Springboot:

First what is Spring and what is Spring boot?

Spring: Spring is java framework, used to build enterprise ready application.

if we are developing application using spring framework, we need to do lot of configuration. for example we need to use hibernate in our spring application then we need to do lot of configuration and we need to add so many libraries and so many jar files based on our requirement. due to this developer will need to focus on both configuration and convention.

this is very hectic for developer. so that developers decided to develop something easily that will allows us to only concentrate on convention not configuration.for that they have created the springboot.

Springboot:

Spring boot is an extension to spring framework. its not an different framework. Springboot itself using spring framework internally.

Benefits of Springboot:

-> Spring boot provides us to build Rapid application development(means what ever application we are developing we can develop easily by using spring framework)

-> Managing dependencies(Springboot will provides lot of starter templates -> in that starter template all the dependencies will include for that particular task -> for example if we want to work with JDBC then we have Springboot-starter-JDBC, Here all the dependencies has included to perform that particular task(giving connection to DB everything)

if we want to work with JPA --> springboot-starter-JPA available (here all the dependencies are included to perform particular task)

and we have springboot-starter-test --> by using this we can test our application(using JUnit and Mockito)

Ex: If we want to include hibernate in our spring application, we need to do configuration and if we want to include JDBC we need to do configuration. means we need to do lot of configurations. i will become very hectic for developers. for that spring boot came.

-> Springboot provides AutoConfiguration.(means if we want to use hibernate in our application just we need to add dependency, then remaining all configurations Spring only will take care)

-> Embedded Server(means in traditional way if we want to deploy our application in server.. first we need to create Application(code) and then we need to create WAR file for our entire application and then we need to deploy that WAR file in server(Tomcat, JBoss...etc). But by using springboot we wont create War file, just we will create Jar file, and server(Tomcat,JBoss...etc) will be embedded in that Jar file alone. so we can say by using springboot we can develop production ready application.

Dependency Injection:

In previous spring framework also we have discussed dependency Injection

Means for example we have created student class and we are creating object of student class by using (Student s = new Student()), here we are creating object manually. and if we have 100 classes, and all the classes are inter related, then creating object manually by using new keyword its very difficult and our application will become tightly coupled. so Dependency Injection concept came. Here we wont create object manually by using new keyword, while creating classes only spring will store all the classes(beans) and objects in container. just we need to tell spring like this class needs object of that class and that class needs object of this class like that, remaining all the work spring only will take care(means if we want object of particular class spring only will give that object we wont create object manually).

Inversion of control:

above example only we can take, in that example we wont create any object, just spring only will create and it will give that object if we want. here we will give control to the spring for creating objects and configurations everything. we are not doing any configurations. that giving control to spring framework is called Inversion of Control.

Spring Initializer:

By using Spring.io we can create project and also we can add all dependencies while creating only.

to do this we can go google and create

SpringBoot starters:

so many springboot starters available in Spring boot. based on our requirement we can use those starters.

like..

spring-boot-starter-web --> in this only tomcate and everything got configured

spring-boot-starter-test --> this is for testing

....

....

....

etc

just we have added dependencies here while creating project. remaining all configurations spring only has taken care. spring.io itself created pom.XML file and given all configurations. thats the advantage of springboot.

After creating springboot project by default @SpringBootApplication annotation will be there.if we open that annotation, we can see all the annotations availabe for that project.

SpringApplication.run(SpringFrameworkDemoApplication.class, args); >> this method will Start our springboot application

and we can exclude some classes which we wont use in our application(example: if we dont need hibernate in our application we can exclude that) >> by using EnableAutoConfiguration Annotation.

Component Scan >> All the classes object present in the particular package will be scanned by ComponentScan and stores all classes and objects inside container. if we have different package then we need to create component scan for that package also.

@Controller:

if we create class and if we give @Controller annotation. then that class will acts as controller. inside that controller @Component Annotation will be there. by default it will be controller and Component of container also.

@RestController:

if we are creating Rest API then we should made that class as RestController, not normal Controller.

RestController always returns ResponceBody not JSP page(becuase JSP is old technology)

@RequestMapping:

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

|-> here we have given end point, if we call this end point below method of that end point will get exceuted

|-> and there we have made that method as GET (same as get in JSP)

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

public String Hello(){

return "hello world";

}

Properties file:

Note: By default Springboot will provide all the configurations, if we want to change any configuration we can do with the help of properties file(which will be present in our project only)

for example: by default our application will be autoconfigured to Tomcat server with 8080 port. if we want to change the configuration i.e.., if we want to change port number from 8080 to 8082. it will be possiblke with the help of properties file.

server.port = 8082 .. by using this property we can change port of our tomcat server.

After giving this.. go to google >> type localhost:8082/ >> we will get output

To remember all these properties to change configuration it will be difficult. so we have spring boot documentation, in that which property is for which Configuration got clearly mentioned. by refering that documentation we can make use of those configurations.

@GetMapping:

instead of using RequestMapping and making method as GET by using RequestMethod.GET, simply we can use GetMapping annotation and we can give end point

@GetMapping("/")

if we want to use post method or delete method based on our requirement we can use those annotations.

Note:

we can run our spring boot application not only in IDE, we can run in Terminal(command prompt also) by using following command

SpringBoot Devtools:

when we creating appliation, if we want to made any changes we need to restart our server for every change. it will be difficult

so springbooot has provided one dependency springboot devtool, if we use this dependency we dont need to restart server for every change. it will restart automatically when we made any changes.

to add that devtool dependency, we need to go spring initializer and we need to create one sample project and add dependency by searching Devtools, and then just copy that particular dependency in pom.xml and paste that dependency in our pom.xml file.

After adding DevTool Dependency, we need to go settings in INtellij and

1st step: file->settings->build,execution,deployment. Go to ->compiler->build project automatically.

2nd step: File -> Settings... -> Advanced Settings then check Allow auto-make to start

by following these two steps, we can enable auto restart our server in IntelliJ. so that for every code changes we dont need to restart server. automatically it can restart.

we can achieve this by using Springboot DevTools

Architecture & Example:

REST API

GET POST PUT DELETE

|

service layer (this is bussiness layer, all bussiness logic will presnt in this layer)

|

Data Access/Repository Layer (this layer deals with database operations)

|

Database (this is database, now we are using H2 Databse, once we can familiar with everything we can use Mysql database)

Adding dependency H2 and JPA:

to add those dependencies, just go to Spring initializer and click on add dependencies and type h2 and add, then type JPA and add.-> two dependencies has added.

and we need to configure H2 database in springboot application, to do that we should add configuration in properties file

spring.h2.console.enabled=true

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=password

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

these configuration will be available in google.

After adding these configuration in properties file, simply we can run our application.

and then go to google type

localhost:8082/h2-console >> we will get h2 database console >> in that console we need to fill all details as per above configuration and give password and then test connection >> we should get Test Successfull. and then give connect >> it will connect to H2 database.

Creating components:

@Entity: if we make any class as entity then it can deal with database.--> if we make entity then that class acts as a table,(that class name = table name)

@Id: Every table should have primary key right >> to make primary key we can use ID annotation

@GeneratedValue(strategy = GenerationType.AUTO): we need to define how we have created primary key(means we have made that primary key as auto generated. for that we should use that annotation)

@service >> to make any class as service we should use this annotation>> bussiness logic

@Repository: To make class as Repository we should use this >> to deal with data base

@RequestBody: what ever the data we are receiving as request body, if we receive JSOn data, this annotation will convert that JSON data into java object(Department object)

@Autowired : if we want to create object of other class inside one class we will use Autowire annotation. we cannot create object using new keyword, because it will give tight coupling

we can acheive this AutoWired in 2 ways

1.contructor

2.setter

in spring boot we won't create any tables in database. JPA itself will create table for us

Testing API's using rest client:

Rest Client examples are: Postman, insomnia, vs code....etc. we have so many rest clients.

To install postman refer videos.

After installing postman create one request Get and post based on our requirement.

path: http://localhost:8080/lakshmi

and go to body -> raw -> text -> JSON

(For me here post method not working in H2 database, so I am using Mysql database. To use mysql database we need to add mysql dependency in pom.xml file and we should give configuration in application.properties file)

Mysql Configuration:

spring.datasource.url=jdbc:mysql://localhost:3306/dep  
spring.datasource.username=root  
spring.datasource.password=JesusLife@1998  
spring.jpa.hibernate.ddl-auto=update  
  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect

in json we can send request, to send request we need to use following.

{

   "departmentCode": "IT-56",

    "departmentName": "Engineering",

    "departmentLocation": "Bangalore",

    "departmentBranch": "CSE"

}

Post: <http://localhost:8080/lakshmi>

Get: <http://localhost:8080/lakshmi>

Both Get and post working in Mysql database.

**@PostMapping:**

@PostMapping  
public Department01 saveDepartment(@RequestBody Department01 department){  
 return departmentService.saveDepartment(department);

In PostMapping we have used @RequestBody -> this annotation will convert our JSON file into java object(1.e., Department Object)

**@GetMapping:**

@GetMapping  
public List<Department01> fetchAllDetails(){  
 return departmentService.fetchAllDetails();  
}

If we use getMapping Annotation then that method will act as get method. This method will return All department details, so we made return type as List.

**@RequestMapping("/lakshmi")**

Normally in GetMapping and PostMapping method also we will give path. Same like by using RequestMapping we can give path, by using that path we can perform Get and Post Methods by using Postman.

**Fetching Data by ID:**

And if we want to Get particular department details, then we need to following code

@GetMapping("/lakshmi/{id}")  
public Department01 getDepartmentByID(@PathVariable("id") Long departmentId){  
 return departmentService.getDepartmentById(departmentId);  
}

In above code we have given {id} -> here we can give the particular ID which we want to fetch details.

Here we have used **@PathVariable("id") ->** This path variable will set the ID value to departmentID.

|-> This ID will come from postman

**Deleting Data:**

we can delete department by ID by using @DeleteMapping Annotation.

@DeleteMapping("/departments/{id}")  
public String deleteDepartmentByID(@PathVariable("id") Long departmentID){  
 departmentService.deleteDepartmentByID(departmentID);  
 return "Department deleted Successfully!!";  
}

**Updating Data:**

For Updating we will use @PutMapping Annotation.

@PutMapping("/departments/{id}")  
public Department01 updateDepartment(@PathVariable("id") Long departmentID, @RequestBody Department01 department01){  
 return departmentService.updateDepartment(departmentID,department01);  
}

Here we need to update department details for particular ID, so we need both department ID and Department Object which needs to be updated.

So that in above code we have taken two parameters (DepartmentID and department01) and we have used @ PathVariable("id") and @RequestBody Annotations because,

@PathVariable(“id”) -> This Annotation will assign id value to the departmentID parameter in above code

@RequestBody -> This annotation will convert JSON data into java object (department01)

And we need to make one condition here, If any null or “” values comes from JSON, that null values we can’t update in our Department object. To do so we can use below code.

@Override  
public Department01 updateDepartment(Long departmentID, Department01 department01) {  
 Department01 deDB = departmentRepository.findById(departmentID).get();  
 if(Objects.*nonNull*(department01.getDepartmentName()) && !"".equalsIgnoreCase(department01.getDepartmentName())){  
 deDB.setDepartmentName(department01.getDepartmentName());  
  
 }  
 if(Objects.*nonNull*(department01.getDepartmentLocation()) && !"".equalsIgnoreCase(department01.getDepartmentLocation())){  
 deDB.setDepartmentLocation(department01.getDepartmentLocation());  
 }  
 if(Objects.*nonNull*(department01.getDepartmentCode()) && !"".equalsIgnoreCase(department01.getDepartmentCode())){  
 deDB.setDepartmentCode(department01.getDepartmentCode());  
  
 }  
 if(Objects.*nonNull*(department01.getDepartmentBranch()) && !"".equalsIgnoreCase(department01.getDepartmentBranch())){  
 deDB.setDepartmentBranch(department01.getDepartmentBranch());  
  
 }  
 return departmentRepository.save(deDB);

Department01 deDB = departmentRepository.findById(departmentID).get();

In above line we have fetched the Department object for particular ID. And we have stored that object in deDB.

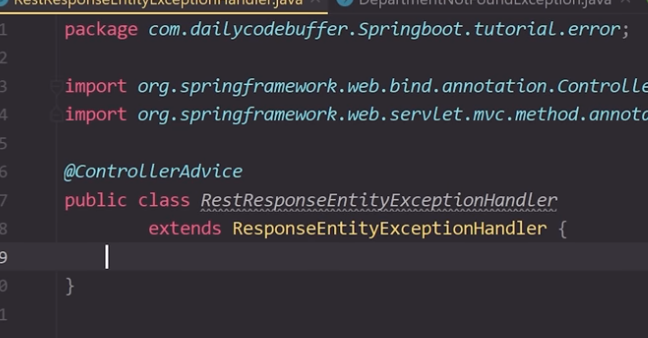
if(Objects.*nonNull*(department01.getDepartmentName()) && !"".equalsIgnoreCase(department01.getDepartmentName())){  
 deDB.setDepartmentName(department01.getDepartmentName());

In this code we have checked the Object (Coming from JSON) that we need to update in DB, wheather it has Null values or “”. If any null values or “” available then we can’t update that data, just we need to skip. If Object is not null and “” then we can update that data in DB.

**@ControllerAdvice:**

Which class wants to handle all the exceptions in whole application, then those class we made as @ControllerAdvice.

If any exception occurs, then that exception will be handled by ResponseEntityExceptionHandler



And after handling exception we need to send response as well right, for that we will use @ResponseStatus annotation. In this crud operations we have created one one exception that is DepartmentNotFoundException. We need to create one method that method will handle this particular exception and pass the response as well.

public ErrorMessage department01NotFoundException(){  
  
}

here we have created method and we made return type as ErrorMessage, and we need to create ErrorMessage class in that class we need to mention what response we need to return clearly.

In above code ErrorMessage we should send as response, so we need to wrap that into responseEntity.

public ResponseEntity<ErrorMessage> department01NotFoundException(){  
  
}

@ControllerAdvice  
@ResponseStatus  
public class RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler {  
 @ExceptionHandler(Department01NotFoundException.class)  
 public ResponseEntity<ErrorMessage> department01NotFoundException(Department01NotFoundException exception, WebRequest request){  
 ErrorMessage message = new ErrorMessage(HttpStatus.*NOT\_FOUND*, exception.getMessage());  
  
 return ResponseEntity.*status*(HttpStatus.*NOT\_FOUND*).body(message);  
  
 }  
}

In above code we have @ControllerAdvice annotation -> so this class will handle all exceptions in this whole application. And then we have used @ResponseStatus annotation -> after handling exception we should return status right for that we will use that annotation

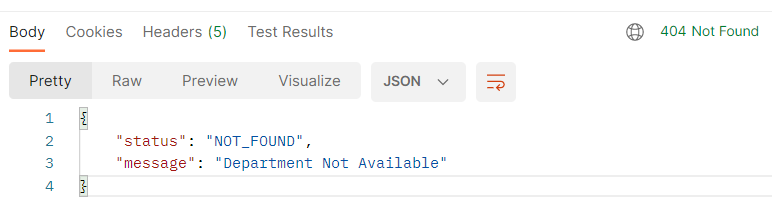
And then to handle particular exception we can create particular method, in this application we have created one exception Department01NotFoundException -> to handle this particular exception we have created one method department01NotFoundException -> and we need to send response in particular format for that we have created one class ErrorMessage, in that class we have created 2 variables status and message. That ErrorMessage class we have used as return type in this method.

But we have wrapped that ErrorMessage with ResponseEntity -> because we should send this as Response. And then we passes 2 parameters in this method -> one is DepartmentNotFoundException and another one is WebRequest -> present we are not requesting anything -> but in future if we want we can make use of this webRequest.

And then we have created object for that errorMessage -> and we passed status and message of that exception and then we have wrapped that status and exception with ResponseEntity and we have returned.

Note: we can create multiple methods for multiple exceptions. And also we can create one method and handle multiple exceptions also.

Output:



And the main advantage in spring boot is if we want to change the database we don’t need to modify everything, just we need to add configurations(MYSQL) in properties file and we need to add mysql driver dependency in pom.xml file.

Unit Testing:

To do unit testing we need to add JUnit and mockito

Mocking:

we have different layers, service layer, controller layer and repository layer. If we want to test only controller layer but controller layer is dependent on service layer(dependent means the we have used methods and functionality of service layer inside controller layer). While testing it will go to service layer right. For that we should mock service layer methods. if any service layer methods or functionalities came we should return something without going to service layer.

Similarly while we are testing service layer, service layer will dependent on the repository layer right -> so we should mock the repository layer. If any repository layer methods or functionalities came it should return something without going to repository layer. It is called mocking.

Unit Testing Service Layer:

Right click on service layer -> go to test -> it will create serviceTest layer.

@SpringBootTest  
class DepartmentServiceTest {  
  
  
}

@springBootTest: by using this annotation spring boot will know it will be particular Test class

And we can test all the methods present in the service layer and while testing we sould give unique method names. By seeing method name only we should find out what will this method will do.

assertEquals() -> this method will take 2 parameters

@BeforeEach -> if we use beforeEach this particular method will be called for each test case. For example if we have 5 test cases -> this method will be called 5 times.

@BeforeAll -> this method will be called only once for all the test cases. If we have 5 test cases this method will be called once for all 5 test cases.

@Test  
public void whenValidDepartmentName\_thenDepartmentShouldReturn(){  
 String departmentName = "IT";  
 Department01 found = departmentService.getDetailsByName(departmentName);  
 *assertEquals*(departmentName, found.getDepartmentName());  
}

In above code @Test annotation from Junit Jupiter package we used. If we use this annotation then that method will act as test method.

And we need to mock repository layer (because while testing service layer it has dependent on respository layer right).

@MockBean  
private DepartmentRepository departmentRepository;

@MockBean -> it will mock particular layer

@Builder -> Normally if we have 10 parameters, if we want to create object using only 2 parameters then we need to create one constructor which accepts only 2 parameters right like wise if we have different number of arguments we can create different no of constructor this was boring right and also code length will increase. So builder annotation came. -> if we use builder we can create object with multiple number of arguments as our wish.

@DisplayName("get data based on valid name")

@DisplayName -> this annotation will display the name which we given here in output

@Disabled -> If we have multiple test cases, and if we want to disable particular test case then we can use this annotation.

@Test  
@DisplayName("get data based on valid name")  
@Disabled  
public void whenValidDepartmentName\_thenDepartmentShouldReturn(){  
 String departmentName = "IT";  
 Department01 found = departmentService.getDetailsByName(departmentName);  
 *assertEquals*(departmentName, found.getDepartmentName());  
}

Unit Testing Repository Layer:

Testing repository layer little bit difficult, because it will deal with database right. So while testig we can’t store any junk data.

We can test repository layer by using many ways, first we can test repository layer by using @DataJPATest Annotation. If we use this annotation it will keep data until testing after testing it will flush out all the data. So that there will be no junk data available

And we can also create database Test and in memory database also for testing purpose.

@DataJpaTest  
class DepartmentRepositoryTest {

Here we are using @DataJpaTest Annotation -> this will keep data until testing after testing it will flush out all data

@DataJpaTest  
class DepartmentRepositoryTest {  
  
 @Autowired  
 private DepartmentRepository departmentRepository;  
 @Autowired  
 private TestEntityManager entityManager;  
 @BeforeEach  
 void setUp() {  
 Department01 dep = Department01.*builder*()  
 .departmentId(1L)  
 .departmentName("mechanical")  
 .departmentLocation("Hyderabad")  
 .departmentBranch("gachibouli").build();  
 entityManager.persist(dep);  
 }  
 @Test  
 public void whenFindByID\_thenReturnDepartment(){  
 Department01 dep01 = departmentRepository.findById(1L).get();  
 *assertEquals*(dep01.getDepartmentName(), "mechanical");  
  
  
  
 }

Unit Testing Controller Layer:

@WebMvcTest(DepartmentController.class)  
class DepartmentControllerTest {  
  
 @Autowired  
 private MockMvc mockMvc;  
 @MockBean  
 private DepartmentService departmentService;  
  
 private Department01 department01;  
 @BeforeEach  
 void setUp() {  
 department01 = Department01.*builder*()  
 .departmentId(1L)  
 .departmentName("IT-05")  
 .departmentBranch("hyderabad")  
 .departmentLocation("dlf")  
 .departmentCode("123")  
 .build();  
  
  
 }  
  
 @Test  
 void saveDepartment() throws Exception {  
 Department01 inputDepartment = Department01.*builder*()  
 .departmentName("Non IT")  
 .departmentBranch("Civil")  
 .departmentLocation("pune")  
 .departmentCode("Civil-123")  
 .build();  
 Mockito.*when*(departmentService.saveDepartment(inputDepartment)).thenReturn(department01);  
 mockMvc.perform(*post*("/lakshmi")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .content("{\n" +  
 " \"departmentName\": \"Non IT\",\n" +  
 " \"departmentBranch\": \"Civil\",\n" +  
 " \"departmentCode\": \"Civil-123\",\n" +  
 " \"departmentLocation\": \"pune\"\n" +  
 "}")  
  
 )  
 .andExpect(*status*().isOk());  
  
 }  
  
 @Test  
 void getDepartmentByID() throws Exception {  
 Mockito.*when*(departmentService.getDepartmentById(1L)).thenReturn(department01);  
  
 mockMvc.perform(*get*("/lakshmi")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 )  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$.departmentName").value(department01.getDepartmentName()));  
 }  
}

Note: this code correct but not getting output, just refer process for interview purpose

Adding Configurations in properties file:

Application.properties file

spring.datasource.url=jdbc:mysql://localhost:3306/dep01  
spring.datasource.username=root  
spring.datasource.password=JesusLife@1998  
spring.jpa.hibernate.ddl-auto=update  
  
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect  
spring.batch.job.enabled = false  
welcome.message = welcome to springboot tutorial

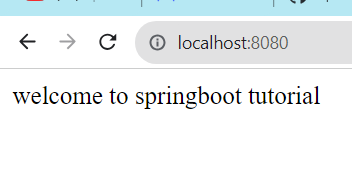
here we have created configuration in key value pair.

HelloController.java

@RestController  
public class HelloController {  
 @Value("${welcome.message}") // this line will take value from properties file  
 private String WelcomeMessage;  
  
 @GetMapping("/")  
 public String Hello(){  
 return WelcomeMessage;  
  
 }  
}

Here we have created Hello Controller and we got values from properties file and it will display in google

Output:



Yaml File:

In Yaml file -> we can add configurations

Advantage in Yaml is it provides human readable format and it will remove all the redundant data from Yaml file.

For Example in properties file we will add below configurations

spring.datasource.url=jdbc:mysql://localhost:3306/dep01  
spring.datasource.username=root  
spring.datasource.password=JesusLife@1998

In above code spring.datasource is repeating. In Yaml it will remove that repeating code and provide human readable format.

And we can convert our application.properties configuration into yml file configuration by using some online tools.

Type properties to yaml in google and copy our properties configuration and paste it into online tool it will give yaml code.

Springboot profiles:

What is profile in springboot?

Normally we have created application(Springboot serive, repository .. etc). these all applications we will deploy on different environments right for example we will deploy out application in production, AWS dev….

Like wise in yaml file we have given configuration for MySql but we can use different environments also

We can acheive this by using profiles.

Application.yaml file

spring:  
 profiles:  
 active: qa  
---  
spring:  
 spring:  
 config:  
 activate:  
 on-profile: dev  
 datasource:  
 password: JesusLife@1998  
 url: jdbc:mysql://localhost:3306/dep01\_dev  
 username: root  
 jpa:  
 hibernate:  
 ddl-auto: update  
 properties:  
 hibernate:  
 dialect: org.hibernate.dialect.MySQLDialect  
welcome:  
 message: welcome to springboot tutorial  
  
---  
  
spring:  
 spring:  
 config:  
 activate:  
 on-profile: prod  
 datasource:  
 password: JesusLife@1998  
 url: jdbc:mysql://localhost:3306/dep01\_prod  
 username: root  
 jpa:  
 hibernate:  
 ddl-auto: update  
 properties:  
 hibernate:  
 dialect: org.hibernate.dialect.MySQLDialect  
welcome:  
 message: welcome to springboot tutorial  
  
---  
  
spring:  
 spring:  
 config:  
 activate:  
 on-profile: qa  
 datasource:  
 password: JesusLife@1998  
 url: jdbc:mysql://localhost:3306/dep01\_qa  
 username: root  
 jpa:  
 hibernate:  
 ddl-auto: update  
 properties:  
 hibernate:  
 dialect: org.hibernate.dialect.MySQLDialect  
welcome:  
 message: welcome to springboot tutorial

Running spring boot with multiple profiles:

We can deploy our application in qa, dev or prod environment. But how can we deploy our application -> we can deploy our application in the form jar file right. For that we need to create jar file for our application.

To create jar file

Go to terminal -> type mvn clean install -> (clean will clean the target directory and install will install the package)

This command will successfully execute when we completed all the code and test cases succesffully.

* Cd target (it will go into our target directory)
* Type dir (it will give our application jar file)
* And now we need to deploy this jar file. Here we are using springboot so we don’t need to install external server. Servers will be embedded internally in springboot
* Java –jar SpringbootCrudOperation -1.0.0.jar –spring.profiles.active=prod (in last line we are giving environemnet, in that particular environment our jar file will deploy)

Spring boot Actuator:

Whenever we deploy our application in any of the environment for example if we deploy in prod environment we should monitor that application. Wheather our application is healthy and memory utilization everything we should monitor. So here spring boot actuator provides us to monitor our application health and different metrices.

For that we need to add dependency for actuator in pom.xml file

After adding dependency if we run our application we can able see in logs as Exposing 1 endpoint(s) beneath base path '/actuator'

After that go to google and type http:8080/actuator -> then it will provide u following



We have lot of actuator end points, but spring boot by default disabled everything. It has enables important things(health, info end points) only. But we can able to enable those end points also

To endable end points we can add below code in application.yml file

management:  
 endpoints:  
 web:  
 exposure:  
 include: "\*"

After writing above command we will get 13 end points will get enable. They are cache, beans , health, info, loggers…. Etc

Custom actuator endpoint:

We can able to create end point also

And also we can exclude particular end point as well