once per Spring container.
- Prototype - New bean created with every request or reference.

Web-aware Context Bean Scopes

- Request - New bean per servlet request
- Session - New bean per session.
- Global Session - New bean per global HTTP session (portlet context).

Q. What happens when we wire a prototype bean with Singleton bean?

Ans:
http://www.logicbig.com/tutorials/spring-framework/spring-core/injecting-singleton-with-prototype-bean/

using ScopedProxyMode, to get different instances when wired a prototype with singleton,

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

ApplicationContextAware Interface:

```
import java.util.List;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;


public class Triangle implements ApplicationContextAware {

    private Point pointA;
    private Point pointB;
    private Point pointC;
    private ApplicationContext context = null


    public Point getPointA() {
        return pointA;
    }


  @Override
  public void setApplicationContext(ApplicationContext context)
          throws BeansException {
      this.context = context;

  }
```

BeanNameAware:

To know the bean name configure in spring.xml

BeanDefinition Inheritance:

```xml
<beans>

    <bean id="parenttriangle" class="org.koushik.javabrains.Triangle">
        <property name="pointA" ref="pointA" />
    </bean>

    <bean id="triangle1" class="org.koushik.javabrains.Triangle" parent="parenttriangle">
        <property name="pointB" ref="pointB" />
        <property name="pointC" ref="pointC" />
    </bean>

    <bean id="triangle2" class="org.koushik.javabrains.Triangle" parent="parenttriangle">
        <property name="pointB" ref="pointB" />
    </bean>


    <bean id="pointA" class="org.koushik.javabrains.Point">
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>

    <bean id="pointB" class="org.koushik.javabrains.Point">
        <property name="x" value="-20" />
        <property name="y" value="0" />
    </bean>
```

```xml
<beans>

    <bean id="parenttriangle" class="org.koushik.javabrains.Triangle">
        <property name="points">
        <list>
            <ref bean="pointA" />
        </list>
        </property>
    </bean>

    <bean id="triangle1" class="org.koushik.javabrains.Triangle" parent="parenttriangle">
        <property name="points">
        <list merge="true">
            <ref bean="pointB" />
        </list>
        </property>
    </bean>

    <bean id="triangle2" class="org.koushik.javabrains.Triangle" parent="parenttriangle">
        <property name="pointB" ref="pointB" />
    </bean>
```

Abstract:

```
<beans>

    <bean id="parenttriangle" class="org.koushik.javabrains.Triangle" abstract="true">
        <property name="points">
        <list>
            <ref bean="pointA" />
        </list>
        </property>
    </bean>

    <bean id="triangle1" class="org.koushik.javabrains.Triangle" parent="parenttriangle">
        <property name="points">
        <list merge="true">
            <ref bean="pointB" />
        </list>
        </property>
    </bean>

    <bean id="triangle2" class="org.koushik.javabrains.Triangle" parent="parenttriangle">
        <property name="pointB" ref="pointB" />
    </bean>


    <bean id="pointA" class="org.koushik.javabrains.Point">
```

ShutDown hook (applicable only for desktop app):

```
*/
public static void main(String[] args) {

    AbstractApplicationContext context = new ClassPathXmlApplicationContext("spring.xml");
    context.registerShutdownHook();
    Triangle triangle = (Triangle) context.getBean("triangle");
    triangle.draw();

}
```

Context is destroyed when the main method ends.

InitializingBean:

```java
import java.util.List;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Triangle implements InitializingBean, DisposableBean {

    private Point pointA;
    private Point pointB;
    private Point pointC;
```

```java
@Override
public void afterPropertiesSet() throws Exception {
    System.out.println("InitializingBean init method called for Triangle");

}
```

```java
@Override
public void destroy() throws Exception {
    // TODO Auto-generated method stub

}
```

Another way to use lifecycle methods:

```xml
<beans>
    <bean id="triangle" class="org.koushik.javabrains.Triangle" autowire="byName" init-method="myInit" destroy-method="

    </bean>
```

```xml
<beans default-init-method="myInit" default-destroy-method="cleanUp">
    <bean id="triangle" class="org.koushik.javabrains.Triangle" autowire="byName" >

    </bean>
```

BeanPostProcessor:

```
public class DisplayNameBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
            throws BeansException {
        System.out.println("In After Initialization method. Bean name is " + beanName);
        return bean;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException {

        System.out.println("In Before Initialization method. Bean name is " + beanName);
        return bean;
    }

}
```

```
    <bean class="org.koushik.javabrains.DisplayNameBeanPostProcessor" />
</beans>
```

BeanFactoryPostProcessor:

```
public class MyBeanFactoryPP implements BeanFactoryPostProcessor {

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
            throws BeansException {
        System.out.println("My Bean Factory Post Processor is called");

    }

}
```

```
<bean class="org.koushik.javabrains.MyBeanFactoryPP" />
```

```
Jun 2, 2011 5:50:15 AM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@41ac1fe4: startup date [Thu Jun 02 05:50:15 EDT 2011]; root of context h
Jun 2, 2011 5:50:15 AM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [spring.xml]
My Bean Factory Post Processor is called
Jun 2, 2011 5:50:16 AM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@4229abJe: defining beans [triangle,pointA,pointB,
Point A = (0, 0)
Point B = (-20, 0)
Point C = (20, 0)
```

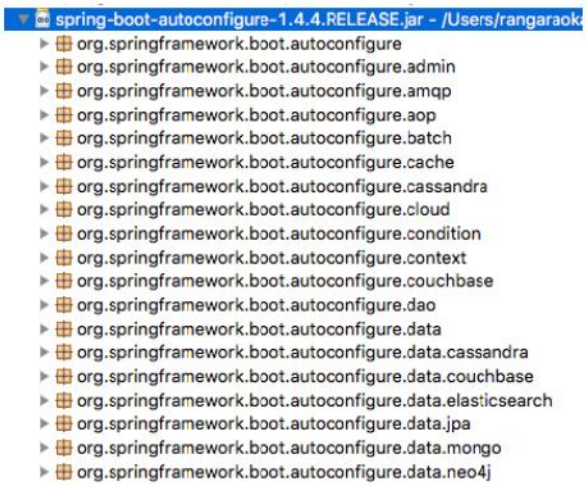
PropertyPlaceholderConfigurer:

```
<bean id="pointA" class="org.koushik.javabrains.Point">
<property name="x" value="${pointA.pointX}" />
<property name="y" value="${pointA.pointY}" />
ean>

<bean class="org.koushik.javabrains.MyBeanFactoryPP" />
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
    <property name="locations" value="pointsconfig.properties" />
</bean>

eans>
```

http://www.springboottutorial.com/spring-boot-auto-configuration

Where is Spring Boot Auto Configuration implemented?

All auto configuration logic is implemented in `spring-boot-autoconfigure.jar`. All auto configuration logic for mvc, data, jms and other frameworks is present in a single jar.



Some More concepts:

@Autowire

1. By default @Autowire is using autowire by type.

2. To make it wire by name change the property name

e.g. Lets consider Engine is an interface.

```
   @Autowire
private Engine engine;    //autowire by type
@Autowire
Private Engine bmwEngine;   //autowire by name. ( if more than one implementer is present)
```

Second Approach to resolve conflict is:

Use

```
@Component
@Primary
Class BMWEngine{
}
```

Note: @Primary has higher propriety than autowire by name.

Third **Approach:**

Use @Qualifier("someName")

Q. Difference between GOF Singleton and Spring Singleton ?

Ans:

1. When we talk about GOF Singleton that is one instance per JVM.

2. When we talk about Spring singleton, it's one instance per Application context.

3. For GOF Singleton even if multiple application contexts are running in the same JVM, you should just have one instance of that specific class.

https://www.journaldev.com/2676/spring-mvc-interceptor-example-handlerinterceptor-handlerinterceptoradapter