Springboot devtools dependency

    - no need to restart the appl each and everytime when we make any changes in appl, the appl will automatically restarted

<dependency>

      <groupId>org.springframework.boot</groupId>

      <artifactId>spring-boot-devtools</artifactId>

      <scope>runtime</scope>

      <optional>true</optional>

    </dependency>

Springboot profiling

     - In enterprise appl we have many env like dev, prod, test etc and each env needs specific configuration related to that enn and configured in appliation.properties or yml file, we cant configure everything in single application.properties file, so we have to use different properties file in different env and that concept is called profiles

1. We need to create different properties file related to env like application-profilename.properties (ie) application-dev.properties, application-prod.properties

2. By default Springboot reads all configuration from application.properties, to configure which env properties file to be read we use

    spring.profiles.active=profilename

@Value - used to read single property from properties file to controller prg

@Profile - used to programmatically control files based on profile

application.properties

server.port=1000

message=Welcome default user

spring.profiles.active=dev

application-dev.properties

server.port=1001

message=Welcome development user

application-prod.properties

server.port=1002

message=Welcome production user

@RestController

public class ProfileController {

              @Value("${server.port}") //SpEl - Spring Expression Language

              private Integer port;

              @Value("${message}")

              private String msg;

              @GetMapping("/profile")

              public String getProfileInfo() {

                             return "Hello from Controller: "+msg+" runs on port no "+port;

              }

}

@Configuration

@Profile("prod")

public class ProfileConfig {

              @PostConstruct

              public void print() {

                             System.out.println("This method should be printed only for prod profile");

              }

}

By default Springboot reads all configuration from application.properties or application.yml

application.properties

1. It is represented as sequence of key value pair

server.port=1001

server.servlet.context-path=/app

spring.profiles.active=dev

2. This files is supported only in Java lang

3. support key,value pair but both key and value should be only in the form of String

4. If we want to handle multiple profile then we have to create different properties file

application.yml

1. It is represented as hierarchial format

server:

   port: 1001

   servlet:

      context-path: /app

spring:

   profiles:

      active: dev

2. This file is supported in Java, Python etc

3. support scalar datatype, map, list, key value pair

4. If we want to handle multiple profile then we can handle in single yml file

server:

   port: 1001

   servlet:

      context-path: /app

spring:

   profiles:

      active: dev

---

spring:

   profiles: dev

server:

   port: 1002

---

spring:

   profiles: prod

server:

   port: 1003

By default Springboot reads all configuration from application.properties or application.yml present inside src/main/resources folder. But if we create properties file in different name and in different location, then how Springboot read those properties file

@PropertySource - used to read single properties file present in different name and in different location

@PropertySources - used to read multiple properties file present in different name and in different location

@Value - used to read single property from properties file to controller prg

@ConfigurationProperties - used to map entire properties from properties file to a separate bean object

Springboot provides Lombok dependency  - Java annotation library which helps to reduce boilerplate code. (ie) getter and setter, default constructor, parameterized constructor, toString(), equals(), hashCode(), logging functionality(@Slf4j)

student.properties inside src/main/resources

student.id=100

student.name=Ram

student.address=Chennai

student.age=25

student1.properties inside C:/Training folder

[student.email=ram@gmail.com](mailto:student.email=ram@gmail.com)

student.course=CSE

student.mark=89

@Configuration

[//@PropertySource("classpath:student.properties")](mailto://PropertySource(%22classpath:student.properties%22))

[//@PropertySource("file:\\C:\\Training\\student1.properties")](mailto://PropertySource(%22file::\\Training\\student1.properties%22))

@PropertySources({

              @PropertySource("classpath:student.properties"),

              @PropertySource([file:\\C:\\Training\\student1.properties](file:///C:\Training\student1.properties))

})

@ConfigurationProperties(prefix="student")

//@Getter

//@Setter

@Data  //getter+setter

@NoArgsConstructor

@AllArgsConstructor

@ToString

@EqualsAndHashCode

public class StudentConfig {

    private String name;

    private String address;

    private Integer age;

    private String email;

    private String course;

}

@RestController

public class StudentController {

              @Value("${student.id}")

              private Integer id;

              @Value("${student.mark}")

              private Integer mark;

              @Autowired

              StudentConfig config;

              @GetMapping("/student")

              public String getStudentInfo() {

                             return id+" "+config.getName()+" "+config.getAddress()+" "+config.getEmail()+" "+mark;

              }

}

@Value

1. Access elements one by one

2. Support SpEL

3. Loose Binding/Loose Grammar is not supported (ie) property name always should be matching

4. Validation of properties is not supported

5. support only scalar datatype

@ConfigurationProperties

1. Bulk accessing of properties

2. Dosent support SpEL

3. Loose Binding/Loose Grammar is supported (ie) property name should be matching but we change by special char or cases

4. Validation of properties is supported

5. support scalar datatype as well as objects

mail.properties inside src/main/resources folder

#Scalar datatype

[mail.to=abc@gmail.com](mailto:mail.to=abc@gmail.com)

[mail.from=xyz@gmail.com](mailto:mail.from=xyz@gmail.com)

mail.age=25

mail.first-name=Ram

mail.lastname=Kumar

mail.middlename=T

#Complex datatype

[mail.cc=efg@gmail.com,mno@gmail.com](mailto:mail.cc=efg@gmail.com,mno@gmail.com)

[mail.bcc=uvw@gamil.com,pqr@gmail.com](mailto:mail.bcc=uvw@gamil.com,pqr@gmail.com)

#Nested datatype

mail.credential.username=Ramu

mail.credential.password=abcd

To validate the properties we have to provide spring-boot-starter-validation dependency

              <dependency>

                                           <groupId>org.springframework.boot</groupId>

                                           <artifactId>spring-boot-starter-validation</artifactId>

                             </dependency>

@Validated - to use validation on the properties

@Valid - used to do validation on the nested class properties

@Configuration

@PropertySource("classpath:mail.properties")

@ConfigurationProperties(prefix="mail")

@Data

@AllArgsConstructor

@NoArgsConstructor

@Validated

public class MailConfig {

              @NotNull

    private String to;

              @NotNull

    private String from;

              @Min(value=20)

              @Max(value=40)

    private Integer age;

              @NotNull

    private String firstname;   //Loose binding

              @NotNull

    private String LASTNAME;    //Loose binding

              @NotNull

    private String midddle\_name; //Loose binding

    private String[] cc;

    private List<String> bcc;

    @Valid

    private Credential credential=new Credential();

    @Data

    public class Credential {

               @NotNull

               private String username;

               @Size(min=4,max=8)

               private String password;

    }

}

@RestController

public class MailController {

              @Autowired

              MailConfig config;

              @GetMapping("/mail")

              public String getMailInfo() {

                             return config.getFrom()+" "+config.getFirstname()+" "+config.getBcc()+" "+config.getCredential().getPassword();

              }

}

Springboot Interceptors

     - used to intercept the client request and response

     - Interceptors(applied only to controller) are similar to Filters(only in Servlets), but interceptors are applied to the request that are sending to the controller

     - We have to implement an interface called HandlerInterceptor and override 3 methods

1. preHandle() - perform any operation before sending request to controller

2. postHandle() - perform any operation before sending  response to client

3. afterCompletion() - perform any operation after completing request and response

preHandle() - controller - postHandle() - afterCompletion()

- Create controller prg

@RestController

@Slf4j

public class EmployeeController {

              @GetMapping("/emp")

              public String getEmployeeInfo() {

                             log.info("Inside Controller");

                             return "Employee Information from Controller";

              }

}

- Create interceptor

@Component

@Slf4j

public class TimerInterceptor implements HandlerInterceptor{

              @Override

              public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

                                           throws Exception {

                             log.info("Inside preHandle");

                             request.setAttribute("startTime", System.currentTimeMillis());

                             return HandlerInterceptor.super.preHandle(request, response, handler);

              }

              @Override

              public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,

                                           ModelAndView modelAndView) throws Exception {

                             log.info("Inside postHandle");

              }

              @Override

              public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

                                           throws Exception {

                             log.info("Inside afterCompletion");

                             long totalTime=System.currentTimeMillis()-(long)request.getAttribute("startTime");

                             System.out.println("Total time taken is "+totalTime+" msec");

              }

}

- Configure interceptor for the controller prg

@Configuration

public class EmployeeConfig implements WebMvcConfigurer {

              @Autowired

              TimerInterceptor timerInterceptor;

              @Override

              public void addInterceptors(InterceptorRegistry registry) {

                             //registry.addInterceptor(timerInterceptor);  //this interceptor will be invoked for all controller prg

              registry.addInterceptor(timerInterceptor).addPathPatterns("/emp","/mail");

              }

}

Day – 7

Spring Data JPA

    - Used to persist the data into database

JPA - It is a specification(inbuilt annotation) that interacts with database

Hibernate - ORM tool - It is provider or vendor that provides the implementation of JPA specification

Spring Data JPA - It is an abstraction layer on top of JPA to reduce boilerplate code

Repository interface(marker interface)

    extends

CrudRepository interface(used to perform CRUD operation)

    extends

PagingAndSortingRepository interface(used for pagination and sorting purpose)

    extends

JpaRepository interface(used to perform CRUD and batch operation)

1. Create Springboot project with web, spring data jpa, mysql driver, Lombok dependency

2. Configure db info in application.properties

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto= update

spring.datasource.url=jdbc:mysql://localhost:3306/fullstack

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

3. Create entity class using JPA annotation - uses ORM(Object Relational Mapping) framework which is used to map entity class properties with columns of the db table

Normally we will import javax.persistence.\*, from springboot 3.x there is no more javax, instead we have to jakarta

import jakarta.persistence.\*;

@Entity - used to indicate this class is entity class

@Table(name="stud100") - optional - used to map the entity class properties to the columns of the db table, if it is not given by default it will create the table in name of entity class

public class Student {

    @Id - inidicate it is a primary key column(no duplication, no null values)

    @GeneratedValue(strategy=GenerationType.AUTO(entire db)/IDENTITY(particular table)/SEQUENCE(only oracle)/TABLE) - To indicate PK value to  be autogenerated, if we not provided @GeneratedValue then we have to provide value for PK column

    @Column(name="stuid")

    private Integer studentId;

    @Column(name="sname")

    private String name;

    @Column(name="sage")

    private Integer age;

    @Temporal(TemporalType.DATE/TIME/DATETIME) - use to map java.util.Date to java.sql.Date

    private Date dob;

    private LocalDate doj;

    //private String gender;

    @Enumerated(EnumType.STRING/ORDINAL(0,1,2..)) - used to map enum type to db table

    private Gender gender;

    @Lob - used to map large object to db table like image, audio, video file etc

    @Column(name="stuImage",columnDefination="BLOB")

    private byte[] myImage;

    @Column(name="stumark",scale="5",precision="2") //100.00

    private double marks;

    @Transient - used to ignore the field at the time of persisting

    private boolean status;

}

public enum Gender{

    MALE, FEMALE, OTHERS

}

@Entity

@Table(name="emp100")

@Data

@AllArgsConstructor

@NoArgsConstructor

public class Employee {

              @Id

    private Integer id;

    private String name;

    private String gender;

    private String email;

    private String dept;

    private Double salary;

}

4. Create repository interface

public interface EmployeeRepository extends JpaRepository<Employee, Integer>{

}

5.

@SpringBootApplication

public class SpringDataJpaApplication implements CommandLineRunner {

              @Autowired

              EmployeeRepository empRepo;

              public static void main(String[] args) {

                             SpringApplication.run(SpringDataJpaApplication.class, args);

              }

              @Override

              public void run(String... args) throws Exception {

                             //insertEmployee();

                             //fetchEmployee(102);

                             //fetchAllEmployee();

                             //updateEmployee(102);

                             deleteEmployee(102);

              }

              private void deleteEmployee(int i) {

                             if(empRepo.existsById(i)) {

                                           //Employee e=empRepo.findById(i).get();

                                           //empRepo.delete(e);

                                           empRepo.deleteById(i);

                             }

              }

              private void updateEmployee(int i) {

                             if(empRepo.existsById(i)) {

                                           Employee e=empRepo.findById(i).get();

                                           e.setDept("Sales");

                                           e.setSalary(50000.0);

                                           empRepo.save(e);

                             }

              }

              private void fetchAllEmployee() {

                             List<Employee> l=empRepo.findAll();

                             l.forEach(System.out::println);

              }

              private void fetchEmployee(int i) {

                             System.out.println(empRepo.findById(i).get());

              }

              private void insertEmployee() {

                             //Employee e1=new Employee(100,"Ram","male",[ram@gmail.com](mailto:ram@gmail.com),"HR",20000.0);

                             //empRepo.save(e1);

                             List<Employee> l=new ArrayList<>();

                             Employee e1=new Employee(101,"Sam","male",[sam@gmail.com](mailto:sam@gmail.com),"IT",30000.0);

                             l.add(e1);

                             Employee e2=new Employee(102,"Saj","male",[saj@gmail.com](mailto:saj@gmail.com),"IT",35000.0);

                             l.add(e2);

                             Employee e3=new Employee(103,"Amy","female",[amy@gmail.com](mailto:amy@gmail.com),"Sales",40000.0);

                             l.add(e3);

                             Employee e4=new Employee(104,"Tim","male",[tim@gmail.com](mailto:tim@gmail.com),"HR",36000.0);

                             l.add(e4);

                             Employee e5=new Employee(105,"John","male",[john@gmail.com](mailto:john@gmail.com),"HR",45000.0);

                             l.add(e5);

                             Employee e6=new Employee(106,"Lim","female",[lim@gmail.com](mailto:lim@gmail.com),"IT",42000.0);

                             l.add(e6);

                             empRepo.saveAll(l);

              }

}

Different ways to communicate with db

1. Using predefined methods

      - T save(T t) - store single object in db

      - void saveAll(Iterable i) - store multiple object into db

      - Optional findById(int id) - return single object based on id

      - List<T> findAll() - returns all object

      - boolean existsById(int id)

      - void deleteById(int id)

      - void delete(T t)

      - void deleteAll()

2. Using Custom JPA method

      - derived methods used for fetching the data based on other properties except id

      - name of custom JPA method should be start with findBy/readBy/getBy/queryBy

      - declare the custom JPA method inside repository interface, so spring data JPA will automatically write the logic on behalf of the user

Pattern matching operator - LIKE operator - 2 wild card character

    % - anything

    \_ - single value

public interface EmployeeRepository extends JpaRepository<Employee, Integer>{

     List<Employee> findByDept(String dname);

     List<Employee> readByDeptAndSalaryLessThan(String dname, double sal);

     List<Employee> queryByNameLike(String name);

     List<Employee> getBySalaryBetween(double sal1,double sal2);

     List<Employee> getByNameLikeAndSalaryGreaterThanEqual(String name,double sal);

     List<Employee> findByDeptIsNull();

}

private void customMethod() {

                             //List<Employee> l=empRepo.findByDept("HR");

                             //l.forEach(System.out::println);

                             //List<Employee> l=empRepo.readByDeptAndSalaryLessThan("HR", 30000.0);

                             //l.forEach(System.out::println);

                             List<Employee> l=empRepo.queryByNameLike("R%");

                             l.forEach(System.out::println);

              }

3. Limiting the records based on custom JPA method

      - using first or top keyword

     Employee findFirstByOrderBySalaryDesc();

     Employee findTopByOrderBySalaryAsc();

     List<Employee> findFirst3ByOrderBySalaryDesc();

     List<Employee> findTop5ByOrderBySalaryAsc();

     List<Employee> findFirst3ByDeptOrderBySalaryDesc(String dname);

4. Counting the records using custom JPA

       - using countBy keyword

     long countByDept(String dname);

     long countByNameEndingWith(String name);

     long countBySalaryGreaterThanEqual(double sal);

//long l=empRepo.countByDept("HR");

                             //System.out.println(l);

                             long l=empRepo.countByNameEndingWith("am")

                             System.out.println(l);

5. If we want to perform joins or subqueries then we cant write custom JPA method, in that case we have to write our own queries using @Query

    JPAQL - query the entity class

    SQL - query the table directly

     @Query("select e from Employee e") //JPAQL

     List<Employee> fetchAllEmployee();

     @Query(value="select \* from emp100",nativeQuery=true)  //SQL

     List<Employee> fetchAllEmployee1();

     @Query("select e from Employee e where e.id=100")

     Employee fetchEmployee();

6. Passing parameters to the query - 2ways

1. Positional parameter - using ?1,?2,?3,....

@Query("select e from Employee e where e.name like ?1 and e.salary=?2")

     List<Employee> findEmpByNameAndSalary(String name, double sal);

2. Named parameter - using : followed by any name (ie) :abc,:xyz,:id:,:a,:b

@Param - used to assign value to the named parameter

@Query("select e from Employee e where e.name like :ename and e.salary=:esal")

     List<Employee> findEmpByNameAndSalary1(@Param("ename")String name, @Param("esal")double sal);

@Query(value="select \* from emp100 where name like :ename and salary=:esal",nativequery=true)

     List<Employee> findEmpByNameAndSalary1(@Param("ename")String name, @Param("esal")double sal);

7. If we want to perform any DML operation(insert/update/delete) using @Query, apart from @Query we have to provide 2 more annotations called @Modifying and @Transactional

     @Modifying

     @Transactional

     @Query("update Employee e set e.salary=e.salary+e.salary\*:percent/100 "

                             + " where e.dept=:dname")

     int updateSalary(@Param("dname")String dname,@Param("percent")double percentage);

8. Instead of writing queries in repository interface we can write queries in entity class and refer by their name, which is called as Named query

    JPAQL - @NamedQuery

    SQL - @NamedNativeQuery

     List<Employee> fetchEmployeeBySalary(double sal);

     List<Employee> fetchEmployeeBySalary1(@Param("sal")double sal);

@Entity

@Table(name="emp100")

@Data

@AllArgsConstructor

@NoArgsConstructor

@NamedQuery(name="Employee.fetchEmployeeBySalary",

            query="select e from Employee e where e.salary=?1") //JPAQL with positional parameter

@NamedNativeQuery(name="Employee.fetchEmployeeBySalary1",

             query="select \* from emp100 where e.salary=:sal",

             resultClass=Employee.class)  //SQL with Named parameter

public class Employee {

              @Id

    private Integer id;

    private String name;

    private String gender;

    private String email;

    private String dept;

    private Double salary;

}

9. For sorting purpose we can use Sort class instead of using orderBy clause

//List<Employee> findByDeptOrderBySalaryDesc(String dname);

List<Employee> findByDept(String dname,Sort s);

List<Employee> l=empRepo.findByDept("HR",Sort.by("salary").descending());

l.forEach(System.out::println);

List<Employee> l=empRepo.findByDept("HR",Sort.by("name").ascending());

l.forEach(System.out::println);

10. Consider we have entity class with 100 properties, but we need to display only some 10 properties, but if we provide List<Employee> then it will return all 100 properties

    Consider we have emp100 table, after performing some operation we want to display only specific information like total count of male and female employees like

mysql> select count(gender) as "Totalcount", gender from emp100 group by gender;

+------------+--------+

| Totalcount | gender |

+------------+--------+

|          5 | male   |

|          2 | female |

+------------+--------+

2 rows in set (0.00 sec)

@Query(value="select count(gender) as "Totalcount", gender from emp100 group by gender",nativeQuery=true)

List<Employee> countGenderWise(); --wrong - it displays all employee info

@Query(value="select count(gender) as "Totalcount", gender from emp100 group by gender",nativeQuery=true)

List<Object[]> countGenderWise(); --correct - but it is not a good practise to return as                                                                                                                                                                                                                                            Object[]

@Query(value="select count(gender) as "Totalcount", gender from emp100 group by gender",nativeQuery=true)

Map<Integer,String> countGenderWise(); --wrong - because Spring data jpa dosent support                                                                                                                                                                                                                                              Map as return type

Constructor method - used when we want to display only specific properties from entity class

- create separate class based on what properties u want to display

@Data

@AllArgsConstructor

@NoArgsConstructor

public class GenderCount {

   private long count;

   private String gender;

}

- Declare method in repo intf

@Query("select new com.pack.SpringDataJPA.GenderCount(count(e.gender),e.gender) from Employee e group by e.gender")

     List<GenderCount> countGenderWise();