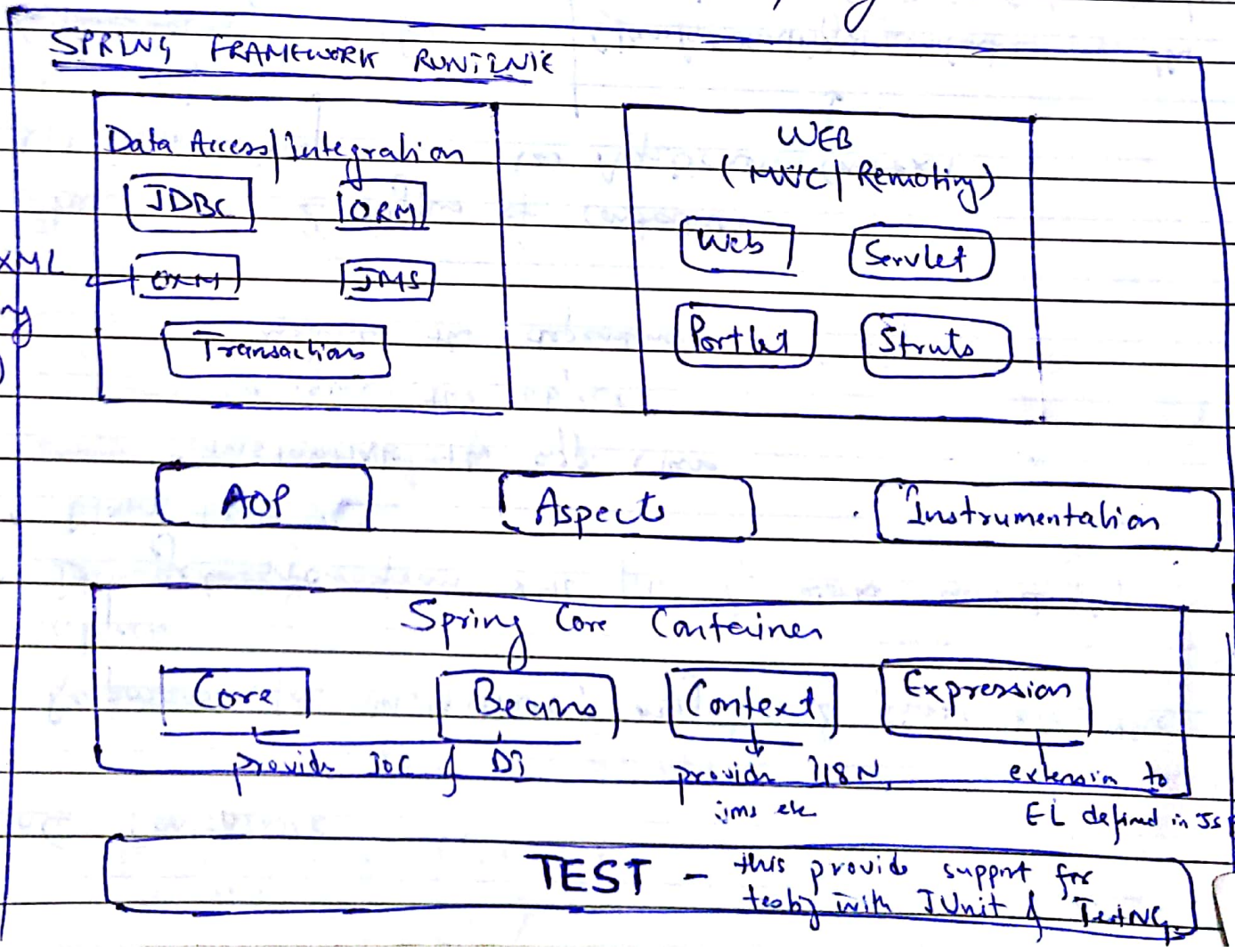
28/02/18

SPRING

- Spring Modules:
  divided into 7 parts
  i) Test
  ii) Core Container
  iii) AOP
  iv) Aspect
  v) instrumentation
  vi) Data Access/integration
  vii) Web MVC/Remoting



∘ IoC CONTAINER:

* Responsible for instantiate, configure & assemble the objects
* It gets info. from XML file & works accordingly.
* Main task is:
  - instantiate the app. class
  - configure the object
  - assemble the dependencies bet^{w} obj.

* There are 2 types of container:

(1.) BeanFactory

↓

XmlBeanFactory is the imp. class for BeanFactory interface.

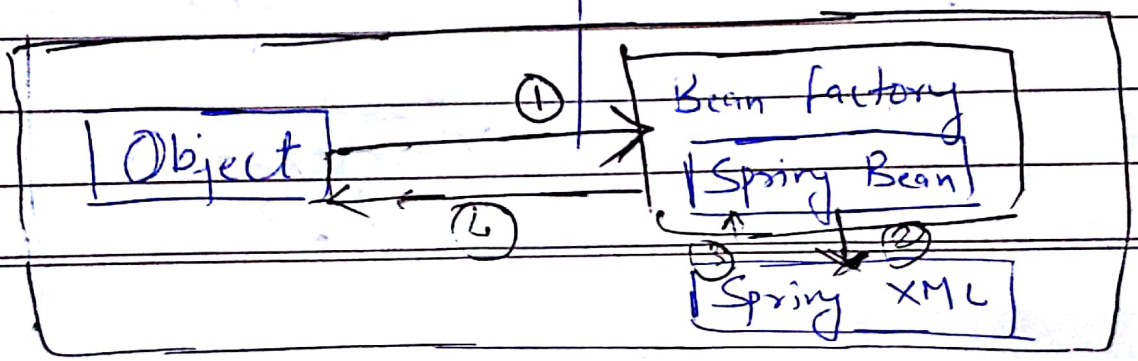
```
Resource res = new ClassPathResource
                ("applicationContext.xml");
BeanFactory fact = new XMLBeanFactory(res);
```

(2) ApplicationContext

↓

ClassPathXmlApplicationContext is the imp. class for ApplicationContext interface.

```
ApplicationContext context =
    = new ClassPathXmlApplicationContext
    ("applicationContext.xml");
```



5. ~~Context / Config~~

* DEPENDENCY INJECTION :

it is performed in 2 ways :

(1.) By Constructor (2.) By Setter Method

(1.) By CONSTRUCTOR :

<constructor-arg> of <bean> is used for constructor injection.

Example :

① Employee.java

```
package com.javapoint;
public class Employee {
  private int id;
  private String name;

  public Employee() {
      sysO.("Default Constructor");

  public Employee(int id)
  { this.id = id; }
```

```
public Employee (String name)
  {  this.name = name;}

public Employee (int id, String name)
  {  this.id = id;
     this.name = name;
  }

void show() {
    sysо (id + " " + name);
  }
}
```

⊙ applicationContext.xml :

We provide info. into the bean by this file.

The constructor-arg element invokes the constructor.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans
   xmlns = "
   xmlns:xsi = "
   xmlns:p = "
   xsi:schemaLocation = "
   <bean id = "emp" class = "com.javapoint.Employee" >
      <constructor-arg value = "10" type = "int" ></constructor-arg>
   </bean>
</beans>
```

① Test.java :

```
public class Test {
    p.s.v.m (String [] args) {
        Resource r = new ClassPathResource ("applicationContext.xml");
        BeanFactory factory = new XmlBeanFactory ();
        Employee e = (Employee) factory.getBean ("emp");
        e.show ();
    }
}
```

* CONSTRUCTOR INJECTION WITH DEPENDENT OBJECT :

ref attribute is used to define the reference of another object, such way we are passing the dependent obj. as an constructor argument.

```
<?xml version = "1.0" ...
<beans> ... :
    <bean id = "a1" class = "com.javapoint.Address" >
        <constructor-arg value = "Parul" > </constructor-args >
        <constructor-arg value = "Ganjwal" > </constructor-args>
    </bean>
    <bean id = "e" class = "com.javapoint.Employee" >
        <constructor-arg value = "12" type = "int" > </constructor-arg>
        :
```

```
<constructor-arg value="Parul"></constructor-arg>
<constructor-arg>
    <ref bean="a1" />
</constructor-arg>
</bean>
</beans>
```

* CONSTRUCTOR INJECTION WITH COLLECTION EXAMPLE:

We can inject collection values by constructor in spring framework. There can be used 3 element inside the constructor-arg element:

(i) list (ii) set (iii) map

Question.java (POJO):

```
public class Question {
    private int id;
    private String name;
    private List<String> answers;
```

```
public void displayInfo(){
    System.out.println(id + " " + name);
    System.out.println("Answers are:");
    Iterator<String> itr = answers.iterator();
    while(itr.hasNext()){
        System.out.println(itr.next());
    }
}
```

applicationContext.xml :

```
<beans ....>
    <bean id="q" class="com.javapoint.Question">
        <constructor-arg value="111"></constructor-arg>
        <constructor-arg value="What is Java"></constructor-arg>
        <constructor-arg>
            <list>
                <value>Java is a prog. lang.</value>
                <value>Java is a Platform</value>
                <value>Java is best prog. lang</value>
            </list>
        </constructor-arg>
    </bean>
```

* HOW TO INHERIT BEAN IN SPRING :

By using parent attribute of bean, we can specify the inheritance relationship bet" the beans.

applicationContext.xml :

```
<beans>
    <bean id="e1" class="com.javapoint.Employee">
        <constructor-arg value="101"></constructor-arg>
        <constructor-arg value="Parul"></constructor-arg>
    </bean>

    <bean id="address1" class="com.javapoint.Address">
        <constructor-arg value="B-704 S.H"></constructor-arg>
        <"  value="Pune"></...>
        <"  value="MH"></...>
    </bean>

    <bean id="e2" class="com.javapoint.Employee" parent="e1">
        <constructor-arg ref="address1"></constructor-arg>
    </bean>
</beans>
```

(2) DI By Setter Method :

We can inject DI by setter method also.
<property> sub-element of <bean> is used for setter injection. Here, we are going to inject.

(i.) primitive & string based values
(ii.) Dependent obj
(iii.) Collection values.

① applicationContext.xml :

```
<beans>
  <bean id="obj" class="com.jampoint.Employee">
    <property name="id">
    <value>10</value>
    </property>
    <property name="name">
    <value>Parul</value>
    </property>
    <property name="city">
    <value>Pune</value>
    </property>
  </bean>
</beans>
```

Pojo fields

* Diff. Between Constructor & Setter Injection:

(1.) Partial Dependency:
can be injected using Setter Injection but not possible by constructor.
Suppose there are 3 properties in a class, having 3 arg constructor & setters methods. In such case, if you want to pass info. for only 1 property, it is possible by only setter method only.

(2.) Overriding:
Setter injection overrides the constructor injection. If we use both, IoC container will use the setter injection.

(3.) Changes:
We can easily change the value by using setter injection. It doesn't create a new bean instance always like constructor. So, setter injection is flexible than constructor injection.

* AUTOWIRING IN SPRING :

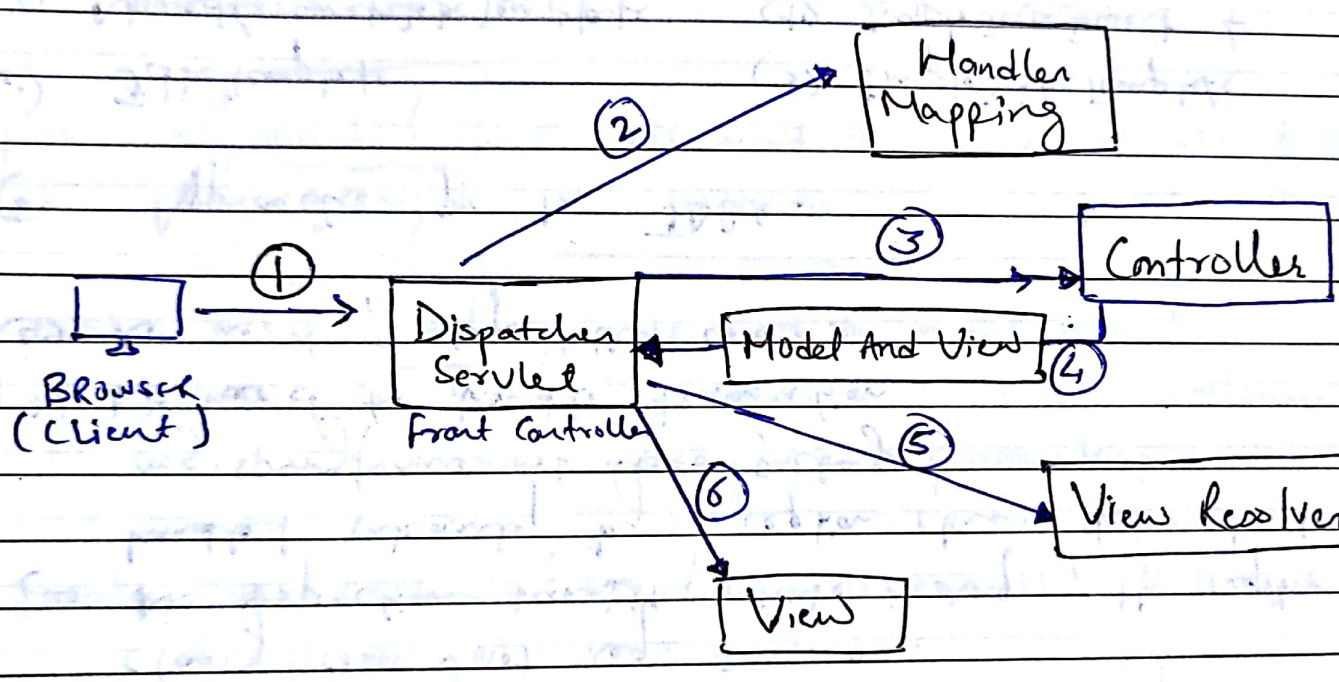
- Autowiring enables us to inject object dependency implicitly. It internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive & string value. It works with only reference.

- AUTOWIRING MODES :

No.	MODE	DESCRIPTION
(1.)	no	It is the default autowiring mode. It means no autowiring by default.
(2.)	byName	This inject object Dependency accordig to the name of bean. in such case property name & bean name must be same. It internally calls setter method.
(3.)	byType	This inject object dependency accordg to the type. i.e, property name & bean name can be diff. It internally calls setter method.
(4.)	constructor	This inject the dependency by callig the constructor of a class. It call the constructor having large no. of parameter.
(5.)	autodetect	Deprecated since Sprig 3

11/03/18

SPRING MVC



@Controller: used to mark class as a Controller.

@RequestMapping: used to map the request URL. It is applied on a method.

The method annotated with @RequestMapping return instance of ModelAndView Controller with mapped name, message name & msg. value. The msg. value will be displayed on jsp page.

To use Annotation based Spring use- <context:annotation-config> in spring.xml

Spring JdbcTemplate :

- it is a powerful mechanism to connect to DB & execute SQL queries.
- It internally uses JDBC api.

① Advantages of JDBC Template :

(1.) St. for creating connection, statement, closing resultset, closing conn. not req. anymore.
(2) No Exception handling code is req., the exption is handled internally by exception class defined in org.springframework.dao package.
(3.) No need to handle transaction.
(4.) No need for repetative code.

② Approaches pr. in JDBC.

(1.) JdbcTemplate
(2) NamedParameterJdbcTemplate
(3) SimpleJdbcTemplate
(4) SimpleJdbcInsert & SimpleJdbcCall.

⊙ Methods of Spring JdbcTemplate class :

METHODS	DESCRIPTION
(1.) public int update (String query)	is used to insert, update & delete records.
(2.) public int update (String query, Object...args)	is used to insert, update & delete records using PreparedStatement using given parameters.
(3.) public void execute (String query)	is used to execute DDL query.
(4.) public T execute (String sql, PreparedStatementCallback action	execute the query by using PreparedStatement callback.
(5.) public T query (String sql, ResultSetExtractor rse)	used to fetch record using ResultSetExtractor.
(6.) public List query (String sql, RowMapper rse	used to fetch records using RowMapper.

* applicationContext.xml :

The DriverManagerDataSource is used to contain the info. about the database such as Driver class name, connection URL, username & password.

```
<beans>
  <bean id="ds" class="org.springframework.jdbc.datasource.
                       DriverManagerDataSource">
    <property name="driverClassName"
              value="com.mysql.jdbc.Driver" />
    <property name="url"
              value="jdbc:mysql://localhost:3306/dbname" />
    <property name="username" value="root" />
    <property name="password" value="root" />
  </bean>
```

There is a property named a dataSource in the JdbcTemplate class of DriverManagerDataSource type. So, we need to provide reference of DriverManager DataSource obj. in JdbcTemplate class for datasource property.

```
<bean id="jdbcTemplate" class="org.springframework.jdbc.
                                core.JdbcTemplate">
    <property name="datasource" ref="ds" />
</bean>
```

Now, we are using the JdbcTemplate object in the DaO class, so we can pass it either by setter method or by constructor also.

```
<bean id="edao" class="com.javatpoint.EmployeeDaO">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
</beans>
```

⊙ Example of PreparedStatement in Spring:

We can use paramaterized query using Spring JdbcTemplate by the help of execute() method of JdbcTemplate class.

To use this, we pass the instance of PreparedStatementCallback in the execute method.

⊙ Method of PreparedStatementCallback Interface.

It has only one method i.e. doInPreparedStatement

Syntax:

```
public T doInPreparedStatement (PreparedStatement ps)
        throws SQLException, DataAccessException
```

EmployeeDao.java

```
public boolean saveEmployeeByPreparedStatement (final Employee e){
    String query = "insert into employee values (?,?,?)";
    return jdbcTemplate.execute (query,
        new PreparedStatementCallback<Boolean> () {

@override
public Boolean doInPreparedStatement (PreparedStatement ps)
        throws SQLException, DataAccessException {
    ps.setInt (1, e.getId());
    ps.setString (2, e.getName());
    ps.setFloat (3, e.getSalary());
    return ps.execute();
}
});
}
```

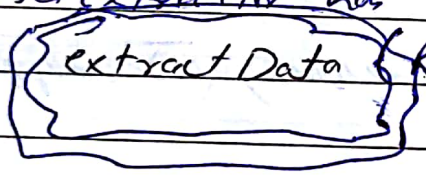
⊙ ResultSetExtractor : Fetching Record using Spring JdbcTemplate.

Record can be fetched using query() method when, we need to pass instance of ResultSetExtractor

Syntax:

public T query (String sql, ResultSetExtractor<T> rse)

ResultSetExtractor has one method i.e.




```
public List<Employee> getAllEmployees (){
    return template.query ("Select * from Employee",
        new ResultSetExtractor<List<Employee>> () {

@Override
public List<Employee> extractData (ResultSet rs)
        throw SqlExceptia, DataAccessException {

List<Employee> empList = new ArrayList<Employee>();
while (rs.next()) {
```

```
Employee emp = new Employee ();
emp.setId (rs.getInt(1));
emp.setName (rs.getString(2));
emp.setSalary (rs.getFloat(3));
emplist.add(emp);
}
return emplist;
}
});
}
}
```

⊙ RowMapper Example:

we can use RowMapper interface to fetch the records from DB using query() method.
Here, we need to pass instance of RowMapper

Syntax:

public T query (String sql, RowMapper <T> rm)

RowMapper Interface allows to map a row of the relations with the instance of user-defined class.

It iterates the ResultSet internally & adds it into the collection.

Method of RowMapper:

```
public T mapRow (ResultSet rs, int rowNumber) throws
                                        SQLException
```

```
public List <Employee> getAllEmployeeRowMapper () {
    return template.query ("select * from Employee"
                    new RowMapper <Employee> () {
    @Override
    public Employee mapRow (ResultSet rs, int rowNumber)
                        throws SQLException {
        Employee emp = new Employee ();
        emp.setId (rs.getInt (1));
        emp.setName (rs.getString (2));
        emp.setSalary (rs.getFloat (3));
        return emp;
    }
    });
}
```

⊙ Named Parameter Example :

While inserting data instead of ? we use name parameters. So, it is better to remember the data for the columns.

insert into employee values (:id, :name, :salary).

to use this we will call execute() method of this interface.

Syntax :

```
public T execute (String sql, Map map,
                  PreparedStatementCallback psc)
```

```
public void save (Employee e){
    String query = "insert into Employee values
                    (:id, :name, :salary)";

    Map<String, Object> map = new HashMap<>();
    map.put("id", e.getId());
    map.put("name", e.getName());
    map.put("salary", e.getSalary());
```

```
    template.execute(query, map, new PreparedStatementCallback(){
        @Override
        public Object doInPreparedStatement(PreparedStatement ps)
                throws SQLException, DataAccessException {
            return ps.executeUpdate();
        }
    });
}
```

SPRING MVC KEY POINTS :

- @Controller :
  It is the class that take care of all the client req. & send them to the configured resources to handle it.
  In Spring MVC :
  org.springframework.web.servlet.DispatcherServlet is the front controller that utilizes the context based on the spring bean config.

-Controller class is responsible to handle different kind of client req. based on the req. mappings.
It is used with @RequestMapping to define handler methods for specific URI mapping.

@Component : It is used to define Bean (Pojo) class.

@Repository : It is used for the DAO classes.

@Service : used for the class which act as a Service class. Usually business classes that provide some services.

⊙ Dispatcher Servlet Vs ContextLoaderListener :

DispatcherServlet :
it is the front controller in the Spring MVC App. & it loads the Spring bean conf. file & initializes all the beans that are configured.

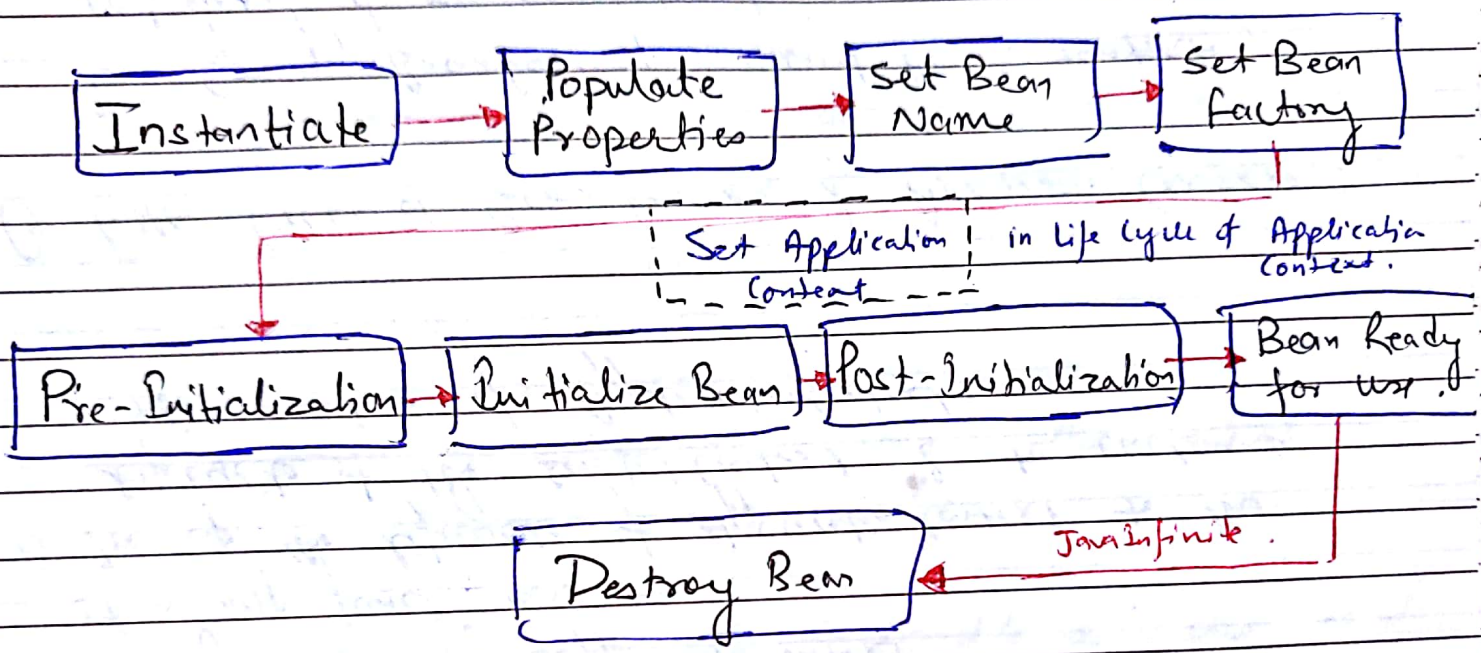
ContextLoaderListener :
it is the listener to start up & shut down Spring's root WebApplicationContext
it's imp funct" are :-
i) Tie up the lifecycle of ApplicationContext to the lifecycle of the ServletContext & to automate the creation of ApplicationContext.

⊙ Life Cycle of Bean Factory & Application Context :

in Spring, BeanFactory is used to instantiate, configure & manage objects

POO.

Life Cycle of Bean factory :



SPRING BOOT

- Spring Boot can be defined as something that lets you bootstrap your your Spring application.

- Spring.io website says:
Spring boot makes it easy to create stand-alone, production grade Spring based applications that you can "just run".

FEATURES:

① OPINIONATED:
It say that this is a good starting point & it make certain configuration changes & certain decision for you & say start with this & then see what to do. & change things if required.

② CONVENTION OVER CONFIGURATION:
if you want to 100 diff. thing then it says let do 80 diff things & left that be the

default & now only if you want to do 20% of the work, then only then you have to figure it out. in 80% no conf. needed.

③ STAND ALONE APPLICATION:
in Spring we were building a WAR file.
in Boot, we don't have to find a container & run & it is ready to run app.

④ PRODUCTION READY:
No need to do extra to do & deploy on Prod.

- HOW TO CREATE A SPRING-BOOT PROJECT:

1) Using Maven Approach:

STEP 1:
Create maven project.

STEP 2:
Select both checkbox (upper checkbox) & click Next

Step 3:

Group id: io.javabrains.springbootquickstart

It is the package name that uniquely identify the namespace for the project.

Artifact id: course-api

it is the name of the project

Version: 0.0.1-SNAPSHOT

Version of the component.

Name: Java Brains Courses API

Click Finish ↵

Till now we have created a Maven Project.

Now steps to convert this mave project to Spring Boot Project.

Step 1:

add <parent> tag as:

This is basically declaring that our proj is basically

child of this parent proj:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
</parent>
```

Step 2:
Declare dependencies.

1) spring-boot-starter-web

Step 3:
Create a class file with main() method.
this file will be annotated with @SpringBootApplication
& it will have a call to a class with method run i.e.

```
SpringApplication.run(className.class, args);
```

SpringApplication — static class
run — static method

This method does following things:

(1.) Set up default configuration
(2) Start the Spring App. Context.
(3.) Performs the Classpath scan.
(4.) Start Tomcat Server.

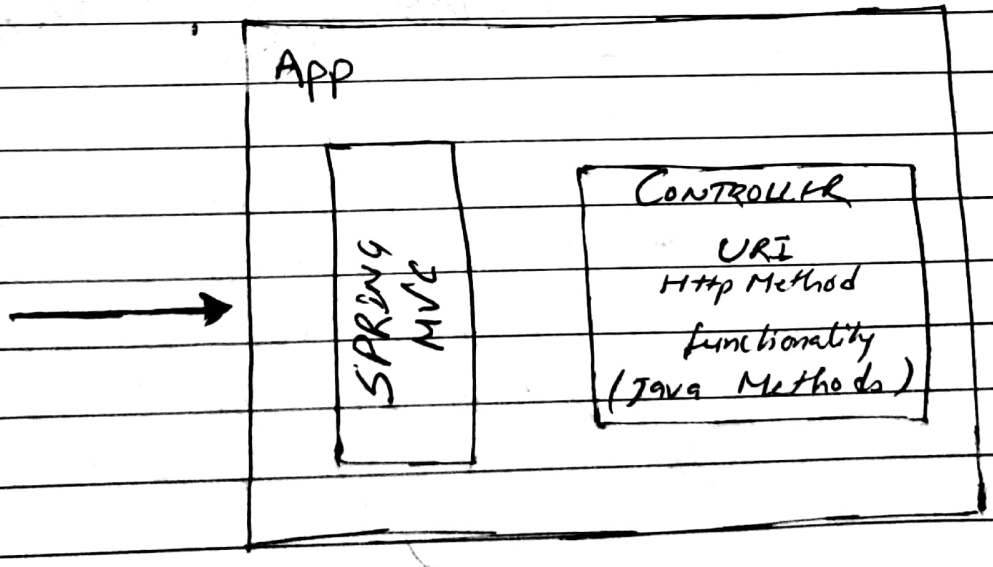
⊙ BILL OF MATERIALS:

The parent tag which was difined above under the pom.xml says what jars version it needs to download with the another tag of dependency.
So, if version is: 1.4.2.RELEASE then, it download jar of springframework of version 4.3.4, tomcat a 8.0 etc.
But, if i change it to 1.0.0.RELEASE, then it will degrate of now springframewrk becomes 4.0.3, tomcat a 7.0 etc.

• EMBEDDED TOMCAT SERVER :

(1) Convenience
(2) Servlet container config is now App. config
(3) Standalone App.
(4) Useful for microservices architecture.
Ex: Best ex is Engage app. where earlier we have to deploy multiple projects on tomcat but now them can be deploy using the single config file.

• SPRING MVC CONTROLLER :



- WAYS TO IMPLEMENT SPRING-BOOT :

1. Using a Maven Project
2. Using Spring Initializr : (http://start.spring.io)
3. Spring Boot CLI
4. STS IDE.

⊙ APPLICATION.PROPERTIES :

We can change the default values which we get by default from the Spring-boot setup using a file which should be present under resources folder i.e. application.properties

Ex :- server.port - 8081

⊙ SPRING DATA JPA : The Data Tier.

JPA : Java Persistence API.

It is a specification which lets us do ORM i.e. Object Relational Mapping.

ORM has - Class ⟷ Table.

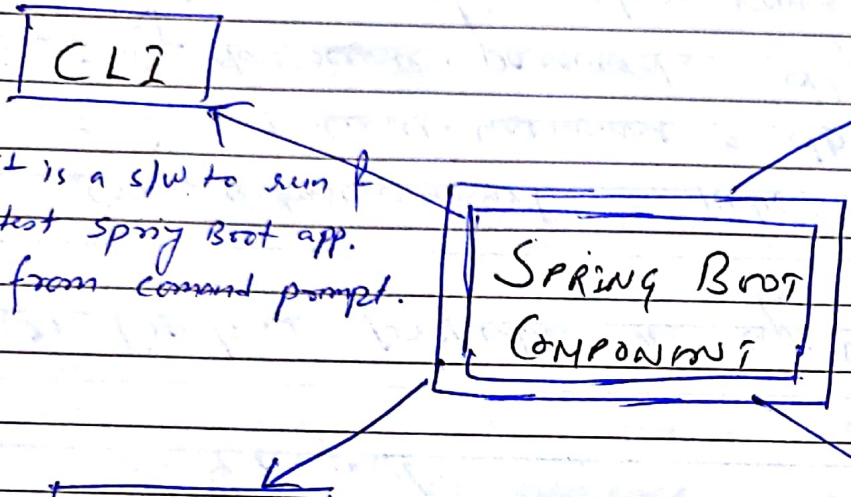
⑥ Spring-Boot CRUD Operation With JPA:

Steps to follow:

1) Add data-jpa & db dependency in the pom.xml
2) On the pojo class use below Annotation.
   - @Entity : This say to create class as a table
   - @Id : This say to keep this field a Pk.

⑦ Spring-Boot Actuator:

- It is a sub-project of Spring Boot. It provides several production-grade services to your app out of the box.
- If actuator is configured in your Spring-Boot app, you can interact & monitor your app by invoking diff. HTTP endpoint exposed by Spring-Boot Actuator.
- Ex:- Health, bean details, version details, conf., logger details etc.



- HOW TO RELOAD CHANGES ON SPRING BOOT WITHOUT RESTART SERVER :

We have to include following dependency in pom.xml

```
<dependency>
    <groupId> org.springframework </groupId>
    <artifactId> springloaded </artifactId>
    <version> 1.2.6.RELEASE </version>
</dependency>
```

- How To Implement Security for Spring-Boot App.

We have to add spring security starter to the boot app.

```
<dependency>
    <groupId> org.springframework.boot </groupId>
    <artifactId> spring-boot-starter-security </artifactId>
</dependency>
```

- How to Configure Datasource using Spring Boot.

(1) Use either spring-boot-starter-jdbc or spring-boot-starteldata-jpa & include a JDBC driver on classpath.

(2) Declare properties in application.properties file.

```
spring.datasource.url = jdbc:mysql://localhost:3306/dbname
spring.datasource.username = dbuser
spring.datasource.password = dbpass
spring.datasource.driver-class-name = com.mysql.jdbc.Driver.
```

⊙ Spring Bean Scope :

① Singleton : Only once per Spring Container.
it is the default scope.

② Prototype : New beans are created with every request or reference.

③ Request : New bean per servlet request.

④ Session : New bean per session

④ Global-Session : New bean per global HTTP Session.

⊙ Lifecycle Callbacks in Spring :

Spring provides callback method for the life-cycle of the beans.
This runs when the beans get created & runs when the beans get destroyed.

To close applicationContext we have to use AbstractApplicationContext & this calls the registerShutdownHook() method to close & destroy the beans.

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

Now, to call method when beans is initialize & destroy,
we have to implement an interface named as :
InitializingBean & DisposableBean.

For InitializingBean a method named - afterPropertiesSet() is called, it is executed one bean finishes initialization.

For DisposableBean a method named - destroy() is called, it is executed before beans get destroy.

2nd Way to implement Lifecycle Callbacks:

- We have to create our own my-init() & my-destroy() method inside a bean (Pojo class)

Then, we have to configure our defined methods in our bean pr. in the spring.xml file. as.

```
<beans>
    <bean id="triangle" class="org.paral.Triangle"
          init-method="my-init" destroy-method="my-destroy">
    </bean>
</beans>
```

if Req. is like the user-defined init() & destroy() need to be called on all the beans then, the method should be defined in main <beans> tag.

```
<beans default-init-method="my-init"
       default-destroy-method="my-destroy">
    .
    .
</beans>
```

⊙ BeanFactoryPostProcessor Vs BeanPostProcessor

(1.) It works on the bean definition or configuration meta-data of the bean before bean are actually created.

(1.) A ~~BeanPostProcessor~~ bean Post-Processor is a java class which implements the BeanPost-Processor Interface.

(2.) It has an overrider method postProcessBeanFactory(ConfigurableListableBeanFactory beanfactory);

(2.) It contains 2 callback methods —
postProcessAfterInitialization()
postProcessBeforeInitialization()

(3.) Default BeanFactory class.
→ PropertyPlaceHolderConfigurer
By defining the value in the Spring.xml we can use property file.

```
<bean class = org.springframework.beans.factory.config.
                PropertyPlaceHolderConfigurer "$7>
    <property name = "locations" value = "resources.properties" />
</bean>
```

⊙ How Spring App. Starts :

- Configuration related to the beans & its wiring is all defined in our spring conf. file which is present at the basepath.

- We can call the config file either by :-
  1.) BeanFactory
  2) By ApplicationContext.

- Using BeanFactory :

```
Resource res = new ClassPathResource("spring.xml");
BeanFactory fact = new XMLBeanFactory(res);
```

spring.xml : This xml file contains all bean config. about our application.

```
BeanFactory fact = new XMLBeanFactory(res);
```

→ This will bring Spring IOC container into our prog. it will be created by reading our config. file assigned to res object.
Spring IOC container is called BeanFactory & this

container is responsible for creating the bean obj & for injecting its dependencies throughout our app.

Now, we can get required obj from Spring container by calling getBean() method.
While calling this method, we need to pass the bean id as a parameter like -
. getBean(beanid), & this method always returns Object class obj & we need to typecast this into our bean type.

```
WelcomeBean wb = (WelcomeBean) fact.getBean("id1");
```

spring.xml:

```
<beans>
  <bean id="id1" class="com.capg.WelcomeBean">
    <property name="message" value="Welcome" />
  </bean>
</beans>
```

- Diff. b/t @RestController & @Controller

@Controller : it is marked to classes to act it as Spring MVC Controller.

@RestController : it is convenience annotation that does nothing more than adding the @Controller & @ResponseBody Annotation.

@RestController = @Controller + @ResponseBody.

@RequestBody : it maps to the HttpRequest body to a transfer or domain obj, enabling automatic deserialization of the inbound HttpRequest body onto a java object.

@ResponseBody : it tells the controller that the object returned is automatically serialized into JSON & passed back into the HttpResponse obj.

③ @MappedSuperClass:

if any field is common across our entities in an application (say id field), then to restrict the use of the field across all the entity class, we should create an AbstractEntity class & extend that in all the Entity class present.

```
@MappedSuperClass
public class AbstractEntity {
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private Long id;
```

④ @OneToOne:

In a Reservation table, a passanger & a flight will have a OneToOne Mapping.
i.e. every Reservation will have a Passanger & a flight.

```
@OneToOne
private Passanger passanger;
@OneToOne
private Flight flight;
```

⊙ @ModelAttribute :

it refers to the property of the Model Object.
Lets say we have a register page with the Model say User.
Then, we can have Spring MVC supply this object to a Controller method by using @ModelAttribute.
This will take all the form field values into the Model object & perform action on that object.

⊙ ModelMap :

- it is used to send a msg. back to the JSP page.
inside a method, we call create an object of ModelMap & return the msg. using :

modelMap.addAttribute("msg", "Login failed");

⊙ Command to kill Busy Port

→ netstat -ano | findstr :portnum.

This command will give you the PID (Process Identifier) which is using the defined Port

→ taskkill /PID pidnum. /F

eg.

e.g. netstat -ano | findstr :8080

it gives PID as :-

Listening 3740

Now, hit :- taskkill /PID 3740 /F