**Creation of Spring Boot Project –**

* Using spring suite tool
* Using spring initializer online site.
* Spring Web dependency: Restful Web Service, Inbuilt tomcat server
* Spring Dev tools: Reloads or Refresh directly when there are any changes made.

Application properties file is used for making application-level configuration.

Whenever any class has @Configuration Annotation then we can perform some spring related configuration in that class.

@ComponentScan Annotation will scan all the files present in the application and make them enable to execute.

**Creation of Hello World Controller Class –**

* Create a class HelloWorldController. Here Hello World called as resource as it is RESTful web service.
* @Controller annotation is used on class to make as a restful class. We can write GET, PUT, POST etc.
* @ResponseBody annotation have some html converters and it allows to send return value to HTML response body.
* The combination of above two is @RestController. We can use this above the class name.
* Now this class can handle the web requests and send responses.

@RestController

Public class HelloWorld {

Public static void main (String args[]){

# <http://localhost:8080/hello-world>

@GetMapping(“/hello-world”)

public String printData () {

Return “Hello World”;

}

}

**Returning Java Bean Class single and multiple students data –**

* Create a Java Class with setters and getters

@RestController

Public class StudentController {

Public static void main (String args[]){

# <http://localhost:8080/student>

@GetMapping(“/student”)

public Student printData () {

Return new Student (“Likhith”, “Kumar”);

}

# <http://localhost:8080/students>

@GetMapping(“/students”)

public List<Student> all () {

List<Student> s = new ArrayList<> ();

// add multiple students

return s;

}

}

* The response will be in the form of JSON with two fields first name and last name which are present in the Java Bean class.
* Jackson library converts the class object to JSON format.

**REST API with path variable –**

<https://localhost:8080/students/id> Here ID is a path variable.

Using Path variable annotation, we bind the URI template path variable to the method variable

@GetMapping(“/student/{firstName}”) Here first name is called as URI template path variable

Public Student getStudent (@PathVariable (“firstName”) String firstName) {

Return new Student (firstName,” Random”);

}

* Similarly, we can pass many path variables to the URI and map using a path variable annotation.

**REST API with Query Parameters/ Request Parameters –**

<https://localhost:8080/students?firstName=likhith?lastName=kumar>

The query parameters will be after ? and they will be in the form of key value pairs. If there are multiple we append using & symbol.

@Request param annotation is used to bind the parameter to the method variable

@GetMapping(“/student/query”)

Public Student getQueryParam (@RequestParam (name = “firstName”) String firstName,

@RequestParam (name =” lastName”) String lastName) {

Return new Student (firstName, lastName);

}

**REST API CRUD Project**

Spring boot Architecture – 3 Layered

POSTMAN 🡪 ***Controller Layer*** 🡪 ***Service Layer*** 🡪 ***DAO Layer*** 🡪 DB

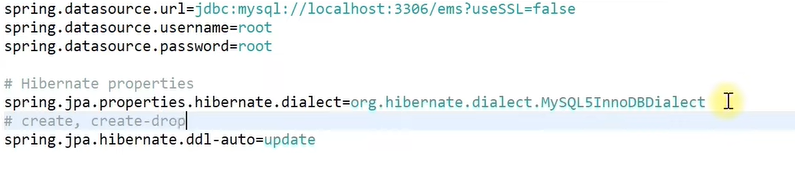
For request sending REST API layer Business Logic Persistence Logic

* DAO is also called as Repository layer
* It is responsible for communicating with the database.
* The logic that interacts with data base s/w and manipulates database data through the insert, update, delete and select operations is called persistence logic.
* POSTMAN is used for sending request in place of front end. It acts like a client.
* Spring data JPA used for interacting with database. It internally uses Hibernate JPA
* Lombok is used for reducing the boiler plate code like getters, setters etc.

Packages List:

1. Service
2. Controller
3. Exception
4. Model
5. Repository

* Open MySQL workbench and create an instance. Create a database.
* Then go to application properties file for configuring the database
* We need to give properties like
  + URL
  + Username
  + Password
  + Dialect – for providing the database queries
  + DDL – to automatically create tables in the database.
    - Value is Update to create or update existing table
    - Other values are create, create and drop etc.



Ems is database name

**Model or Entity Creation:**

In the model package create employee class

@Entity // to make this an JPA entity

@Table (name= “employees”) // to map this class to table in the database. If not given table name is class name

public class Employee {

@Id // for making this attribute as primary key. If not given in the body, it is auto generated.

@GeneatedValue (strategy = GenerationType.IDENTITY)

private long id;

@Column(name=”first\_name”, nullable = false) // not null condition

private string firstName;

@Column(name=”last\_name”, nullable = false) // not null condition

private string lastName;

@Column(name=”email”)

private string email;

// Getters and setters

}

**Employee Repository Interface:**

* Go to repository package

// 1st parameter is the type of Entity and 2nd parameter is type of the primary key

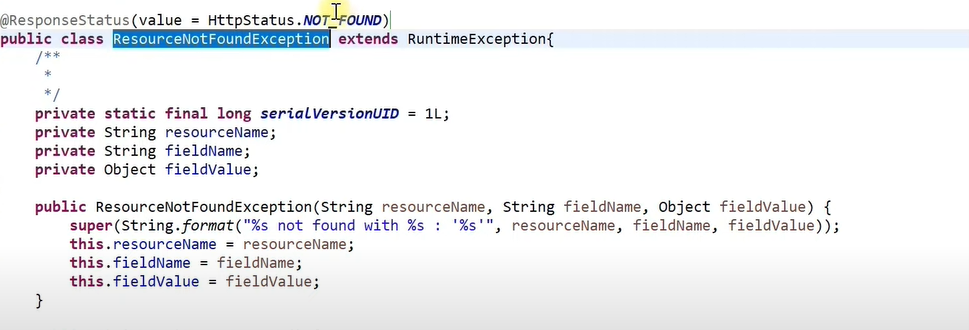
public interface EmployeeRepository extends JpaRepository <Employee, Long> {

}

* Now we can have access to CRUD methods. If we want anything new we can add here.
* We need not add @Repository annotation as JPA repository internally has this annotation

**Custom Resource Not Found Exception**

* Go to the exception package and create a class with some name
* Here we are sending the response status as Resource Not found if this exception occurs.



**Employee Service class – Contains Business Logic**

* Create a class inside the service package.
* Create EmployeeRepositoy class for the usage. Here emprepo
* Mark the class with annotation @Service

Methods:

// Adding to the database: POST

public Employee save (Employee employee) {

return emprepo.save (employee);

}

// Sending all employees

public List<Employee> getAllEmployee(){

return emprepo.findAll();

}

// Get employee by id

Public Employee getEmployeeById(long id){

Optional<Employee> employee = emprepo.findById(id);

// find by id returns optional object. We get actual object by using get method

If(employee.isPresent()){

return employee.get();

}

// or else throw custom exception we have created

return new ResorceNotFoundException(“Employee”, “Id”, id);

OR

return emprepo.findById(id).orElseThrow(

()-> new ResorceNotFoundException(“Employee”, “Id”, id));

}

// Update employee by id

public Employee updateById(Employee employee, long id){

Optional<Employee> exist-employee = emprepo.findById(id);

// find by id returns optional object. We get actual object by using get method

If(employee.isPresent()){

Employee emp = exist-employee.get();

Emp.setFirstName(employee.getFirstname());

// similarly for last name and email

emprepo.save(emp);

Return emp;

}

// or else throw custom exception we have created

return new ResorceNotFoundException(“Employee”, “Id”, id);

}

// Delete Employee By Id

Public void deleteEmployee(long id){

emprepo.findById(id).orElseThrow(

()-> new ResorceNotFoundException(“Employee”, “Id”, id));

emprepo.deleteById(id);

}

**Employee Controller Class:**

* Create a class in controller package.
* Add annotation @RestController to the class
* User @RequestMapping annotation to create some base URL for all the APIs.
  + @RequestMapping(“/api/employees”);
* Create an object for the employee service class. Here it is service.
* Response Entity allows us to send some additional information like status code as the return value.

Method:

// adding employee to database

@PostMapping(“/addEmployee”)

public ResponseEntity <Employee> addEmployee (@RequestBody Employee employee){

Employee emp = service.save (employee);

return new ResponseEntity <Employee> (emp, HttpStatus.CREATED);

}

// Get all employees REST API

@GetMapping(“/allEmployees”)

public List<Employee> getAll(){

return service.getAllEmployees();

}

// Get Employee by Id

@GetMapping(“/employee/{id}”)

public ResponseEntity <Employee> getbyId(@PathVaribale (“id”) long id){

return new ResponseEntity <Employee> (service. GetEmployeeById(id), HttpStatus.OK);

}

// Update employee : Request body for converting JSON to java object

@PutMapping(“/employee/{id}”)

public ResponseEntity <Employee> update(@RequestBody Employee emp, @PathVaribale (“id”) long id){

return new ResponseEntity <Employee> (service.updateById(emp, id), HttpStatus.OK);

}

@DeleteMapping(“/employee/{id}”)

public ResponseEntity <String> delete(@PathVaribale (“id”) long id){

service.deleteEmployee(id);

return new ResponseEntity <String> (“Employee Deleted Successfully, HttpStatus.OK);

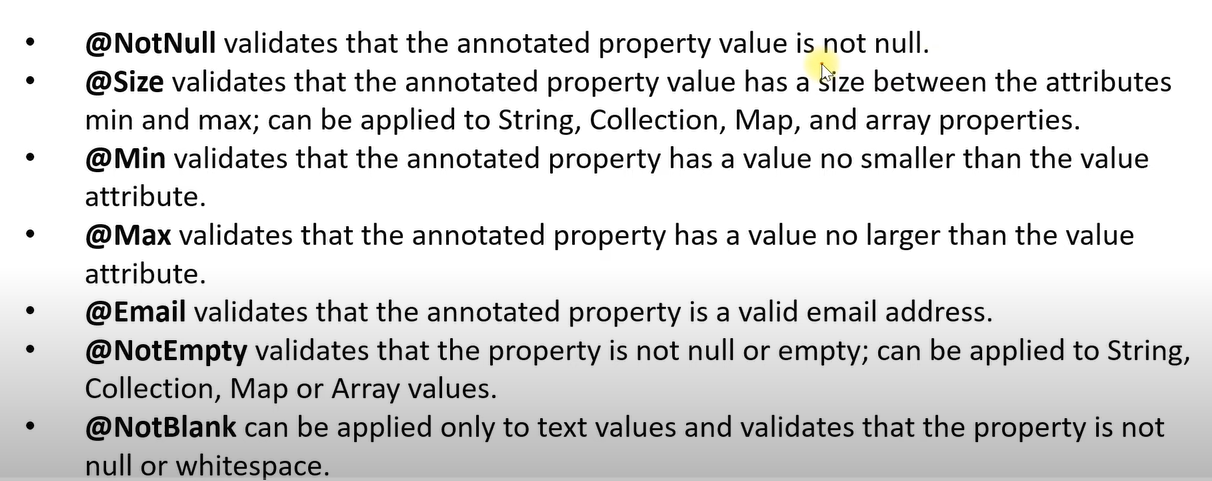
}

Link: <https://github.com/RameshMF/spring-boot-tutorial-course>

**Validations in Spring boot**

**Using Hibernate Validator:**

* Hibernate validator is one of the reference implementations of the Java Bean Validator.



* To make it work we need to add spring-boot-started-validation dependency while creation or in pom.xml file.

Create a Model named User:

@Table(name=”users”)

@Entity

class User{

@Id // for making this attribute as primary key. If not given in the body, it is auto generated.

@GeneatedValue (strategy = GenerationType.IDENTITY)

private long id;

@Column(name=”name”, nullable = false)

// user is not null or not empty and it should have at least 2 characters.

@NotEmpty

@Size (min=2, message = “name must have at least 2 characters”)

private String name;

// email must be a valid one and it should not be null

@NotEmpty

@Email

private String email;

// password is not empty and minimum length is 8

@NotEmpty

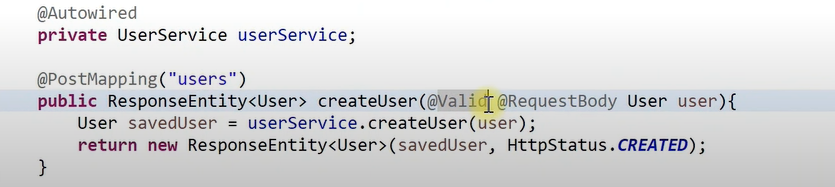
@Size (min = 8, message = “at least 8 characters”)

private String password;

// constructors and getter setters

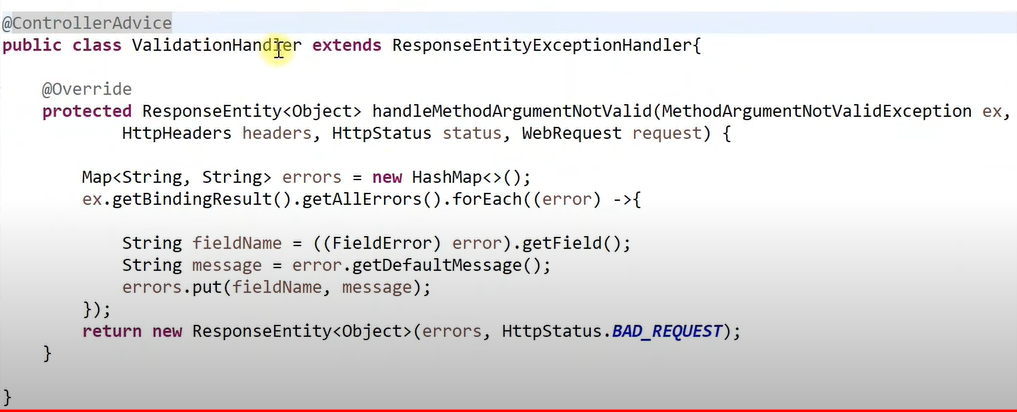
}

* To make validations work we need to add the @Valid annotation in the Controller class.



Making a Custom Response Message:

* Inside controller package create a ValidationHandler class
* We need to extends some classes and override method in order to create a custom response message.
* We need to use @ControllerAdvice annotation to handle.



* We are stroring all error messages in the form of key value pairs and returning them.
* The messages are the messages we have provided inside the annotation.

Link - <https://github.com/RameshMF/springboot-validation>

**Spring Boot Custom validation annotations:**

We need to create a Annotation interface and annotation class for defining how validation should occur.

