REST API =  Representational State Transfer Application (Application programming interface) ,JSON(javascript object notation)

SOAP API = Simple Object Access Protocol. It is an XML-based messaging protocol ,

Note: (@Path is used in JAX-RS Jersy implementation for controller class making restful , @Controller is used in webservice and @RestController is used in webservice to make it restful service )

URI

http://hostName:port/apllicaitonContextPath/

Flow of spring data jpa (in MVC)

DAO\_LAYER :

PART A

1st: make one pojo class >>> and annotate with @Entity >>> in the same class we can do @Id,@column in properties level and @Table in class level .

Again we can do @oneToMany , @manyToONe, @manyToMany ,@JoinTable, @joinColumn,mappedBy

@OneToMany : in one to many, @joinColumn is always owners table’s primary key . Eg. If person and phone are two class interrealated to eachother, one person can carry many phone . so phone comes under the person class as properties . Then we do @OneTomany and @joincolumns on the getter method of owner class person .

**# How to define maping relation like: @oneToMany , @manyToONe, @manyToMany,@oneToOne**

ANS: Suppose customer class has a reference of Address class as a property . Now customer class is owner and address class become target . And we need to start from owner class . let one customer may have multiple address , so this relation is @oneToMany and called unidirectional . Because we can get address detail by cutomerId but can not get customer detail by addressId .

Case : in practical, one address can be shared by many customer Like 4020 esters rd may have number of customer . this means @ManyToOne

Note: always remember, under customer table – addressId will be act as foreign key . and CustomerID is like primary key .

@OneToMany(cascade = CascadeType.ALL,fetch=FetchType.Lazy)

@JoinTable(name = "STUDENT\_PHONE", joinColumns = { @JoinColumn(name = "STUDENT\_ID") }, inverseJoinColumns = { @JoinColumn(name = "PHONE\_ID") })

public Set<Phone> getStudentPhoneNumbers() {

return this.studentPhoneNumbers;

}

Here : two tables are being joined ( person table and phone table) :-- both table’s primary key is taken as joincolumn . but child’s table(phone table) joinColumn is given inverseJoinColumns .

>>> cascading : --- is nothing but it gives instruction if changes made in parent table then it’s affect should be seen in child table .

Eg: person may have multiple phone , so here we have two different table person and phone . so if we save person in repository it will save only person not phone to its related table . So , if I want to save list of phone in to phone table at the same time while saving person in person table : cascade.save or cascade.persist need to use . Cascade.delete will delete all phone related to person if we delete person from repository .

So cascading is used to bring affect in related entity or table if we make changes in parent table table

Fetch Type : Lazy and Eager: ( however by default: from JPA 2.0 >> OneToMany-lazy, ManyToOne-Eager,ManyToMany:Lazy, oneToOne-Eager , but in hibernate all is lazy by default )

Eager: It is a design pattern in which data loading occurs on the spot.

Lazy : In Lazy loading , associated data loads only when we explicitly call getter or setter method .

Eg: if we use eager then when we instantiate person , at the same time all the associated phone details will be loaded

If we use lazy then when we instantiate person , at that time only property “phone’ will load but data related to phone will not load . To load the phone data we need to call getter method Like: person.getPhone();

PART B :

When completing Pojo class part , we need to connect pojo class to database . for this make in interface and extend it to JPARepository<pojo, integer>

NOTE: @Repository annotation can be used to in user defined interface which extends JPARepository , But it is optional because Spring auto detects this interface as repository .

Eg:

interface Personrepository extends JPARepository<Person, integer>{

Person save (Person person) // this is abstract method to save new person in database

Person gerPerson(int id) // this is abstract method to retrieve peron by its id .

}

Now this interface is responsible to map this pojo class to database and connect. This JPARepository will crate the table person in database and give the unique identifier by Integer .

Latter on , we can implement whatever abstract method mentioned in this interface to service

SERVICE LAYER ::

Service layer is the class which perform business logic and act as a middle man between repository and web controller . Here we make pojo class which is responsible to tranfer data from controller to service layer and service layer to controller . data coming from repository will be copied to VO and transfer to controller . Like wise data coming from controller copied to Entity or repository class and transfer to database . VO(visual object)

We make interface class and implemented class . Under implemented class we do business logic . latter on this interface will be autowired to controller class and controller class will access all the implemented class method and properties ,

Controller --🡪 VO (Service layer)copy to VO and paste to Entity and save to repository ---🡪 DB

Controller 🡨-----VO (service layer) copy to entity and paste to VO🡨----- entity🡨-------------DB

Controller --🡪 🡨------ UI (JSP,HTML,ThymeLeaf,angularJs,React etc)

##Used annotation in service layer : @Service , @Transactional , @AutoWired

@service it is specialization of @Component , it will make annoted class as service class and create its bean in spring container .

@Transactional : If we need to execute multiple query and its effect dependent on one to other , lets consider if one of the dependent query got failed then it affects whole process . So process should be rolled back if successfully run all query then commit. This process is called Transaction . eg. If I will sell coffee then when I sell coffee it should be appeared in sales table and at the same time it should update stock of coffee . this scenario creates transaction and we should run both query for sales and stock at the same time .

Transaction is treated as an aspect. Spring will apply the required transaction at run time using spring AOP .

We can use @Transactional in method level and class level

Note: Always use @Modifying annotation when we need to change the data already in table like update , delete and insert

@Repository : it is specialization of @Component , it will make annoted class as Repository class and create its bean in spring container .

@AutoWired : Autowiring feature of spring framework enables **us to inject the object dependency implicitly**. It internally uses setter or constructor injection. Autowiring can't be used to inject primitive and string values.

Here in service layer we do @AutoWired to Repository interface class so we can inject all the implemented class and method .

Eg .

@AutoWired

Personrepository personrepository; // now this is injected into service class and can access all the class and method that is associated with this .

WEB LAYER:

Here we autowired interface make in service class .

@Controller : it is specialization of @Component , it will make annoted class as service class and create its bean in spring container.

@RestController : @RestController is a specialized version of the controller. It includes the @Controller and @ResponseBody annotations, and as a result, simplifies the controller implementation:

@RequestBody :

@RequestParam and @PathVariable : and @MatrixVariable

@RequestParam and @PathVariable can both be used to extract values from the request URI, but they are a bit different. While **@RequestParams extract values from the query string, @PathVariables extract values from the URI path**:

@PathVariable: example:

localhost:999/v3/customer/findPayee/21 ---- >> This is uri path and @PathVariable pick its value 21 from here uri Path.

,

@GetMapping("/customer/findPayee/{payeeId}")

**public** PayeeInfoVO getAllPayees(@PathVariable **int** payeeId){ //using pathvariable

PayeeInfoVO payee= payeeInfoservice.findPayeeById(payeeId);

**return** payee;

}

**@RequestParams ExAmple :**

localhost:999/v3/customer/PayeeList?customerId=sam@gmail.com >>>> Here @RequestParam picks value [sam@gamil.com](mailto:sam@gamil.com) from query string ([CusotmerId=sam@gmail.com](mailto:CusotmerId=sam@gmail.com))

@GetMapping("/customer/PayeeList")

public List<PayeeInfoVO> getAllPayees(@RequestParam String customerId){ //using requestParam

List<PayeeInfoVO> payeeList= payeeInfoservice.findAllPayee(customerId);

return payeeList;

}

@MatrixVariable Example:

@RequestBody :

Simply put, **the @RequestBody annotation maps the HttpRequest body to a transfer or domain object, enabling automatic deserialization** of the inbound HttpRequest body onto a Java object.

Spring automatically deserializes the JSON into a Java type, assuming an appropriate one is specified.

By default, **the type we annotate with the @RequestBody annotation must correspond to the JSON sent from our client-side controller: meaning the type which annotate with @RequestBody will automatically serialize to json .**

localhost:999/v3/customer/payee/update

@PostMapping("/customer/payee/update")

**public** ApplicationResponseVO updatePayees(@RequestBody PayeeInfoVO payeeInfoVO){

ApplicationResponseVO response = **new** ApplicationResponseVO();

String message = payeeInfoservice.updatePayee(payeeInfoVO);

response.setCode(200);

response.setStatus("Success");

response.setMessage(message);

**return** response;

}

Here @RequestBody will map to PayeeInfoVO object and this acts like httpRequestBody

@ResponseBody:

The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

In given example we have used our custom responsetype (ApplicationResponseVO) with annotation @ResponseBody .

/localhost:999/v3/customer/payee/update

@PostMapping("/customer/payee/update")

@ResponseBody

**public** ApplicationResponseVO updatePayees(@RequestBody PayeeInfoVO payeeInfoVO){ /

ApplicationResponseVO response = **new** ApplicationResponseVO();

String message = payeeInfoservice.updatePayee(payeeInfoVO); //it will provide "update successfully" message

response.setCode(200);

response.setStatus("Success");

response.setMessage(message);

**return** response;

}

*ResponseEntity* :

*ResponseEntity* represents an HTTP response, including headers, body, and status. While *@ResponseBody* puts the return value into the body of the response, ResponseEntity also allows us to add headers and status code.

users.add(new User(3, "Tom"));

HttpHeaders headers = new HttpHeaders();

headers.add("Responded", "UserController");

return ResponseEntity.accepted().headers(headers).body(users);

@AttributeVariable:

Note: (@Path is used in JAX-RS Jersy implementation for controller class making restful , @Controller is used in webservice and @RestController is used in webservice to make it restful service )

@Value(“${value.name})

Long phoneNo;

This @Value is used to read the spring environment variable ( like application.properties)

@PostConstruct: = Equivalent to or similar to constructor