Spring Fundamentals:

* Spring will create the objects of the class. It would be responsible for any updates. If there is any change in the class name spring will handle this. We won’t be using the new keyword to create an object. In the configuration file of spring, we will not only mention the class name but also initialize.
* All the objects created in each layer (like we create Dao object in business layer, business layer object in service layer) all this would be done by the spring framework. This is dependency injection. One class is dependent on the other class for its services. These dependencies should be automatically resolved.
* Spring is responsible for dependency injection. Spring would be responsible to create objects of all the layers
* Laws of demerit.
* Non Invasive approach – This approach gives the benefit to avoid sharing the data from the developer. In this approach the class names will come from some xml file.
* Context.xml – It is the main configuration file for spring
* Every object which you needs to be created you need to mention the class name in context.xml
* Frameworks try to handle unchecked exception
* Autowiring is done on constructor, field
* Resource annotation given for dependency injection.
* Interception – It can be used for security, transaction handling, logging
* AOP – Aspect Oriented Programming depends on Interception
  + Advice/Aspect – Before/After
  + Join Points – on which method call Aspect is called.
  + PointCut – Defined on advices to decide the join points
  + Special Compiler – When we need a special compiler to call the before and after methods.
  + Load Time weaving – When the special class loader is used to compile the code in the sequence of before->join points -> after
  + Proxy Objects :
    - Before()
    - Abc()
    - After()
* Aspects types
  + Try{
    - Method()
    - afterReturning.
  + }catch(){
  + afterThrowing
  + }finally {
  + After
  + }
* Multi Context architecture – If we have some common beans needed in 2 or more WebApplicationContext(
* Whenever we start the TomcatServer then it searches for the Servlet and then the init() method is called. This in turn searches for appServlet-servlet.xml in WEB-INF folder. This Dispatcher servlet (provided) by Spring is configured in web.xml.
* Spring provides ready front controller
* Controlling method is responsible to suggest/return the View name(JSP, velocity etc)
  + Interact with service layer with @Autowired annotations
  + Data Transformation
  + Model and View – It is like a bag which has the data, jsp etc. Then it will be taken by the Dispatcher Servlet and then given to the request of the servlet
* When we need to add alias to the JSP then we go to the servlet xml and then add the below code

<beans:bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<beans:property name=*"prefix"* value=*"/WEB-INF/jsps/"* />

<beans:property name=*"suffix"* value=*".jsp"* />

* + </beans:bean>
  + Here value=*"/WEB-INF/jsps/" will give the folder where the jsps are stored*
* Component scan will let spring to reach to the package and then it will search for annotations
* Context.xml will be mapping every annotation to the context.xml and create bean
* @RequestParam will automatically bring the request parameter with the name and assign it to the variable empID
  + **public** ModelAndView getEmpDetails(HttpServletRequest request, @RequestParam("id") **int** empID)
  + **It is doing the below task**

/\*String strEmpId = request.getParameter("id");

int empID = Integer.parseInt(strEmpId);\*/

<listener>

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

</listener>

* The above listener searches for a particular config (context.xml) file whose name is applicationContext.xml. hence we rename the context.xml file to applicationContext.xml
* Dispatcher servlet mentioned in the web.xml create applicationcontext.xml. It gets all the beans from the context.xml. So we imported the context.xml in the appServlet-servlet.xml file which Dispatcher Servlet.
* When we need to change the name of the appServlet-servlet.xml, then in the <servlet> tags we declare the <init-param>
* Then we need different Application Context (parent and child (WebApplicationContext) so we wrote <listener> tags.
* This listener searches for the applicationContext.xml by default. So we should rename context.xml to applicationContext.xml
* When we need to change the applicationContext.xml then we need to include the <context-param> tag and mention the name there
* <%-- 'command' comes from the model.addAttribute("command", emp) in the Controller --%>

Context Loader Listener

* Spring.xml – We defined the following in this file
  + Dao
  + Service
  + Datasource
* springWebFin
  + DispatcherServlet
  + Controller
  + Jsp – internal view resolver – it gives the path of the jsps with prefix and suffix
  + All the request which will come from finance will go to finance
* springWebFin
  + DispatcherServlet
  + Controller
  + Jsp – internal view resolver – it gives the path of the jsps with prefix and suffix
* Ownership table – The table where we can get the data wrt the other table column is called the ownership table

@OneToOne //For one employee we have one department only so one to one

@JoinColumn(name="DEPTNO") //Since Emp table is the owner table (has the foreign key of Dept table) hence we are giving joincolum in the emp table

**public** Dept getDept() {

**return** dept;

}

* Above code will get the record from the dept table with matching deptno.

**public** **void** setDept(Dept dept) {

* **this**.dept = dept;
* And then that data from the getDept would be set in dept object in the setter table
* When there is one to many relationship from one side(table) then in that side(class) we should declare a Set for the (many relationship) table
* @OneToMany(mappedBy="dept") //This is the relation from dept to emp. One dept has many employees