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");

              }

}