**Spring - Core**

1.**IOC**

2.**MVC**

3.**DAO** <==> **ORM** -->**Persistant API**

1.**Spring JdbcTemlate**,2.**Spring DataJPA,** 3. **Hibernate**

4.**AOP** ----> **@Before, @After:** **To enable or disable the some aditional functionality** without touching the actual business logic.

5.**Security**

==============================================================================

**Inversion of control:(IOC)**: **Dont create dependent beans, IOC container will create and provide where ever it required and do dependency injection.**

It is a design principle, normally object finds its dependencies on by itself and then calls them. Here, this is reversed: The dependencies are handed to the object by external sources when it's created.

Don't call around for your dependencies, we'll give them to you when we need you.

No need to create objects explicitly IOC container creates the instances ,just need to add configuration in xml.

**IOC(inversion of control) containers:** **read xml,instatiates beans ,do dependency injection and manages lifecycles**

**Main asset of IOC is Dependency injection.**

**Dependency Injection:** **injecting the Dependent beans either through field ,constructor, setting level injection**. Can be done using **@Autowired annotation**. means injecting the dependent obj to target class obj(How:arg constructor,setters)

DependencyInjection:is a design pattern and implements IOC principle.injecting the dependent properties by(external sources) setter or constructor arguments in a loosly coupled manner.

DI is by using java runtimepolymerphism.

**IOC containers:**

1.**For core:BeanFactory**=new XmlBeanFactory(new FileSystemResource(.xml));

2.**For J2EE:ApplicationContext, ConfigurableApplicationContext**=new ClasspathXMLApplicationContext(.xml):

**3. For Web:WebApplicationJdbcTemlate**

**@Autowired**: **To do dependency injection** .bydefault is **byType**(first check type if more than one then **byName** even **no name match** then **@Qualifier**);

* **@Autowire** with **variables**(no need setters and getters), also on top of **setter, constructor**.

@**Qualifier**("id"): if there are 2 beans are of same type then **Qualifier** annotated bean will be loaded. bydefault it will get **@Required**

**@Primary:** if there are 2 beans are of same type then **Primary** annotated bean will be loaded.

**@Required:** dependent bean **should not be null** if null then **application startup fails.**

**Note: if there are 2 beans are of same type then if no valid qualifier then application startup fails**

**BeanFactory:**Lazy container (at first request/reference it creates the singleton beans(object))

**ApplicationContext**:Eager (at the time of loading the context.xml it creates objects):Singleton beans loaded on application startup

Spring can carete beans for private constructor classes.

Class c=Class.forName();

Constructor[] cc=c.getDeclaredConstructors();cc[0].setAccessible(true)//to access privte constructors.c[0].newInstance();

**Type of depencency injections**: setter,constructor,abstract(lookup-method), Field level

**Type of dependencies**: Primitive,Secondary,primitive array,secondary array,collection

**Setter:** no duplicate propertes ,Constrcutor:order by index or type,can allow override value by giving same index

**Scope**s:**singleton**,**prototype**,**request**,**session**,**global-session**

**Singletons are per container per bean basis**

**Scopes**: by default spring create singleton beans

1.**singleton**: only one bean created per container on application startup

2.**Prototype**: new bean created for each request and reference

**MVC**:

3.**request**: for each input request

4.**session**: for each user session

5.**global session:**

Dependecy-check(none,spimple,object,all,required(on setter method)),depends-on(depenent instance created first),p:name space(p:propertyName="" or p:propertyName-ref=""),c:namespace(c:propertyName)

<context:annotation-config/>

**@Autowiring(byName,byType,constructor,autodetect,no):Auto dependency injection(can inject only secondary types);autowire-candidate=false;**

**Autowiring processs**:(**bytype**->**byname**->**qualifier**->**primary**->**application starup fail**): first it do autoire bytype if there are 2 beans are of same type then it do byname if there is no variable name match then it checks Qualifier if not then checks Primary if not then application startup fail by saying multiple beans of same type.

**@Service**

class Service{

**@Autowired**

**@Qualifier("abc")**

**private Repository repository;**

}

**@Configuration**: To define the bean configuration

**@Qualifier("id"):** if there are multiple beans of same type @Qualifier used to distinguish between them.

**@Primary(by default):** if there are multiple beans of same type by default @Primary bean will be loaded if you want other bean need to add @Qualifier

**@Required(mandatory bean):** dependent bean shoulnot be null, application startup fails if the dependent bean is null;

**@Configuration**

class A{

**@Bean("abc")**

public Repository m1(){

return new MySQLRepository();

}

**@Primary**

**@Bean("xyz")**

public Repository m2(){

return new OracleRepository();

}

}

**SterioType**:@**component**(non-mvc),@**repository**(DAO),@ **@required**

(business),@**controller**

| @**Component** │ generic stereotype for any Spring-managed component |:no special functionality ,spring managed bean

│ @**Repository**│ stereotype for persistence layer │:do database functionality

│ @**Service** │ stereotype for service layer │:do business logic

│ @**Controller**│ stereotype for presentation layer (spring-mvc): handle request and return response

@**Component**: It is a basic auto component scan annotation.

@**Repository**:bean post processor adds an advisor to any bean that’s annotated with @Repository so that any platform-specific exceptions are caught and then rethrown as one of Spring’s unchecked data access exceptions.

@**controller**:We cannot switch this annotation with any other like @Service or @Repository, even though they look same. The dispatcher scans the classes annotated with @Controller and detects @RequestMapping annotations within them.

@**Service**:

Inject Static property autowiring:class=MethodInvoicationFactoryBean-<property name=arguments valu="arg values" --name=staticMethod() value="static methods name"

For predifined Classes pivate constructor can not be called==Illigal access exception==>we can also make illigal access by putting throw in private constructor.

Factory methods called using :<bean class="Calender" factory-method="getInstance">==>it is sigleton static method and

same config for factory static method(<bean class="LoggerFactory" factory-method="getLogger">==gets Logger object).

<id="sf" class="sessionFactory"><id="s" factory-bean="sf" factory-method="openSession"/>==>factory instance method

**Factory Design pattern:** **Aim is to hide instatiation logic and return instance.**

Ex:

Scanner s= new Scanner(System.in);

String carName=s.next();

TataCar t = Tatafactory.getCar(carName);

t.drive(); ---> **method call decided at runtime so runtime polimerphism.**

NanoCar extends TataCar

PunchCar extends TataCar

class Tatafactory {

static TataCar getCar(String carName){

if(carName.equals("nano")){

return new NanoCar(engine, tyres);

} elss if(carName.equals("punch")){

return new PunchCar(engine, tyres);

}

}

}

Lifecycle: IntializingBean(afterPropertiesSet()), disposableBean(destroy):init-method="x()",destroy-method="y()":@PostConstruct,@Predestroy

<bean interface or abstract class><LookUp-method name="abstractMethodName" bean=""></bean>:If you want to get abstract method return object we add this;

<bean old class><replace-method name='old-methods' replacer='new class ref'>impliments Methods replacer:object reimpliment(object ,methods,parameters\_object[]){new implimentation}

ApplicationContextAware==>setApplicationContext()

BeanNameAware==>setBeanName()

BeanPostProcessor==>Object postprocessBeforeInitialization(Object,beanName) and Object postprocessAfterInitialization(Object,beanName)

BeanFactoryPostProcessor==>void postProcessBeanFactory(beanFactory) ==callled before singletons created.ex:PropertyPlaceholderConfigurer

@Autowiring(by type)=(@Resource(by name) in JDK=@inject(by type) in JDK)

@Component== @Named in jdk

PropertyPlaceholderConfigurer ==>to replace placeholders in context.xml by taking from .propertyfile

I18N for lanaguage support

L10N for business support like currency

@Lazy===to remove cyclic dependency check

Use @Lazy Annotation

Use Setter/Field Injection

1--request.getSession().setAttribute()/getAttribute()/invalidate()

2--response.add(new cookie("key","value"))/cookie.getValue()/setValue()cookie.setmaxAge(0)->method(@CookieValue(vaue="cookieName" ,defaultVal=""))

<form method="POST" action="uploadFile" enctype="multipart/form-data">

File to upload: <input type="file" name="file"><br />

Name: <input type="text" name="name"><br /> <br />

<input type="submit" value="Upload"> Press here to upload the file!

</form>

uploadFileHandler(@RequestParam("name") String name,@RequestParam("file") MultipartFile file) {byte[] bytes = file.getBytes();}

MultipartFile[]

**Spring MVC**

==============================================================================

**ModelAndView** m=new ModelAndView("viewName")

m.addObject("key","value")

**Frontcontroller**-->**Dispatcher servlet**

**HandlerMapppper**:BeanNameHandlerMapping :used to find appropriate controller by url path

**ViewResolver**:InternalResourceViewResolver :used to find appropriate View page by view Name

Without annotation:Extend AbstarctController class==override handleRequestInternal method()

Annaotation:<Context:component-scan base-package="com.xyz"/>so no need to extend AbstarctControler,no need BeanNameHandlerMapping and controller mapping in -servlet.xml

use @controller on top of class ,@RequestMapping("/path") on top of a method

**MultiActionController:**Multiple @RequestMapping in a single class on top of methods

In Url Use:/class/method

@**RequestMapping**("/class")

publci class A{

@**RequestMapping**("/method",Method.GET)

public **ModelAndView** method(){

**ModelAndView** m=new ModelAndView("viewName")

m.addObject("key","value");

**return m;**

}

}

<mvc:annotation-driven/> if you want use @PathVariable annotation

**PathVariable**:take variable value from path /method/pathvariable

publci class A{

@**RequestMapping**("/method/{**pathvariable**}")

public ModelAndView method(@**PathVariable**("**pathvariable**") String variable){

**return new ModelAndView("login", st);**

}

}

**Multiple Path variables**:/method/pathvariable1/pathvariable2

publci class A{

@**RequestMapping**("/method/{**pathvariable1**}/{**pathvariable2**}")

public ModelAndView method(@**PathVariable** **Map**<**String,String**> pathVariablesMap){

}

}

**Method with Http methodType:@RequestParam**:

@RequestMapping(value="/url" ,method=RequestMethod.POST)

public ModelAndView method(@**RequestParam**("studentName") **String** name,@**RequestParam**("studentHobby") **String** hobby){

}

public ModelAndView method(@**RequestParam**(value="studentName", **defaultValue**="xyz") **String** name,@**RequestParam**("studentHobby") **String** hobby){

}

**Multiple RequestParams:**

@RequestMapping(value="/url" ,method=RequestMethod.POST)

public ModelAndView method(@**RequestParam** **Map**<String,String> paramMap){

}

@**ModelAttribute**("modelObject"):auto data-binding possible by using this for built in datatype properties

To inject request input properties directly to model object(model properties=input names),no need to manually get from request and set to response.

**access by ${student1.name} at html/jsp**

@**RequestMapping**(value="/url" ,method=RequestMethod.POST)

public **ModelAndView** method(@**ModelAttribute**("student1") **Student** student1){

}

**ModelAttribute annotated method will be executed first** for all the request methods in the controller class

@**ModelAttribute**

public void addCommonObjects(**Model model**){

**model.addAttribute(**"header","common header for all views")

}

**Data Binding with a User-Defined Type**:input==>address.city,adress.pincode output==>student1.adress.city

**BindingResult:**To cath all the binding related errors and show to users:in UI **<form:errors paths="student1.\*"/>**

@RequestMapping(value="/url" ,method=RequestMethod.POST)

public ModelAndView method(@ModelAttribute("student1") Student student1,BindingResult result){

if(**result.hasErrors()**){

return new ModelAndView("viewName")

}

}

**To do custom binding:**Property Editing by existing property editors:FileEditor,classEditor,CustomNumberEditor are some propertyEditor classes

@**initBinder**

public void addCommonObjects(**WebDataBinder binder**){

binder.setDisallowedFields(new String[]{"phone"})

**binder.registerCustomEditor(**Date.class,"dob",new CustomDateEditor(new SimpleDateFormat("yyyy\*\*\*\*MM\*\*\*\*dd"),false))

}

we can write our own Custom Property Editor:Extends PropertyEditorSupport

override:setAsText(String studentName){setValue("MR."+studentName)}

**@Valid :Form-Validation:**When ever spring doing databinding task it throws respective error if not satisfy the condition

**JSR** **supported api library like hibernate**

public **class** **Student**{

**@Size(min=2,max=30 ,message="lenth between {min} and {max}")**

In property file Size.student1.studentHobby="{0} lenth between {2} and {1}"

private String studentHobby;

}

@RequestMapping(value="/url" ,method=RequestMethod.POST)

public ModelAndView method**(@Valid @ModelAttribute("**student1**") Student** student1,**BindingResult** result){

if(**result.hasErrors()**){

return new ModelAndView("**errorViewName**")

}

}

**@pattern(regexp="[^0-9]\*"), @Past,@Future,@Max(222),@Min(1), @NotNull, @NotEmpty**

**Custom Form-Validator**:@IsValidHobby(listOfValidHobbies="a|b|c" message="")

@Constraint(validatedBy=HobbyValidator.class)

public @interface IsValidHobby{

String listOfValidHobbies() default="a|b|c";

String message() default="please enter valid hobby";

}

public class HobbyValidator implements **ConstraintValidator**<IsValidHobby,String>{

String listOfValidHobbies=null;

initialize(IsValidHobby isValidHobby){

listOfValidHobbies=isValidHobby.listOfValidHobbies();

}

**boolean** **isValid**(String studentHobby,ConstraintValidatorContext ctx){

**return** listOfValidHobbies.contsians(studentHobby);

}

}

**Interceptor**:<mvc:interceptors><mvc:mapping path="/url"><bean class=""></mvc:interceptors>

extend HandlerInterceptorAdapter class and override boolean **preHandleMethod**(req,res,handler) ,**postHandleMethod**(req,res,object,ModelAndView),**afterComplete**(req,res,handler,Exception)

**Internationalization:**

<springTag:message code=lable.studentName>

url?sitelanguage=en

ResourceBundleMessageSource id="messageSource"==>property name="baseName" value="/WEB-INF/propertyFileName"

LocaleChangeInterceptor ==>property name="paramName" value="sitelanguage"

CookieLocaleResolver id="localeResolver"==>property name="defaultLocale" value="en"

**MultipleThemes:**

<link rel="stylesheet" href="<springTag:theme code="styleSheet">"

ResourceBundleThemeSource id="themeSource"==>property name="baseNamePrefix" value="theme-"

ThemeChangeInterceptor ==>property name="paramName" value="siteTheme"

CookieThemeResolver id="themeResolver"==>property name="defaultThemeName" value="green"

<mvc:resources mapping="/resources/\*\*" location="/WEB-INF/">

**ExceptionHandling**: using GlobalExceptionHandlerClass @**ControllerAdvice**

@**ExceptionHandler**(value=NullPointerException.class)

String handleNullPointerException(Exception e){//log exception, return "NullPointerException.jsp"}

Common exception for all controller classess:

@ControllerAdvice===SimpleMappingExceptionResolver

publci class GlobalExceptionHandlerClass

--------------------------------

<bean class="SimpleMappingExceptionResolver"

<property name="exceptionMappings"><map><entry key="" value=""></map></property><property name="defaultErrorView" value="Exception"/>

<property name="warnLogCategory" value="MVCLoggerClass"/>

**ErrorCodes**:

**100**=Continue===>**Information** responses

**200**=ok, **201**=created, **202**=accepted===>**successfull** responses

**301**=Moved Permanently===>**Redirection** messages

**400: Client errors**

**400**=Bad Request(input request validation failures Ex:**Invalid data format& invalid data**)

**401**=unauthrized(**authentication** failure,**bad credentials, invalid credentials**)

**403**=**Forbidden**(**authorization**, because of role, **invalid role**),

**404**=Resource Not Found,

**408**=Request Timeout===>Client error responses

**500: Server errors**

**500**=Internal Server Error(if you **didn’t handle the exception**)

**503**=Service Unavailable(overloaded(**busy**) or down for **mainatenance**),

**504**=Gateway Timeout===>Server error responses

@**ResponseStatus**(value=HttpStatus.INTERNAL\_SERVER\_ERROR)

@**ExceptionHandler**(value=RuntimeException.class)

**Spring Restful WebServices:**

**HTTP methods:**

**Idempotent**:on multiple hits of the same request there wont be any change(or create) in the response.if you hit same request multiple times you will get same response object.

**Idempotent**:GET,PUT,DELETE:multiple same requests will produce the same result.

**GET**:

1.**To retrieve the resource** from server(**200--ok**)

2.**Data present in the URL(visible so not secure).**

3.**Length of the url 2048 characters**

4.Bookmarked,cached,parameters saved in the browser history

5.only ascii characters allowed (binaries not allowed like file)

6.Idempotent ,so on refresh will produce the same result no modifications.

**POST**:

1.**To create a resource at the sever (201--created)**

2.Data send through the request body(not visible so secure)

3.No length restriction

4.Not-Bookmarked,Not-cached,request parameters not saved in the browser history

5.No restriction on characters,binaries(file) also allowed

6.Non-Idempotent,so on refresh browser alerts the user that 'do you want to re-submitted(performed earlier)'

7.**on multiple hits of the same request will create multiple resources.**

8.calling a POST request repeatedly may have side effects of creating the same resource multiple times.

**PUT**:

1.To create/update the resources

2.If the record already present then update if not it wil create new

3.Idempotent

4.on multiple hits of the same request will not change(or create) the response(but POST will change)

5.calling a PUT request repeatedly don't have any side effects such as creating the same resource multiple times.

**Post create vs PUT create:**

POSTing twice with the same data means create two identical users with different ids. PUTing twice with the same data creates the user the first and updates him to the same state the second time (no changes).

Since you end up with the same state after a PUT no matter how many times you perform it, it is said to be "equally potent" every time - idempotent

When you know the complete URL of the thing you want to create or overwrite, a PUT method should be used.

If you only know the URL of the category or sub-section of the thing you want to create something within, use the POST method.

**PATCH**:

1.To only partial modifications to the resource

2.The difference between PATCH and PUT, is that a PATCH request is non-idempotent(like a POST request).

3.Non-Idempotent

**DELETE**:

1.To Delete the resource at the specified URL

2.Idempotent

**HEAD**:

1.HEAD method is almost identical to GET, except without

**Restful services:**

**Exposes resources.**

**service Provider** and **consumer** can be written in **different prog languages** and not require to present in same system.

**To communicate:** 1.Need medium(Internet using HTTP)and 2.format of i/p and o/p messages(SOAP messages)

**SOAP**:It is a protocol (language) used to do communication between client and server.(provider and consumer).

**Exposes behaviour.Ex add()**

WSDL:webservice description language==>to know actual location of service(url) and description of all exposed methods and its syntaxes

UDDI:universal description and discovery integration==>contains WSDL

**REST**:It is a **architecture**(design) which uses the existing **Http protocol to comunicate between client and server**.

**Exposes resource.Ex: Caliculater to add, HTTP GET,PUT,DELETE,POST**

**REST is simpler over SOAP** :consumption:In SOAP consumer need WSDL(webservice description lang) ,need to generate stubs and invoke stubs to call serviceProvider.In REST we have **HTTPUrlConnection GET,POST,DELETE,PUT**

SOAP message in XML format,so need to parse .

**REST message is JSON/XML** ,parsing of **json** is **very fast and easy than xml.**

SOAP is secure ,has ws-security specification(token ,username,password,sign,encrypt messages)

**REST is less secure uses basic authentication.o-auth,token**

UDDI=universal description and discovery integration===HATEOAS=hpermedian as the enigine of application state

SOAP mostly uses HTTP protocal while sending messages over a network.(also uses SMTP,FTP,JMS)

**REST**:It is an **architecture** or design ,**uses existing protocal to communicate between producer and consumer(HTTP).**

**Exposes resources.**

**Request for resource and gets representational state of the resource.**

**message:JSON,XML...**

**uses HTTP protocal to transfer messages.**

**stateless**(cookie created at server & sent to client)

**Spring Resttemplate: To call a microservice from other** usingget,post,put,delete,exchange **methods**, it is **synchronous**(handle requests **after completing the previous**)

**Web client:** To call a microservice from other , it is **asynchronous**(can handle requests parallelly)

**Request**:

**url**(constant url + pathvariables + queryparams),

**request headers,**

**request body,**

**cookies**

**http method**

**Response**:

**response Headers,**

**response body**

**Http Status**

**Request**

**httpMethod**=**GET**, **url**:https://www.xyz.com/students/10?name=sridhar&coll=Mahaveer

**headers**:

**Controller:**

@**RestController**

class A {

@**Autowired**

Service service;

@**GetMapping**(path ="/staudents/{id}", produces=MediaType.APPLICATION\_JSON\_VALUE)

public **ResponseEntity**<Student> method(@**PathVariable**("id") Integer id, @**RequestParam**("name") String name/@**RequestParam** Map<String, String> req)){

Student s=service.getStudent(req);

**HttpHeaders** h= new **HttpHeaders**();

h.add("header1", "value1");

return new **ResponseEntity**<Student>(s,h,**HttpStatus.OK**)

}

}

**@PostMapping:**

Input validations can be written in requst model classes with the help of **@Valid** annotation in method argument.Ex**:@NotNull,@size,@pattern,customValidation**

**Request/Response Model:**

@**JsonInclude**(JsonInclude.Include.**NON\_NULL**)

@**JsonIgnoreProperties**({"age"})

@JsonPropertyOrder({"student\_name","age"})

class Student{

@**JsonProperty**("student\_name") //**to explicitly show this name**

@**Size**(min=15,max=20)

@**NotNull**

private String studentName;

@**Pattern**("regex")

private String email;

}

**DB Entity Model:**

@**Entity**

@**Table**(name="Student\_Details")

class StudentsAudit{

@**Autogenerated**

private id;

@**Column**("Student\_Name")

private String studentName;

@Column("Student\_Hobby")

private String studentHobby;

@Column("Student\_RegistrationDate")

private Date registeredDate;

@Column("Student\_UpdatedDate")

private Date updatedDate;

}

@**Service**

class **Service**{

@**Autoired**

private **StudentRespository** **respository**;

Student saveStudent(Student student){

StudentsAudit st = new StudentsAudit();

st.setName(student.getName());

StudentsAudit sa= **respository.save(st);**

student.setId(sa.getId());

return student;

}

}

@**Repository**

**interface** StudentRespositoryextends **JpaRepository<StudentsAudit, Long>{**

}

**Controller**:

@**RestController**

class A {

@**Autoired**

**Service** service;

@**PostMapping**(path ="/students", **produces=MediaType.APPLICATION\_JSON\_VALUE,** **consumes=MediaType.APPLICATION\_JSON\_VALUE**)

public **ResponseEntity**<Student> method(@**Valid** @**RequestBody** **Student** **studentObj**)){

Student s=service.saveStudent(studentObj);

**HttpHeaders** h= new **HttpHeaders**();

h.Add("header1", "value1");

return new **ResponseEntity**<**Student**>(s,h,**HttpStatus.Created**)

}

}

**@RestControlAdvice** : to handle exceptions that are thrown from controller == **Global Exception handler**

**@RestControlAdvice**

class **GloblaErrorHandler** {

@**ExceptionalHandler**(**RuntimeException.class**)

public **ResponseEntity**<Student> **method**(){

Student s= new Student();

**Error e = new Error();**

e.setErrorCode("5000")

e.setErrorDescription("some error occured")

**s.setError(e);**

**HttpHeaders h= new HttpHeaders();**

h.add("header1", "value1");

**return new ResponseEntity<Student>(s,h,HttpStatus.BAD\_REQUEST)**

}

}

**@Valid is for request data validation ---> if not valid client errors 400 series**

**@ResponseBody**<==>**converts Java object to equivalent desired** (json/xml)<==>add jackson jars to parse java object to json or xml

**@PutMapping**

**@DeleteMapping**

**@RestController**:**no need to set @ResponseBody** annotation on each RequestMapping method

**Headers**:Need to mention Accept:application/xml or json, no need to do any coding changes

**Get:**

produces=MediaType.APPLICATION\_XML\_VALUE:To restrict service to produce only xml(also need related Accept:application/xml otherwise error)

@**RequestMapping**(value**="/students**" method=**RequestMethod.GET**,produces=MediaType.APPLICATION\_XML\_VALUE)

public **List<Student>** getStudents()

**Put:**

@**RequestBody**:**Converts input request body data(json/xml) to Java object**

@**RequestMapping**(value="/students/{name}/{age}" method=**RequestMethod.PUT**)

public boolean updateStudents**(@pathVariable("name")** String studentName**, (@pathVariable("age")** String studentAge **, @RequestBody Student** studentObj**)**

need to mention input content type to understand the input by server.Content-Type:application/json

consumes=MediaType.APPLICATION\_XML\_VALUE:To restrict service to consume only xml(also need related Content-Type:application/xml otherwise error)

**@RequestMapping**(value="/students**/{name}**" method=**RequestMethod.PUT**,consumes=MediaType.APPLICATION\_XML\_VALUE)

public boolean updateStudents**(@pathVariable("name") String** studentName**,@RequestBody Student** studentObj)

--------------------------------------------------

**ResponseEntity<Type>: return something**

**HttpHeaders: responseHeaders**

**HTTPStatus: return response status**

--------------------------------------------------

**HttpHeaders httpHeaders=new HttpHeaders();**

**h.add("key","value");**

---------------------------------------------------------------------------------

**return new ResponseEntity<Student>(new Student(),httpHeaders,HttpStatus.OK);**

---------------------------------------------------------------------------------

**To send inserted record serviceUrl in response header:**

httpHeaders.add("location",ServletUriComponentBuilder.fromCurrentrequest.path("/{name}").builtAndExpand(student.getName()).toUri().toString())

**void ResponseEntity.noContent().build();**

**void ResponseEntity.created(location).build();**

==============================================================================

**Security:**

<filter-name>springsecurityfilterchain

<filter-class>DelegatingFilterProxy

**@RestController @GetMapping("/"),@PostMapping("/")**

**Richardson Maturity model:**grades the api meturity

**level 0:**expose a service endpoint at some URI.==>plain old xml,http used as transport b/w clinet and server,only one url,one http post method

**level 1:**resources ,multiple resource endpoints but single http post method

**level 2**:HTTP verbose, all HTTP methods and different resoure endpoints used

**level 3**:supports HATEOAS == send links as a response,client no need to aware of links(json+self documenting hyper media as a response)

**Spring Boot**

**Spring framework is mainly about dependecy injection**, writing Unit Testing, JDBC, and ORM framework support.

**Spring Boot over Spring:**

1. In Spring we need to **do lot of bean configuration** like in MVC we need to configure Dispatcher servlet, handler Mapper, View resolver, Error handle.
2. In Spring we need **add all maven dependencies with compatable versions**.

**Spring boot** has **Auto configuration** and **Auto dependency features** to solve the spring issues.

**Springboot makes the easy and fast application develop using starters and autoconfiguration.**

Spring Boot: To build production ready application

1.**Spring boot starter projects**: Spring boot adds the basic depencies along with compatable versions that are required for an application based on type.

2.**spring boot auto configuration**: SpringBoot autoconfigure the beans if the class present in the application classpath. Auto configure(define) the beans if the class present in the classpath.

3**.embedded server**

4.**actuator**

5.**default error handler**

6.**devtools**

**Disadvantages**: Boot for new projects not for existing projects migration.

**Spring boot starters:**

1.baisc dependencies that are required for a web application along with compatible versions. For compatable dependencies that are needed to develop our application ex:spring-boot-starter-web

2.**Auto configurations** like viewresolver,ResourceBundle,dispatcherServelt,datasource,hibernate session factory,transcationManager

3.**spring-boot-starter-actuater**:for monitoring the application in prod(metrics,healthcheck)

4.**spring-boot-devtools**:for instant building application when saved.

**Spring Boot Starters**: No need to mention dependency version for dependencies in pom.xml

**<spring-boot-starter-parent>:** gets **compatable** dependency versions , plugins, java

<parent>

<artifactId>**spring-boot-starter-parent**<artifactId>

</parent>

**Spring Boot Starter Web:**

**<spring-boot-starter-web>:** gets all the **basic** dependencies required for **a web application** and rest services.

<dependency>

<artifactId>spring-boot-starter-web</artifactId> //no need version its taken care by parent

</dependency>

**<spring-boot-starter-web-services>:** **for soap services**

**Set Java version:** To **explicitly mention** requied **version**(default parent provided verion overridded)

<properties>

<java.version>1.8</java.version>

</properties>

**Spring Boot plugin:** **To create jar/war** and run spring boot application

<build>

<plugins>

<plugin>

<artifactId>**spring-boot-maven-plugin**</artifactId>

</plugin>

</plugins>

</build>

**Create Spring Boot Application Launcher** (java class)

**@SpringBootApplication(@Configuration, @ComponentScan(package), @EnableAutoConfiguration)**

@**ComponentScan**("**package**")

publci class XYZ{

public static void main(String args[])

{

**ApplicationContext** a=**SpringApplication**.**run**(XYZ.class,args);

}

@**Bean**("abc")

**Object** m1(){

**retrun new Bean();**

}

}

checks @**Configure** **annotations** and **instantiates** **beans** and autoconfigure the mappings

**bydefault tomcat start runs when we run the java class**

**spring-boot-starter-parent:**

actulal dependencies are in **:<parent>spring-boot-dependencies</parent>**:contains all compatable dependency versions

**spring-boot-starter-web**:

actulal dependencies are in **:<parent>spring-boot-starters</parent>:**

contains **tamcat** as a default **embidded** **web** **container**

**hibernate-validator**

**jackson-databind**

**spring-webmvc**, **web**

**spring core**, **beans**, **aop**

**logging**

**Spring Boot Autoconfiguration:** auto configure lot of things

even for mvc no need of dispacher servlet configure

default error page provided

error controller,viewresolver,messageConverters

**@ConditionalOnClass**

**@ConditionalOnBean**

**@ConditionalOnMissingBean**

positive and negative matches

**Spring-Boot-AutoConfigure.jar** contains **all autoconfiguration classes**.

**DispacherServletAutoConfiguration** Matched ===>@**ConditionalOnClass** found **DispacherServlet**(**spring-webmvc.jar**) **class found in classpath**

@AutoConfigureOrder(Ordered.HIGHEST\_PRECEDENCE)

@Configuration

@ConditionalOnWebApplication

**@ConditionalOnClass(DispatcherServlet.class)**

@AutoConfigureAfter(EmbeddedServletContainerAutoConfiguration.class)

public class DispatcherServletAutoConfiguration {

@Bean(name = DEFAULT\_DISPATCHER\_SERVLET\_BEAN\_NAME)

public DispatcherServlet dispatcherServlet(WebMvcProperties webMvcProperties) {

DispatcherServlet dispatcherServlet = new DispatcherServlet();

return dispatcherServlet;

}

}

@Configuration

**@ConditionalOnClass(DataSource.class)**

@ConditionalOnSingleCandidate(DataSource.class)

@AutoConfigureAfter(DataSourceAutoConfiguration.class)

public class **JdbcTemplateAutoConfiguration** {

private final DataSource dataSource;

public JdbcTemplateAutoConfiguration(DataSource dataSource) {

this.dataSource = dataSource;

}

@Bean

@ConditionalOnMissingBean(JdbcOperations.class)

public JdbcTemplate jdbcTemplate() {

return new JdbcTemplate(this.dataSource);

}

@Bean

@ConditionalOnMissingBean(NamedParameterJdbcOperations.class)

public NamedParameterJdbcTemplate namedParameterJdbcTemplate() {

return new NamedParameterJdbcTemplate(this.dataSource);

}

**MessageConverterAutoConfiguration** ==> **@conditionalOn ObjectMapper** present in **jakson-databind**

Mapping servlets:dispatcherServelt to /

Mapped {[/error]}

=======================

**spring-boot-starter-web-service** for soap services

**spring-boot-starter-test** for junit,mockito

**spring-boot-starter-jdbc**

**spring-boot-starter-security --**basic authentication

**spring-boot-starter-data-jpa**

**spring-boot-starter-data-rest**

**if you add hibernate then it configure datasource, session-factory:**

if it finds hibernate jar in its classpath then it configure datasource, session factory beans with default values. you can override values in application.properties

application.properties(.yml): for any dynamic properties

**start.spring.io(spring initializer):**

give any groupid and artifact and dependecy module like -web,test==>Generate project

1.individual properties: @**Value**

@**Value("${welcome.message}"**)

private String welcomeMessage;

2.set of properties: @**ConfigurationProperties**:

@Component

@**ConfigurationProperties**("welcome")

public class BasicConfiguration {

private String message;

private String description;

}

In **application.properties:**

**welcome.message**=value of property

**welcome.description**=value of property

**spring.datasource.url=**url

**Load beans based on profile:** active profile beans will be loaded

**@profile(“dev”)**==>dynamically configure bean in dev env

**@profile(“qa”) )**==>dynamically configure bean in qa env

**java -jar -Dspring.profiles.active=dev myproject.jar**

**spring.profiles.active=dev**

**Configuration files for different env:** current **active profile** .properties file will be loaded among the available.

**application.properties**

**application-dev.properties**

**application-prod.properties**

**Actuators:** Spring boot **exposes endpoints to monitor the metrics.**

/actuator/:exposes rest services to monitor the application in prod

/autoConfig:positive and negative matches of beans

/configProperties: pre configurations

/metrics

/healthcheck

/heapdump

/env

/beans

/trace

**Spring Boot Data JPA operations:**

save(Object),saveALL(List<Object>)

findOne(ID)/findAll()/findAll(List<ID>)

deleteById(Id);/delete(Iterable);/delete(Object)/deleteAll()--insert then delete if not present

**Queries with methodname**:removes the common prefix

**With named queries**:@**NamedQuery**(name="findByName") on pojo then add this method in repo interface

**with @Query:**

@**SpringBootApplication** ==>

@**Configuration**(to define beans , makes the class as source of bean definations for app-context)+

@**EnableAutoConfiguration**(tells SpringBoot to start adding beans based on classpath)+

@**ComponentScan**(to scan the components present in a given package,tells spring to look for components,Configurations,services,controllers in the current package)

**In-meomoryDB**: H2

spring.h2.console.enabled=true(localhost:8080/h2-console)

spring.datasource.url/username/password/driver-class=

**actuators**: /info/health(status,free)/loggers/beans/env/metrics/trace(last 100)/mapping

**Custom endpoint:** class customEnd **implemets** **EndPoint**<List<Student>> ==>getId(),isSensitive(),isEnabled(),invoke();

**Logging**:slf4, logback(xml in resources), log4j

logging.level.root=DEBUG/INFO/WARN/ERROR

logging.file=

logging.patter.console/file=

**Schedular**:To schedule a particular task at a **particular time** or date.

@**EnableScheduling** on class

@**Sheduled**(fixedRate= or **cronExpression**)==> \*/10 6 5 \* \* \* for every 10 sec from 5:6:00

customScheduleMethod(){}

**Spring batch:** to do batch operations on bulk data ex: like reading from a db and writing into other db or file.

**TestCases**:

@**RunWith**(**SpringRunner**.class)

@**SpringBootTest**(instatiate whole application context)/@WebMvcTest(instatiatate only web layer)

class x{

@Autowired

MockMvc mockMvc

test(){

mockMvc.perform(MockMvcReqBuilder.get("/test")).andExpect(status().isOk).andExpect(content().string(containsString("hello")))

}

**ErrorManagement**:

common error handler across all the controller

@**ControlerAdvice**

Class CustomClass{

@**ExceptionHandler**(**Exception**.class)

**ResponseEntity**<Error> method(Exception e){

return new **ResponseEntity**<Error>(new Errror(),HttpStatus.Internal\_server\_error.value());

}

}

**Caching**:

@**EnableCaching** on class

@**Cacheable**("persons")/CacheEvict/CachePut/Caching regroups/CacheConfig

Person getPersons(){

}

=============================================================================

@Component("name")

@**Scope**(value="**prototype**")

**Class**

**context**.**getBean**(**A.class**);**created** for **every request and reference**

**WebApp folder structor:**

src/main/java

src/main/resources

src/test/java

src/test/resources

src/main/webapp/.jsp

**bydefault boot does not support .jsp we need to add tomcat-jasper**

return string(viewName);

spring.mvc.view.prefix/suffix=/pages/.jsp

request.getSession.setAttribute("name","value"); in jsp ${name}

method(@RequestParam("x") x,HttpSession session,HttpServeletRequest req,HttpServletResponse res)

ModelAndView m=new ModelAndView();

m.setViewNmae("page");

m.addObject("name","value");

@RequestMapping(value="")

method(Student s)==>no need(with form data)@RequestBody,it is required when you are sending using RAW json data

**JPA,H2,Web:**

**data.sql** in resources will load when we **start the application**

need to write interface CustomRepo extends **JPARepository**<**Student**,**Integer**> extends PagingAndSorting extends CrudRepository

@Controller

class Controller{

@Autowire

CustomRepo repo;

@RequestMapping("/add",produces={"application/xml","application/json"},consumes={"application/xml","application/json"})

method(){

repo.save();/delete();no update()

repo.findOne(Id);/repo.getOne(Id);

repo.findById();repo.findByIdGreaterThan(Id);

repo.findByName();need to declare abstart method in repo interface

Option<List<Student>> =repo.findAll();

}

}

**JPQA**: in repo interface

@**Query**("from student where name=?1 order by name")

List<Student> findByNameSort(String name)

**Formats:**In request Headers.Need Jackson-json/xml jars

**Accept**:application/xml-->**cleint requesting for xml response**

**ContentType**:application/xml-->**server sending xml response**

**GET**:Retrieve :@GetMapping("/Students/1")/@GetMapping("/Students")

**POST**:create :@PostMapping("/students")

**PUT**:create/update :@PutMapping("/students")==need id in req /@PutMapping("/students/1")==no need id in req

**DELETE**:Delete :@DeleteMapping("/students/1")

**PATCH**:partialUpdate

**Controller accepting request and calling repo methods**: otherthan that it is not doing anything.

So we can remove controller by DataREST(@**RepositoryRestResource**)

@**RepositoryRestResource**(collectionResourceRel="/students",path="/students")

interface Repo extends JPARepository<Student,Integer>

==============================================================================

1.To build Production ready application.(embedded servers)

2.Convension over config:removed xml conf and only annotation

3.Standadized For microservices

4.Cloud support

==> In **pom.xml:**

<parent>starter-parent</parent>

<properties>java</properties>

<dependencies>-starter-web</dependencies>

packaging==>jar <build><plugins>maven-plugin

**Lombok**:To reduce boilerplate code

@Data==tostring,eauals,hascode,getter/setter,required

@AllArgsConstructor

@NoArgConstructor

@ToString

@Entity

**Hateoas**: Hypher media as the engine of application state

Model extends ResourcesSupport

in controller method-->Link l=ControllerLinkBuilder.linkTo(controller.class).slash(u1.getUserName()).withSelfRel();u1.add(l);

**Swagger**:/swagger-ui.html

@**EnableSwagger2**

@Configuration

class Config{

new Docket().select().apis("package").paths("/rest\*").build().apiInfo("","","","");

}

@**ApiOperation**("")

@**ApiResponses**({@Apiresonse(code=100,message="100 response")})

@GetMapping("/rest")

@**Api**("") on controller class

@**ApiModelProperty**("") on model class property

**JWT**:**Authorization**: Bearer <token>

**Header**.**Payload**(Claim).**Signature**

**Header**:Base64Url

{

"alg": "HS256",

"typ": "JWT"

}

**Claim**: iss (issuer), exp (expiration time),Issued At, sub (subject), aud (audience)

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

**Signature**:

HMACSHA256(base64UrlEncode(header) + "." +base64UrlEncode(payload),secret)

**sping boot over Spring :**

In spring we need to write lot of config to setup application:

1.compatable maven dependecies

2.web.xml-dispather servlet

3.context.xml -> beans config

4.Install tomcat and add tomat mavne plugin

5.deploy and run

for a web App we need to configure: spring boot do these as a part of auto configuration

1.dispatcher servlet

2.tomcat server

3.view resolver

4.error handler

5.Jackson http message converter

**springboot-starter-web**: springboot-startter, tomcat ,jakson databinder,sping -webmvc/web,hibernate validator

**manifest.yml:** we need to provide a manifest.yml file, to bind the service to the application.

manifest.yml file tells the cf cli how to deploy your application. You can put configuration in here, but only because your manifest.yml file can be used to define environment variables.

In other words, your manifest.yml file can be used to set specific environment variables from which your application can later read configuration values. Your application cannot directly read manifest.yml.

applications:

- name: spring-boot-bootstrap

memory: 768M

random-route: true

path: ../target/spring-boot-bootstrap-cf.jar

env:

SPRING\_PROFILES\_ACTIVE: cloud,mysql

services:

- spring-bootstrap-db

**Jasypt** (Java Simplified Encryption) Spring Boot provides utilities for encrypting property sources in Boot applications.

**Whenever we need to store sensitive information in the configuration file** – that means we're essentially making that information vulnerable; this includes any kind of sensitive information, such as credentials, but certainly a lot more than that.

By using Jasypt, we can provide encryption for the property file attributes and our application will do the job of decrypting it and retrieving the original value.

input: abcd1234 (Actual password to be encrypted)

password: hello (the secret key chosen by you)

algorithm: PBEWithMD5AndDES (default algorithm used)

OUTPUT: scEjemHosjc/hjA8saT7Y6uC65bs0swg (Encrypted value of input)

encrypted.property=ENC(scEjemHosjc/hjA8saT7Y6uC65bs0swg)

**$mvn-Djasypt.encryptor.password=secretkey spring-boot:run**

lookup using JNDI name and its repective config is at server

**SSL**:Establish encrypted link b/w webserver and browser:data security

**1-way:**server send its public certificate to client(verify and stored in client trust srore).cleint send date to server by encrypting with the publlic cert.

**2-way:**server sends its public cert to cleint,cleint verifies the server cert and then sends clients public cert to server.(stored in trust store)

server.ssl.key-alias=selfsigned

server.ssl.key-password=

server.ssl.key-store=classpath:ssl-server.jks

server.ssl.key-provider=sun

server.ssl.key-type=jks

keystore explorer:create new keystore-->create keyValpair->RSA-->CN,OU,OL,ST,C

with http request it will give bad request-->allow safety

cleint service set jks to RestTemplate(load the keystore(jks)) and call the server service.