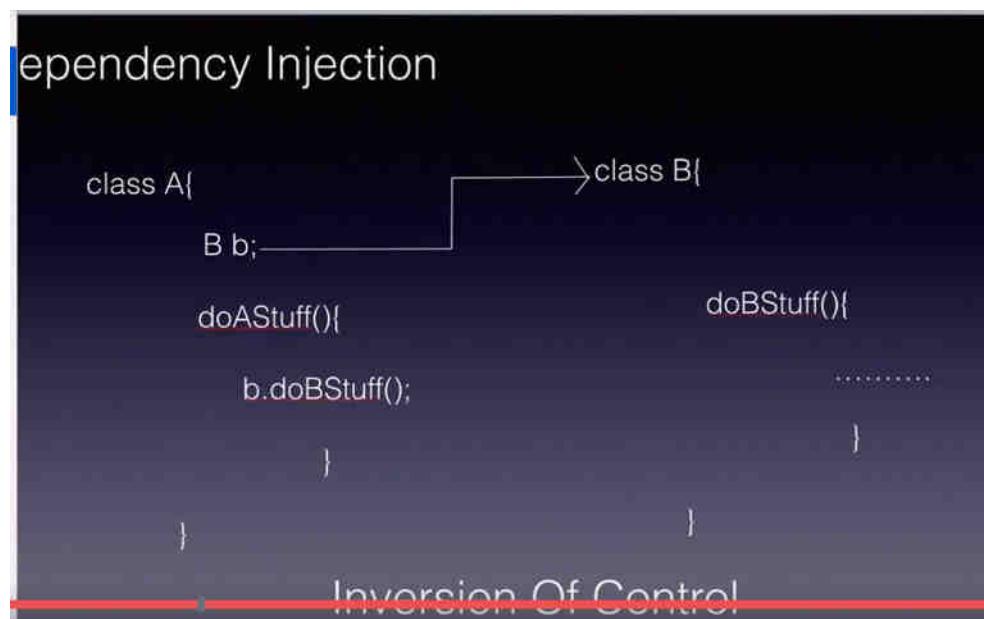


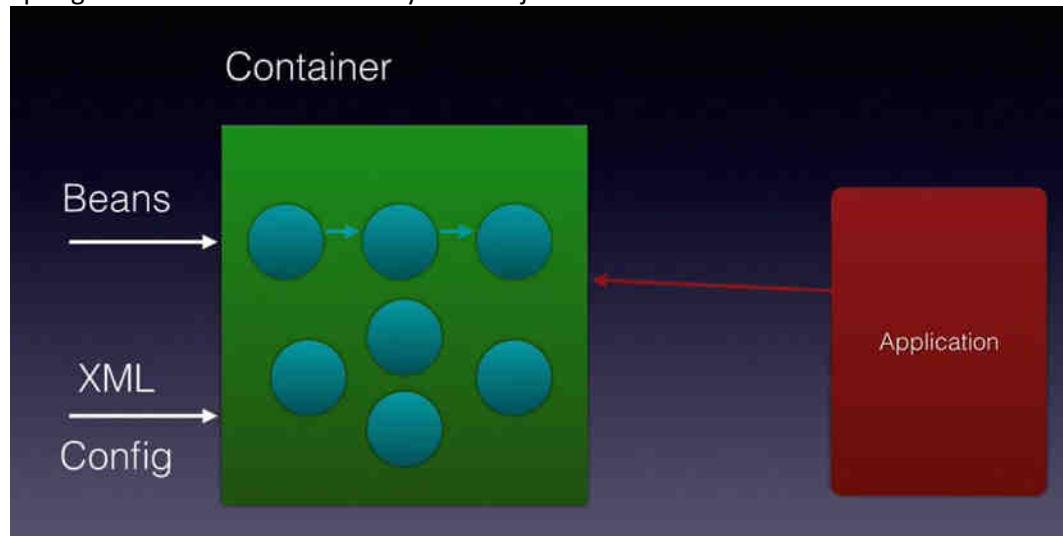
<https://dev.mysql.com/downloads/file/?id=480823>(download mysql with workbench)

1Mysql  
root  
2Mysql workbench :- gui pw-test  
3connect workbench to my sql server

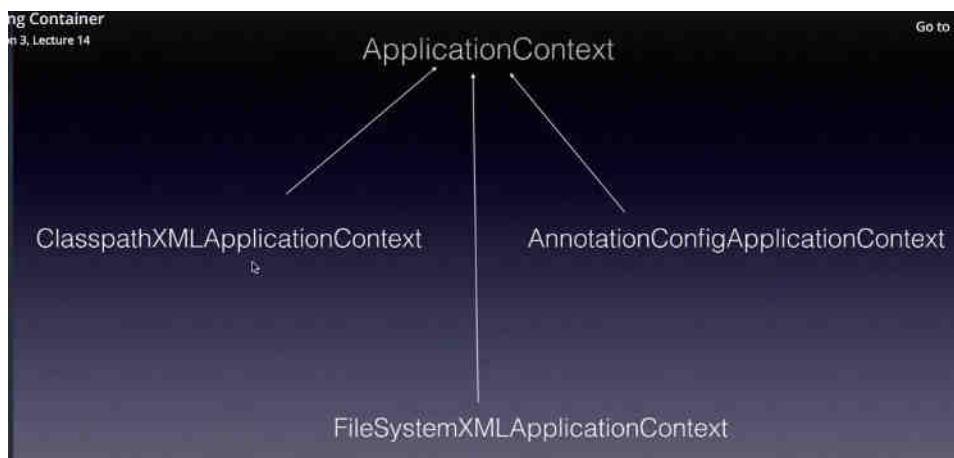
Container will provide b object



Spring container: it maintain lifecycle of object from creation to destruction

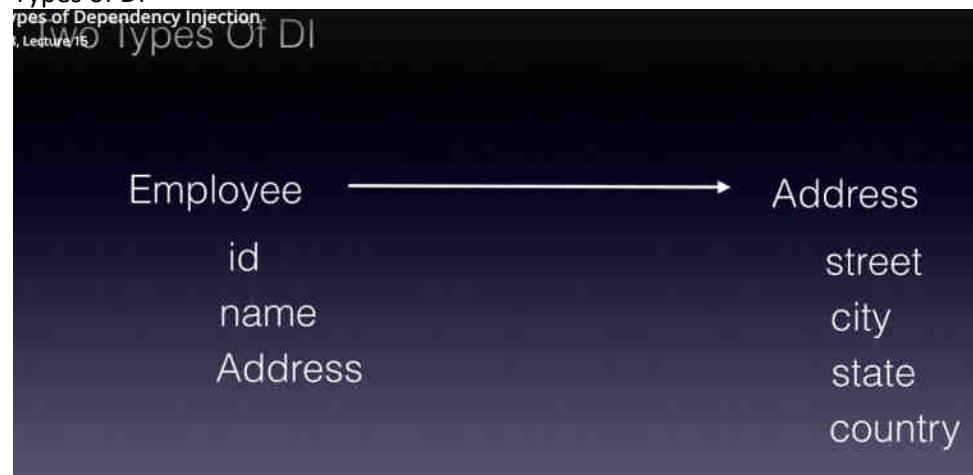


ApplicationContext is

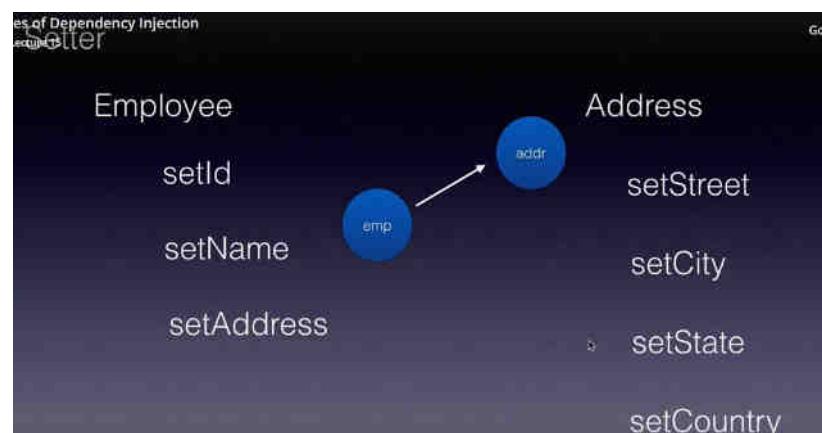


### Types of DI

#### Two Types Of DI



1setter



2constructor



### Spring configuration file

#### Types of data

1)

Types of Data

Primitive Types Dependencies

- byte
- short
- int
- long
- float
- double
- boolean
- char
- string

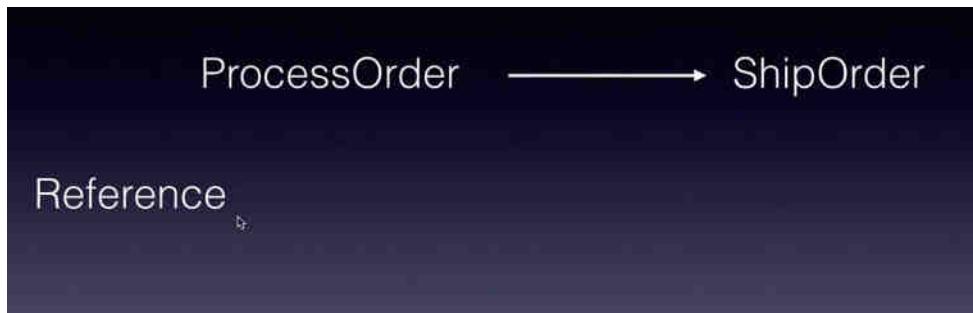
0:33 / 1:30 Transcript Browse Q&A Add Bookmark

This is a screenshot of a presentation slide. The title 'Types of Data' is displayed prominently. Below the title, the text 'Primitive Types Dependencies' is shown. To the right of this text is a vertical list of primitive data types: byte, short, int, long, float, double, boolean, char, and string. At the bottom of the slide, there are navigation controls including 'Transcript', 'Browse Q&A', and 'Add Bookmark', along with a timer indicating the video is at 0:33 of 1:30.

2)



3



Injecting Primitive Types  
Section 3, Lecture 4B

```
<bean class="Product">          value as element
    <property name="id">
        <value>10</value>      value as attribute
    </property>
</bean>
```

p schema/p namespace

=====

===== section 4)

## 3 Steps to DI

Create the POJO

Create the configuration file

Create a test class

Requirement need to inject id and name from container

Note: we define a bean element for each object we want to instantiate

```
public class Employee {  
  
    private int id;  
    private String name;  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Value as element

```
<bean name="emp" class="com.bharath.spring.springcore.Employee">  
    <property name="id">  
        <value>10</value>  
    </property>  
    <property name="name">  
        <value>saurabh</value>  
    </property>  
</bean>
```

Value as attribute

```
<bean name="emp" class="com.bharath.spring.springcore.Employee"
      <property name="id" value="10" />
      <property name="name" value="saurabh" />
  </bean>
```

Value using p schema or p namespace

```
<bean name="emp" class="com.bharath.spring.springcore.Employee" p:id="10" p:name="saurabh">
</bean>
```

Create a test class

```
public class Test {

    public static void main(String arr[]) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
        Employee emp = (Employee) ctx.getBean("emp");
        System.out.println("Employee id : " + emp.getId());
        System.out.println("Employee name : " + emp.getName());
    }
}
```

#### Collection types

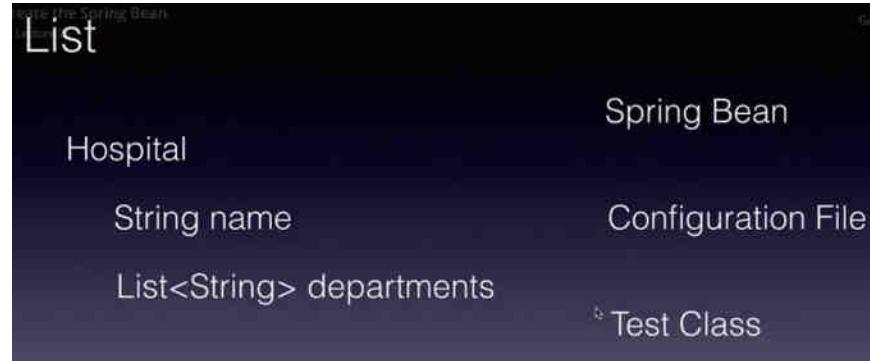
List

Set

Map

Properties

#### 1)List



```
public class Hospital {  
    private String name;  
    private List<String> departments;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public List<String> getDepartments() {  
        return departments;  
    }  
  
    public void setDepartments(List<String> departments) {  
        this.departments = departments;  
    }  
}
```

```
<bean name="hospital" class="com.bharath.spring.springcore.list.Hospital"  
    <property name="name">  
        <value>Global Hospital</value>  
    </property>  
    <property name="departments">  
        <list>  
            <value>Front Office</value>  
            <value>In Patient</value>  
            <value>Out Patient</value>  
        </list>  
    </property>  
</bean>  
</beans>
```

Note: by default arraylist will be created  
Syso(hospital.getDepartments.getClass());

```
public class Test {  
    public static void main(String[] args) {  
        ApplicationContext context = new ClassPathXmlApplicationContext(  
            "com/bharath/spring/springcore/list/listconfig.xml");  
        Hospital hospital = (Hospital) context.getBean("hospital");  
        System.out.println(hospital.getName());  
        System.out.println(hospital.getDepartments());  
    }  
}
```

It will create bean by name of hospital and inject name and list property in setter method

#### Note

If only one element in list then list is optional. No need to write <list>

```
9  
10<bean name="hospital" class="com.bharath.spring.springcore.list.Hospital">  
11    <property name="name">  
12        <value>Global Hospital</value>  
13    </property>  
14    <property name="departments">  
15        <value>Front Office</value> .  
16    </property>  
17</bean>  
18</beans>
```

We can create empty list also by defining  
<list> </list>( without any values)

## 2. Set create , configure and test

#### Note

everything applies to list will be apply to set also

Default implementation of set spring uses is LinkedHashSet

```

public class CarDealer {
    private String name;
    private Set<String> models;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<String> getModels() {
        return models;
    }

    public void setModels(Set<String> models) {
        this.models = models;
    }
}

<bean id="carDealer"
      class="com.bharath.spring.springcore.set.CarDealer">
    <property name="name">
        <value>Bangalore dealer</value>
    </property>

    <property name="models">
        <set>
            <value>bmw</value>
            <value>audi</value>
            <value>ferari C</value>
        </set>
    </property>

</bean>

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("com/bharath/spring/springcore/set/
        CarDealer carDealer = (CarDealer) context.getBean("carDealer");
        System.out.println(carDealer.getName());
        System.out.println(carDealer.getModels());
    }
}

```

### 3) Map

Map

Customer

```
int id;  
Map<Integer, String> products;
```

```
4  
5 public class Customer {  
6 |  
7     private int id;  
8  
9     private Map<Integer, String> products;  
0  
1     public int getId() {  
2         return id;  
3     }  
4  
5     public void setId(int id) {  
6         this.id = id;  
7     }  
8  
9     public Map<Integer, String> getProducts() {  
0         return products;  
1     }  
2  
3     public void setProducts(Map<Integer, String> products) {  
4         this.products = products;  
5     }  
6  
7     @Override  
8     public String toString() {  
9         return "Customer [id=" + id + ", products=" + products + "]";  
0     }  
1  
2 }
```

We can define key and value in 4 different ways

```
<bean name="customer" class="com.bharath.spring.springcore.map.Customer"
    p:id="20">
    <property name="products">
        <map>
            <entry key="100" value="IPhone" />
            <entry key="200">
                <value>IPad</value>
            </entry>
            <entry value="Macbook Pro">
                <key>
                    <value>300</value>
                </key>
            </entry>
            <entry>
                <key>
                    <value>400</value>
                </key>
                <value>Macbook AIR</value>
            </entry>
        </map>
    </property>
</bean>
```

#### 4)Properties

```
Properties:
    Languages
        Properties countryAndLangs;
```

```

public class CountriesAndLanguages {

    private Properties countryAndLang;

    public Properties getCountryAndLang() {
        return countryAndLang;
    }

    public void setCountryAndLang(Properties countryAndLang) {
        this.countryAndLang = countryAndLang;
    }

    @Override
    public String toString() {
        return "CountriesAndLanguages [countryAndLang=" + countryAndLang + "]";
    }
}

<bean id="countryAndLang"
      class="com.bharath.spring.springcore.properties.CountriesAndLanguages">

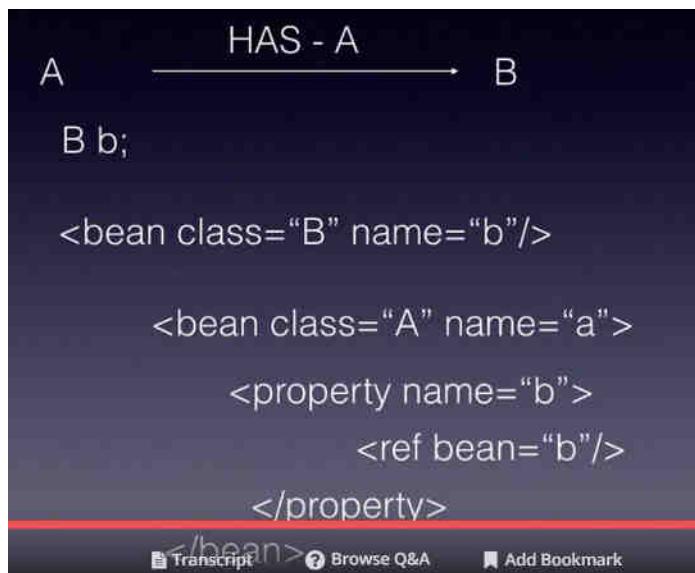
    <property name="countryAndLang">
        <props>
            <prop key="India">Hindi</prop>
            <prop key="USA">English</prop>
        </props>
    </property>
</bean>

public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext(
        "com/bharath/spring/springcore/properties/config.xml");

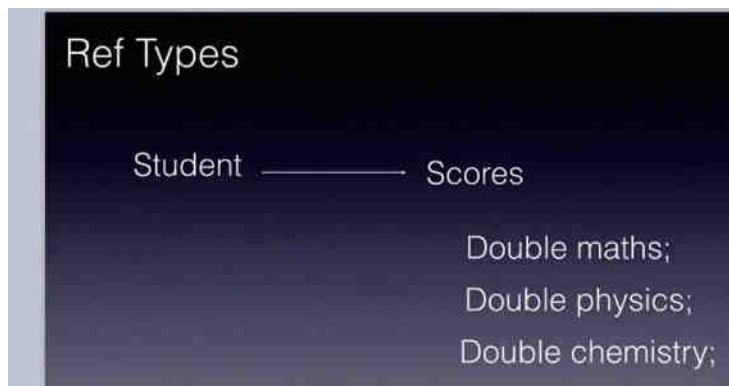
    CountriesAndLanguages countryAndLang = (CountriesAndLanguages) context.getBean("countryAndLang");
    System.out.println(countryAndLang);
}

```

## References Types



### Reference type usecase



```

public class Student {
    private Scores score;

    public Scores getScore() {
        return score;
    }

    public void setScore(Scores score) {
        this.score = score;
    }
}

```

```
public class Scores {  
  
    private Double physics;  
    private Double chemistry;  
    private Double maths;  
  
    public Double getMaths() {  
        return maths;  
    }  
  
    public void setMaths(Double maths) {  
        this.maths = maths;  
    }  
  
    public Double getPhysics() {  
        return physics;  
    }  
  
    public void setPhysics(Double physics) {  
        this.physics = physics;  
    }  
}
```

### Reference types configuration and test

Note: reference type always start with dependent bean

```
<bean name="scores" class="com.bharath.spring.springcore.reftypes.Scores"  
      p:maths="80" p:physics="90" p:chemistry="78" />  
  
<bean name="student" class="com.bharath.spring.springcore.reftypes.Student">  
    <property name="scores">  
        <ref bean="scores" />  
    </property>  
  
    </bean>  
  
</beans>
```

### Ref as attribute

```
<bean name="scores" class="com.bharath.spring.springcore.reftypes.Scores"  
      p:maths="80" p:physics="90" p:chemistry="78" />  
  
<bean name="student" class="com.bharath.spring.springcore.reftypes.Student">  
    <property name="scores" ref="scores"/>  
    </bean>  
  
</beans>
```

## As p schema

```
<bean name="scores" class="com.bharath.spring.springcore.reftypes.Scores"
      p:maths="80" p:physics="90" p:chemistry="78" />

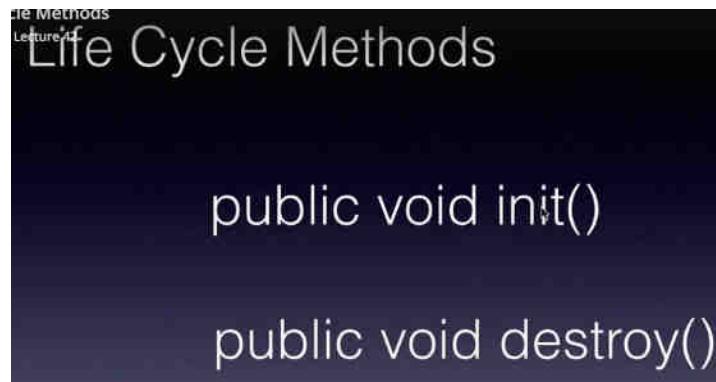
<bean name="student" class="com.bharath.spring.springcore.reftypes.Student" p:scores-ref="scores"/>

public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("com/bharath/spring/springcore/reftypes/config.xml");
        Student stud = (Student) context.getBean("student");
        System.out.println(stud);
    }
}
```

## Section 5 ( life cycle method)

Init() can be used for connecting to database , webservice configurstion

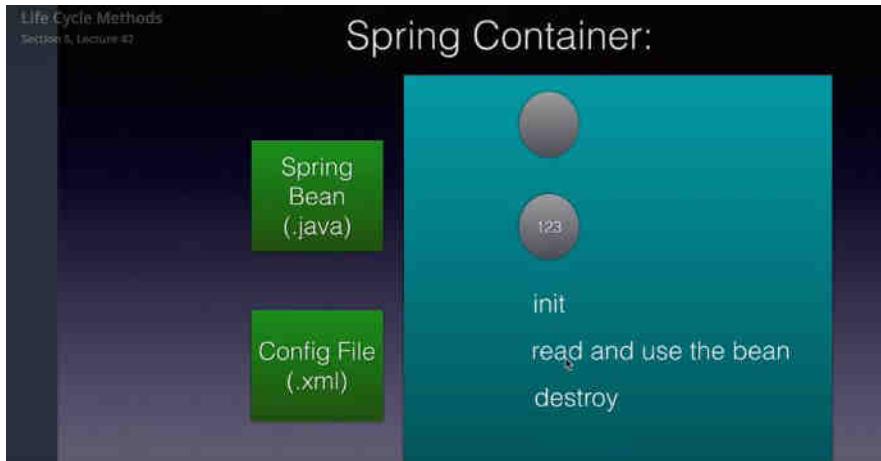
Destroy() can be use for cleanup of code



Note:

**First object is initialized then value will set by setter method then init method is called**

**Init method is called just after object creation and destroy method is called just before object clean up**



We can configure bean life cycle method in three ways

- Life Cycle Methods  
Section 5, Lecture 42
- 1) XML Configuration
  - 2) Spring Interfaces
  - 3) Annotations

#### 1) Life cycle method using xml configuration

Name of init method and destroy method can be anything but we need to configure

```

3 public class Patient {
4
5     private int id;
6
7     public int getId() {
8         return id;
9     }
10
11     public void setId(int id) {
12         System.out.println("inside setter method");
13         this.id = id;
14     }
15
16     public void hi() {
17         System.out.println("inside hi method");
18     }
19
20     public void bye() {
21         System.out.println("inside destroy method");
22     }
23
24     @Override
25     public String toString() {
26         return "Patient [id=" + id + "]";
27     }
28 }
29

```

```
<bean name="patient" class="com.bharath.spring.springcore.lc.xmlconfig.Patient" p:id="10" init-method="hi" destroy-method="bye"/>
```

```

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcore/lc/xmlconfig/config.xml");

        Patient patient = (Patient) context.getBean("patient");
        System.out.println(patient);
    }
}

```

```
//output
INFO: Loading XML bean de
inside setter method
inside hi method
Patient [id=10]
```

**Note:** but destroy method is not called in above example to call destroy method we need to enable hook in spring

### Configure pre shutdown hook

registerShutdownHook present in AbstractApplicationContext which tell spring container that We need to invoke destroy method before container destroyed

```
6 public class Test {  
7  
8     public static void main(String[] args) {  
9         AbstractApplicationContext context = new ClassPathXmlApplicationContext(  
10             "com/bharath/spring/springcore/lc/xmlconfig/config.xml");  
11         Patient patient = (Patient) context.getBean("patient");  
12         System.out.println(patient);  
13         context.registerShutdownHook();  
14     }  
15 }
```

### 2)Life cycle method using spring interface

**Note:** registerShutdownHook is required

```

public class Patient implements InitializingBean, DisposableBean {
    private int id;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        System.out.println("inside setter method");
        this.id = id;
    }
    public void hi() {
        System.out.println("inside hi method");
    }
    public void bye() {
        System.out.println("inside destroy method");
    }
    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("inside after property set");
    }
    @Override
    public void destroy() throws Exception {
        System.out.println("inside destroy method");
    }
}

```

```

<bean name="patient" class="com.bharath.spring.springcore.Lc.interfaces.Patient" p:id="10"/>

6 public class Test {
7
8    public static void main(String[] args) {
9        AbstractApplicationContext context = new ClassPathXmlApplicationContext(
10            "com/bharath/spring/springcore/lc/interfaces/config.xml");
11
12        Patient patient = (Patient) context.getBean("patient");
13        System.out.println(patient);
14        context.registerShutdownHook();
15    }
16 }
17

```

### 3. Life cycle method using annotation

Init method marked as @PostConstruct

Destroy method marked as @PreDestroy

**Note :** by default spring support all the annotation is disabled. We need to enable them

```
1 public class Patient {  
2     private int id;  
3  
4     public int getId() {  
5         return id;  
6     }  
7  
8     public void setId(int id) {  
9         System.out.println("inside setter method");  
10        this.id = id;  
11    }  
12  
13    @PostConstruct  
14    public void hi() {  
15        System.out.println("inside hi method");  
16    }  
17  
18    @PreDestroy  
19    public void bye() {  
20        System.out.println("inside destroy method");  
21    }  
22  
23    @Override  
24    public String toString() {  
25        return "Patient [id=" + id + "]";  
26    }  
27}
```

**Point 1)** want to support only above annotation then need to include

```
<bean name="patient" class="com.bharath.spring.springcore.Lc.annotations.Patient" p:id="10"/>  
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor"/>  
</beans>
```

```
5 public class Test {  
6  
7     public static void main(String[] args) {  
8         AbstractApplicationContext context = new ClassPathXmlApplicationContext(  
9             "com/bharath/spring/springcore/lc/annotations/config.xml");  
10  
11         Patient patient = (Patient) context.getBean("patient");  
12         System.out.println(patient);  
13         context.registerShutdownHook();  
14     }  
15 }
```

configuring support for all annotation

<context:annotation-config/> is used to support all annotation

```
<bean class="com.bharath.spring.springcore.lc.annotations.Patient"  
      name="patient" p:id="123" />  
  
<context:annotation-config />
```

The screenshot shows a presentation slide with a dark background. At the top, there is a navigation bar with links like 'Cycle Methods Summary', '1.5. Life Cycle Methods', 'Transcript', 'Browse Q&A', 'Add Bookmark', and 'Comments'. The main content area has a title 'Life Cycle Methods' and a subtitle 'XML Configuration'. Below this, there are two columns of methods:

|         |                               |
|---------|-------------------------------|
| init    | init-method<br>destroy-method |
| destroy | @PostConstruct<br>@PreDestroy |

Below these columns is a section titled 'Annotations'.

At the bottom of the slide, there is a red horizontal bar containing the XML code: <context:annotation-config />.

## Section -6

### Dependency check - bean and test creation

Dependency Check - Bean and Test Creation

### Dependency Check

The screenshot shows a presentation slide with a dark background. At the top, there is a navigation bar with links like 'Transcript', 'Browse Q&A', 'Add Bookmark', and 'Comments'. The main content area has a title 'Dependency Check' and a subtitle 'Prescription'.

```
int id;  
  
String patientName;  
  
List<String> medicines;
```

#### Note

- 1) we can use @Required only at setter method
2. Spring accepted default value if we didn't provide any value in spring.xml
3. If we use Required annotation then we need to give value in spring.xml for that attribute otherwise beanInitializationException we will get

```

public class Prescription {

    private int id;
    private String patientName;
    private List<String> medicines;

    public int getId() {
        return id;
    }

    @Override
    public String toString() {
        return "Prescription [id=" + id + ", patientName=" + patientName + ", medicines=" + medicines + "]";
    }

    @Required
    public void setId(int id) {
        this.id = id;
    }

    public String getPatientName() {
        return patientName;
    }

    public void setPatientName(String patientName) {
        this.patientName = patientName;
    }

    public List<String> getMedicines() {
        return medicines;
    }

    public void setMedicines(List<String> medicines) {
    }
}

<context:annotation-config/>
<bean id="prescription" class="com.bharath.spring.springcore.dependencycheck.Prescription" p:id="10"/>

```

/beans

```

public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext(
        "com/bharath/spring/springcore/dependencycheck/config.xml");

    Prescription prescription = (Prescription) context.getBean("prescription");
    System.out.println(prescription);
}
}

```

Output//

```
Prescription [id=10, patientName=null, medicines=null]
```

Inner bean - create the bean and config ( nested bean and bean as property)

We will configure address as inner bean for employee

Inner Beans - Create the bean and config  
Section 6, Lecture 52

|                  |                |
|------------------|----------------|
| Employee         | Address        |
| int id;          | int hno;       |
| Address address; | String street; |
|                  | String city;   |

```
4
3 public class Employee {
4
5     private int id;
6     private Address address;
7     @Override
8     public String toString() {
9         return "Employee [id=" + id + ", address=" + address + "]";
10    }
11    public int getId() {
12        return id;
13    }
14    public void setId(int id) {
15        this.id = id;
16    }
17    public Address getAddress() {
18        return address;
19    }
20    public void setAddress(Address address) {
21        this.address = address;
22    }
23 }
24 }
```

```

public class Address {

    private int hno;
    private String street;
    private String city;
    @Override
    public String toString() {
        return "Address [hno=" + hno + ", street=" + street + ", city=" + city + "]";
    }

    public int getHno() {
        return hno;
    }

    public void setHno(int hno) {
        this.hno = hno;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }
}

<beans>
    <bean class="com.bharath.spring.springcore.innerbeans.Employee"
          name="employee" p:id="123">
        <property name="address">
            <bean class="com.bharath.spring.springcore.innerbeans.Address"
                  p:hno="700" p:street="Mira Mesa Blvd" p:city="San Diego"/>
        </property>
    </bean>
</beans>

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "com/bharath/spring/springcore/innerbeans/config.xml");

        Employee employee = (Employee) context.getBean("employee");
        System.out.println(employee);
    }
}

```

In above scenario Employee has a relation with address so we are using inner beans

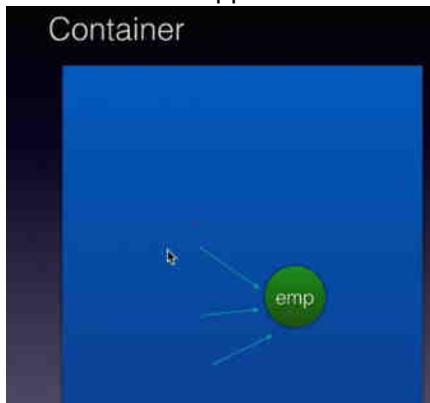
### Bean Scopes

"it is number of object created in a container of particular bean type"



1) By default scope is **singleton** and container will create only one object no matter how many time bean is read

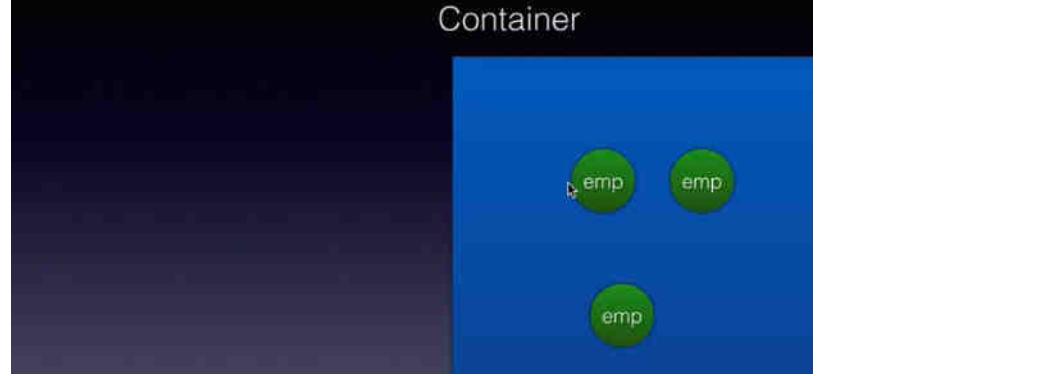
And used in our application



## 2. Prototype

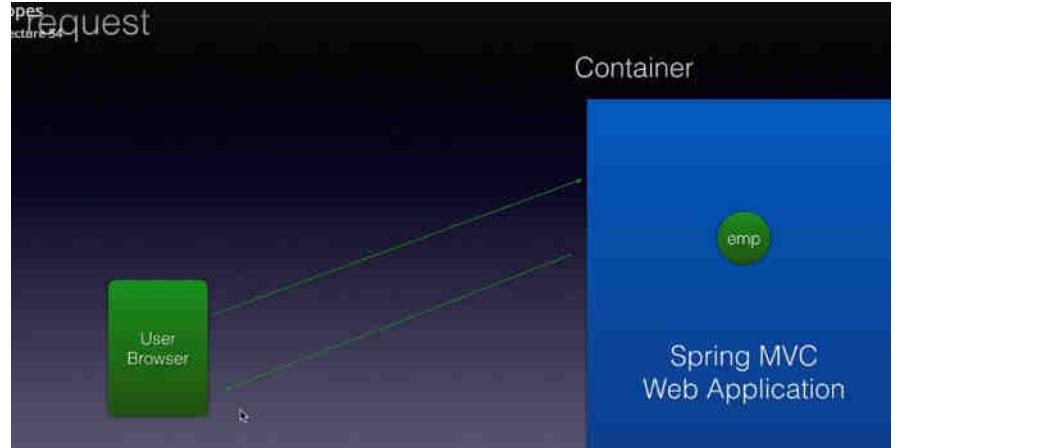
If we marked bean as prototype then every time container will create new object as it read or used in application

## prototype



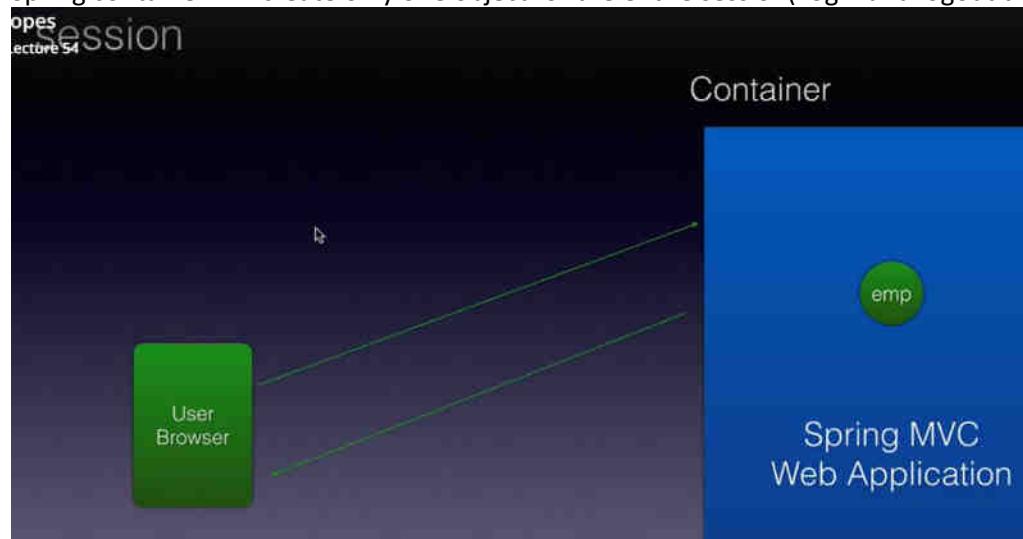
## 3) Request

Spring will create object per user request



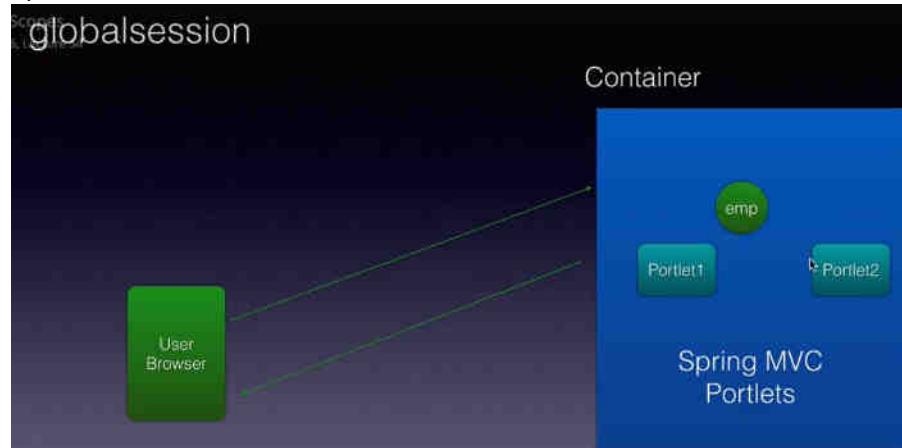
## 4. Session

Spring container will create only one object for the entire session( login and logout time)



When new user object is created then another object will be instantiated

### 5) Global session



### Scopes in action

We can check by printing hashCode

```
<bean class="com.bharath.spring.springcore.innerbeans.Employee"
      name="employee" p:id="123" scope="prototype">
    <property name="address">
      <bean class="com.bharath.spring.springcore.innerbeans.Address"
            p:hno="700" p:street="Mira Mesa Blvd" p:city="San Diego"/>
    </property>
</bean>
```

```
ApplicationContext context = new ClassPathXmlApplicationContext(
    "com/bharath/spring/springcore/innerbeans/config.xml");
Employee employee = (Employee) context.getBean("employee");
System.out.println(employee.hashCode());

Employee employee2 = (Employee) context.getBean("employee");
System.out.println(employee2.hashCode());
```

Different hashCode we will get

| Dependency Check, Inner Beans and Scope Summary |  | Scopes        |
|---|--|---------------|
| Section 6, Lecture 56                           |  |               |
| @Required                                       |  | singleton     |
| BeanInitializationException                     |  | prototype     |
|   |  | request       |
|   |  | session       |
| Inner Beans                                     |  | globalSession |
| <bean>  |  |               |
| <property>                                      |  |               |
| <bean>  |  |               |

## section 7

### Constructor Injection

to inject an object using constructor

| Create the Bean and Configuration |                      | Go to ... |
|-----------------------------------|----------------------|-----------|
| Section 7, Lecture 57             |                      |           |
| Constructor Injection             |                      |           |
| <constructor-arg>                 | As Tag               |           |
| <value> b </value>                |                      |           |
| <constructor-arg>                 | As Attribute         |           |
| <constructor-arg>                 | C Schema/C Namespace |           |
| <ref bean="a"/>                   |                      |           |
| </constructor-arg>                |                      |           |

```
1 public class Employee {  
2     private int id;  
3     public Employee(int id, Address address) {  
4         this.id = id;  
5         this.address = address;  
6     }  
7     private Address address;  
8     @Override  
9     public String toString() {  
10         return "Employee [id=" + id + ", address=" + address + "]";  
11     }  
12 }  
  
1 public class Address {  
2     private int hno;  
3     private String street;  
4     private String city;  
5     @Override  
6     public String toString() {  
7         return "Address [hno=" + hno + ", street=" + street + ", city=" + city + "]";  
8     }  
9     public int getHno() {  
10         return hno;  
11     }  
12     public void setHno(int hno) {  
13         this.hno = hno;  
14     }  
15     public String getStreet() {  
16         return street;  
17     }  
18     public void setStreet(String street) {  
19         this.street = street;  
20     }  
21 }
```

```

<bean name="address"
      class="com.bharath.spring.springcore.constructorinjection.Address"
      p:hno="26" p:street="hosur" p:city="bangalore" />

<bean name="employee"
      class="com.bharath.spring.springcore.constructorinjection.Employee">
    <constructor-arg>
      <value>12</value>
    </constructor-arg>

    <constructor-arg>
      <ref bean="address" />
    </constructor-arg>

  </bean>
</beans>

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcore/constructorinjection/config.xml");

        Employee employee = (Employee) context.getBean("employee");
        System.out.println(employee);

    }
}

```

#### As Element , Attribute and C schema

```

<bean class="com.bharath.spring.springcore.constructorinjection.Address"
      name="address" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcore.constructorinjection.Employee"
      name="employee">
    <constructor-arg value="123"/>
    <constructor-arg ref="address"/>
  </bean>

</beans>

```

#### C : schema

```
<bean class="com.bharath.spring.springcore.constructorinjection.Address"
      name="address" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcore.constructorinjection.Employee"
      name="employee" c:id="123" c:address-ref="address"/>
| 
</beans>
```

### Ambiguity problem

```
public class Addition {

    Addition(int a, int b) {
        System.out.println("inside constructor Int");
    }

    Addition(double a, double b) {
        System.out.println("inside constructor Double");
    }

    Addition(String a, String b) {
        System.out.println("inside constructor String");
    }
}
```

```
<bean name="addition"
      class="com.bharath.spring.springcore.constructorinjection.ambiguity.Addition">

    <constructor-arg value="10" />
    <constructor-arg value="20" />

</bean>
/beans|
```

```

6 public class Test {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new ClassPathXmlApplicationContext(
0             "com/bharath/spring/springcore/constructorinjection/ambiguity/config.xml");
1
2         Addition addition = (Addition) context.getBean("addition");
3         System.out.println(addition);
4
5     }
6 }
7

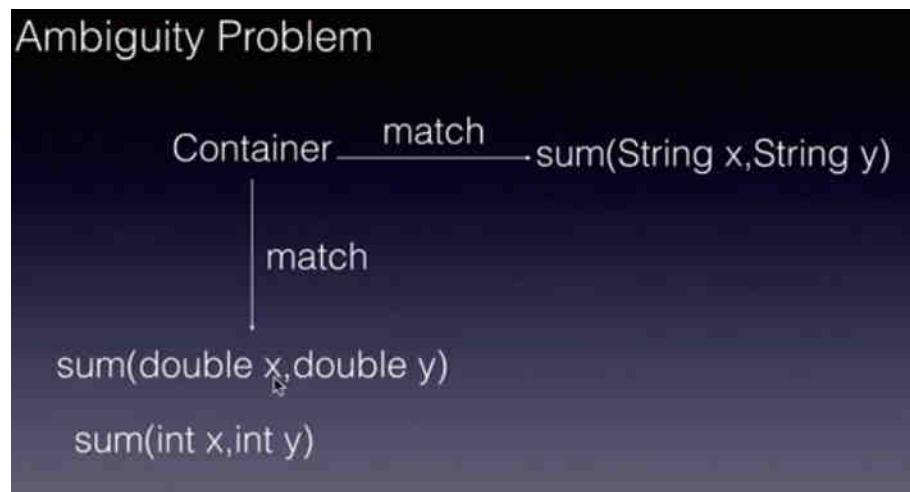
//output
inside constructor String
com.bharath.spring.springcore.constructorinjection.ambiguity.Addition@6767c1fc

```

**Problem :** spring printing in random way

**Solution:** Using the type attribute

Spring first find string method if it is not there then in which order constructor is given it will consider that



Ambiguity problem can be fixed by

```
<constructor-arg>  
    type  
    index  
    name
```

Type(int, double) index(int, double) name(name of variable)

```
<http://www.springframework.org/schema/context/spring-context.xsd>  
  
<beans>  
    <bean  
        class="com.bharath.spring.springcore.constructorinjection.ambiguity.Addition"  
        name="addition">  
        <constructor-arg value="10" type="int"/>  
        <constructor-arg value="20" type="int"/>  
    </bean>  
  
</beans>
```

#### Ambiguity problem variation

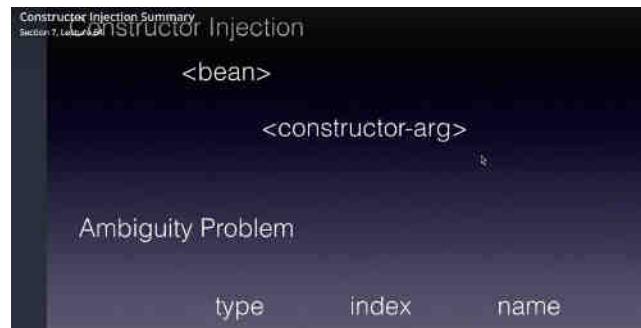
Note : if we want order in constructor then we have to use index in config.xml

```
<http://www.springframework.org/schema/context/spring-context.xsd>  
  
<beans>  
    <bean  
        class="com.bharath.spring.springcore.constructorinjection.ambiguity.Addition"  
        name="addition">  
        <constructor-arg value="20.3" type="double" index="1"/>  
        <constructor-arg value="10" type="int" index="0"/>  
    </bean>  
  
</beans>
```

```

3 public class Addition {
4
5     Addition(int a,double b){
6         System.out.println("Inside the Constructor");
7         System.out.println(a);
8         System.out.println(b);
9     }
10 }
11

```



## Section 8

### Bean Externalization or reading Properties

If we change anything it can be reuse without compiling code

| Externalization Introduction<br>8, Lecture 2 |                                     | Go |
|--|-------------------------------------|----|
| Bean Externalization or Reading Properties   |                                     |    |
| database.properties                          | property place holder configuration |    |
| dbName                                       | Create the properties               |    |
| port   |                                     |    |
| userName                                     | Link the properties                 |    |
| password                                     | User properties in xml and inject   |    |

### Database.properties

```
1 dbServer=saurabhserver  
2 #default port  
3 dbPort=3306  
4 dbUser=test  
5 dbPass=test
```

Note: we will create pojo class and linked these properties with pojo  
Configure and test

Note: in constructor argument value we want to read from properties file

```
1 public class MyDao {  
2     private String dbServer;  
3     MyDao(String dbServer) {  
4         this.dbServer = dbServer;  
5     }  
6     @Override  
7     public String toString() {  
8         return "MyDao [dbServer=" + dbServer + "]";  
9     }  
10 }  
  
<context:property-placeholder location="com/bharath/spring/springcore/propertyplaceholder/database.prop"  
11 <bean id="mydao" class="com.bharath.spring.springcore.propertyplaceholder.MyDao">  
12 <constructor-arg>  
13 <value>${dbServer}</value>  
14 </constructor-arg>  
15 </bean>  
</beans>
```

```

5 public class Test {
6
7     public static void main(String[] args) {
8
9         ApplicationContext context = new ClassPathXmlApplicationContext(
10             "com/bharath/spring/springcore/propertyplaceholder/config.xml");
11
12         MyDao mydao = (MyDao) context.getBean("mydao");
13         System.out.println(mydao);
14
15     }
16
17 }

```

### BeanDefinitionStoreException

If the property does not defined in properties file and trying to access from config.xml then what will happen ?

Then it will give below exception:-

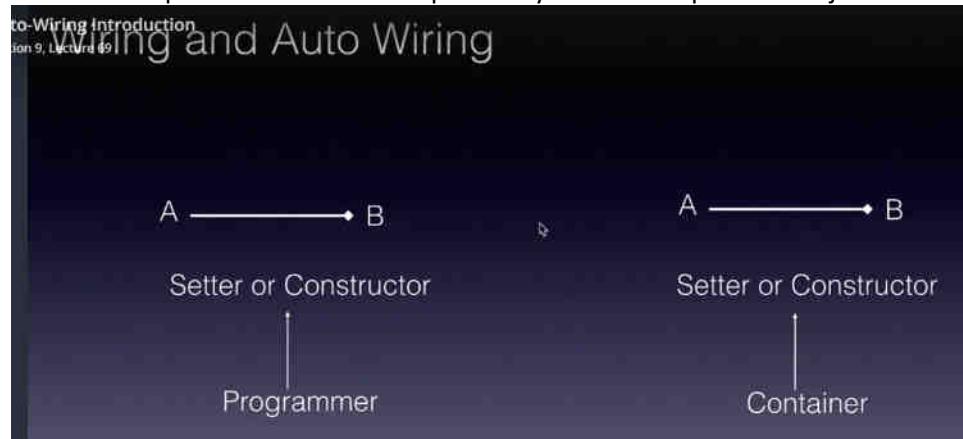
**WARNING: Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.BeanDefinitionStoreException: Invalid bean definition [**

## Section 9

### Wiring and Autowiring introduction ( wiring of beans together)

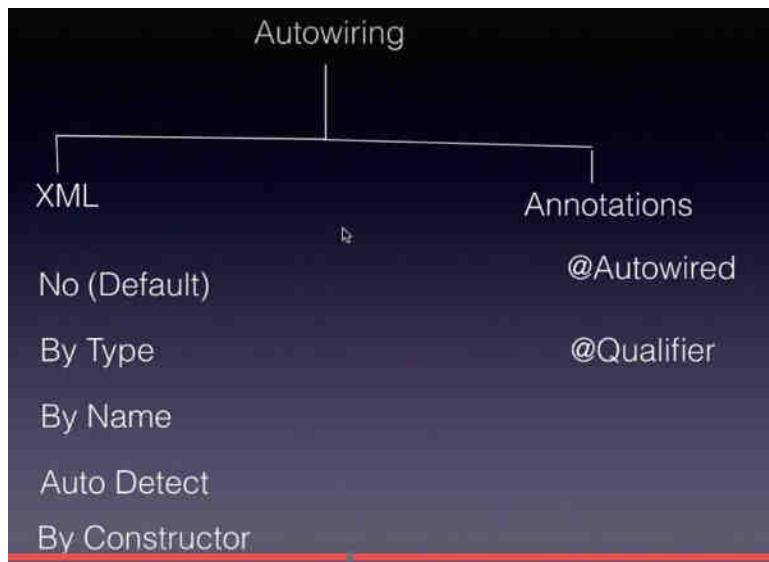
a----->b

When a is depend on b then b is dependency then a is dependent object



If it is done by programmer then it is called wiring and by container then it is called autowiring

**Note:** By Type and by name use setter injection



### Autowiring by type

Spring container will look all the dependencies in pojo class take the types and search in config.xml

```
2
3 public class Employee {
4
5     private Address address;
6
7     @Override
8     public String toString() {
9         return "Employee [address=" + address + "
10    }
11
12    public Address getAddress() {
13        return address;
14    }
15
16    public void setAddress(Address address) {
17        this.address = address;
18    }
19
20 }
```

```

3 public class Address {
4
5     private int hno;
6     private String street;
7     private String city;
8     @Override
9     public String toString() {
0         return "Address [hno=" + hno + ", street=" + street + ", city=" + city + "]";
1     }
2
3
4     public int getHno() {
5         return hno;
6     }
7
8     public void setHno(int hno) {
9         this.hno = hno;
0     }
1
2     public String getStreet() {
3
<bean name="address"
      class="com.bharath.spring.springcoreadvanced.autowiring.Address"
      p:hno="26" p:street="hosur" p:city="bangalore" />
<bean name="employee"
      class="com.bharath.spring.springcoreadvanced.autowiring.Employee" autowire="byType" >
</bean>
</beans>
```

```

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcoreadvanced/autowiring/config.xml");

        Employee employee = (Employee) context.getBean("employee");
        System.out.println(employee);

    }
}

//output
INFO: Loading XML bean definitions from class path resource [com/bharath/spring/springcoreadvanced/autowiring/config.xml]
Employee [address=Address [hno=26, street=hosur, city=bangalore]]
```

### Three things about auto-wiring by type

**Point 1:-** If no bean match is found no exception we will get



```
Address.java Employee.java Test.java config.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"
4   xmlns:p="http://www.springframework.org/schema/p"
5   xmlns:c="http://www.springframework.org/schema/c"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7   http://www.springframework.org/schema/beans/spring-beans.xsd
8   http://www.springframework.org/schema/context
9   http://www.springframework.org/schema/context/spring-context.xsd">
10
11
12
13   <bean class="com.bharath.spring.springcoreadvanced.autowiring.Employee"
14     name="employee" autowire="byType"/>
15
16 </beans>
```

terminated> Test [14] [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_121.jdk/Contents/Home/bin/java -jar /Library/Java/JavaVirtualMachines/jdk1.8.0\_121.jdk/Contents/Home/jre/lib/charsets.jar -Dfile.encoding=UTF-8 com.bharath.spring.springcoreadvanced.autowiring.Test  
Mar 06, 2017 2:12:33 PM org.springframework.context.support.ClassPathXmlApplicationContext prepareRefresh  
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@5a1f33: startup date [Mar 06, 2017 2:12:33 PM]; root of context hierarchy  
Mar 06, 2017 2:12:33 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh  
INFO: Loading XML bean definitions from class path resource [config.xml]  
Employee [address=null]

**Point 2:** if it is duplicate it will throw a exception so always use matching one only

```
<bean class="com.bharath.spring.springcoreadvanced.autowiring.Address"
      name="address" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Address"
      name="address1" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Employee"
      name="employee" autowire="byType"/>

</beans>
```

### autowiring By Name ( consider above example)

By name: will work based on reference variable

#### Point 1

Spring container will inject null if it cant find bean

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.
4   xmlns:p="http://www.springframework.org/schema/p"
5   xmlns:c="http://www.springframework.org/schema/c"
6   xsi:schemaLocation="http://www.springframework.org/schema/beans
7   http://www.springframework.org/schema/beans/spring-beans.xsd
8   http://www.springframework.org/schema/context
9   http://www.springframework.org/schema/context/spring-context.xsd">
10
11
12
13
14   <bean class="com.bharath.spring.springcoreadvanced.autowiring.Employee"
15     name="employee" autowire="byName"/>
16
17 </beans>

```

<terminated> Test (14) [Java Application] /Lib  
 Mar 06, 2017 2:57:14 PM org.springframework.context.support.ClassPathXmlApplicationContext info  
 INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@713333  
 Mar 06, 2017 2:57:14 PM org.springframework.context.support.ClassPathXmlApplicationContext info  
 INFO: Loading XML bean definition  
 Employee [address=null]

Note : byName we can have same bean defined multiple times

```

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Address"
      name="address1" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Address"
      name="address" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Employee"
      name="employee" autowire="byName"/>

/beans>

```

autowiring by constructor (consider above example)

```

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Address"
      name="address" p:hno="123" p:street="mira mesa" p:city="san diego" />

<bean class="com.bharath.spring.springcoreadvanced.autowiring.Employee"
      name="employee" autowire="constructor"/>

/beans>

```

```
3 public class Employee {  
4     } Employee(Address address) {  
5         this.address = address;  
6     }  
7  
8     @Override  
9     public String toString() {  
10         return "Employee [address=" + address + "]";  
11     }  
12  
13     public Address getAddress() {  
14         return address;  
15     }  
16  
17     public void setAddress(Address address) {  
18         this.address = address;  
19     }  
20  
21     private Address address;  
22  
23 }  
24  
25
```

### Using the @Autowired annotation

#### 1)at setter level

```
4  
5 public class Employee {  
6     }  
7     private Address address;  
8  
9     @Override  
0     public String toString() {  
1         return "Employee [address=" + address + "]";  
2     }  
3  
4     public Address getAddress() {  
5         return address;  
6     }  
7     @Autowired  
8     public void setAddress(Address address) {  
9         this.address = address;  
0     }  
1  
2 }
```

```

3 public class Address {
4
5     private int hno;
6     private String street;
7     private String city;
8     @Override
9     public String toString() {
0         return "Address [hno=" + hno + ", street=" + street + ", city=" + city + "]";
1     }
2
3
4     public int getHno() {
5         return hno;
6     }
7
8     public void setHno(int hno) {
9         this.hno = hno;
0     }
1
2     public String getStreet() {
3
4

```

```
<context:annotation-config>
```

```

<bean name="address"
      class="com.bharath.spring.springcoreadvanced.autowiring.annotations.Address"
      p:hno="26" p:street="hosur" p:city="bangalore" />

<bean name="employee"
      class="com.bharath.spring.springcoreadvanced.autowiring.annotations.Employee">
</bean>
'beans>
```

```

6 public class Test {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new ClassPathXmlApplicationContext(
0             "com/bharath/spring/springcoreadvanced/autowiring/annotations/config.xml"
1
2         Employee employee = (Employee) context.getBean("employee");
3         System.out.println(employee);
4
5     }
6 }
```

Using @Autowired at field and constructor level

```
public class Employee {  
    Employee(Address address) {  
        this.address = address;  
    }  
  
    @Override  
    public String toString() {  
        return "Employee [address=" + address + "]";  
    }  
  
    public Address getAddress() {  
        return address;  
    }  
  
    @Autowired  
    private Address address;  
}
```

Using @Qualifier along with @autowired annotation

```
@Qualifier("address123")  
NoSuchBeanException
```

Qualifier annotation tell the container it should find a bean with given name and then inject that bean

Instead of field name

If the container cant find bean with given name then it will give NoSuchBeanException

```

public class Employee {
    @Autowired
    @Qualifier("address2")
    private Address address;

    @Override
    public String toString() {
        return "Employee [address=" + address + "]";
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}

<context:annotation-config />

<bean name="address1"
      class="com.bharath.spring.springcoreadvanced.autowiring.annotations.Address"
      p:hno="26" p:street="hosur" p:city="bangalore" />

<bean name="address2"
      class="com.bharath.spring.springcoreadvanced.autowiring.annotations.Address"
      p:hno="26" p:street="hosur" p:city="bangalore" />

<bean name="employee"
      class="com.bharath.spring.springcoreadvanced.autowiring.annotations.Employee">
</bean>
'beans>

```

Note: if bean name is not there but we use required=false

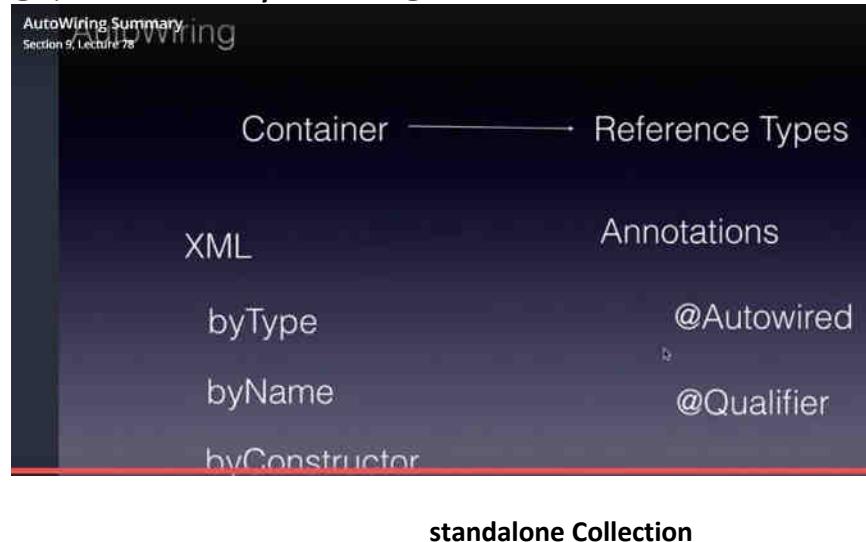
```
public class Employee {  
    @Override  
    public String toString() {  
        return "Employee [address=" + address + "]";  
    }  
  
    public Address getAddress() {  
        return address;  
    }  
  
    @Autowired(required=false)  
    @Qualifier("address789")  
    private Address address;  
}
```

Markers Properties Servers Data Source Explorer Snippets

```
minated> Test (15) [Java Application] /Library/Java/JavaVirtualMachines/jdk  
08, 2017 3:20:38 PM org.springframework.context.support.  
D: Refreshing org.springframework.context.support.ClassPa  
08, 2017 3:20:38 PM org.springframework.beans.factory.xml  
D: Loading XML bean definitions from class path resource  
Employee [address=null]
```

Note: we will use **@Qualifier** if there is a multiple bean with same type( there is a confusion for spring container which bean need to consider)

**@Qualifier** will always use with **@Autowired** annotation



## Standalone Collections

util schema

Add the namespaces on the bean element

```
<util:CN CN-class="" id="">          Re-Usability  
    <value/> or <entry>           Type  
</util>
```

```
public class ProductsList {  
    private List<String> productNames;  
  
    public List<String> getProductNames() {  
        return productNames;  
    }  
  
    public void setProductNames(List<String> productNames) {  
        this.productNames = productNames;  
    }  
}
```

```
1  <?xml version="1.0" encoding="UTF-8"?>  
2  <beans xmlns="http://www.springframework.org/schema/beans"  
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:context="http://www.springframework.org/schema/context"  
4      xmlns:p="http://www.springframework.org/schema/p" xmlns:c="http://www.springframework.org/schema/c"  
5      xmlns:util="http://www.springframework.org/schema/util"  
6      xsi:schemaLocation="http://www.springframework.org/schema/beans  
7          http://www.springframework.org/schema/beans/spring-beans.xsd  
8          http://www.springframework.org/schema/context  
9          http://www.springframework.org/schema/context/spring-context.xsd  
10         http://www.springframework.org/schema/util  
11         http://www.springframework.org/schema/util/spring-util.xsd">  
12  
13  <util:list list-class="java.util.LinkedList" id="productNames">  
14      <value>Mac Book</value>  
15      <value>Iphone</value>  
16  </util:list>  
17  
18  <bean  
19      class="com.bharath.spring.springcoreadvanced.standalone.collections.ProductsList"  
20      name="productsList">  
21      <property name="productName">  
22          <ref bean="productNames" />  
23      </property>  
24  </bean>  
25  
26 </beans>
```

```

public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcoreadvanced/standalone/collections/config.xml");
        ProductsList pl = (ProductsList) context.getBean("productsList");
        System.out.println(pl);
    }
}

```

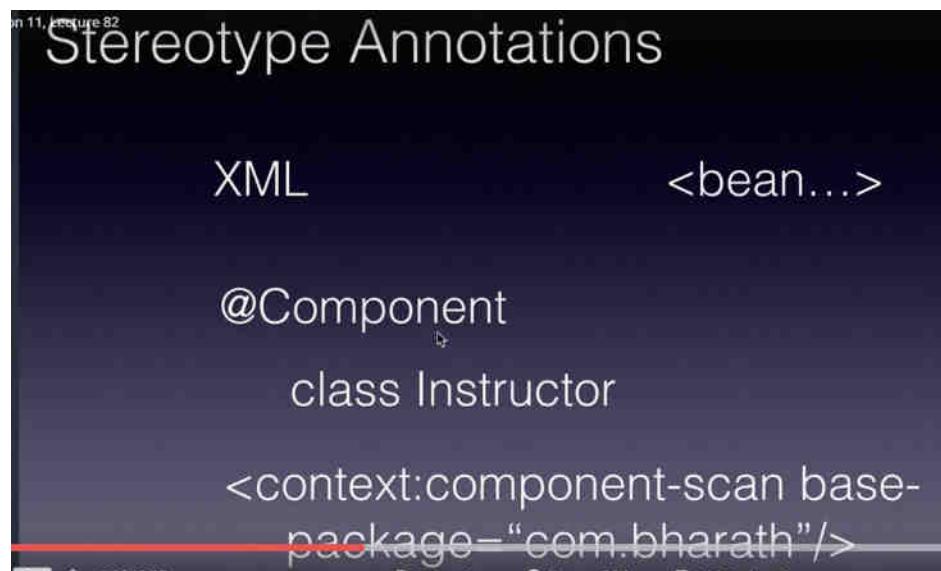
This can be injected to any number of bean it is not local

### Section 11)

**Stereotype annotation** : are used to create object

**Important point Note:** is equivalent to bean tag use in xml

Spring container will scan all package mark with com.bharat and its sub package for classes marked with @Component ---telling spring to create an object particular class



Spring container by default creates object when we will mark class by @ Component

We can use @Component only on the classes we create not on inbuilt classes

```
@Component  
class Instructor  
  
Instructor instructor = new Instructor();
```

## User Defined Types

**Important point: bean name is instructor (name of class) spring will create it but in camel case**

### create a object using annotation

How to create object using annotation instead of xml configuration ?

@Component : tells spring container and object of this class should be created

```
@Component  
public class Instructor {  
  
    private int id;  
    private String name;  
  
    public int getId() {  
        return id;  
    }  
  
    public void setId(int id) {  
        this.id = id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public String toString() {  
        return "Instructor [id=" + id + ", name=" + name + "]";  
    }  
}  
  
> <context:component-scan  
     base-package="com.bharath.spring.springcoreadvanced.stereotype.annotations"></context:component-scan>  
</beans>
```

```

6 public class Test {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new ClassPathXmlApplicationContext(
10             "com/bharath/spring/springcoreadvanced/stereotype/annotations/config.xml");
11
12     Instructor instructor = (Instructor) context.getBean("instructor");
13     System.out.println(instructor);
14 }
15 }
```

### Using different Object names

By default @Component uses class name as a bean but in camel case but can change it

To name a particular bean using component

```

4
5 @Component("inst")
6 public class Instructor {
7
8     private int id;
9     private String name;
10
11    @Override
12    public String toString() {
13        return "Instructor [id=" + id + ", name=" + name + "]";
14    }
15
16    public int getId() {
17        return id;
18    }
19 }
```

### Using @ Scope annotation

How to configure scope using annotation instead of xml configuration

**Note:** we use scope always with @Component annotation

|                             |               |
|-----------------------------|---------------|
| Using the @Scope annotation |               |
| Section 11, Lecture 85      |               |
|                             | singleton     |
|                             | prototype     |
| @Scope                      | request       |
|                             | session       |
|                             | globalsession |

```

5
6 @Component("inst")
7 @Scope("prototype")
8 public class Instructor {
9
10    private int id;
11    private String name;
12
13    @Override
14    public String toString() {
15        return "Instructor [id=" + id + ", name=" + name + "]";
16    }
17
18    public int getId() {
19        return id;
20    }
21
22    public void setId(int id) {
23
24    }
25
26 public class Test {
27
28     public static void main(String[] args) {
29
30         ApplicationContext context = new ClassPathXmlApplicationContext(
31             "com/bharath/spring/springcoreadvanced/stereotype/annotations/config.xml");
32         Instructor instructor = (Instructor) context.getBean("inst");
33         System.out.println(instructor.hashCode());
34
35         Instructor instructor2 = (Instructor) context.getBean("inst");
36         System.out.println(instructor2.hashCode());
37
38     }
39
40 }
41
42
43 //output
44 le spring container creating different object everytime we ask object of a type
45 INFO: Loading XML bean
46 1613255205
47 1897115967

```

### Using @Value annotation with primitive

**@Value:** is used to inject value using annotation not from config.xml

Using @Value Annotation with primitives  
Section 11, Lecture 8a

## @Value

|                  |  |
|------------------|--|
| Primitive Types  | @Value("20")<br><br>@Value("Core Java")        |
| Collection Types | util:Cn id="myList"<br><br>@Value("#{myList}") |
| Object Types     | @Autowired                                     |

The screenshot shows an IDE interface with several tabs at the top: listconfig.xml, ProductsList.java, config.xml, Test.java, Instructor.java, and others. The Instructor.java tab is active, displaying the following Java code:

```
1 package com.bharath.spring.springcoreadvanced.stereotype.annotations;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.context.annotation.Scope;
5 import org.springframework.stereotype.Component;
6
7 @Component("inst")
8 @Scope("prototype")
9 public class Instructor {
10
11     @Value("10")
12     private int id; bean
13     @Value("Bharath Thippireddy") N
14     private String name;
15
16     @Override
17     public String toString() {
18         return "Instructor [id=" + id + ", name=" + name + "]";
19     }
20
21     public int getId() {
22         return id;
23     }
24
25     public void setId(int id) {
26         this.id = id;
27     }
28
29     public String getName() {
30         return name;
31     }
32
33     public void setName(String name) {
```

Annotations @Value("10") and @Value("Bharath Thippireddy") are highlighted with a red box. Handwritten text "bean" is written next to the first annotation, and "N" is written next to the second one. The right side of the screen shows the output of the application running, displaying the loaded beans.

**Note:** annotation always overwrite the value we assign

```
listconfig.xml ProductsList.java config.xml Test.java Instructor.java
Marker Proper Server Data S Snipp

1 package com.bharath.spring.springcoreadvanced.stereotype.annotations;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.context.annotation.Scope;
5 import org.springframework.stereotype.Component;
6
7 @Component("inst")
8 @Scope("prototype")
9 public class Instructor {
10
11     @Value("10")
12     private int id = 15;
13     @Value("Bharath Thippireddy")
14     private String name = "John";
15
16     @Override
17     public String toString() {
18         return "Instructor [id=" + id + ", name=" + name + "]";
19     }
20 }
```

```
<terminated> Test (17) [Java Application] /Library/java/javaVirtualMachine
Mar 10, 2017 2:36:15 PM org.springframework.context.support.ClassPathXmlApplicationContext
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@733...
Mar 10, 2017 2:36:15 PM org.springframework.context.support.ClassPathXmlApplicationContext
INFO: Loading XML bean definitions from class path resource [listconfig.xml]
Instructor [id=10, name=Bharath Thippireddy]
Instructor [id=10, name=Bharath Thippireddy]
```

### using @Value with collection type

How to inject collection Type using @ Value annotation

```
@Component("inst")
@Scope("prototype")
public class Instructor {

    @Value("10")
    private int id = 15;
    @Value("Bharath Thippireddy")
    private String name = "John";

    @Value("#{topics}")
    private List<String> topics;

    @Override
    public String toString() {
        return "Instructor [id=" + id + ", name=" + name + ", topics=" + topics + "]";
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}
```

```
<context:component-scan base-package="com.bharath.spring.springcoreadvanced.stereotype.annotations"/>

<util:list list-class="java.util.LinkedList" id="topics">
    <value>Java Web Services</value>
    <value>Core java</value>
    <value>XSLT</value>
</util:list>

/beans>
```

## Autowiring Object

```
@Component("inst")
@Scope("prototype")
public class Instructor {
    @Value("10")
    private int id = 15;
    @Value("Bharath Thippireddy")
    private String name = "John";
    @Value("#{topics}")
    private List<String> topics;
    @Autowired
    private Profile profile;
    @Override
    public String toString() {
        return "Instructor [id=" + id + ", name=" + name + ", topics=" + topics + ", profile=" + profile + "]";
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
}
```

```
@Component
public class Profile {

    @Value("java instructor")
    private String title;
    @Value("mnc company")
    private String company;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getCompany() {
        return company;
    }
}
```

```
<context:component-scan
    base-package="com.bharath.spring.springcoreadvanced.stereotype.annotations"></context:component-scan>
<util:list list-class="java.util.LinkedList" id="topics">
    <value> java</value>
    <value>spring</value>
    <value>web service</value>
</util:list>
/beans>
```

```
public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcoreadvanced/stereotype/annotations/config.xml");

        Instructor instructor = (Instructor) context.getBean("inst");
        System.out.println(instructor);

    }
}
```

Stereotype Annotations Summary  
Section 12, Lecture 90

@Component

```
<context:component-scan base-package = "com.app.beans"
```

@Scope      @Value      @Autowired

## Section 12

### Spring expression language

# ----- tells spring container to evaluate a value

Introduction  
Section 12, Lecture 90

SpEL      Spring Expression Language

Expression      @Value

Classes, Variable, Methods, Constructors and Objects

and symbols

char, numerics, operators, keywords and special symbols which return a value

```
@value("#{66+44}")  
@value("#{5>6?22:33}")
```

static methods      object methods      variables

↓

```
1  @Component("inst")
2  public class Instructor {
3      @Value("#{10+20}")
4      private int id;
5      @Value("saurabh")
6      private String name;
7      @Value("#{topics}")
8      private List<String> topics;
9
10     @Autowired
11     private Profile profile;
```

### Using static method

How to invoke static method inside the expression

Syntax

```
T(class).method(param)
```

```
9
10 @Component("inst")
11 @Scope("prototype")
12 public class Instructor {
13
14     @Value("#{T(java.lang.Math).abs(-99) }")
15     private int id = 15;
16     @Value("Bharath Thippireddy")
17     private String name = "John";
```

Output// we will get 99 not -99

```
INFO: LOADING AND DUMPING STATEMENTS FROM CLASS PATH RESOURCE [com/universal/spring/app]
Instructor [id=99, name=Bharath Thippireddy, topics=[Java Web Services, Core java,
```

### Accessing static variables and creating object

To invoke constructor with value in expression

```

1 @Component("inst")
2 @Scope("prototype")
3 public class Instructor {
4
5     @Value("#{new Integer(88) }")
6     private int id = 15;
7     @Value("Bharath Thippireddy")

```

T(class).method(param)

We are assigning value to field id

```

1 @Scope("prototype")
2 public class Instructor {
3
4     @Value("#{T(java.lang.Integer).MIN_VALUE }")
5     private int id = 15;
6     @Value("Bharath Thippireddy")

```

### Creating the String type

How to use string type in expression

**Note : we can't use double code in expression language**

```

1 package com.bharath.spring.springcoreadvanced.stereotype.annotations;
2
3 import java.util.List;
4
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.beans.factory.annotation.Value;
7 import org.springframework.context.annotation.Scope;
8 import org.springframework.stereotype.Component;
9
10 @Component("inst")
11 @Scope("prototype")
12 public class Instructor {
13
14     @Value("#{T(java.lang.Integer).MIN_VALUE }")
15     private int id = 15;
16     @Value("#{ 'Bharath Thippireddy' }")
17     private String name = "John";
18 }

```

```

<terminated> Test (17) [Java Application] /Library/Java/JavaVirtualMachin
Mar 10, 2017 4:51:27 PM org.springframework.context.support.
INFO: Refreshing org.springframework.context.support.
Mar 10, 2017 4:51:27 PM org.springframework.beans.factory.
INFO: Loading XML bean definitions from class path reso
Instructor [id=-2147483648, name=Bharath Thippireddy,
Instructor [id=-2147483648, name=Bharath Thippireddy,

```

To invoke method on string

```

9
10 @Component("inst")
11 @Scope("prototype")
12 public class Instructor {
13
14     @Value("#{T(java.lang.Integer).MIN_VALUE}")
15     private int id = 15;
16     @Value("#{'Bharath Thippireddy'.toUpperCase()}")
17     private String name = "John";
18
19     @Value("#{topics}")
20
21     @Autowired
22
9
10 @Component("inst")
11 @Scope("prototype")
12 public class Instructor {
13
14     @Value("#{T(java.lang.Integer).MIN_VALUE}")
15     private int id = 15;
16     @Value("#{new java.lang.String('Bharath Thippireddy')}")
17     private String name = "John";
18
19     @Value("#{topics}")
20     private List<String> topics;
21
22     @Autowired

```

#### Expressing Boolean types:-

```

@Component("inst")
@Scope("prototype")
public class Instructor {
    @Value("#{T(java.lang.Integer).MIN_VALUE}")
    private int id = 15;
    @Value("#{new java.lang.String('Bharath Thippireddy')}")
    private String name = "John";
    @Value("#{topics}")
    private List<String> topics;
    @Value("#{2+4>5}")
    private boolean active;
    @Autowired
    private Profile profile;
}

```

#### section -13

#### Interface Injection (through xml)

## Interface Injection



`OrderBOImpl` depends on `OrderDAOImpl` but in config.xml we will configure bean as implementation class.

But in class we will define as interface and spring will inject object of its implementation class

(think like `OrderBOImpl` is service class and `OrderDAOImpl` is dao class)

1)

```
public interface OrderBO {
    void placeOrder();
}

public class OrderBOImpl implements OrderBO {
    private OrderDAO dao;

    @Override
    public void placeOrder() {
        System.out.println("inside order BO");
        dao.createOrder();
    }

    public OrderDAO getDao() {
        return dao;
    }

    public void setDao(OrderDAO dao) {
        this.dao = dao;
    }
}
```

```

public interface OrderDAO {
    void createOrder();
}

public class OrderDAOImpl implements OrderDAO {
    @Override
    public void createOrder() {
        System.out.println("Inside order DAO createOrder()");
    }
}

<bean id="dao" class="com.bharath.spring.springcoreadvanced.injecting.interfaces.OrderDAOImpl" />
<bean id="bo" class="com.bharath.spring.springcoreadvanced.injecting.interfaces.OrderBOImpl">
    <property name="dao" ref="dao"/>
</bean>
</beans>

public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcoreadvanced/injecting/interfaces/config.xml");
        OrderBO bo = (OrderBO) context.getBean("bo");
        bo.placeOrder();
    }
}

//output
-----
inside order BO
Inside order DAO createOrder()

```

**Interface injection ( using annotation)**

```
public interface OrderBO {  
    void placeOrder();  
}  
  
7 @Component("bo")  
3 public class OrderBOImpl implements OrderBO {  
3  
3@Autowired  
1     @Qualifier("dao2")  
2     private OrderDAO dao;  
3  
4@Override  
5     public void placeOrder() {  
5         System.out.println("inside order BO");  
7         dao.createOrder();  
3     }  
9  
3@  
1     public OrderDAO getDao() {  
2         return dao;  
2     }  
3  
4@  
5     public void setDao(OrderDAO dao) {  
5         this.dao = dao;  
5     }  
7 }  
  
: public interface OrderDAO {  
:     void createOrder();  
: }  
  
+  
+ @Component("dao")  
+ public class OrderDAOImpl implements OrderDAO {  
+  
3@  
3     @Override  
3     public void createOrder() {  
3         System.out.println("Inside order DAO createOrder()");  
L  
2     }  
3  
+ }  
+
```

```

@Component("dao2")
public class OrderDAOImpl2 implements OrderDAO {

    @Override
    public void createOrder() {
        System.out.println("Inside OrderDAOImpl2 createOrder()");
    }
}

<!-->
<context:component-scan
    base-package="com.bharath.spring.springcoreadvanced.injecting.interfaces" />
</beans>

```

```

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springcoreadvanced/injecting/interfaces/config.xml");

        OrderBO bo = (OrderBO) context.getBean("bo");
        bo.placeOrder();
    }
}

```

## Section 14

### Spring JDBC

**Point : jdbc connection code we need to write in every class**



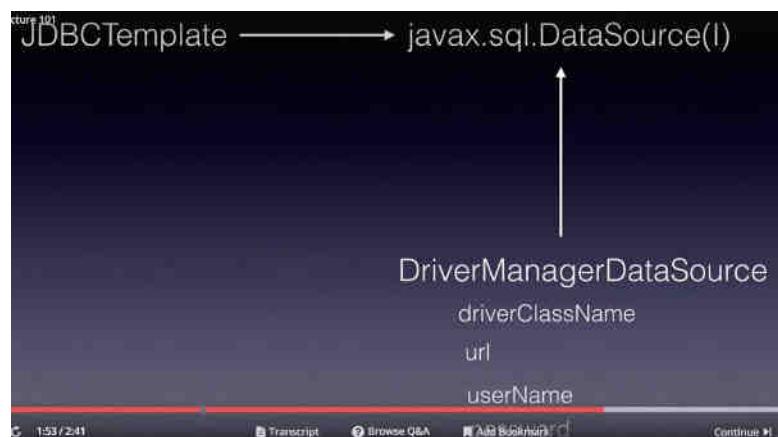
JDBC + Template  
Technology Design Pattern

Template

Common Code

Developer loves JDBCTemplate

Java.sql.DataSource() is a interface and DMDS is implementation class which create connection with database and it give connection to JDBCTemplate



JDBCTemplate

```
update(String sql) int
update(String sql, Object...args) int
```

insert, update and delete

Create employee table in database

```
create database mydb;  
use mydb;  
create table employee(id int,firstname varchar(20),lastname varchar(20));  
select * from employee;
```

### step to use JDBCTemplate

**Note:** JDBCTemplate depends on DriverManagerDataSource  
Internally jdbcTemplate use dataSource to create connection

| Steps to use the JDBC <small>T</small> emplate |                             |
|--|-----------------------------|
| DriverManagerDataSource                        | dataSource                  |
| driverClassName                                | com.mysql.jdbc.Driver       |
| url  | jdbc:mysql://localhost/mydb |
| username                                       | root                        |
| password                                       | test                        |
| JDBC <small>T</small> emplate                  | jdbcTemplate                |
| dataSource                                     |                             |

Create the test class and use the JDBCTemplate

### configure the DataSource and JDBCTemplate

**Note:**

Use ctrl shift T to search package name

```
<bean class="org.springframework.jdbc.datasource.DriverManagerDataSource"  
      name="dataSource" p:driverClassName="com.mysql.jdbc.Driver" p:url="jdbc:mysql://localhost/mydb"  
      p:username="root" p:password="test" />  
  
<bean class="org.springframework.jdbc.core.JdbcTemplate" name="jdbcTemplate"  
      p:dataSource-ref="dataSource" />  
  
</beans>
```

Use JdbcTemplate to perform insert operation

```

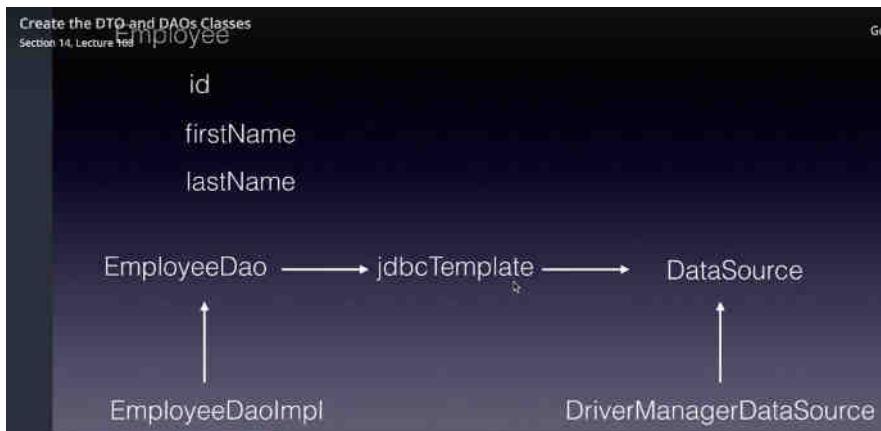
<bean
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    name="dataSource" p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost/mydb" p:username="root"
    p:password="test" />

<bean class="org.springframework.jdbc.core.JdbcTemplate"
    name="jdbcTemplate" p:dataSource-ref="dataSource" />

public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("com/bharath/spring/springjdbc/config.xml");
        JdbcTemplate jdbcTemplate = (JdbcTemplate) context.getBean("jdbcTemplate");
        String sql = "insert into employee values(?, ?, ?)";
        int result = jdbcTemplate.update(sql, new Integer[2], "saurabh", "kesarwani");
        System.out.println("number of record inserted are = "+result);
    }
}

```

Create the dto (entity class) and dao class and uses jdbcTemplate in dao class



a)

```
package com.bharath.spring.springjdbc.employee.dto;

public class Employee {
    private int id;
    private String firstName;
    private String lastName;

    @Override
    public String toString() {
        return "Employee [id=" + id + ", firstName=" + firstName + ", lastName=" + lastName + "]";
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
}

b)
package com.bharath.spring.springjdbc.employee.dao;

import com.bharath.spring.springjdbc.employee.dto.Employee;

public interface EmployeeDao {
    int create(Employee employee);

}
```

c)

```

1 package com.bharath.spring.springjdbc.employee.dao.impl;
2
3*import org.springframework.jdbc.core.JdbcTemplate;□
4
5 public class EmployeeDaoImpl implements EmployeeDao {
6
7     private JdbcTemplate jdbcTemplate;
8
9     @Override
10    public int create(Employee employee) {
11        String sql = "insert into employee values(?, ?, ?)";
12
13        int result = jdbcTemplate.update(sql, employee.getId(), employee.getFirstName(), employee.getLastName());
14        return result;
15    }
16
17    public JdbcTemplate getJdbcTemplate() {
18        return jdbcTemplate;
19    }
20
21    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
22        this.jdbcTemplate = jdbcTemplate;
23    }
24
25 }
26
27
28 }
```

d)

```

<bean
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    name="dataSource" p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost/mydb" p:username="root"
    p:password="test" />

<bean class="org.springframework.jdbc.core.JdbcTemplate"
    name="jdbcTemplate" p:dataSource-ref="dataSource" />

<bean id="employeeDao"
    class="com.bharath.spring.springjdbc.employee.dao.impl.EmployeeDaoImpl">
    <property name="jdbcTemplate">
        <ref bean="jdbcTemplate" />
    </property>
</bean>

'beans>
```

e)

```
1 public class Test {  
2  
3     public static void main(String[] args) {  
4         ApplicationContext context = new ClassPathXmlApplicationContext(  
5             "com/bharath/spring/springjdbc/employee/test/config.xml");  
6  
7         EmployeeDao dao = (EmployeeDao) context.getBean("employeeDao");  
8  
9         Employee emp = new Employee();  
10        emp.setId(2);  
11        emp.setFirstName("shagun");  
12        emp.setLastName("kesarwani");  
13  
14        int result = dao.create(emp);  
15        System.out.println("number of record inserted are = " + result);  
16    }  
17}  
18}
```

#### Update the row

Note:

Employee.java and config.xml are same as above

a)

```
1 package com.bharath.spring.springjdbc.employee.dao;  
2  
3 import com.bharath.spring.springjdbc.employee.dto.Employee;  
4  
5 public interface EmployeeDao {  
6     int create(Employee employee);  
7  
8     int update(Employee employee);  
9  
0 }  
1
```

b)

```

public class EmployeeDaoImpl implements EmployeeDao {
    private JdbcTemplate jdbcTemplate;

    @Override
    public int create(Employee employee) {
        String sql = "insert into employee values(?, ?, ?)";

        int result = jdbcTemplate.update(sql, employee.getId(), employee.getFirstName(), employee.getLastName());
        return result;
    }

    @Override
    public int update(Employee employee) {
        String sql = "update employee set firstname=? , lastname=? where id = ?";
        int result = jdbcTemplate.update(sql, employee.getFirstName(), employee.getLastName(), employee.getId());
        return result;
    }

    public JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}

c)
public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
            "com/bharath/spring/springjdbc/employee/test/config.xml");

        EmployeeDao dao = (EmployeeDao) context.getBean("employeeDao");

        Employee emp = new Employee();
        emp.setId(1);
        emp.setFirstName("saurabh");
        emp.setLastName("kesarwani");

        //int result = dao.create(emp);
        int result = dao.update(emp);
        System.out.println("number of record updated are = " + result);
    }
}

```

delete a row

a)

```
4
5 public interface EmployeeDao {
6     int create(Employee employee);
7
8     int update(Employee employee);
9
10    int delete(int id);
11 }
12
13 b)
14
15 public class EmployeeDaoImpl implements EmployeeDao {
16
17     private JdbcTemplate jdbcTemplate;
18
19     @Override
20     public int create(Employee employee) {
21         String sql = "insert into employee values(?, ?, ?)";
22
23         int result = jdbcTemplate.update(sql, employee.getId(), employee.getFirstName(), employee.getLastName());
24         return result;
25     }
26
27     @Override
28     public int update(Employee employee) {
29         String sql = "update employee set firstname=? , lastname=? where id = ?";
30         int result = jdbcTemplate.update(sql, employee.getFirstName(), employee.getLastName(), employee.getId());
31         return result;
32     };
33     @Override
34     public int delete(int id) {
35         String sql = " delete from employee where id=?";
36         int result = jdbcTemplate.update(sql, id);
37         return result;
38     }
39
40     public JdbcTemplate getJdbcTemplate() {
41         return jdbcTemplate;
42     }
43
44     public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
45         this.jdbcTemplate = jdbcTemplate;
46     }
47 }
```

c)

```

5
6 public class Test {
7
8     public static void main(String[] args) {
9         ApplicationContext context = new ClassPathXmlApplicationContext(
10             "com/bharath/spring/springjdbc/employee/test/config.xml");
11
12         EmployeeDao dao = (EmployeeDao) context.getBean("employeeDao");
13
14         Employee emp = new Employee();
15         emp.setId(1);
16         emp.setFirstName("saurabh");
17         emp.setLastName("kesarwani");
18
19         // int result = dao.create(emp);
20         // int result = dao.update(emp);
21         int result = dao.delete(1);
22         System.out.println("number of record updated are = " + result);
23     }
24 }

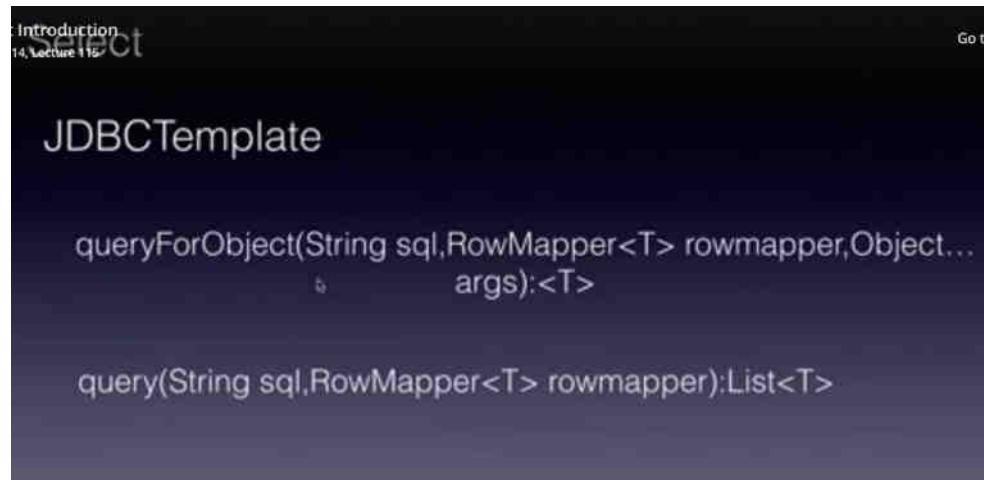
```

### Select Introduction

#### Important:

**queryForObject** : return single record

**Query**: return multiple record



RowMapper: is interface in spring framework which we need to implement and it contains method

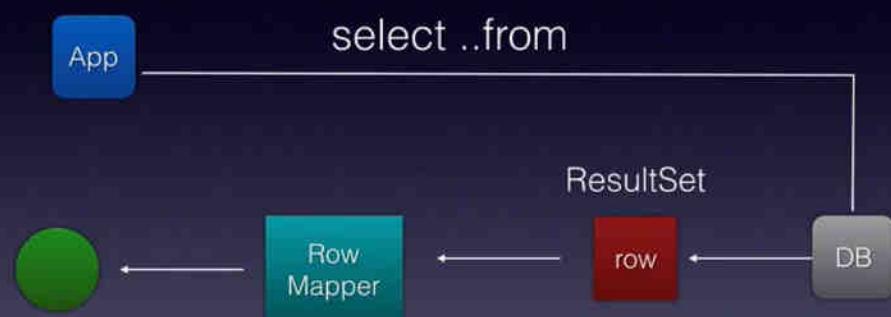
It map jdbc ResultSet that come back into object of class that we create

## Select Introduction

Section 14, Lecture 115

### RowMapper<T> (I):

ResultSet → Object



Employee mapRow(ResultSet rs)

emp.set(rs.get(1));

Select query run into database it return row as resultSet and RowMapper we will implement and override

Method mapRow with argument ResultSet . It get the result from ResultSet and we will set in our object

**Create the read method and row mapper**

#### 1. Fetching single record

a)

```

public interface EmployeeDao {
    int create(Employee employee);

    int update(Employee employee);

    int delete(int id);

    Employee read(int id);
```

}

b)

```

public class EmployeeDaoImpl implements EmployeeDao {

    private JdbcTemplate jdbcTemplate;

    @Override
    public int create(Employee employee) {
        String sql = "insert into employee values(?, ?, ?)";

        int result = jdbcTemplate.update(sql, employee.getId(), employee.getFirstName(), employee.getLastName());
        return result;
    }

    @Override
    public int update(Employee employee) {
        String sql = "update employee set firstname=? , lastname=? where id = ?";
        int result = jdbcTemplate.update(sql, employee.getFirstName(), employee.getLastName(), employee.getId());
        return result;
    };

    @Override
    public int delete(int id) {
        String sql = " delete from employee where id=?";
        int result = jdbcTemplate.update(sql, id);
        return result;
    }

    @Override
    public Employee read(int id) {
        String sql = " select * from employee where id=?";

        EmployeeRowMapper rowMapper = new EmployeeRowMapper();
        Employee employee = jdbcTemplate.queryForObject(sql, rowMapper, id);
        return employee;
    }
}
```

c)

```
2
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5
6 import org.springframework.jdbc.core.RowMapper;
7
8 import com.bharath.spring.springjdbc.employee.dto.Employee;
9
10 public class EmployeeRowMapper implements RowMapper<Employee> {
11
12     @Override
13     public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {
14
15         Employee emp = new Employee();
16         emp.setId(rs.getInt(1));
17         emp.setFirstName(rs.getString(2));
18         emp.setLastName(rs.getString(3));
19         return emp;
20     }
21
22 }
23
24 d)
25
26 public class Test {
27
28     public static void main(String[] args) {
29         ApplicationContext context = new ClassPathXmlApplicationContext(
30             "com/bharath/spring/springjdbc/employee/test/config.xml");
31
32         EmployeeDao dao = (EmployeeDao) context.getBean("employeeDao");
33
34         Employee emp = new Employee();
35         emp.setId(1);
36         emp.setFirstName("saurabh");
37         emp.setLastName("kesarwani");
38
39         // int result = dao.create(emp);
40         // int result = dao.update(emp);
41         int result = dao.delete(1);
42         Employee employee = dao.read(2);
43         System.out.println("number of record updated are = " + employee);
44     }
45
46 }
47
48 }
```

## Reading multiple records

a)

```
>
7 public interface EmployeeDao {
3     int create(Employee employee);
3
3     int update(Employee employee);
L
2     int delete(int id);
3
4     Employee read(int id);
3
5     List<Employee> read();
7
3 }
3
```

b)

```
@Override
public List<Employee> read() {
    String sql = "select * from employee";
    EmployeeRowMapper rowMapper = new EmployeeRowMapper();
    List<Employee> result = jdbcTemplate.query(sql, rowMapper);
    return result;
}
```

c)

```
public class EmployeeRowMapper implements RowMapper<Employee> {

    @Override
    public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

        Employee emp = new Employee();
        emp.setId(rs.getInt(1));
        emp.setFirstName(rs.getString(2));
        emp.setLastName(rs.getString(3));
        return emp;
    }
}
```

d)

```

}
public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "com/bharath/spring/springjdbc/employee/test/config.xml");

        EmployeeDao dao = (EmployeeDao) context.getBean("employeeDao");

        Employee emp = new Employee();
        emp.setId(1);
        emp.setFirstName("saurabh");
        emp.setLastName("kesarwani");

        // int result = dao.create(emp);
        // int result = dao.update(emp);
        /*
         * int result = dao.delete(1); Employee employee = dao.read(2);
         */
    }

    List<Employee> result = dao.read();
    System.out.println("number of record updated are = " + result);
}

```

Spring will give list of employee

```
//output
SERVE CERTIFICATE VERIFICATION.
number of record updated are = [Employee [id=2, firstName=neha, lastName=kesarwani], Employee [id=1,
```

```

        autowire JdbcTemplate
to use autowire
    @Component("employeeDao")
    public class EmployeeDaoImpl implements EmployeeDao {
        @Autowired
        private JdbcTemplate jdbcTemplate;
    }
    @Override

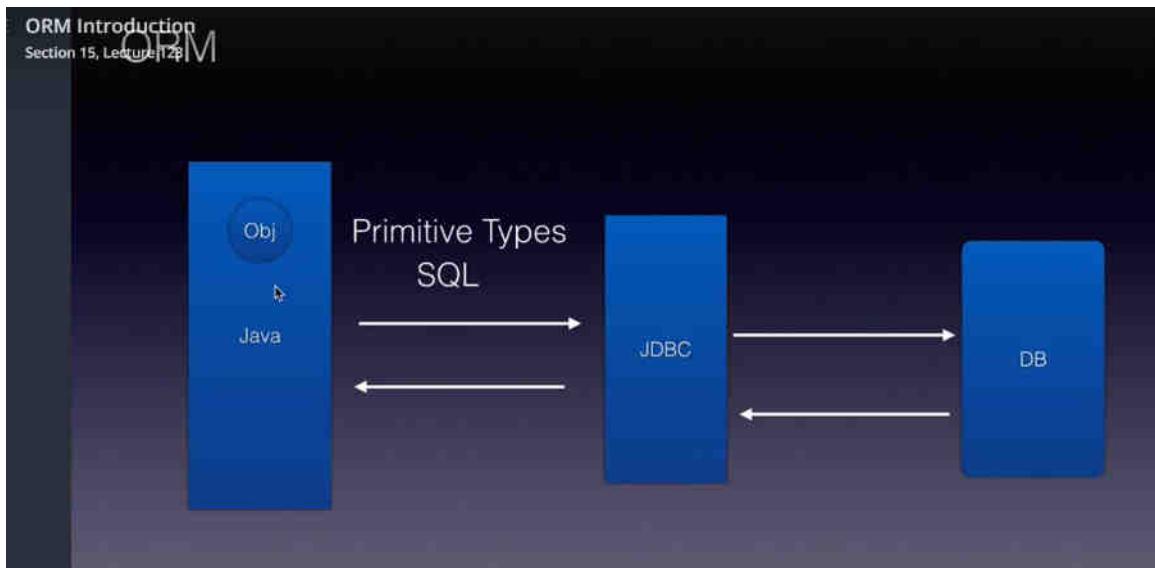
```

section -15

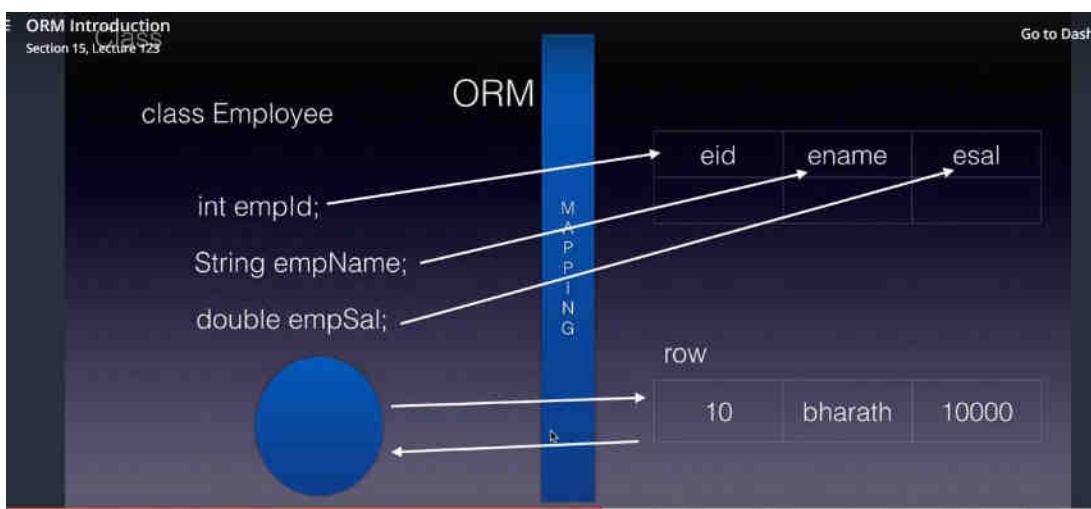
**ORM Introduction**

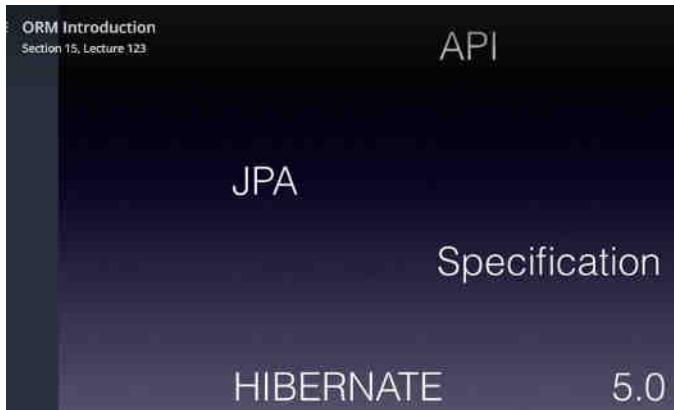
Developer take a primitive data and create sql statement and execute against db and when response come back we convert data into java object which is hectic that's why orm comes

NOTE: in springJdbc there is no mapping means pojo class variable and table column name can be different



Field name and table column name must be same we provide mapping orm automatically covert object into data and vice versa without any writing sql query





### **Spring orm introduction**

It removes all the boilerplate code like creating session

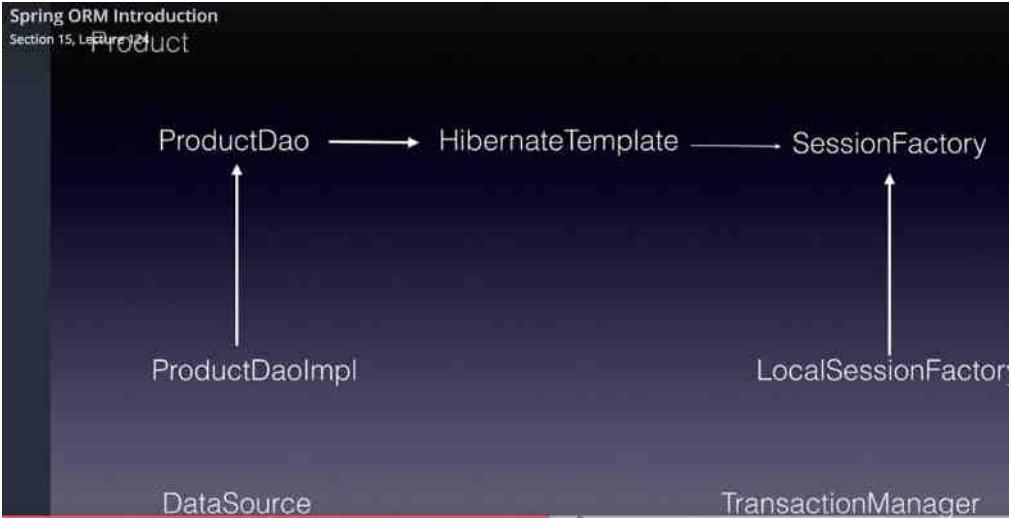
loadAll return list of object behind the scene it will generate sql statement . Every method take object



Database code will be written in dao class which depends on HibernateTemplate and it will get connection

From SessionFactory interface which is implemented by class LocalSessionFactoryBean provided by spring.

And SessionFactory need a dataSource we also need TransactionManager to write operation to database to ensure that whether all operation falling in one transaction



2

`LocalSessionFactoryBean` takes 3 arguments



`Hibernate.dialect` is responsible to generate sql and `hibernate.show_sql` it will tell hibernate to show sql statement on console

# Hibernate Properties

key : value

hibernate.dialect = org.hibernate.dialect.MYSQLDialect

hibernate.show\_sql = true

## Mapping an entity to a database table

@Entity and @Id is mandatory we use @Table and @Column when table name and column name is different

From the database

## JPA Mapping:

### XML and Annotations

@Entity

    @Table

    @Id

    @Column

```
@Entity  
 @Table(name="emp")  
 public class Employee{  
     @Id  
     @Column(name="id")  
     private int id;  
     @Column(name="firstname")  
     private int firstName;  
     @Column(name="lastName")  
     private int lastName;
```

#### **create the product table in the database**



```
use mydb;  
create table product(id int,name varchar(20),description varchar(100),price decimal(8,3));  
select * from product;
```

Here 8 is total no of digit and 3 can be go after decimal

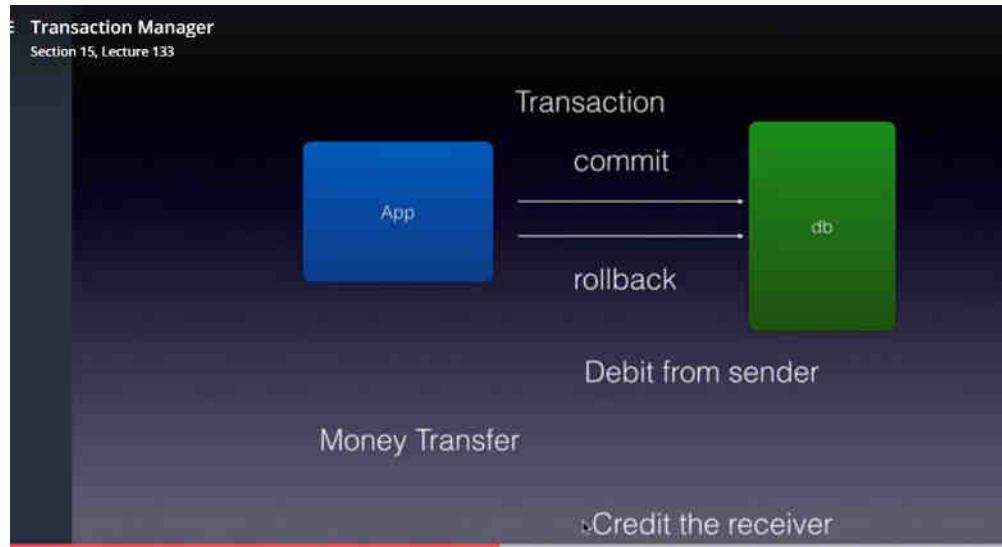
NOTE:

We need two dependency more spring-orm and hibernate-core

#### **TransactionManager (to write in database)**

Everything occur in single transaction

Debit and credit should occur in single transaction otherwise it should roll back



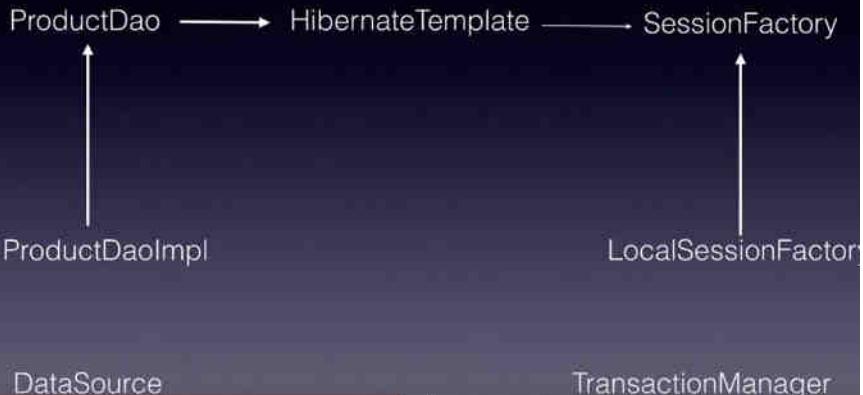
Spring provides `transactionManager`.

`HibernateTransactionManager` is a bean for this we need to define `<tx:annotation-driven/>` in configuration file

And use `@Transactional` at method level if any exception occur in that method then spring will roll back it otherwise it will execute that method in single transaction



Step to create and configure hibernate



a)

```
use mydb;

create table product(id int,name varchar(20),description varchar(100),price decimal(8,3));

select * from product;
```

b. Create entity( pojo ) class

```
package com.bharath.spring.springorm.product.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "product")
public class Product {
    @Id
    @Column(name = "id")
    private int id;
    @Column(name = "name")
    private String name;
    @Column(name = "description")
    private String desc;
    @Column(name = "price")
    private double price;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}
```

c. Create interface

```
1 package com.bharath.spring.springorm.product.dao;
2
3 import com.bharath.spring.springorm.product.entity.Product;
4
5 public interface ProductDao {
6
7     int create(Product product);
8 }
```

d. Create implementation class

```
1 package com.bharath.spring.springorm.product.dao.impl;
2
3 import java.io.Serializable;
4
5 @Component("productDao")
6 public class ProductDaoImpl implements ProductDao {
7     @Autowired
8     HibernateTemplate hibernateTemplate;
9
10    @Override
11    @Transactional
12    public int create(Product product) {
13        Integer result = (Integer) hibernateTemplate.save(product);
14        return result;
15    }
16
17 }
```

e. Create config.xml

hibernateTemplate had dependency on sessionFactory which had dependency on dataSource

```

<tx:annotation-driven />
<context:component-scan base-package="com.bharath.spring.springorm.product.dao.impl" />
<bean
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
    name="dataSource" p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost/mydb" p:username="root"
    p:password="test" />

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"
    p:dataSource-ref="dataSource">
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>
    <property name="annotatedClasses">
        <list>
            <value>com.bharath.spring.springorm.product.entity.Product</value>
        </list>
    </property>
</bean>

<bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate5.HibernateTemplate"
    p:sessionFactory-ref="sessionFactory">
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager"
    p:sessionFactory-ref="sessionFactory">
</bean>
beans>

```

f. Test for insert record

```

public class Test {

    public static void main(String[] args) {

        ApplicationContext context = new ClassPathXmlApplicationContext(
                "com/bharath/spring/springorm/product/test/config.xml");
        ProductDao productDao = (ProductDao) context.getBean("productDao");
        Product product = new Product();
        product.setId(1);
        product.setName("i phone");
        product.setDesc("awesomoe");
        product.setPrice(9000);
        productDao.create(product);
    }

}

//show in console
Hibernate: insert into product (description, name, price, id) values (?, ?, ?, ?)

```

#### Implement the update and delete method

a)

```

5 public interface ProductDao {
6
7     int create(Product product);
8
9     void update(Product product);
.0
.1     void delete(Product product);|
.2
.3 }

```

b)

```
1  @Component("productDao")
2  public class ProductDaoImpl implements ProductDao {
3      @Autowired
4      HibernateTemplate hibernateTemplate;
5
6      @Override
7      @Transactional
8      public int create(Product product) {
9          Integer result = (Integer) hibernateTemplate.save(product);
10         return result;
11     }
12
13     @Override
14     @Transactional
15     public void update(Product product) {
16         hibernateTemplate.update(product);
17     }
18
19     @Override
20     @Transactional
21     public void delete(Product product) {
22         hibernateTemplate.delete(product);
23     }
24 }
```

c)

```

3
4 public class Test {
5
6     public static void main(String[] args) {
7
8         ApplicationContext context = new ClassPathXmlApplicationContext(
9             "com/bharath/spring/springorm/product/test/config.xml");
10        ProductDao productDao = (ProductDao) context.getBean("productDao");
11        Product product = new Product();
12        product.setId(1);
13        product.setName("i phone");
14        product.setDesc("awesomoe");
15        product.setPrice(10000);
16        //productDao.create(product);
17        //productDao.update(product);
18        productDao.delete(product);
19    }
20
21 }

```

In case of delete first it will select then delete record

```

Hibernate: select product_.id, product_.description as descript2_0_, product_.name as name3_0_, product_.price as price4_0_ from product
Hibernate: delete from product where id=?

```

### fetch a single record

Read data from database

a)

```

1
2
3 public interface ProductDao {
4
5     int create(Product product);
6
7     void update(Product product);
8
9     void delete(Product product);
10
11    Product find(int id);
12}

```

b) what type of entity should return

```

@Override
@Transactional
public Product find(int id) {
    Product product = hibernateTemplate.get(Product.class, id);
    return product;
}

c)
9 public class Test {
0
1*     public static void main(String[] args) {
2
3         ApplicationContext context = new ClassPathXmlApplicationContext(
4             "com/bharath/spring/springorm/product/test/config.xml");
5         ProductDao productDao = (ProductDao) context.getBean("productDao");
6         /* Product product = new Product();
7         product.setId(1);
8         product.setName("i phone");
9         product.setDesc("awesomoe");
0         product.setPrice(10000);
1         //productDao.create(product);
2         //productDao.update(product);
3         productDao.delete(product);*/
4
5         Product product = productDao.find(1);
6         System.out.println(product);
7     }
8
9 }
a)

//output
Hibernate: select product0_.id as id1_0_0_, product0_.description as descript2_0_0_, product0_.name as name3_0_0_,
Product [id=1, name=iphone, desc=good, price=700.0]

```

fetch all records

Use loadAll()

a)

```
5  
6 public interface ProductDao {  
7     int create(Product product);  
8     void update(Product product);  
9     void delete(Product product);  
10    Product find(int id);  
11    List<Product> findAll();  
12 }  
13  
b)
```

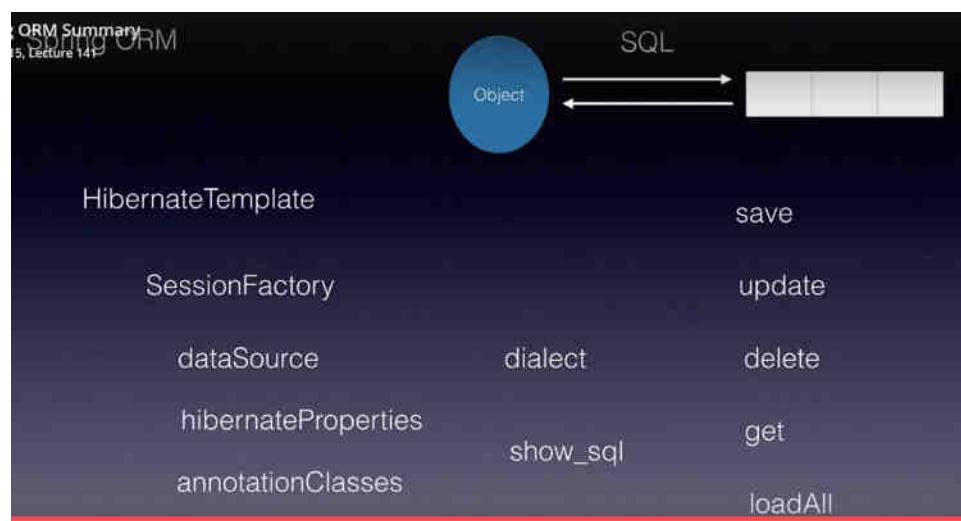
```
@Override  
|  
public List<Product> findAll() {  
    List<Product> products = hibernateTemplate.loadAll(Product.class);  
    return products;  
}
```

c)

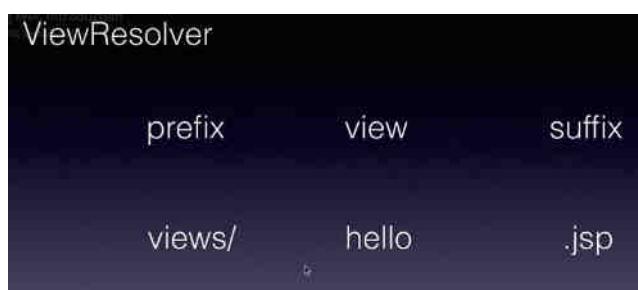
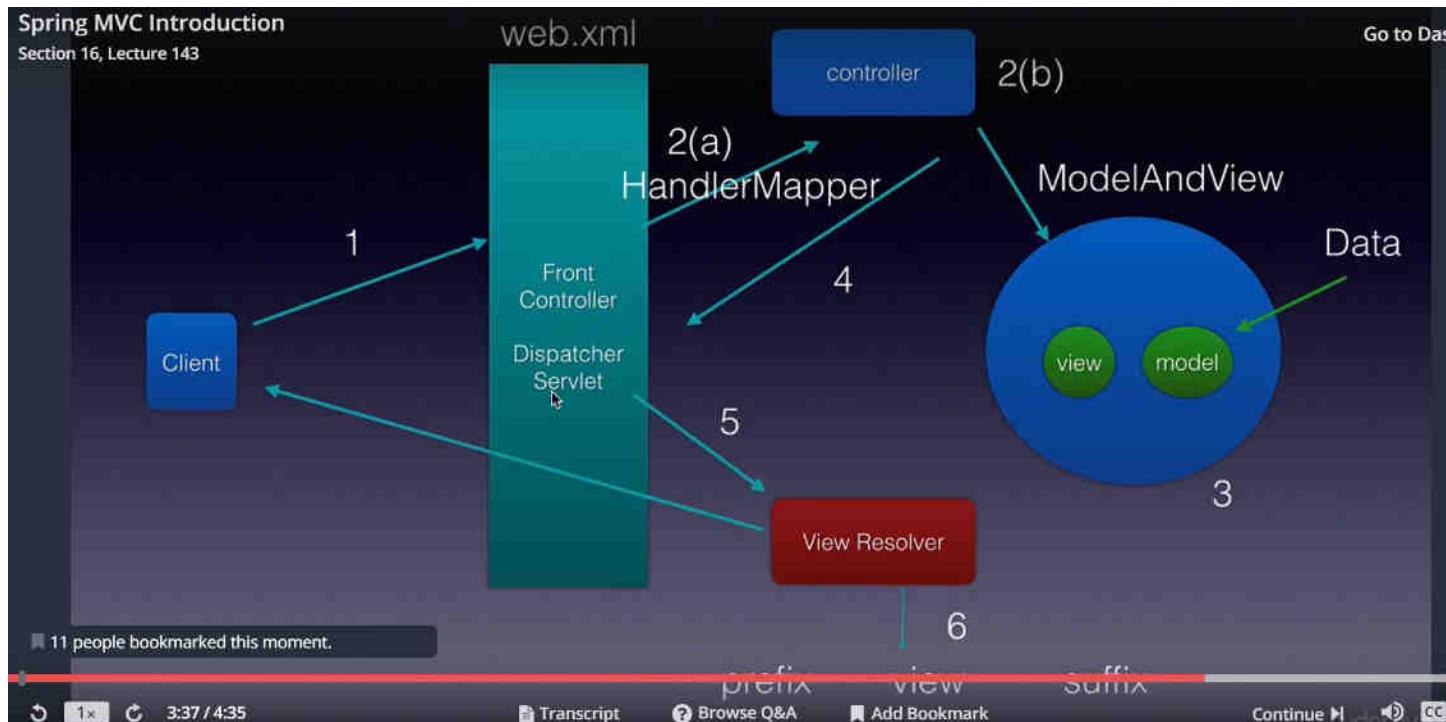
```
List<Product> products = productDao.findAll();  
System.out.println(products);
```

Output

```
[Product [id=1, name=iphone, desc=good, price=700.0], Product [id=2, name=mac, desc=good, price=300.0]]
```



## section-16 ( spring mvc introduction)



1) Configure the dispatcher servlet

## Spring MVC Application Creation Steps:

- Configure the dispatcher servlet
- Create the spring configuration
- Configure the View Resolver
- Create the controller
- Create the folder structure and view

```

5<web-app>
6    <display-name>Hello Spring MVC</display-name>
7
8<servlet>
9    <servlet-name>dispatcher</servlet-name>
10   <servlet-class>org.springframework.web.servlet.DispatcherServlet </servlet-class>
11 </servlet>
12
13<servlet-mapping>
14   <servlet-name>dispatcher</servlet-name>
15   <url-pattern>/</url-pattern>
16 </servlet-mapping>
17 </web-app>
18

```

### 2) create the spring configuration

Name of xml file:-

Servlet name-servlet.xml

Dispatcher-servlet.xml

### 3) configure the view resolver

Dispatcher-servlet.xml

```

<context:component-scan base-package= "com.bharath.spring.springmvc.controller" />
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver"
    name="viewResolver">
    <property name="prefix">
        <value>/WEB-INF/views</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>

```

#### 4) create and configure the controller

```
@Controller
public class HelloController {

    @RequestMapping("/hello")
    public ModelAndView hello() {
        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("hello");
        return modelAndView;
    }
}
```

#### 5) create the view

Hello.jsp

```
1 <% page language="java" contentType="text/html; charset=UTF-8"
2     pageEncoding="UTF-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7 <title>Hello</title>
8 </head>
9 <body>
0 <H1>Hello from Spring MVC!!!</H1>
1 </body>
2 </html>
```

#### section 17) sending data from controller to ui

1)Introduction

## Controller to the UI



### 2) Sending primitive types (and get data in jsp first way)

a)

```
>
>  @Controller
>  public class HelloController {
>    @RequestMapping("/hello")
>    public ModelAndView hello() {
>      ModelAndView modelAndView = new ModelAndView();
>      modelAndView.setViewName("hello");
>      modelAndView.addObject("id",1);
>      modelAndView.addObject("name","saurabh");
>      modelAndView.addObject("salary",500000);
>      return modelAndView;
>
>    }
>  }
```

b. Hello.jsp

```
</head>
<body>
<%
  Integer id=(Integer)request.getAttribute("id");
  String name=(String)request.getAttribute("name");
  Integer salary=(Integer)request.getAttribute("salary");
  out.println("id : "+id);
  out.println("name : "+name);
  out.println("salary : "+salary);
%
</body>
</html>
```

### 3. Using jsp expression language( second way to get data from controller to jsp page)

`${id}` ----- jsp expression will evaluate id value . It will look this value in request scope

```
1  pageEncoding="ISO-8859-1" %>
2 <%@ page isELIgnored="false" %> | 
3 <html>
4 <head>
5 <meta charset="ISO-8859-1">
6 <title>Hello</title>
7 </head>
8 <body>
9   <%
10  Integer id=(Integer)request.getAttribute("id");
11  String name=(String)request.getAttribute("name");
12  Integer salary=(Integer)request.getAttribute("salary");
13  out.println("id : "+id);
14  out.println("name : "+name);
15  out.println("salary : "+salary);
16 <%
17 <br>
18 Id : <b>${id}</b>
19 name: ${name}
20 salary: ${salary}
21 </body>
22 </html>
23
```

### 4. Sending the object data from controller to jsp

```
1 public class Employee {
2
3     private int id;
4     private String name;
5     private double salary;
6
7     public int getId() {
8         return id;
9     }
10
11    @Override
12    public String toString() {
13        return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "]";
14    }
15
16    public void setId(int id) {
17        this.id = id;
18    }
19
20    public String getName() {
21        return name;
22    }
23
24    public void setName(String name) {
25        this.name = name;
26    }
27}
```

```

1
2 @Controller
3 public class ObjectController {
4     @RequestMapping("/readObject")
5     public ModelAndView sendObject() {
6         ModelAndView modelAndView = new ModelAndView();
7         modelAndView.setViewName("displayObject");
8         Employee employee = new Employee();
9         employee.setId(1);
10        employee.setName("saurabh");
11        employee.setSalary(5000000);
12        modelAndView.addObject("employee", employee);
13        return modelAndView;
14    }
15
16 }
17
18
19 <html>
20 <head>
21 <meta charset="ISO-8859-1">
22 <title>Object Details</title>
23 </head>
24 <body>
25     <%=request.getAttribute("employee") %>
26
27     ${employee}
28 </body>
29 </html>

```

//output

http://localhost:8085/springmvc/readObject

Employee [id=1, name=saurabh, salary=5000000.0] Employee [id=1, name=saurabh, salary=5000000]

Create The list controller ( send list of object from controller to ui)

```

1 1 @Controller
2 public class ListController {
3     @RequestMapping("/readList")
4     public ModelAndView sendObject() {
5         ModelAndView modelAndView = new ModelAndView();
6         modelAndView.setViewName("displayList");
7
8         Employee employee = new Employee();
9         employee.setId(1);
10        employee.setName("saurabh");
11        employee.setSalary(500000);
12
13        Employee employee2 = new Employee();
14        employee2.setId(2);
15        employee2.setName("neha");
16        employee2.setSalary(50000);
17
18        ArrayList<Employee> employees = new ArrayList<Employee>();
19
20        employees.add(employee);
21        employees.add(employee2);
22
23        modelAndView.addObject("employees", employees);
24        return modelAndView;
25    }
26
27 }
28
29
30 </head>
31 <body>
32     <%
33     List<Employee> employees = (List<Employee>)request.getAttribute("employees");
34     for(Employee e: employees)
35     {
36         out.println("id = " + e.getId());
37         out.println("name = " + e.getName());
38     }
39
40     %>
41 </body>
42 </html>

```

### Section 18 (sending data from ui to controller)

2 ways we can send data

duction  
18, Lecture 15B

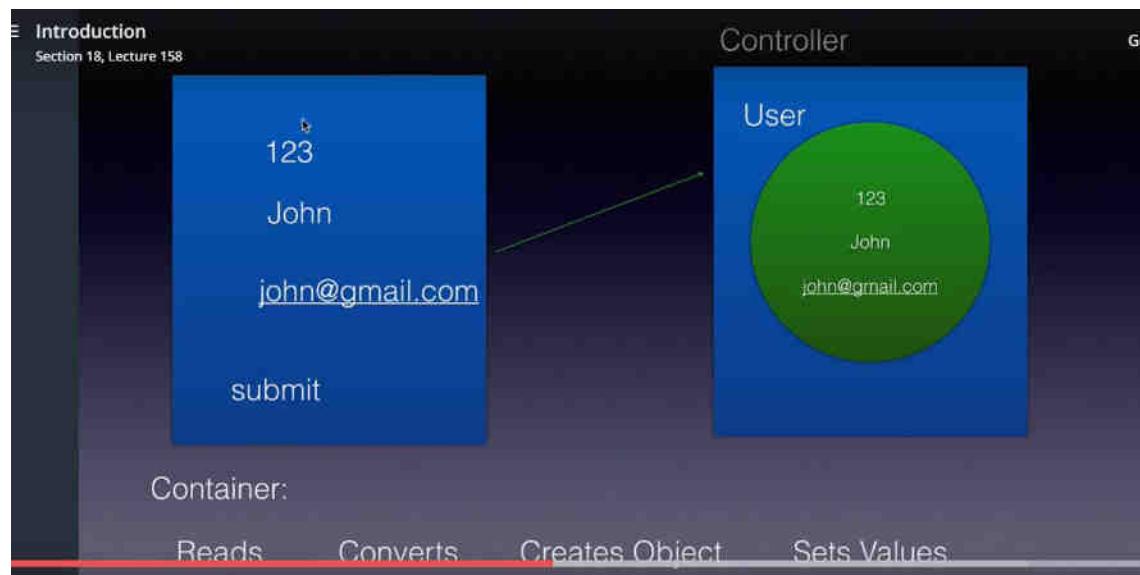
Sending data from UI to Controller:

HTML Form

Query Parameters

Once we submit data spring container do 4 things

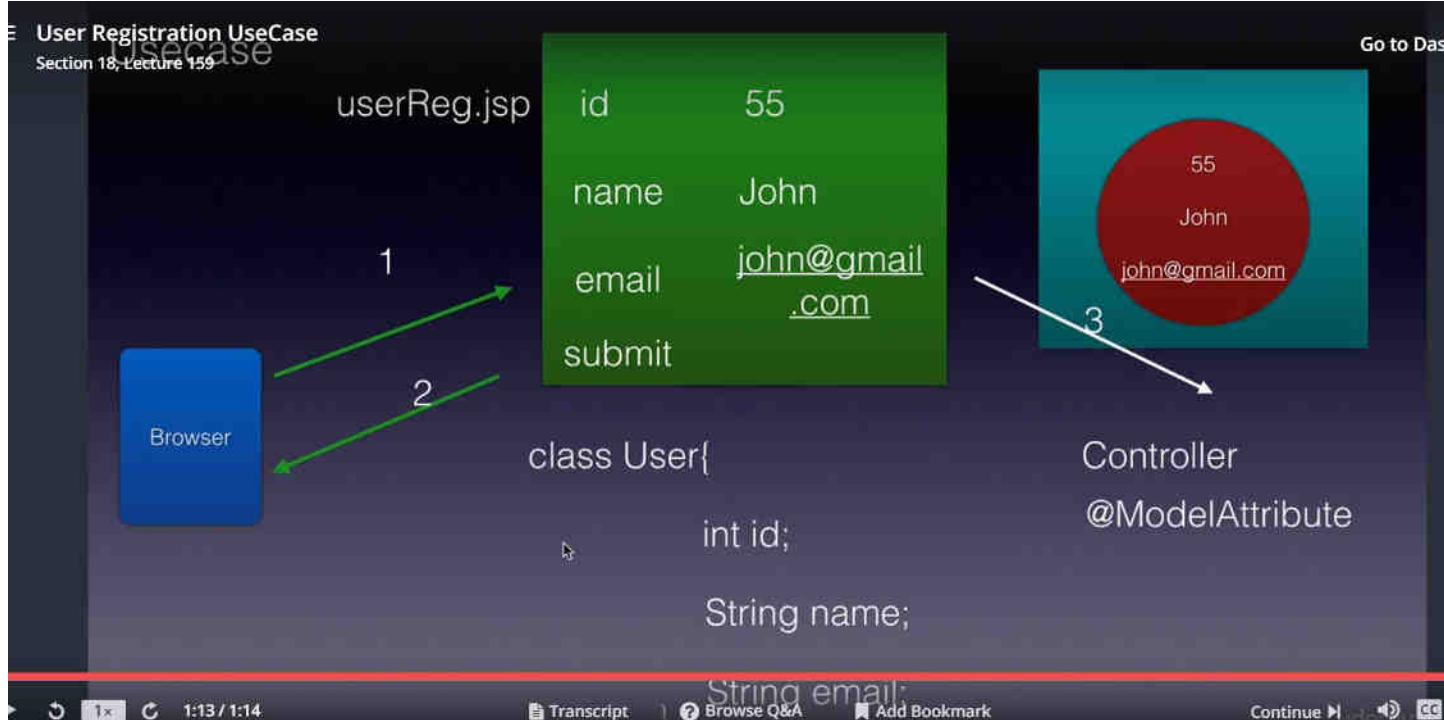
Read data .create object of model class and set values automatically and handover this object to controller



And controller handover this object to ModelAttribute method



### User Registration Use Case



Create the user model and registration view and test it

<http://localhost:8085/springmvc/registrationPage>

a)

```
public class User {
    private int id;
    private String name;
    private String email;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "User [id=" + id + ", name=" + name + ", email=" + email + "]";
    }
}
```

b. userReg.jsp

```
1 <%@ page language="java" contentType="text/html; charset=ISO-88
2     pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>Insert title here</title>
8 </head>
9 <body>
10    <form action="registerUser" method="post">
11        <pre>
12            Id : <input type="text" name="id" /><br>
13            Name : <input type="text" name="name" /><br>
14            Email :<input type="text" name="email" /><br>
15            <input type="submit" name="register" />
16        </pre>
17    </form>
18
19 </body>
20 </html>
```

c)

```
1 @Controller
2 public class UserController {
3
4     @RequestMapping("registrationPage")
5     public ModelAndView showRegistrationPage() {
6         ModelAndView modelAndView = new ModelAndView();
7         modelAndView.setViewName("userReg");
8         return modelAndView;
9     }
0
1     @RequestMapping(value = "registerUser", method = RequestMethod.POST)
2     public ModelAndView registerUser(@ModelAttribute("user") User user) {
3         System.out.println(user);
4         ModelAndView modelAndView = new ModelAndView();
5         modelAndView.addObject("user", user);
6         modelAndView.setViewName("regResult");
7         return modelAndView;
8     }
9 }
0
```

d) regResult.jsp

---

```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2     pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4 <html>
5 <head>
6 <meta charset="ISO-8859-1">
7 <title>user Registration response</title>
8 </head>
9 <body>User Registered successfully. User details are
10 <%= request.getAttribute("user") %>
11 </body>
12 </html>
```

http://localhost:8085/springmvc/registrationPage

**Id :**

**Name :**

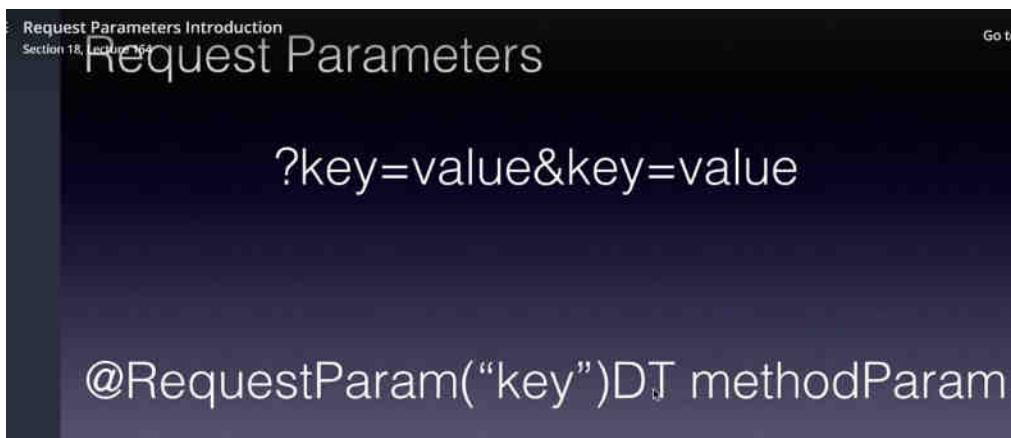
**Email :**

**Submit Query**

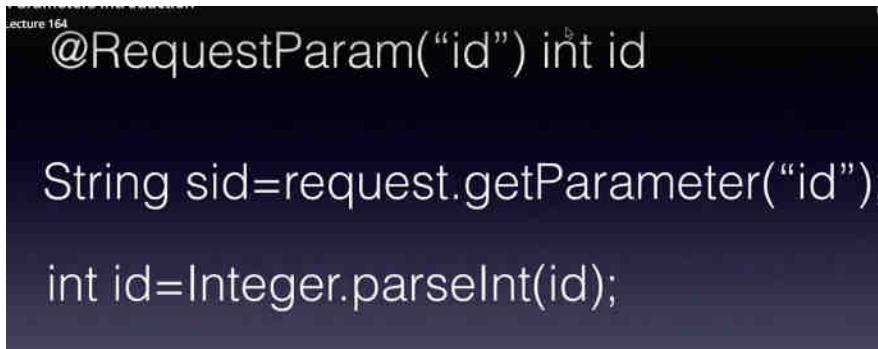
http://localhost:8085/springmvc/registerUser

User Registered successfully. User details are User [id=1, name=saurabh, email=a@gmail]

**Request Parameter Introduction** (Second way of sending data from ui to controller )



Spring will get the value as key and parse it into appropriate value. Spring will do last 2 step



If no value provide by url spring provide default value

```
@RequestParam(value="id",required=false,  
defaultValue="123") int id
```

#### Using @RequestParam annotation

http://localhost:8085/springmvc/showData?id=123&name=saurabh&sal=13000

```
1 @Controller  
2 public class RequestParamsController {  
3  
4     @RequestMapping("/showData")  
5     public ModelAndView showData(@RequestParam("id") int id, @RequestParam("name") String name,  
6                                 @RequestParam("sal") double salary) {  
7         System.out.println("id=" + id);  
8         System.out.println("name=" + name);  
9         System.out.println("salary=" + salary);  
10        return new ModelAndView("userReg");  
11    }  
12}
```

#### Using the required and default value attributes

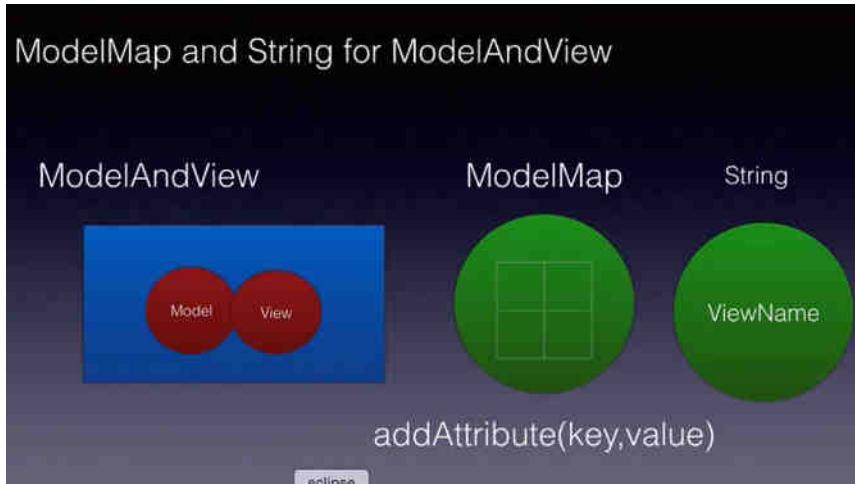
Note: not providing sal in url

http://localhost:8085/springmvc/showData?id=123&name=saurabh

```
1 @Controller  
2 public class RequestParamsController {  
3  
4     @RequestMapping("/showData")  
5     public ModelAndView showData(@RequestParam("id") int id, @RequestParam("name") String name,  
6                                 @RequestParam(value = "sal", required=false, defaultValue="60") double salary) {  
7         System.out.println("id=" + id);  
8         System.out.println("name=" + name);  
9         System.out.println("salary=" + salary);  
10        return new ModelAndView("userReg");  
11    }  
12}
```

section 19: Using modelMap and string view

## ModelMap and String for ModelAndView

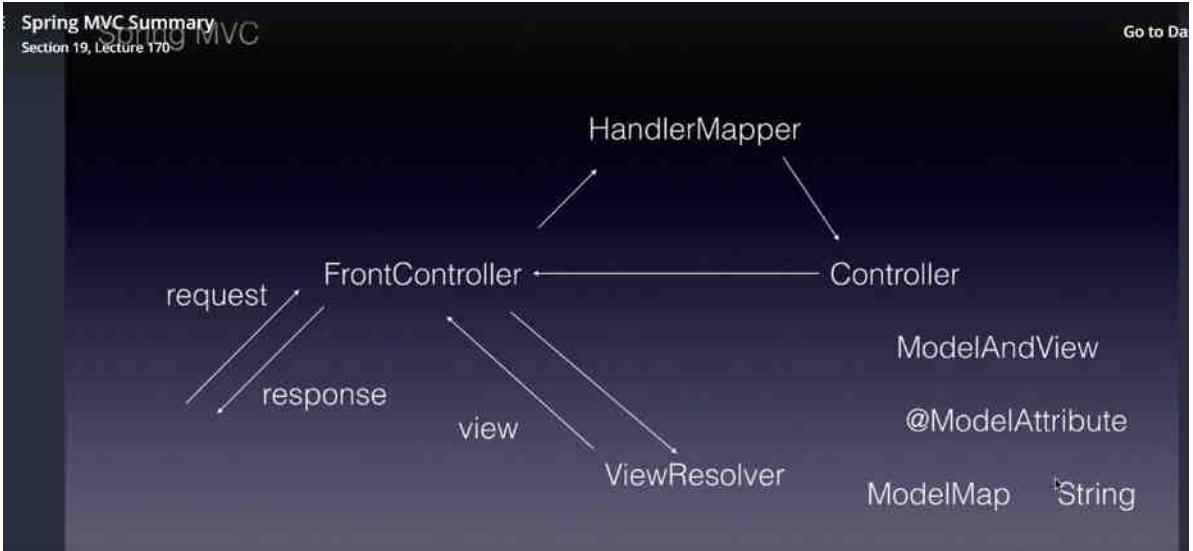


**Note:** if we don't use Model then use ModelMap in place of ModelAndView

We use ModelMap to send data back

userReg.jsp regResult.jsp

```
1  @Controller
2  public class UserController {
3
4      @RequestMapping("registrationPage")
5      public String showRegistrationPage() {
6          return "userReg";
7      }
8
9      @RequestMapping(value = "registerUser", method = RequestMethod.POST)
10     public String registerUser(@ModelAttribute("user") User user, ModelMap model) {
11         System.out.println(user);
12         model.addAttribute("user", user);
13         return "regResult";
14     }
15 }
```



## section 20) spring mvc and orm



a) Create user table in database

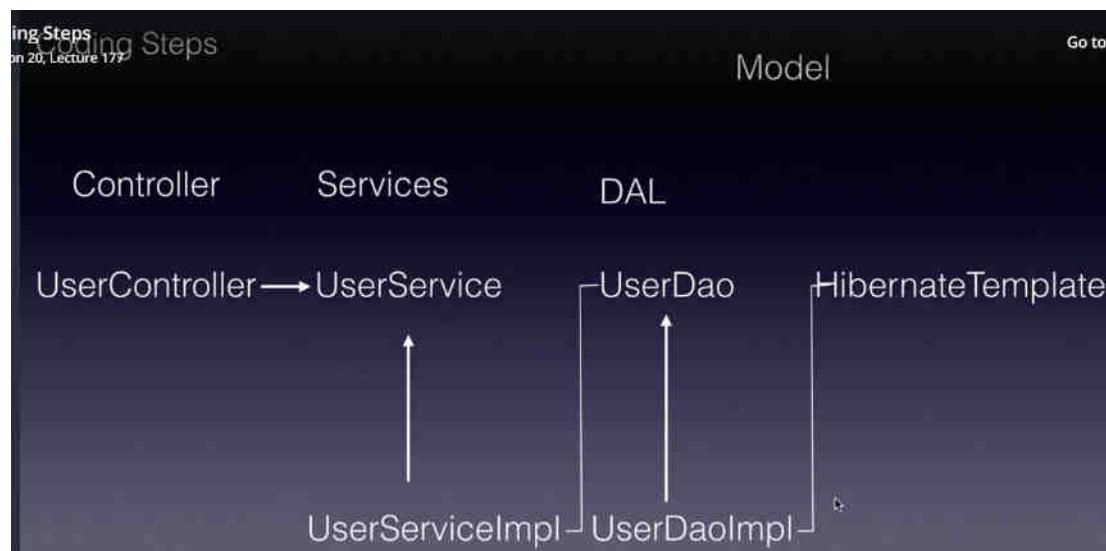
```
use mydb;  
  
create table user(id int,name varchar(20),email varchar(30));  
  
select * from user;
```

Coding step

Layer has a r/s and within the layer is-a R/s

From below structure we are writing class

Note service and interface method name can be different



Note:

It a good practice to write Transactional at service layer

a) web.xml

```
1 |  
2<web-app>  
3     <display-name>Hello Spring MVC ORM</display-name>  
4  
5     <servlet>  
6         <servlet-name>dispatcher</servlet-name>  
7         <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
8     </servlet>  
9     <servlet-mapping>  
0         <servlet-name>dispatcher</servlet-name>  
1         <url-pattern>/</url-pattern>  
2     </servlet-mapping>  
3 </web-app>
```

### b)dispatcher-servlet.xml

```
<context:component-scan base-package="com.bharath.spring.springmvcm.user"/>  
<tx:annotation-driven />  
  
<bean  
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"  
    name="dataSource" p:driverClassName="com.mysql.jdbc.Driver"  
    p:url="jdbc:mysql://localhost/mydb" p:username="root"  
    p:password="test" />  
  
<bean id="sessionFactory"  
    class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"  
    p:dataSource-ref="dataSource">  
    <property name="hibernateProperties">  
        <props>  
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>  
            <prop key="hibernate.show_sql">true</prop>  
        </props>  
    </property>  
    <property name="annotatedClasses">  
        <list>  
            <value>com.bharath.spring.springmvcm.user.entity.User</value>  
        </list>  
    </property>  
</bean>  
  
<bean id="hibernateTemplate"  
    class="org.springframework.orm.hibernate5.HibernateTemplate"  
    p:sessionFactory-ref="sessionFactory">  
</bean>
```

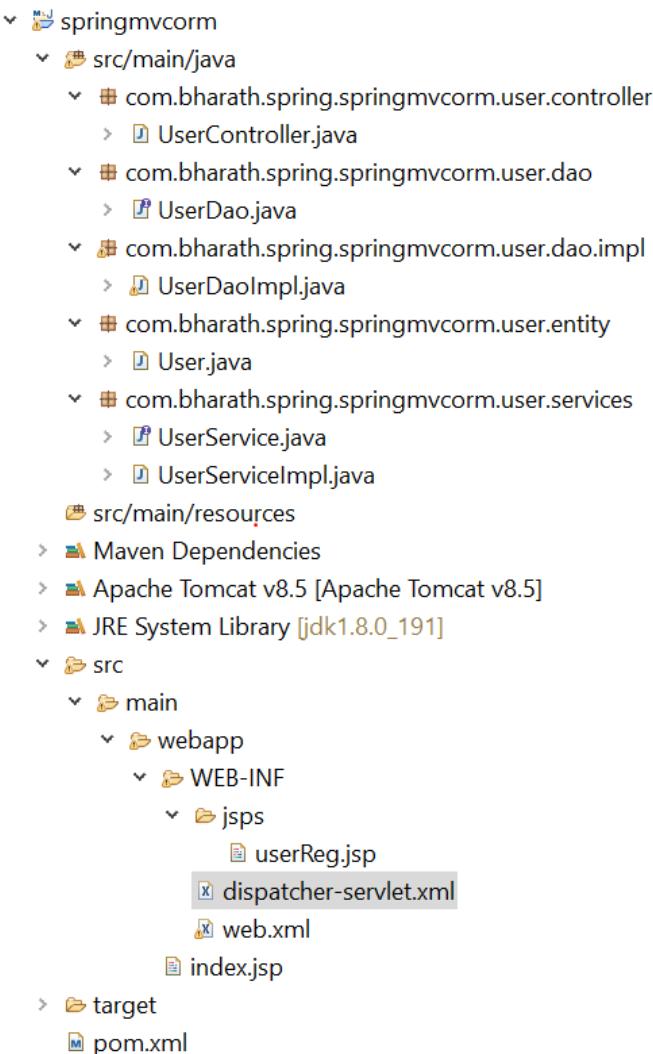
```

<bean id="transactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager"
      p:sessionFactory-ref="sessionFactory">
</bean>

<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
      <value>/WEB-INF/jsp/</value>
    </property>
    <property name="suffix">
      <value>.jsp</value>
    </property>
</bean>

```

c)



d) User.java

```
5  
6 @Entity  
7 @Table(name = "user")  
8 public class User {  
9  
10     @Override  
11     public String toString() {  
12         return "User [id=" + id + ", name=" + name + ", email=" + email + "]";  
13     }  
14  
15     @Id  
16     private int id;  
17     private String name;  
18     private String email;  
19  
20     public int getId() {  
21         return id;  
22     }  
23  
24     public void setId(int id) {  
25         this.id = id;  
26     }  
27  
28     public String getName() {  
29         return name;  
30     }  
31  
32
```

e. UserDao and UserDaoImpl

```
4  
5 public interface UserDao {  
6  
7     int create(User user);  
8 }  
9 |
```

```
1
2 @Repository
3 public class UserDaoImpl implements UserDao {
4
5     @Autowired
6     private HibernateTemplate hibernateTemplate;
7
8     public HibernateTemplate getHibernateTemplate() {
9         return hibernateTemplate;
10    }
11
12    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
13        this.hibernateTemplate = hibernateTemplate;
14    }
15
16    @Override
17    public int create(User user) {
18        Integer result = (Integer) hibernateTemplate.save(user);
19        return result;
20    }
21
22 }
```

f. UserService and UserServiceImpl

```
public interface UserService {
    int save(User user);
}
```

```
1  @Service
2  public class UserServiceImpl implements UserService {
3
4      @Autowired
5      private UserDao userDao;
6
7      public UserDao getUserDao() {
8          return userDao;
9      }
10
11     public void setUserDao(UserDao userDao) {
12         this.userDao = userDao;
13     }
14
15     @Override
16     @Transactional
17     public int save(User user) {
18         // Business logic
19         return userDao.create(user);
20     }
21 }
```

g)UserController

```
12
13 @Controller
14 public class UserController {
15
16     @Autowired
17     private UserService service;
18
19     @RequestMapping("registrationPage")
20     public String showRegistrationPage() {
21         return "userReg";
22     }
23
24     @RequestMapping(value = "registerUser", method = RequestMethod.POST)
25     public String registerUser(@ModelAttribute("user") User user, ModelMap model) {
26         int result = service.save(user);
27         model.addAttribute("result", "user created with id" + result);
28         return "userReg";
29     }
30
31     public UserService getService() {
32         return service;
33     }
34
35     public void setService(UserService service) {
36         this.service = service;
37     }
38
39 }
40
```

## h)userReg.jsp

```
<body>
    <form action="registerUser" method="post">
        <pre>
            Id : <input type="text" name="id" /><br>
            Name : <input type="text" name="name" /><br>
            Email :<input type="text" name="email" /><br>
            <input type="submit" name="register" />
        </pre>
    </form>
    <br>
    ${result}
</body>
</html>
```

Run application

The screenshot shows a web browser window with the URL <http://localhost:8085/springmvccorm/registrationPage>. The page contains a registration form with the following fields:

- Id :
- Name :
- Email :

Below the form is a **Submit Query** button.

**Result will return as id**

**Implement the Load User Method in Dao and Service**

**a)dao (UserDao and UserDaoImpl)**

```
import com.bharath.spring.springmvccorm.user.entity.User;

public interface UserDao {
    int create(User user);
    List<User> findUsers();
}

@Override
public List<User> findUsers() {
    return hibernateTemplate.loadAll(User.class);
}
```

**b. UserService and UserServiceImpl**

```
public interface UserService {
    int save(User user);
    List<User> getUsers();
}
```

```
    @Override
    public List<User> getUsers() {
        return userDao.findUsers();
    }
}
```

c. UserController

```
@RequestMapping("getUsers")
public String getUser(ModelMap model) {
    List<User> users = service.getUsers();
    model.addAttribute("users", users);
    return "displayUsers";
}
```

d. displayUsers.jsp

```
<%@ page isELIgnored="false"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
    <table border="1">
        <tr>
            <th>Id</th>
            <th>Name</th>
            <th>Email</th>
        </tr>
        <c:forEach items="${users}" var="user">
            <tr>
                <td>${user.id}</td>
                <td>${user.name}</td>
                <td>${user.email}</td>
            </tr>
        </c:forEach>
    </table>
</body>
```

**Output**



| Id | Name    | Email       |
|----|---------|-------------|
| 1  | saurabh | a@gmail.com |

Sort by id

User.java

```
public void setEmail(String email) {  
    this.email = email;  
}  
  
@Override  
public int compareTo(User user) {  
    return this.id.compareTo(user.id);  
}
```

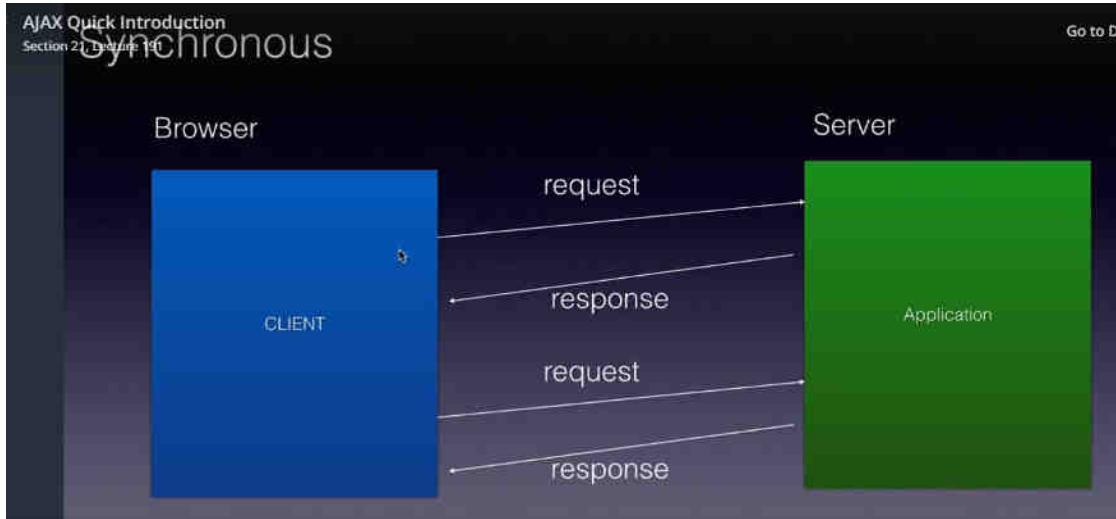
```
UserServiceImpl.java | UserServiceImpl.java | springmvccorm/po | web.xml | springmvccorm/po | displayUsers.js  
13 @Service  
14 public class UserServiceImpl implements UserService {  
15  
16     @Autowired  
17     private UserDao dao;  
18  
19     public UserDao getDao() {  
20         return dao;  
21     }  
22  
23     public void setDao(UserDao dao) {  
24         this.dao = dao;  
25     }  
26  
27     @Override  
28     @Transactional  
29     public int save(User user) {  
30         // Business Logic  
31         return dao.create(user);  
32     }  
33  
34     @Override  
35     public List<User> getUsers() {  
36         List<User> users = dao.findUsers();  
37         Collections.sort(users);  
38         return users;  
39     }  
40 }  
41
```

## section 21 ( Ajax quick introduction)

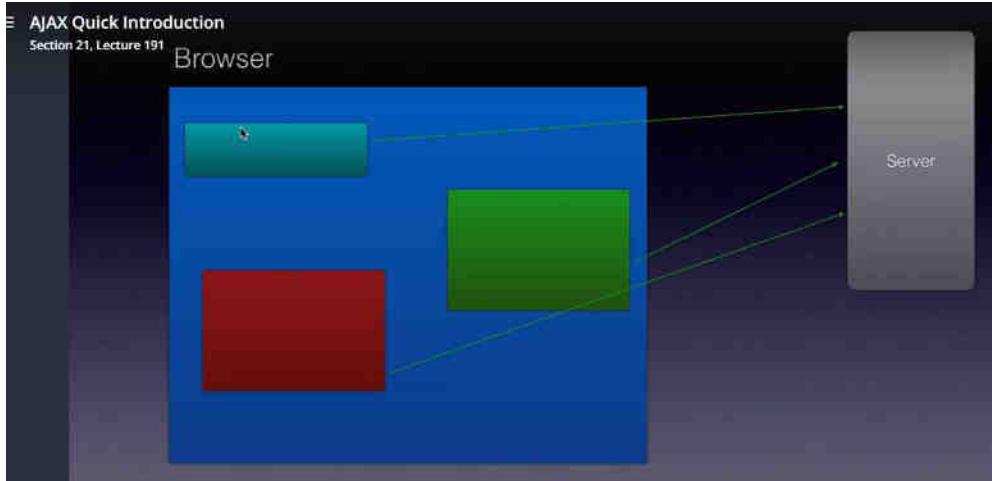
Aysnchronous java script and xml

Request and response everything is in sync.

End user need to wait when response come the only he can do some action



Using Ajax multiple request can be send to server at same time



## Jquery quick introduction

It is layer over java script and hide lot of things of javascript code and start with \$.

It internally uses java script

Everything in jquery is a function

JQUERY Quick Introduction  
Section 21, Lecture 192

# JQuery

# Java Script

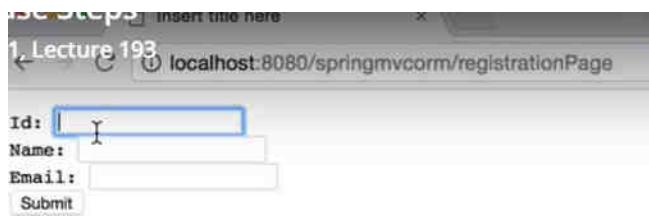
```
$  
  
document.getElementById("userId")  
$("#userId")
```

Url of server data send as request to server and and callback function (success function) invoke when response come back

```
$.ajax({  
    url:"",  
    data:{key:value},  
    success:function(){}
})
```

### User case steps

Id doesn't exist in database. If it exist display error message



The screenshot shows a web browser window with the title "Lecture 193". The address bar shows the URL "localhost:8080/springmvccorm/registrationPage". The page content is a form for user registration. It has three input fields: "Id" (with value "1"), "Name" (with value "Amit"), and "Email" (empty). Below the fields is a "Submit" button.

2 step

a)

## UserCase Steps

Implement the backend validation

Controller ————— Service ————— DAO

b)

Make the AJAX Call

Use JQuery

onChange

AJAX Call

Controller

Handle Response

**Note:** from controller if we want to let know spring don't search for jsp page as return then will use @ResponseBody

@ResponseBody: don't serach for jsp page just return as response to view in response body

```
@RequestMapping("validateEmail")
public @ResponseBody String validateEmail(@RequestParam("id") int id) {
```

**Backened code:-**(consider example as springorm)

- UserDao and UserDaoImpl

```
3* import java.util.List;
4
5 public interface UserDao {
6     int create(User user);
7     List<User> findUsers();
8     User findUser(Integer id);}
```

To get single record give entity class and id

```
1 @Override  
2 public User findUser(Integer id) {  
3     return hibernateTemplate.get(User.class, id);  
4 }
```

b) UserService and UserServiceImpl

```
1 public interface UserService {  
2     int save(User user);  
3     List<User> getUsers();  
4     User getUser(Integer id);  
5 }  
  
-  
  
1 @Override  
2 public User getUser(Integer id) {  
3     return userDao.findUser(id);  
4 }
```

c. UserController

```
1 @Controller  
2 public class UserController {  
3     @Autowired  
4     private UserService service;  
5     @RequestMapping("registrationPage")  
6     public String showRegistrationPage() {  
7         return "userReg";  
8     }
```

```
@RequestMapping("validateEmail")
public String validateEmail(@RequestParam("id") int id) {
    User user = service.getUser(id);
    String msg = "";
    if (user != null) {
        msg = id + " already exist";
    }
    return msg;
}
```

#### front end development

Want to do ajax call when user move from id

userReg

```

10<script>
11    src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>
12<script>
13 $(document).ready(function() {
14     $("#id").change(function() {
15
16         $.ajax({
17
18             url : 'validateEmail',
19             data : {
20                 id : $("#id").value()
21             },
22             success : function(responseText) {
23                 $("#errMsg").text(responseText);
24
25                 if(responseText!=""){
26                     $("#id").focus();
27                 }
28             }
29         });
30     });
31 });
32 });
33 });
34 </script>
35
36 </head>
37 <body>
38     <form action="registerUser" method="post">
39         <pre>
40 Id: <input type="text" name="id" id="id" /><span id="errMsg"></span>
41 Name: <input type="text" name="name" />
42 Email: <input type="text" name="email" />
43 <input type="submit" name="register" />
44 </pre>
45 </form>

```

//output

http://localhost:8085/springmvccorm/registrationPage

|   |                                |                                  |                 |
|---|--------------------------------|----------------------------------|-----------------|
| Id:   | <input type="text" value="1"/> | <input type="button" value="X"/> | 1 already exist |
| Name:                                       | <input type="text"/>           |                                  |                 |
| Email:                                      | <input type="text"/>           |                                  |                 |
| <input type="button" value="Submit Query"/> |                                |                                  |                 |

## AJAX AND JQuery

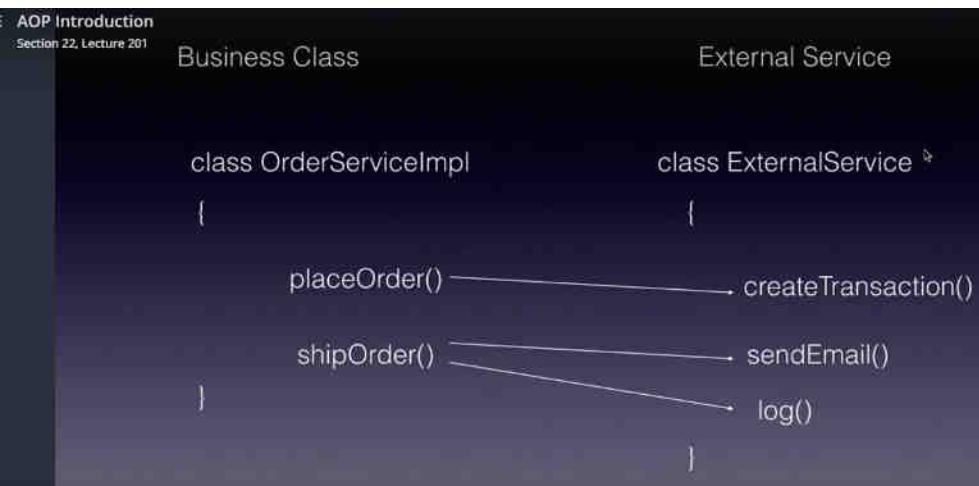
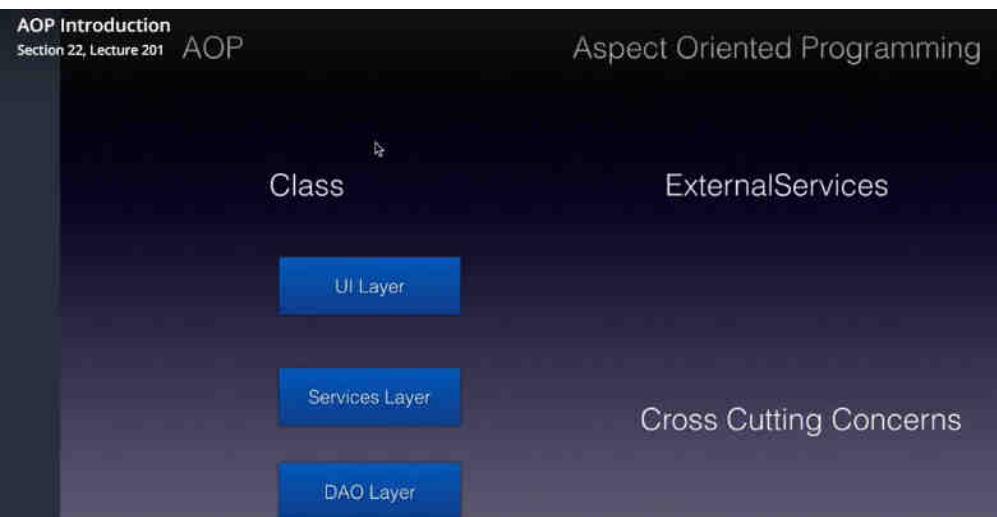
Section 21, Lecture 201

Responsive

JS Abstraction

```
$ajax(  
    by url:  
    data:  
    success:
```

## section 22- aspect oriented programming AOP



## AOP Terminology

**Aspect:** it is a class that represent external services

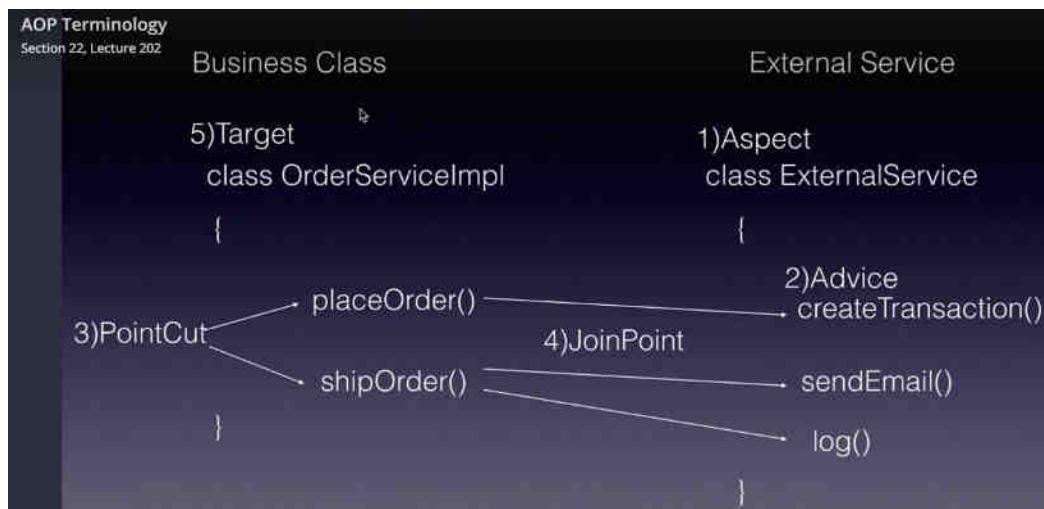
**Advice:** it is method define inside aspect class that needs to be apply business method

**Pointcut :** is an expression that tells which business method in our application need advices

**JoinPoint** is a combination of advice and pointcut. It will tell which business method need which advice.

Note: using joinpoint we can access business method parameter

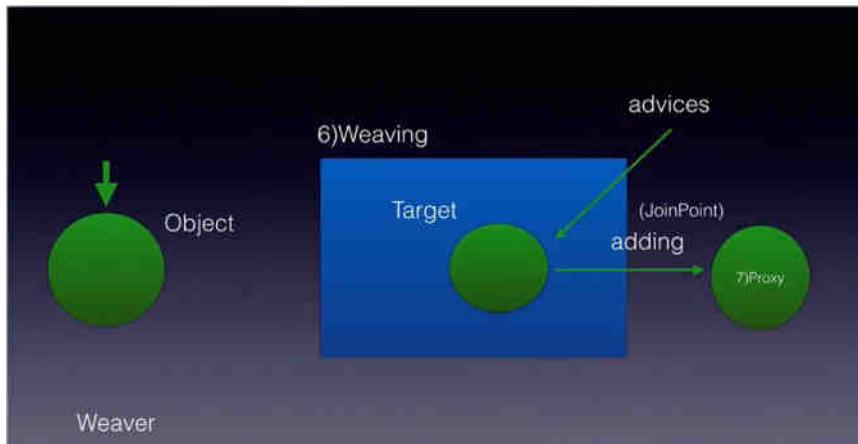
**Target** is an object of business class to which advice need to apply



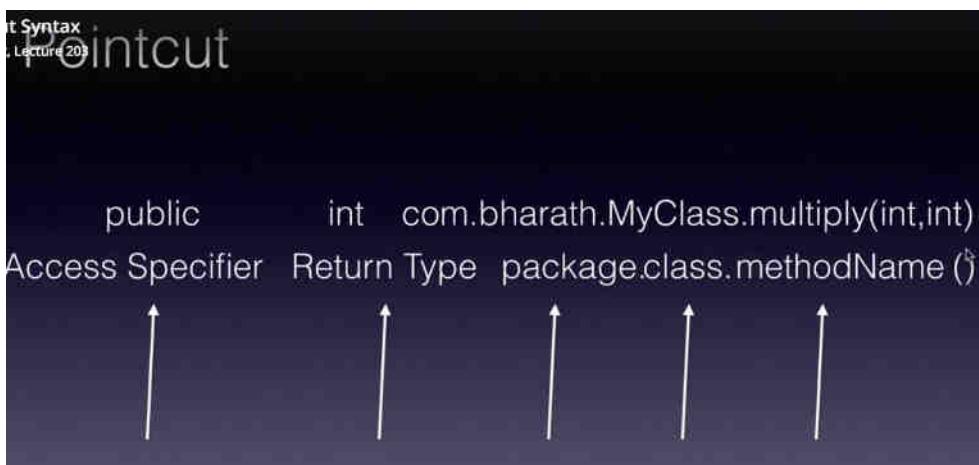
**Weaving** is a process of mixing advice to target object based on joinpoint because joinpoint tell which advice will use for target method

**Proxy:** it is a class that generated as a result of the weaving process. It contain business logic method as well as advice method

And weaving is done by special component weaver



### Pointcut syntax



### Important

Star and

**dot dot** : any number and any types of parameter

| symbols | can be used at                               |
|---------|--|
| *       | AS,RT,PACK,CLASS,MN                          |
| ..      | pack,current and sub (1)<br>Any Parameter(2) |

### Example of pointcut expression

```
public void *Id()
```

```
public int *e*(..)
```

```
public int get(..)
```

```
public * *()
```

```
public void get(..)
```

```
public int *(..)
```

Method name start with get end with anything. It should fall in any class. All the class fall under this package and subpackage. Return type anything

```
public * com.app..*.get*()
```

```
public * *(..)
```

### AOP frameworks



**Third one is outdated**

## Spring AOP

AspectJ Annotation Driven

AspectJ XML Driven

Classic Spring Proxy-Based AOP

### Aspect j annotation driven

**Note:** in AfterReturning we can use value

## AspectJ Annotation Driven

@Aspect

    @Before

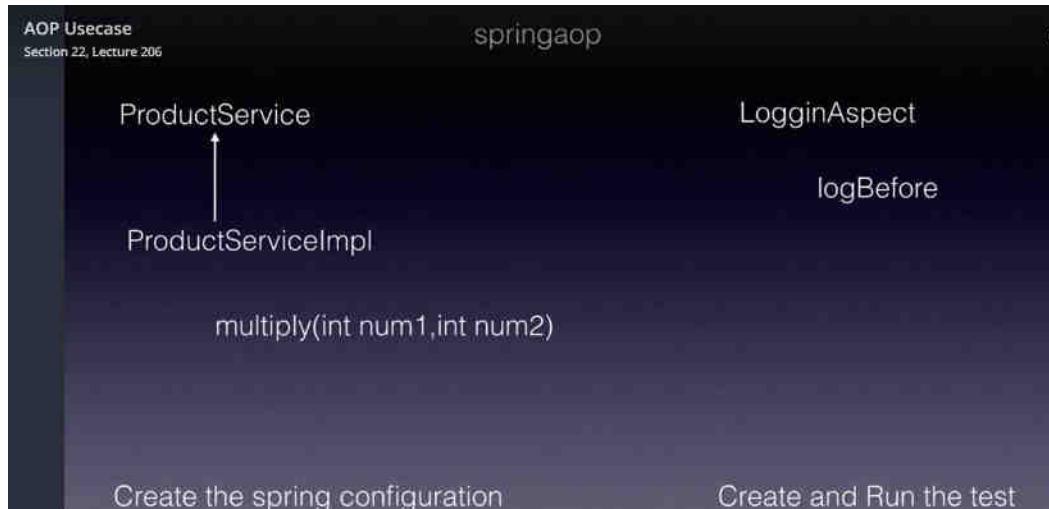
    @After

    @AfterReturning

    @Around

    @AfterThrowing

### AOP usecase



springaop

- src/main/java
  - com.bharath.spring.springaop
    - ProductService.java
    - ProductServiceImpl.java
  - com.bharath.spring.springaop.aspects
    - LoggingAspects.java
  - com.bharath.spring.springaop.test
    - Test.java
- config.xml

```

public interface ProductService {
    .
    int multiply(int num1 , int num2);
}

public class ProductServiceImpl implements ProductService {
    @Override
    public int multiply(int num1, int num2) {
        return num1 * num2;
    }
}
  
```

```

@Aspect
public class LoggingAspects {

    @Before("execution(* com.bharath.spring.springaop.ProductServiceImpl.multiply(..))")
    public void logBefore(JoinPoint joinPoint) {
        System.out.println("before calling the method");
    }

    @After("execution(* com.bharath.spring.springaop.ProductServiceImpl.multiply(..))")
    public void logAfter(JoinPoint joinPoint) {
        System.out.println("After the method invocation");
    }
}

<aop:aspectj-autoproxy />

<bean id="productService"
      class="com.bharath.spring.springaop.ProductServiceImpl" />

<bean id="LoggingAspect"
      class="com.bharath.spring.springaop.aspects.LoggingAspects" />

</beans>

public class Test {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("com/bharath/spring/springaop/test/config.xml");
        ProductService productService = (ProductService) context.getBean("productService");
        System.out.println(productService.multiply(2, 2));
    }
}

//output

```

**before calling the method  
After the method invocation  
4**

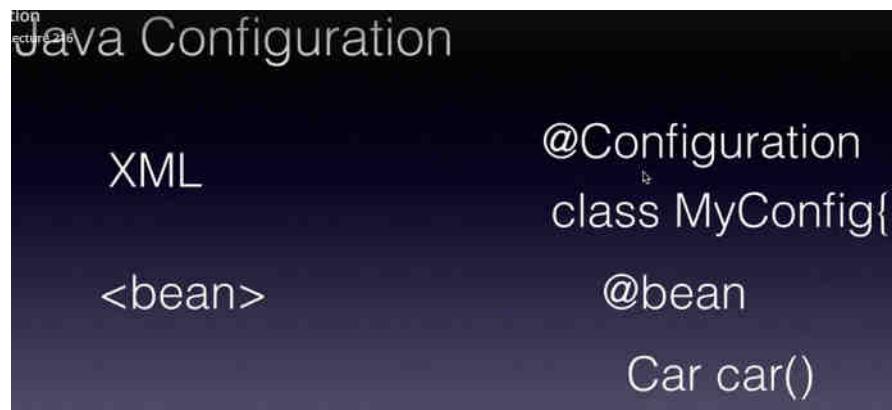
### Section 23: java configuration

Annotating a class with the **@Configuration** indicates that the class can be used by the **Spring** IoC container as a source of bean definitions. The **@Bean** annotation tells **Spring** that a method annotated with **@Bean** will return an object that should be registered as a bean in the **Spring** application context.

**@Configuration** : once the class is marked with this annotation it tell that this class is source of spring bean  
(like xml file)

@bean similar to <bean> used in xml

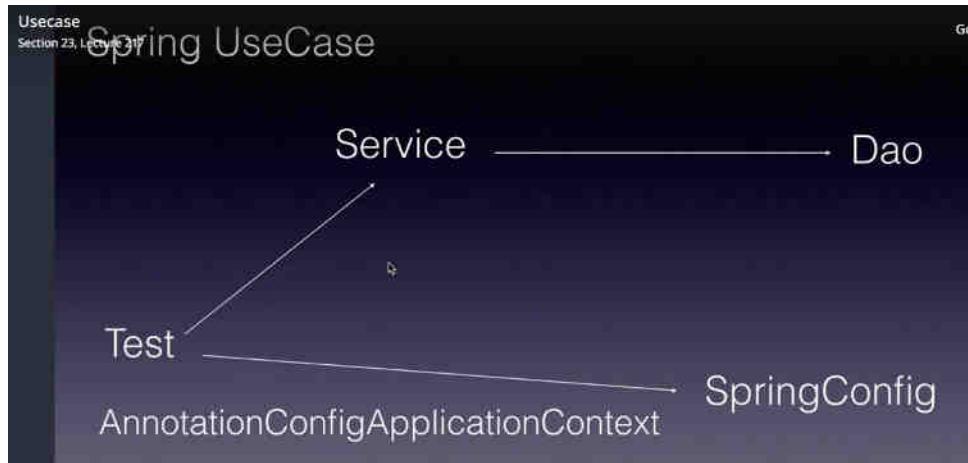
@bean return car object



In Java configuration in place of ClassPathXmlApplicationContext use given below



**UseCase**



a)

```

@Configuration
public class SpringConfig {

    @Bean
    public Dao dao() {
        return new Dao();
    }

    @Bean
    public Service service() {
        return new Service();
    }
}
  
```

```

1  @Component
2  public class Dao {
3
4      public void create() {
5          System.out.println("created");
6      }
7  }
8
9
10 public class Service {
11
12     @Autowired
13     Dao dao;
14
15     public void save() {
16         dao.create();
17     }
18 }
19
20
21 public class Test {
22
23     public static void main(String[] args) {
24         ApplicationContext applicationContext = new AnnotationConfigApplicationContext(SpringConfig.class);
25         Service service = applicationContext.getBean(Service.class);
26         service.save();
27     }
28 }
```

Note:-

We can make separate Configuration file and import it also

```

5
6  @Configuration
7  public class DaoConfig {
8
9      @Bean
10     public Dao dao() {
11         return new Dao();
12     }
13
14 }
```

```
@Configuration  
@Import(DaoConfig.class)  
public class SpringConfig {  
  
    @Bean  
    public Service service() {  
        return new Service();  
    }  
}
```

#### LifeCycle callback

```
@Configuration  
@Import(DaoConfig.class)  
public class SpringConfig {  
  
    @Bean(initMethod="init" ,destroyMethod="destroy")  
    public Service service() {  
        return new Service();  
    }  
}
```

```
public class Service {  
  
    @Autowired  
    Dao dao;  
  
    public void init() {  
        System.out.println("init");  
    }  
  
    public void destroy() {  
        System.out.println("destroy");  
    }  
  
    public void save() {  
        dao.create();  
    }  
}
```

```
5
5 public class Test {
6
7     public static void main(String[] args) {
8         AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext(SpringConfig.class);
9         Service service = applicationContext.getBean(Service.class);
10        service.save();
11        applicationContext.close();
12    }
13}
14
```

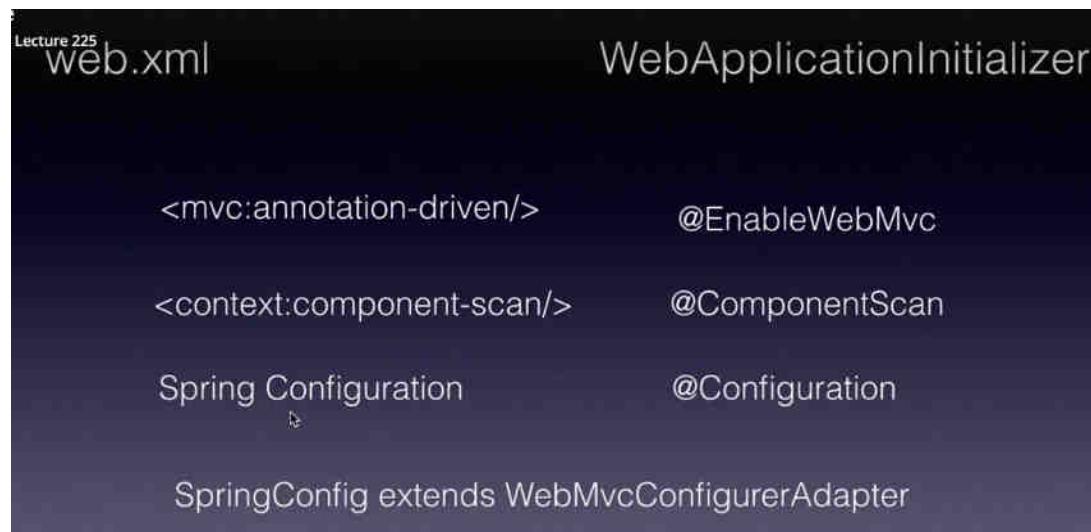
//output

```
init
created
Dec 25, 2018
INFO: Closing
destroy
```

## Section 24: java configuration for web application

### Use case

- 1) web.xml can be equivalent to a class extends WebApplicationInitializer
- 2) dispatcher-servlet.xml equivalent to a class extend to WebMvcConfigurerAdapter and marked this class With @EnableWebMvc



### Migration step:(3 step required)

Need to configure war plugin in pom.xml which tell maven our project don't have web.xml by default which expect it



Pom.xml

Added plugin

```
</plugins>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.4</version>
    <configuration>
        <warSourceDirectory>src/main/webapp</warSourceDirectory>
        <warName>springmvc</warName>
        <failOnMissingWebXml>false</failOnMissingWebXml>
    </configuration>
</plugin>
</plugins>
```

SpringConfig.java

```
1  @EnableWebMvc
2  @ComponentScan("com.bharat.spring.springmvc.controller")
3  @Configuration
4  public class SpringConfig extends WebMvcConfigurerAdapter {
5
6      @Bean
7      public ViewResolver viewResolver() {
8          InternalResourceViewResolver resolver = new InternalResourceViewResolver();
9          resolver.setPrefix("/WEB-INF/views/");
10         resolver.setSuffix(".jsp");
11         return resolver;
12     }
13
14     @Override
15     public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) {
16         configurer.enable();
17     }
18 }
```

## Section 25 : Spring boot

**Opinionated default:** no need of xml(dispatcher servlet) and view resolver but if we want we can write it



One annotation equivalent to 3 annotation



### Spring boot starter project

These configuration will pull all the required jar



### How does spring boot works

Spring have already have configuration( java configuration) need to enabled based on certain condition

**Note:** based on what it found on classpath spring enable it by looking into meta-inf/spring.factories  
Behind the scene

How does Spring Boot work?  
Section 25, Lecture 237

## Starter POMs

Add Jars

META-INF/spring.factories

@Condition

@Configuration

HibernateJpaAutoConfiguration

### Different ways to create a spring boot project( 4 ways)

Different ways to create a Spring Boot Project  
Section 25, Lecture 238

Create a maven project and adder the starter dependencies

User the Spring Initializer

Using and IDE

Using Spring Boot CLI

### Create a spring boot application using spring initializer

<https://start.spring.io/> (to create project)

Created a standalone project

Generate a  with  and Spring Boot

## Project Metadata

Artifact coordinates

Group

com.bharath.spring.boot

Artifact

springboot

## Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

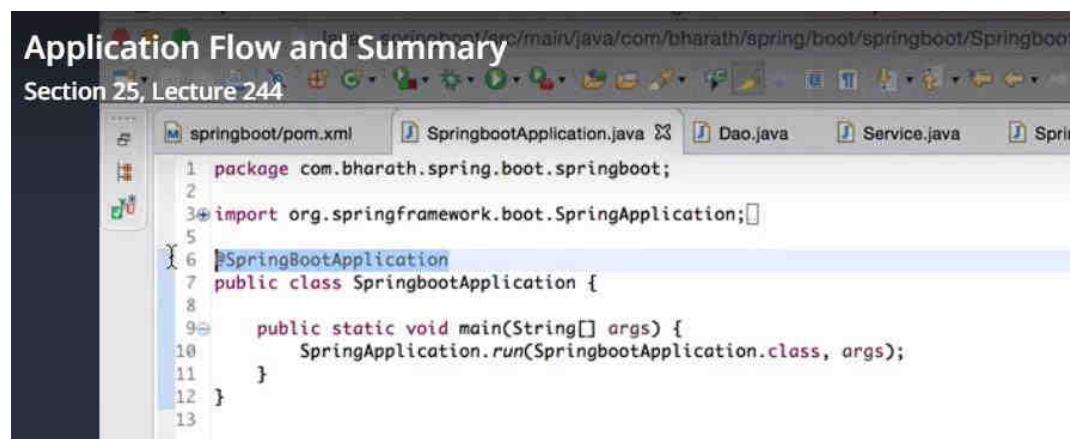
Don't know what to look for? [Open](#) [e full version.](#)

## Import the project into eclipse

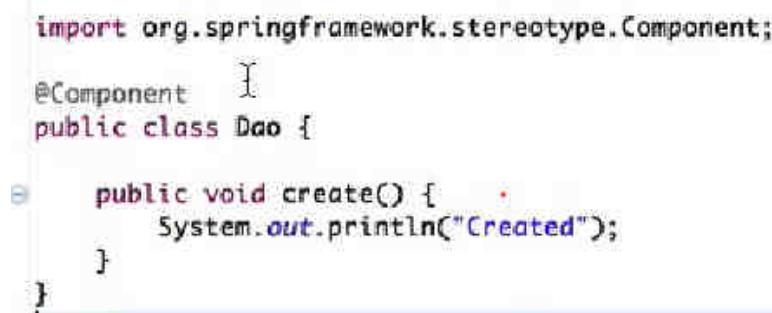
@SpringBootTest will search for @SpringBootApplication

Starting point of spring application which has main method run by container

Application Flow and Summary  
Section 25, Lecture 244



```
1 package com.bharath.spring.boot.springboot;
2
3 import org.springframework.boot.SpringApplication;
4
5 @SpringBootApplication
6 public class SpringbootApplication {
7
8     public static void main(String[] args) {
9         SpringApplication.run(SpringbootApplication.class, args);
10    }
11 }
12
13
```



```
import org.springframework.stereotype.Component;

@Component
public class Dao {

    public void create() {
        System.out.println("Created");
    }
}
```

```

import org.springframework.beans.factory.annotation.Autowired;

@Component
public class Service {

    Dao dao;

    @Autowired
    Service(Dao dao) {
        System.out.println("Service Bean Created");
        this.dao = dao;
    }

    public void save() {
        dao.create();
    }
}

RunWith(SpringRunner.class)
@SpringBootTest
public class SpringbootApplicationTests {

    @Autowired
    ApplicationContext context;

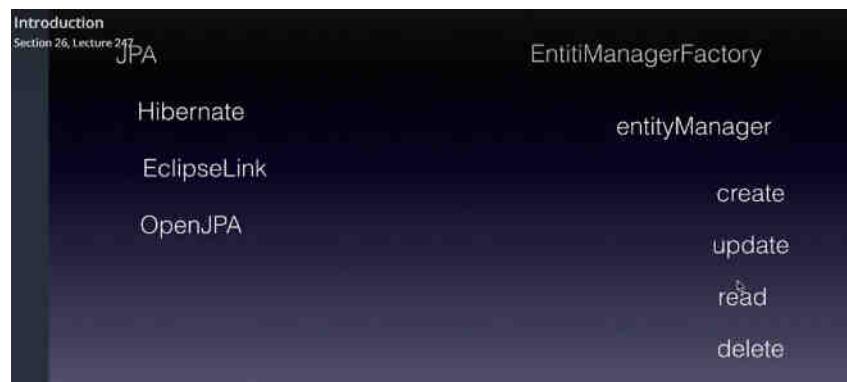
    @Test
    public void testService() {
        Service service = context.getBean(Service.class);
        service.save();
    }
}

```

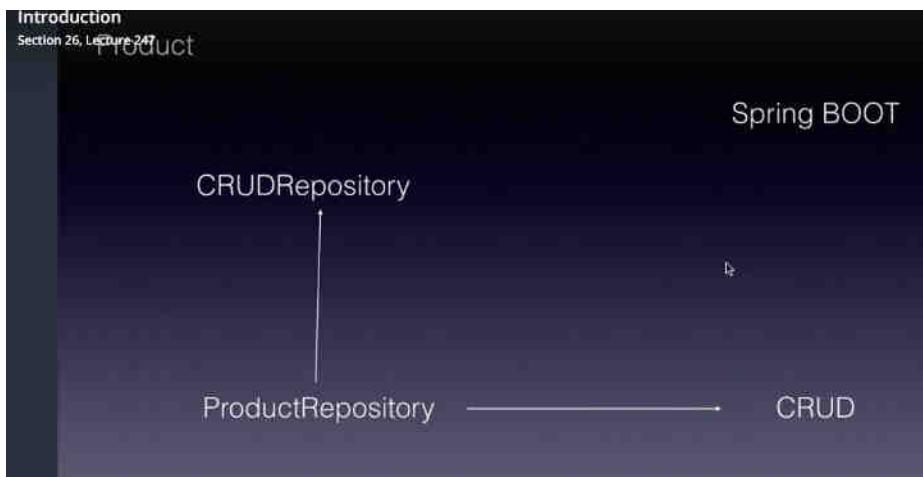
## Section 26: spring data jpa using Spring boot

**Introduction:-**

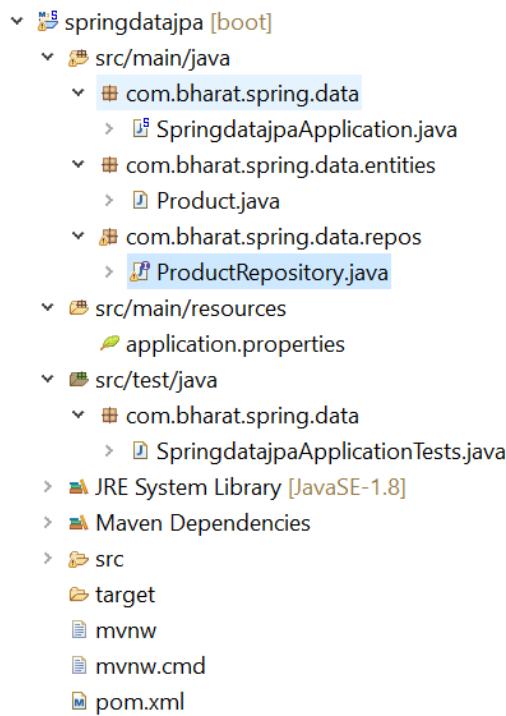
**JPA: java persistence API**



**Just need to extend CRUDRepository to perform crud operation**



### Create the spring data jpa project



A screenshot of the 'application.properties' file in a code editor, showing the following configuration:

```

spring.datasource.name=mydatabase
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=test

```

```
5
6 @Entity
7 public class Product {
8
9     @Id
10    private Long id;
11    private String name;
12    private String description;
13    private double price;
14
15    @Override
16    public String toString() {
17        return "Product [id=" + id + ", name=" + name + ", description=" + description + ", price=" + price + "]";
18    }
19
20    public Long getId() {
21        return id;
22    }
23
24    public void setId(int id) {
25
26        import java.util.Optional;
27
28    public interface ProductRepository extends CrudRepository<Product, Long> {
29
30
31    }
```

```
.2
.3 @RunWith(SpringRunner.class)
.4 @SpringBootTest
.5 public class SpringdatajpaApplicationTests {
.6
.7     @Autowired
.8     ApplicationContext context;
.9
.10    @Test
.11    public void saveProduct() {
.12
.13        ProductRepository repository = context.getBean(ProductRepository.class);
.14
.15        Product product = new Product();
.16        product.setId(1);
.17        product.setName("apple");
.18        product.setDescription("awesome");
.19        product.setPrice(2000d);
.20        repository.save(product);
.21    }
.22    // read a product
.23    System.out.println(repository.findById(1L));
.24
.25    // update a product
.26    product.setPrice(4000d);
.27    repository.save(product);
.28    repository.findAll().forEach(p -> {
.29        System.out.println(p.getPrice());
.30    });
.31 }
.32 }
```

### Section 27: JPA Entity

| Product |             |
|---------|-------------|
| name    | findByName  |
| price   | findByPrice |

**Implement custom finders without writing any sql**

Consider as above example

```
.0
.1 public interface ProductRepository extends CrudRepository<Product, Long> {
.2
.3     List<Product> findByName(String name);
.4
.5 }
.6

.
.
.

    @RunWith(SpringRunner.class)
    @SpringBootTest
    public class SpringdatajpaApplicationTests {

        @Autowired
        ApplicationContext context;

        @Test
        public void saveProduct() {

            ProductRepository repository = context.getBean(ProductRepository.class);

            Product product = new Product();
            product.setId(1);
            product.setName("apple");
            product.setDescription("awesome");
            product.setPrice(2000d);

            System.out.println(repository.findByName("apple"));

            /*repository.save(product);
```

**Find by multiple fields**

```
public interface ProductRepository extends CrudRepository<Product, Long> {

    List<Product> findByName(String name);

    List<Product> findByNameAndPrice(String name, Double Price);

}
```

```

2
3 @RunWith(SpringRunner.class)
4 @SpringBootTest
5 public class SpringdatajpaApplicationTests {
6
7     @Autowired
8     ApplicationContext context;
9
10    @Test
11    public void saveProduct() {
12
13         ProductRepository repository = context.getBean(ProductRepository.class);
14
15         Product product = new Product();
16         product.setId(1);
17         product.setName("apple");
18         product.setDescription("awesome");
19         product.setPrice(2000d);
20
21         System.out.println(repository.findByName("apple"));
22         System.out.println(repository.findByNameAndPrice("apple", 4000d));
23
24         /*repository.save(product);*/

```

## Section 28- spring boot web

### Introduction

1)No need to configure DispatcherServlet and InternalViewResolver



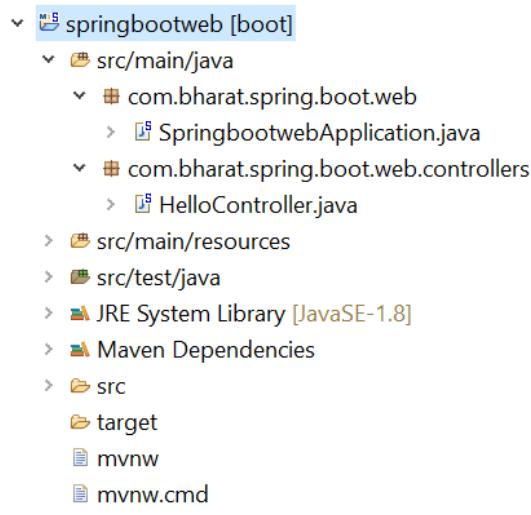
If we want to customize InternalViewResolver properties then we can use application.properties

# Dispatcher Servlet

## Internal View Resolver

### application.properties

```
spring.mvc.view.prefix= # Spring MVC view prefix.  
spring.mvc.view.suffix= # Spring MVC view suffix.
```



```
@SpringBootApplication  
public class SpringbootwebApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(SpringbootwebApplication.class, args);  
    }  
}
```

```
7
3 @Controller
3 public class HelloController {
3
3
L
2@     @RequestMapping("/hello")
3     @ResponseBody
4     public String hello(@RequestParam String name) {
5         return "hello " + name;
5
5
7
3 }
```

//output

By default no root context is there

← → ⌂ ⓘ localhost:8080/hello?name=saurabh

hello saurabh

#### Configuring the application Context path

☰ HelloController.java SpringbootwebApplication.java application.properties

```
1 server.servlet.context-path=/springbootweb
```

← → ⌂ ⓘ localhost:8080/springbootweb/hello?name=saurabh

hello saurabh

Jars instead of wars

Springboot create jars instead of wars

Section 29) creating restful web-service

# REST?



CREATE      READ      UPDATE      DELETE

Http support 4 method to perform crud operation

Uniform Interface and Easy Access:

HTTP Methods:

POST

GET

VERBS

PUT

DELETE

URI: /employees

NOUNS

CREATE



POST /employees

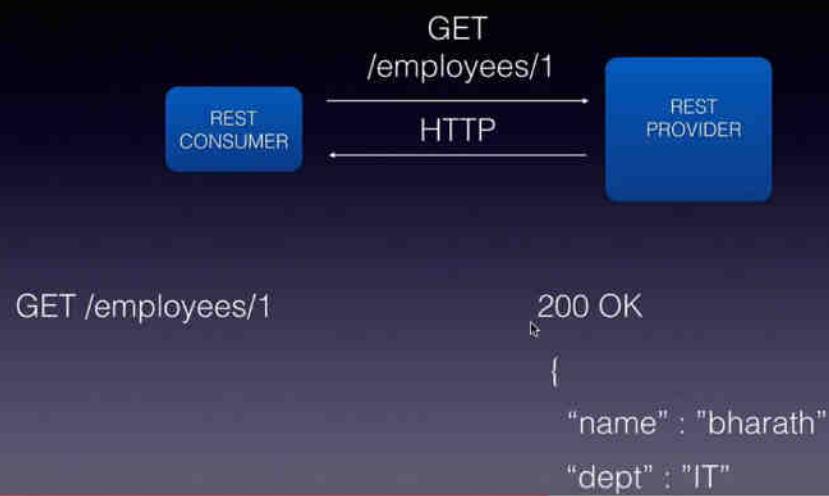
```
{  
  "name" : "bharath"  
  "dept" : "IT"
```

201 Created

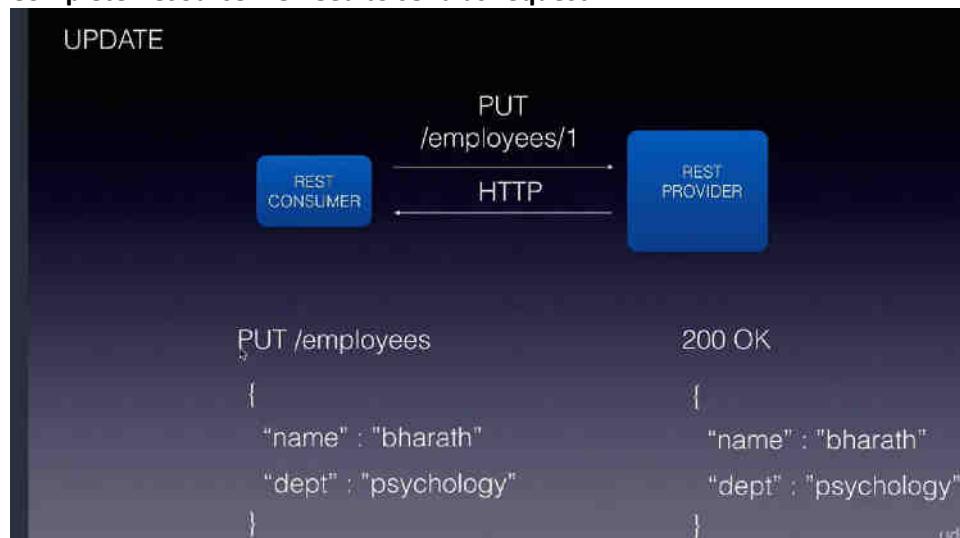
```
{  
  "name" : "bharath"  
  "dept" : "IT"
```

## What is REST API

Section 29, Lecture 264



Complete resource we need to send as request



Http patch method support partial update

## PATCH

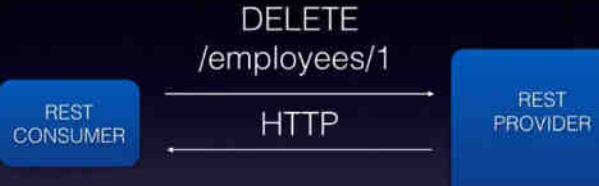


PATCH /employees

```
{  
    "dept" : "psychology"  
}
```

200 OK

```
{  
    "name" : "bharath"  
    "dept" : "psychology"  
}
```



DELETE /employees/123

200 OK

## MULTIPLE FORMATS:

```
<employee>  
    <name>john</name>  
</employee>
```

text/xml

```
employee:{name:john}
```

application/json

```
name=john
```

text/plain

## Rest Using Spring



### Steps:

- Steps
  - Add the maven dependency
  - spring-boot-starter-web
- Create a Rest Controller Class.
- Implement the CRUD Methods

```
0
1 @RestController
2 @RequestMapping("/products")
3 public class ProductController {
4     @Autowired
5     ProductRepository repository;
6
7     @GetMapping
8     public Iterable<Product> getProducts() {
9
10         return repository.findAll();
11     }
12 }
```

localhost:8080/products

GET localhost:8080/products

Params Authorization Headers Body Pre-request Script Tests

| KEY | VALUE |
|-----|-------|
| Key | Value |

Body Cookies (1) Headers (3) Test Results

Pretty Raw Preview JSON ↗

```
1 [  
2   {  
3     "id": 1,  
4     "name": "apple",  
5     "description": "awesome",  
6     "price": 4000  
7   }  
8 ]
```

## Implement create

localhost:8080/products

POST localhost:8080/products

Params Authorization Headers (1) Body Pre-request Script Tests

none  form-data  x-www-form-urlencoded  raw  binary **JSON (application/json)** ↗

```
1 {  
2   "id": 2,  
3   "name": "iphone",  
4   "description": "awesome",  
5   "price": 2000  
6 }  
7  
8 }
```

```

2
3 @RestController
4 @RequestMapping("/products")
5 public class ProductController {
6     @Autowired
7     ProductRepository repository;
8
9     @GetMapping
10    public Iterable<Product> getProducts() {
11
12        return repository.findAll();
13    }
14
15    @PostMapping
16    public Product create(@RequestBody Product product)
17    {
18        return repository.save(product);
19    }
20 }

```

## Update Product

**Repository.save:** knows if particular product is there in db or not if it is there then it will perform update operation

The screenshot shows the Postman application interface. At the top, the URL is set to `localhost:8080/products`. Below the URL, the method is selected as `PUT`. The `Body` tab is active, indicated by a red underline. Under the `Body` tab, the `raw` option is selected, and the `JSON (application/json)` radio button is checked. The JSON payload is displayed in the text area:

```

1
2 {
3     "id": 2,
4     "name": "iphone_new",
5     "description": "awesome",
6     "price": 2000
7 }
8

```

```
@PutMapping  
public Product update(@RequestBody Product product)  
{  
    return repository.save(product);  
}
```

### Read product by id

The screenshot shows the Postman application interface. At the top, there is a code snippet of Java code for a PUT mapping. Below it, the main interface shows a GET request to `localhost:8080/products/2`. The 'Params' tab is selected, showing a single parameter named 'Key' with a value of 'Value'. The 'Body' tab is also selected, displaying a JSON response:

```
1 {  
2     "id": 2,  
3     "name": "iphone_new",  
4     "description": "awesome",  
5     "price": 2000  
6 }
```

The status bar at the bottom right indicates a `200 OK` response.

```
@GetMapping("{id}")
public Optional<Product> getProduct(@PathVariable("id") Long id) {
    return repository.findById(id);

}

@GetMapping("{id}")
public void deleteProduct(@PathVariable("id") Long id) {
    repository.deleteById(id);

}
```