**https://www.geeksforgeeks.org/how-to-set-upstream-branch-on-git/**

## Spring

Spring is java based framework used to build enterprise application.

There are many modules in spring like - Spring-core, spring-mvc, spring-web, spring-batch, spring-data, spring-jpa

But in spring we have to do lot of configuration in order to overcome from this situation developers created spring-boot where configurations will be done automatically.

## Spring boot

is an extension of spring framework. It uses spring internally.

## Benefits of spring boot

1. Rapid application development
2. Managing dependencies is easy as we use starter templates e.g. JPA templates, JDBC templates, Test templates. We can directly use these templates and start working on particular modules.
3. Configuration done automatically for modules for which we want to develop application. Just add starter templates and automatically it takes care of configuration.
4. Embedded server – We do not need external server to deploy war file. We are creating jar file here and our server is embedded with that jar file. We can directly run this jar file in any of the environment. We are always production ready.
5. To create micro services spring boot is the default option.

## Dependency Injection pattern

Traditional way to create objects of different classes in the application is like create object manually for all classes using new keyword.

e.g. Student s = new Student();

But when we have thousands of classes then it is not good idea to create objects like this.

Then comes **Inversion of Control** in picture – It means we are giving the control to framework to create the object for us. E.g. here we are giving privileges to create object for us to springboot.

**Dependency injection is pattern which helps us in achieve inversion of control.**

So when spring boot application context started, object for all the classes get created.

And all those particular beans (objects) stored in one of the container, and whenever we want that bean or object, spring will return us directly.

## Spring Initializr

It is a tool which helps us to create bare bone spring boot application with bare minimum.

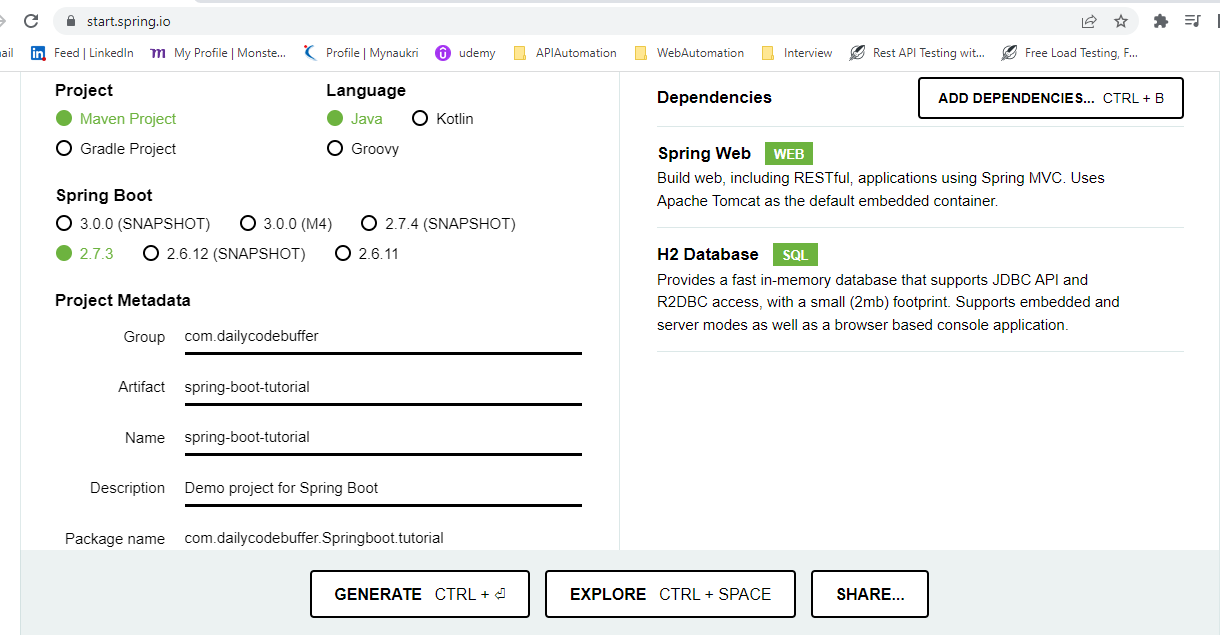
Open url – start.spring.io

Then select Build tool Maven, language java, Spring boot version 2.4.5 stable, packaging jar, java version 11 or 8, project metadata group(com.dailycodebuffer), artefact(spring-boot-tutorial), name, description, package name (com.dailycodebuffer.Springboot.tutorial)

Add dependencies –

Spring web, H2 Database(in memory database)

Click on Generate, it will download project.

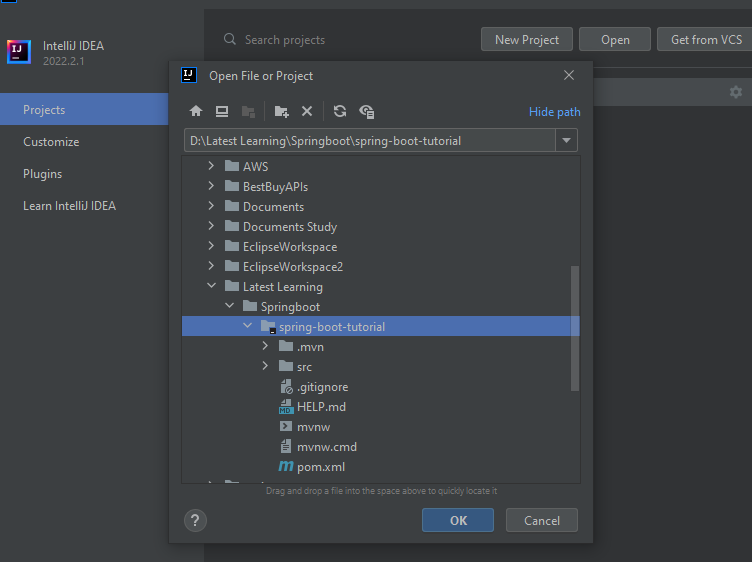
****

Downloaded project - D:\Latest Learning\Springboot\spring-boot-tutorial.zip

## Springboot IDE

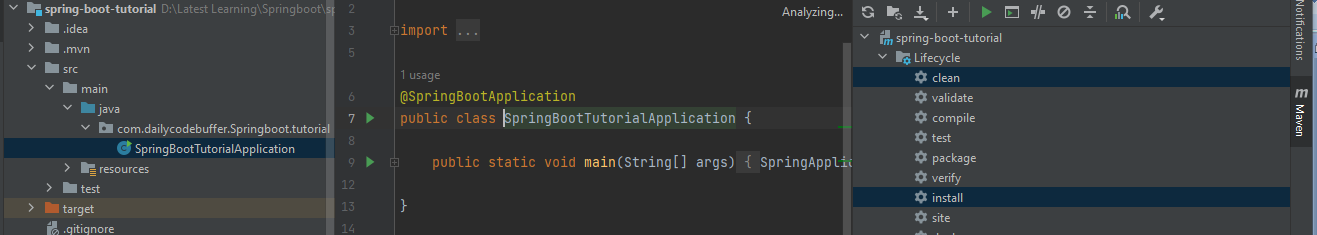
We can use any editor like Spring tool suite (STS), vs-code, eclipse, intellij idea etc with help of plugins

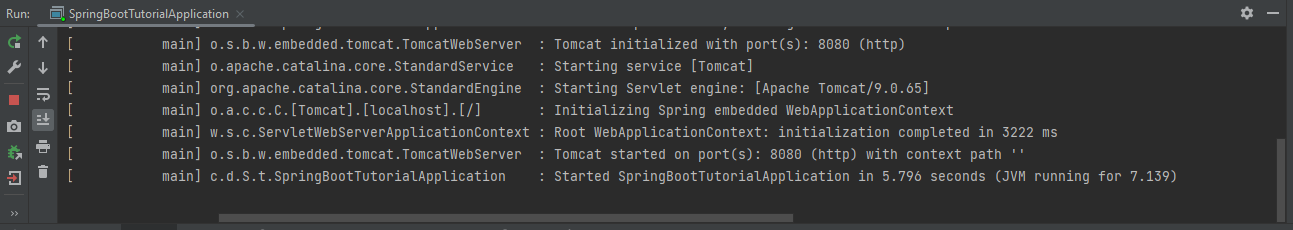
## Open project



## Run project

Do mvn clean install. Then run application from main class

****

****

## Run from command prompt

Navigate to the root of the project via command line and execute the command

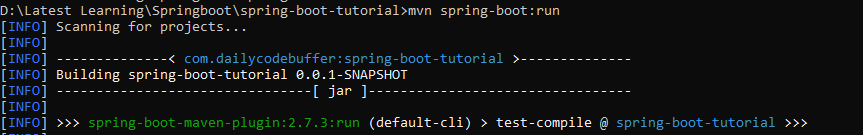
mvn spring-boot:run.

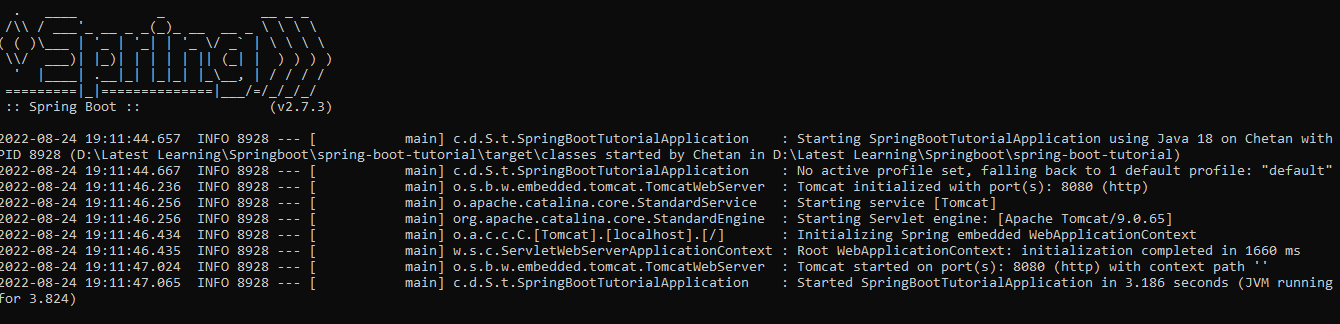
Also we can save these two in maven configurations in IDE like

Clean install

Spring-boot:run

We can also use intelliJ idea terminal also to run these commands





## Spring boot starters

In pom.xml we have parent tag that is our starter parent.

And we have added dependencies for web, h2, which is also our starter. If we click on any of these dependencies from pom.xml then we can see that these starters contain multiple dependencies.

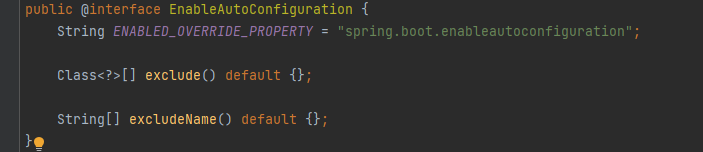
## Key Concepts

@SpringBootApplication

This annotation contains many annotations like

@SpringBootConfiguration 🡪 It tells this is main application start point  
@EnableAutoConfiguration 🡪 It tells to do auto configuration of application

This automatically add all the classes in application for which we added dependencies and those are required but for instance if we not need some classes then we can open below inteface and mention those particular classes under exclude tag.



@ComponentScan 🡪 It scans all the components available in spring boot application and component will be added to spring container when application started

SpringApplication.*run*(SpringBootTutorialApplication.class, args)

*🡪 this method start our application*

## Creating Simple API

Create a package *Controller* and a class named as *HelloControlle*r.

Annotate this class with **@Component** so that this class will be in spring radar and whenever application context starts , the class which we created new will be added to container. So whenever we need this class, spring can give us.

But since this class is a controller class so instead of @Component we will use @Controller. **@Controller has in built @Component**

Again since we are creating Rest application, so instead of using @Controller we can use **@RestController**. It tells that it is controller and has component annotation **and always return ResponseBody** since it is a RestController.

Now create a method which returns us String.

public String helloWorld(){  
 return "Welcome to Daily Code Buffer!!";  
}

Now i want this method to be executed when I hit some endpoint. To achieve this we use **@RequestMapping**

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

Instead of using @RequestMapping and define RequestMethod as Get, we can directly use **@GetMapping.** This reduces the verbose

@GetMapping("/")

Similarly for Post, Delete and other methods we can use these direct annotations available.

## Change Configuration of port

By default it runs on 8080, but we can change it from **application.properties**.

Add property as 🡪 server.port = 8082

[Common Application Properties (spring.io)](https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html)

**Now Run our application and access url in browser** 🡪

<http://localhost:8082/> “/ “ is the path which we provided in our Request mapping

## Devtools dependency

This we can add in our project so whenever there is a changes springboot will detect those changes and restart our application.

To add it open spring intializr and search for sprig boot dev tools dependencies. Click on explore and copy dependency tag and paste it there in our pom.xml.

In IntelliJ Idea to make it work we have to do couple of changes.

Open Actions – shortcut Ctrl+shift+A

Search for Registry and open it.

Search for property –compiler.automake.allow.when.app.running and check it.

**In latest version** go to Advanced Settings and Compiler setting select Allow auto-make to start even if developed application is currently running. – Apply and OK

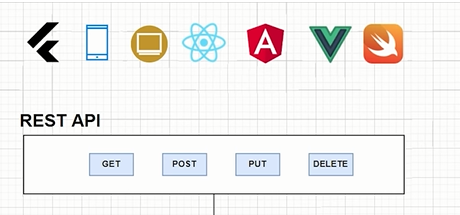
Also open settings- Build - compiler setting – Build project automatically

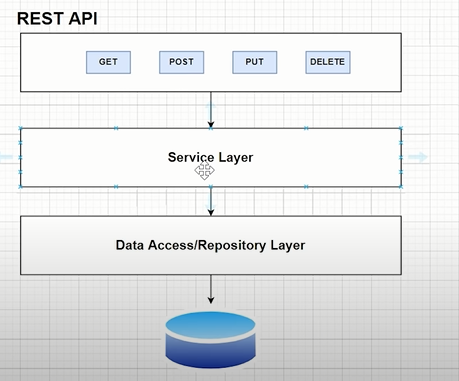
Now Run our application

Try to make some changes.

Observe server restart automatically with latest changes.

## Architecture & Example





We will create Department API’s for Get all department, get 1 department, update one department, add new department, delete department.

We will add all these in our **controller layer**. This layer is just to add the request and get the response back.

In **service layer** we will be writing all our logic. It is called as business layer.

From service layer we call **Data Access/Repository layer**. This is responsible to interact with database. It handle all the database operation. For this layer we need **dependency Spring data JPA**.

**Database** – First we will use in memory database H2. Then after that we will switch to MySql db.

We will access H2 in browser. So we will do some configuration to access it.

## Access H2 database

Add JPA dependency in POM.xml

Open application.properties file and add jpa and h2 properties. We can add these in yaml also refer below article.

spring.h2.console.enabled=true  
spring.datasource.url=jdbc:h2:mem:dcbapp (dcbapp is a database name)  
spring.datasource.driverClassName=org.h2.Driver  
spring.datasource.username=sa  
spring.datasource.password=password  
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

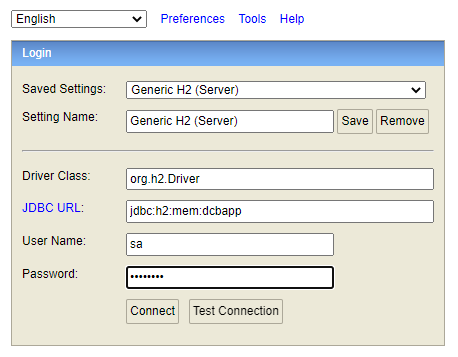
#spring.datasource.url=jdbc:h2:file:/data/demo (If want to store it in file)

<https://www.baeldung.com/spring-boot-h2-database>

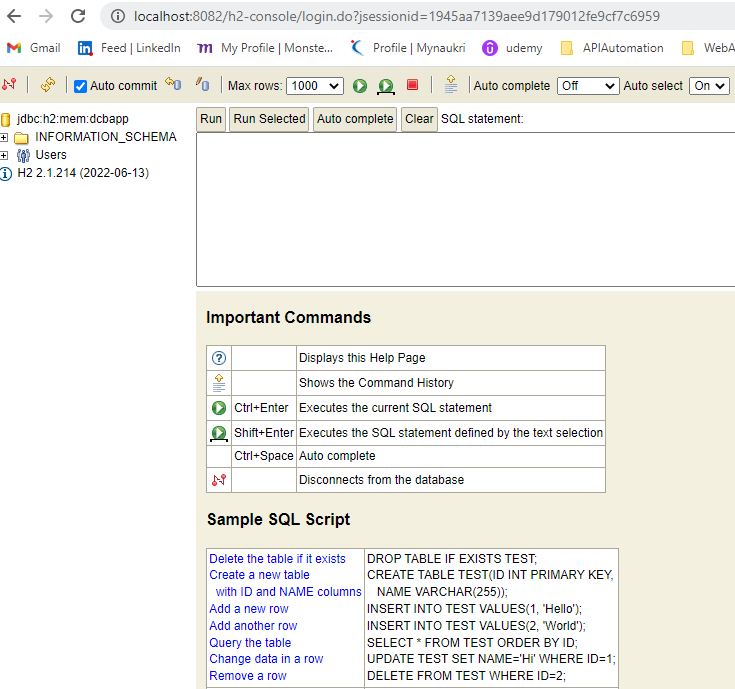
We can access it once our application is running - <http://localhost:8082/h2-console>

If provided path, then access it at http://localhost:8082/h2

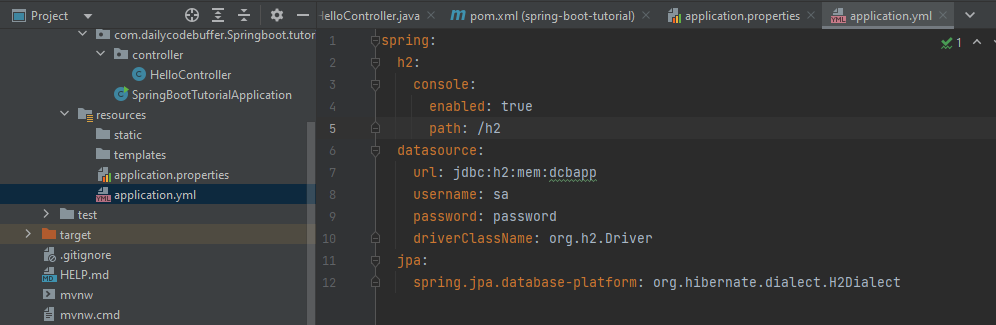
spring.h2.console.path=/h2



After login



Similarly we can use yml file to create H2 database. First comment properties in application.properties file and then create a new **application.yml** file in same location.



Restart the application and we can access it now on localhost:8082/h2. Since we have provided path as h2.

## Creating Components

Create packages controller, entity, service and repository

#### ENTITY

In **entity package** create **Department Class** and create variables and then generate getters and setters, constructor and toString method for those. Also create one default constructor.

Now we have to annotate our class with **@entity**, so that our class can interact with database.

Also as we know there should have one **primary key**, so to make departmentID as primary key we annotate it with **@Id** and to define strategy of how it should generate value we define it with **@GeneratedValue(starategy=GenerationType.AUTO)**

@Id  
@GeneratedValue(strategy = GenerationType.*AUTO*)  
private Long departmentId;

#### CONTROLLER

Now in **controller package** create **DepartmentController class** andannotate with **@RestController.**

#### SERVICE

Now in **service package we create DepartmentService interface** and **DepartmentServiceImpl class** which implements this interfaceand annotate this class with **@Service**

#### REPOSITORY

Now in **repository package** we create **interface Department Repository.**

We annotate with **@Repository**.

And we extent this class with **JpaRepository interface**. Which provides methods to work on database.

We need to pass Entity type and Data type of primary key**. E.g extends JpaRepository<Department, Long>**

## DEPARTMENT SAVE API

Open controller class and create a method which saves department.

Here i want payload which is coming from postman request should come here and convert to entity.

Whatever JSON we are getting as RequestBody from client convert that json to department object.

(@RequestBody Department department)

@PostMapping("/departments")  
public Department saveDepartment(@RequestBody Department department){  
  
}

Now we want to refer DepartmentService in controller, so instead of creating object in traditional way we can use **@Autowired** annotation. Since beans already stored in spring containers, so using autowiring we are telling to refer that bean to variable which we created in any other classes.

@Autowired  
private DepartmentService departmentService;

Now we returning the result of saveDepartment method of departmentService.

We are passing department Object which knows about request payload to saveDepartment method.

return departmentService.saveDepartment(department);

Now in DepartmentService inteface create a method as saveDepartment.

Department saveDepartment(Department department);

Now provide implementation of this method in Impl class

@Override  
public Department saveDepartment(Department department) {  
 return null;  
}

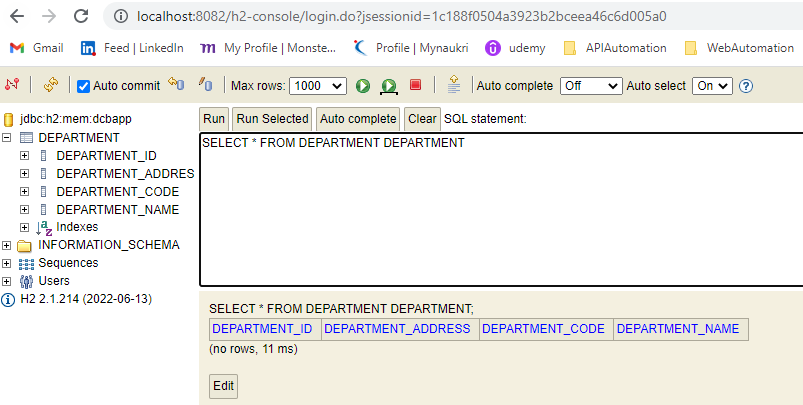
Now in Impl class we have to autowire repository class.

@Autowired  
private DepartmentRepository departmentRepository;

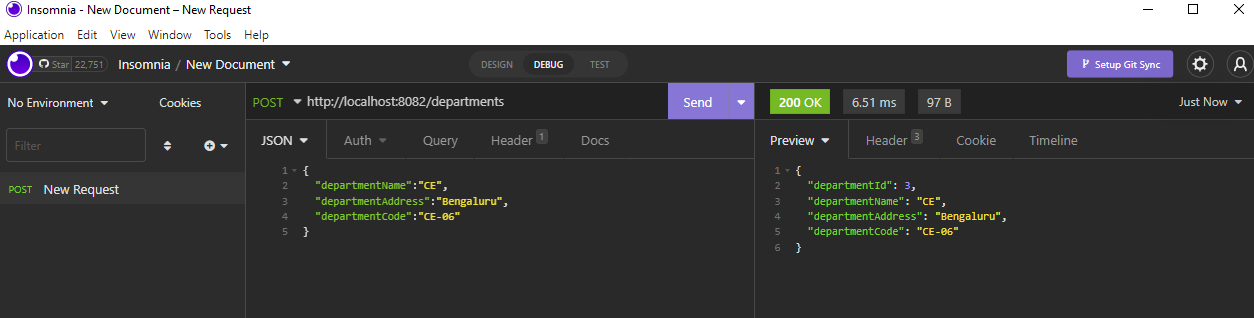
Then instead of returning null we can return save method from jpa repository which we extended in departmentRepository. This save method takes argument as entity.

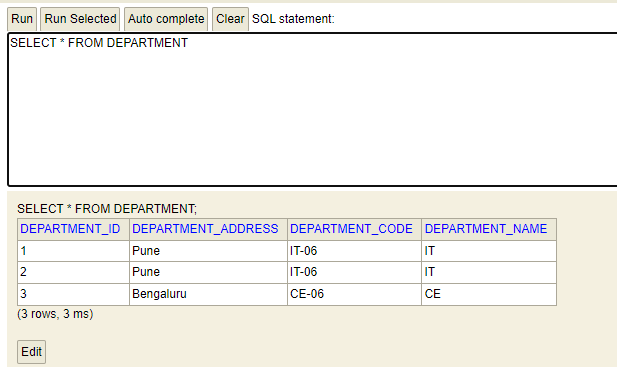
public Department saveDepartment(Department department) {  
 return departmentRepository.save(department);  
}

Spring data JPA will create a table for us automatically.



## TESTING API USING REST CLIENT(Insomnia)





We are able to insert record in H2 database.

## Add swagger UI and Open API to spring boot

<https://www.baeldung.com/spring-rest-openapi-documentation> - **MUST READ**

Add this dependency –

<**dependency**>

<**groupId**>org.springdoc</**groupId**>

<**artifactId**>springdoc-openapi-ui</**artifactId**>

<**version**>1.6.4</**version**>

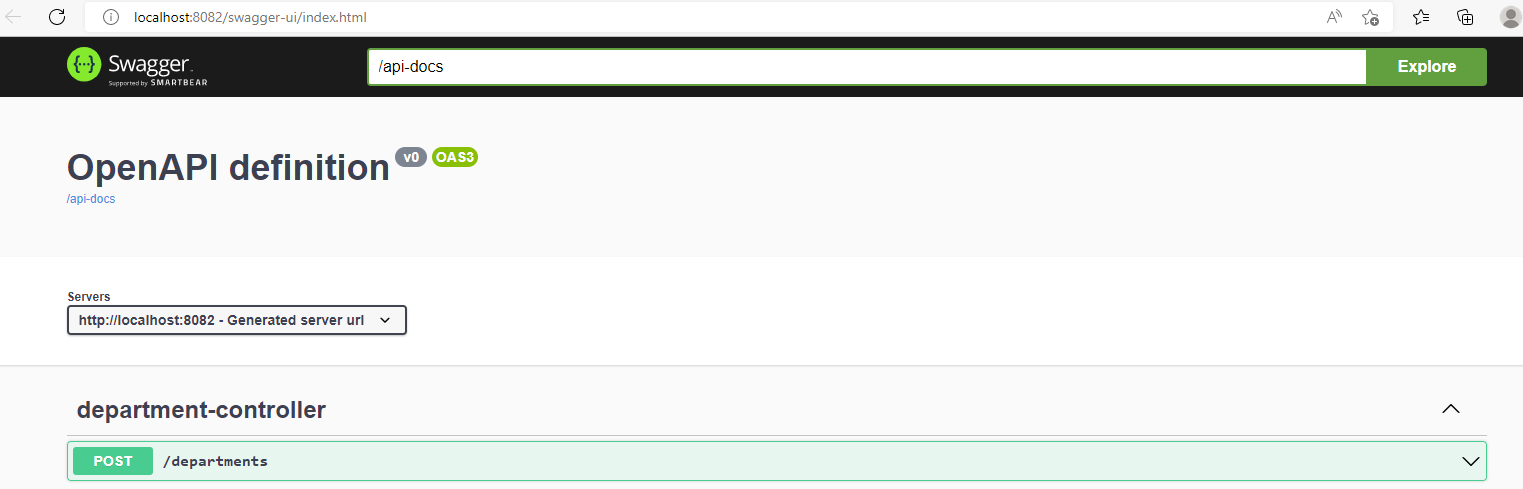
</**dependency**>

Set application.properties to below for custom path.

**springdoc.api-docs.path=/api-docs**

<http://localhost:8082/api-docs/> 🡪 it open json by default.

<http://localhost:8082/swagger-ui.html> --> To view swagger UI



## Get API’s

**Controller –**

@GetMapping("/departments")  
public List<Department> fetchDepartmentList(){  
 return departmentService.fetchDepartmentList();  
}

This will return us List of departments saved in repository.

**Service –**

public List<Department> fetchDepartmentList();

We need to provide implementation for this method

**Service Implementation –**

@Override  
public List<Department> fetchDepartmentList() {  
 return departmentRepository.findAll();  
}

Here we are returning department list with help of findAll method from repository.

## Get API’s (Fetching Data by ID)

@GetMapping("/departments/{id}")  
public Department fetchDepartmentById(@PathVariable("id") Long departmentId){  
 return departmentService.fetchDepartmentById(departmentId);  
}

Here in controller inside GetMapping we have to provide PathParam which is {id}.

So whatever ID is passed by user same has been passed in fetchDepartmentById method of controller.

Also fetchDepartmentById method we have to provide annotation @PathVariable(“id”).

**Service –**

public Department fetchDepartmentById(Long departmentId);

We need to provide implementation for this method

**Service Implementation –**

@Override  
public Department fetchDepartmentById(Long departmentId) {  
 return departmentRepository.findById(departmentId).get();  
}

Here we are returning department details with help of findById method from repository which returns optional, that is why we have to use get() method here.

## Delete API’s (Delete by ID)

@DeleteMapping("/departments/{id}")  
public String deleteDepartmentById(@PathVariable("id") Long departmentId){  
 departmentService.deleteDepartmentById(departmentId);  
 return "Department deleted successfully";  
}

Here in controller inside DeleteMapping we have to provide PathParam which is {id}.

So whatever ID is passed by user same has been passed in deleteDepartmentById method of controller.

Also fetchDepartmentById method we have to provide annotation @PathVariable(“id”).

We are returning String after deletion of record.

**Service –**

public void deleteDepartmentById(Long departmentId);

We need to provide implementation for this method

**Service Implementation –**

@Override  
public void deleteDepartmentById(Long departmentId) {  
 departmentRepository.deleteById(departmentId);  
}

Here we are not returning any details.

## Update API’s

@PutMapping("/departments/{id}")  
public Department updateDepartmentById(@PathVariable("id") Long departmentId,  
 @RequestBody Department department){  
 return departmentService.updateDepartmentById(departmentId, department);  
}

Here in controller inside PutMapping we have to provide PathParam which is {id}.

So whatever ID is passed by user same has been passed in updateDepartmentById method of controller.

Also in Put mapping value is passed by user to update for particular record, we need to have updateDepartmentById one more parameter which is Department with annotation as @RequestBody

**Service –**

public Department updateDepartmentById(Long departmentId, Department department);

We need to provide implementation for this method

**Service Implementation –**

public Department updateDepartmentById(Long departmentId, Department department) {  
 Department depDB = departmentRepository.findById(departmentId).get();  
 if(Objects.*nonNull*(department.getDepartmentName()) &&  
 !"".equalsIgnoreCase(department.getDepartmentName())){  
 depDB.setDepartmentName(department.getDepartmentName());  
 }  
  
 if(Objects.*nonNull*(department.getDepartmentCode()) &&  
 !"".equalsIgnoreCase(department.getDepartmentCode())){  
 depDB.setDepartmentCode(department.getDepartmentCode());  
 }  
  
 if(Objects.*nonNull*(department.getDepartmentAddress()) &&  
 !"".equalsIgnoreCase(department.getDepartmentAddress())){  
 depDB.setDepartmentAddress(department.getDepartmentAddress());  
 }  
  
 return departmentRepository.save(depDB);  
}

Here we are not returning any details.

## Fetch Data by Name API’s

@GetMapping("/departments/name/{name}")  
public Department fetchDepartmentByName(@PathVariable("name") String departmentName){  
 return departmentService.fetchDepartmentByName(departmentName);  
}

Here in controller inside GetMapping we have to provide PathParam which is {name}.

So whatever name is passed by user same has been passed in fetchDepartmentByName method of controller. We have to bind argument departmentName with @PathVariable.

**Service –**

public Department fetchDepartmentByName(String departmentName);

We need to provide implementation for this method

**Service Implementation –**

@Override  
public Department fetchDepartmentByName(String departmentName) {  
 return departmentRepository.findByDepartmentName(departmentName);  
}

Here we are not returning any details.

**Repository Interface –**

public Department findByDepartmentName(String departmentName);

Since we do not have any method in repository interface to find by name, we have to declare one.

**\*\*\***Name is case sensitive, if we want to create a api which ignore case for search department name. We only need to create a method with Keyword as **IgnoreCase** in Repository interface and same has to be called from Service class. Refer below doc for more keywords.

public Department findByDepartmentNameIgnoreCase(String departmentName);

@Override  
public Department fetchDepartmentByName(String departmentName) {  
 return departmentRepository.findByDepartmentNameIgnoreCase(departmentName);  
}

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

Also we can pass native query instead of JPA keywords.

e.g

@Query(value = "select \* from tablename where firstName like %?1", nativeQuery = true)  
public List<User> findByFirstnameEndsWith(String firstname);

## Hibernate Validations

**Scenario - I want my Department Name as mandatory parameter when getting input from user as a json**

1. Add dependency in POM file

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
 <version>3.0.1</version>  
</dependency>

1. In Department entity class we have to add annotation @NotBlank for departmentName.

@NotBlank(message = "Please add department name")  
private String departmentName;

1. In Controller class where we are adding new department, there we have to provide annotation @Valid in saveDepartment method which will check for entity departmentName for validation NotBlank before adding it to DB.

@PostMapping("/departments")  
public Department saveDepartment(@Valid @RequestBody Department department){  
 return departmentService.saveDepartment(department);  
}

1. We can pass multiple validations as well

@Length(max = 5, min =1)  
@Size(max = 10, min = 0)  
@Email  
@Positive  
@Negative  
@PositiveOrZero  
@NegativeOrZero  
@Future  
@Past  
@FutureOrPresent  
@PastOrPresent

## Adding Loggers

**Spring boot comes with the SLF4J library.**

**DepartmentController Class –**

private final Logger LOGGER = LoggerFactory.*getLogger*(DepartmentController.class);

@PostMapping("/departments")  
public Department saveDepartment(@Valid @RequestBody Department department){  
 LOGGER.info("Inside saveDepartment of DepartmentController");  
 return departmentService.saveDepartment(department);  
}

We can simply use the LOGGER object to log the message.

## Project Lombok

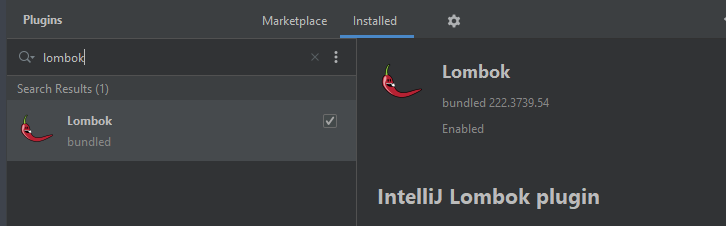
To reduce the boiler plate code

Add Lombok dependency and plugin

<plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <configuration>  
 <excludes>  
 <exclude>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </exclude>  
 </excludes>  
 </configuration>  
</plugin>

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
</dependency>

Also add plugin in IntelliJ



If we use Lombok then with the help of @Data annotation we can generate getters, setters, toString method without doing anything.

@NoArgsConstructor will create default constructor

@AllArgsConstructor will create constructor with all parameters

@Builder is used to design builder pattern.

Except the entity names comment all code.

Then use these annotations at class level.

@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class Department {

## Exception Handling

Create a error package

Create a class DepartmentNotFoundException and this class will extends Exception class.

package com.dailycodebuffer.Springboot.tutorial.error;  
  
public class DepartmentNotFoundException extends Exception {  
 public DepartmentNotFoundException() {  
 super();  
 }  
  
 public DepartmentNotFoundException(String message) {  
 super(message);  
 }  
  
 public DepartmentNotFoundException(String message, Throwable cause) {  
 super(message, cause);  
 }  
  
 public DepartmentNotFoundException(Throwable cause) {  
 super(cause);  
 }  
  
 protected DepartmentNotFoundException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {  
 super(message, cause, enableSuppression, writableStackTrace);  
 }  
}

Now override all methods which is from java.lang.Exception class.

Now open DepartmenServiceImpl class we will add exception handling here in fetchDepartmentById method

@Override  
public Department fetchDepartmentById(Long departmentId) throws DepartmentNotFoundException {  
 Optional<Department> department = departmentRepository.findById(departmentId);  
 if(!department.isPresent()){  
 throw new DepartmentNotFoundException("Department not available");  
 }  
 return department.get();  
}

Since we are throws exception with this method, we need to throw exception in Controller method as well.

@GetMapping("/departments/{id}")  
public Department fetchDepartmentById(@PathVariable("id") Long departmentId) throws DepartmentNotFoundException {  
 return departmentService.fetchDepartmentById(departmentId);  
}

But this will show lot of exception stack trace in response. In order to remove that as well. Do below changes. Whenever there is exception in controller and that exception is thrown then our RestResponseEntityExceptionHandler class will handle that exception and returns response object.

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

Inside Entity package we will create a class ErrorMessage. This class data will be send back as response. So we will create entity which we want to send as response e.g. status and message.

@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class ErrorMessage {  
 private HttpStatus status;  
 private String message;  
  
}

## Changing H2 to MySQL database

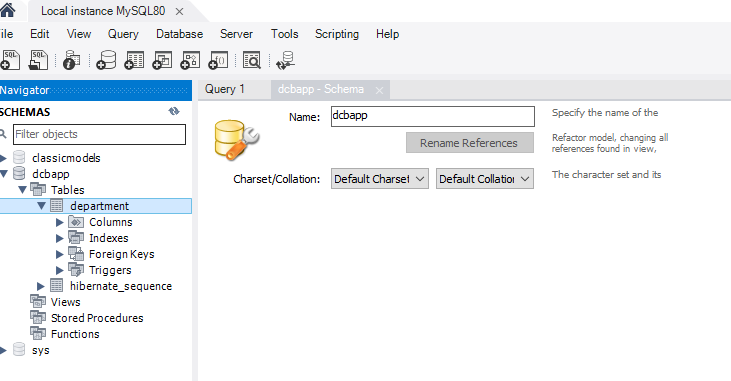
1. In application.properties replace H2 database properties with MySQL database properties

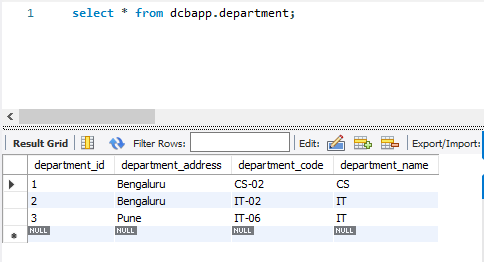
#MySQL Database  
spring.jpa.hibernate.ddl-auto=update  
spring.datasource.url=jdbc:mysql://localhost:3306/dcbapp  
spring.datasource.driverClassName=com.mysql.jdbc.Driver  
spring.datasource.username=root  
spring.datasource.password=Chetan  
spring.jpa.show-sql=true

1. In POM.xml add dependency for MySQL Connection java.

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
</dependency>

1. Now create a schema in mysql workbench which we define in datasource url (dcbapp)





## Unit Testing using Junit5 and Mockito

1. Junit and Mockito library added by default.
2. Unit testing we test Service, Repository and Controller layer.
3. In Unit testing we can mock one of the layer. E.g. To test Controller layer, we can mock service layer as we are focussing on controller component.
4. **@DisplayName** this we can use for reporting purpose.
5. **@Disabled** this we can use to disable any test.
6. **@BeforeEach** gets called before every @Test.
7. **@BeforeAll** gets called only once before @Test start. It will not execute next time for second @Test or so on.
8. **@AfterAll**
9. **@AfterEach**

#### Unit Testing Service Layer

To create a test, just go to the service class for which we want to create tests. Right click on code and generate tests. Select setup checkbox before generate code.

We have to annotate our class with @SpringBootTest annotation.

We have to Autowired class for which we are writing tests.

@SpringBootTest  
class DepartmentServiceTest {  
  
 @Autowired  
 private DepartmentService departmentService;  
  
 @BeforeEach  
 void setUp() {  
 }  
  
 @Test  
 public void whenValidDepartmentName\_thenDepartmentShouldFound(){  
  
 }  
}

Now in below test we are asserting the actual data which we are getting from repository layer because fetchDepartmentByName will call repository layer.

@Test  
@DisplayName("Get data based on valid department name")  
public void whenValidDepartmentName\_thenDepartmentShouldFound(){  
 String departmentName = "EEE";  
 Department found = departmentService.fetchDepartmentByName(departmentName);  
 *assertEquals*(departmentName, found.getDepartmentName());  
}

We have to mock this repository layer because we are not testing repository layer here.

So we have to create object of DepartmentRepository and annotate it with @MockBean

@MockBean  
private DepartmentRepository departmentRepository;

Now we have to create a object of Department class.

We can simply create it by using new Department() , but this is traditional way and arguments need to match.

Here we will use builder method and provide only argument which we want.

@BeforeEach  
void setUp() {  
 Department department = Department.*builder*()  
 .departmentName("EEE")  
 .departmentAddress("Pune")  
 .departmentCode("EEE-06")  
 .departmentId(1L)  
 .build();  
  
 Mockito.*when*(departmentRepository.findByDepartmentNameIgnoreCase("EEE"))  
 .thenReturn(department);  
}

#### Unit Testing Repository Layer

To create a test for repository layer we are not supposed to create data in original database. So we use **@JPATest** which will create a data while our test is running and once test is over it flush the data.

Other approach is we can use in-memory database like H2 database or containerized database (replica of original database)

We will open our repository class and generate test.

Now in our test we have to annotate our class with @DataJpaTest

We have to create object of DepartmentRepository using @Autowired

We have to create object of TestEntityManager class using @Autowired

Inside @BeforeEach method we are saving information in our mock database

**Persist()** method of TestEntityManager class will store the information which we stored in department object.

@DataJpaTest  
class DepartmentRepositoryTest {  
  
 @Autowired  
 private DepartmentRepository departmentRepository;  
  
 @Autowired  
 private TestEntityManager testEntityManager;  
  
 @BeforeEach  
 void setUp() {  
 Department department = Department.*builder*()  
 .departmentName("Mechanical Engineering")  
 .departmentCode("ME - 024")  
 .departmentAddress("BLR")  
 .build();  
 testEntityManager.persist(department);  
 }  
  
 @Test  
 public void whenFindById\_thenReturnDepartment(){  
 Department department = departmentRepository.findById(1L).get();  
 *assertEquals*(department.getDepartmentName(), "Mechanical Engineering");  
 }  
}

#### Unit Testing Controller Layer

To create a test simply open DepartmentController class and generate test, we can select methods from checkbox if want to create a method automatically.

Controller layer get called when we hit endpoint. So here we use @WebMvcTest annotation at class level. This annotation takes argument as DepartmentController.class

From controller layer we are calling service layer so we need to mock service layer. We need to create object of DepartmentService using @MockBean.

We Need to create object of MockMvc using Autowired.

We need to create Department reference variable at class level, and we create object using Builder in @BeforeEach setup method. This department is output department which we return.

Here in actual test we need to create one more Department object and this will be our input object for department to save data.

Then using mockMvc object we have to use perform method and create a request using url, content type and bodyusing MockMvcRequestBuilders post, content-type and content method and then using MockMvcResultMatchers assert if status is ok or not.

In second test we using mock which fetchDepartmentById and return department which we created.

Then using mockMvc we are calling get method with correct url and asserting for status and jsonPath from actual value we receiving and expected value our department object.

package com.dailycodebuffer.Springboot.tutorial.controller;  
  
import com.dailycodebuffer.Springboot.tutorial.entity.Department;  
import com.dailycodebuffer.Springboot.tutorial.error.DepartmentNotFoundException;  
import com.dailycodebuffer.Springboot.tutorial.service.DepartmentService;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mockito;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.boot.test.mock.mockito.MockBean;  
import org.springframework.http.MediaType;  
import org.springframework.test.web.servlet.MockMvc;  
  
import javax.print.attribute.standard.Media;  
  
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  
  
@WebMvcTest(DepartmentController.class)  
class DepartmentControllerTest {  
  
 @Autowired  
 private MockMvc mockMvc;  
  
 @MockBean  
 private DepartmentService departmentService;  
  
 private Department department;  
  
  
 @BeforeEach  
 void setUp() {  
 department = Department.*builder*()  
 .departmentName("Computer Science")  
 .departmentCode("CSE - 32")  
 .departmentAddress("Delhi")  
 .departmentId(1L)  
 .build();  
 }  
  
 @Test  
 void saveDepartment() throws Exception {  
 Department inputDepartment = Department.*builder*()  
 .departmentName("Computer Science")  
 .departmentCode("CSE - 32")  
 .departmentAddress("Delhi")  
 .build();  
 Mockito.*when*(departmentService.saveDepartment(inputDepartment)).thenReturn(department);  
  
 mockMvc.perform(*post*("/departments")  
 .contentType(MediaType.*APPLICATION\_JSON*)  
 .content("{\n" +  
 "\t\"departmentName\":\"Computer Science\",\n" +  
 "\t\"departmentAddress\":\"Delhi\",\n" +  
 "\t\"departmentCode\":\"CSE - 32\"\n" +  
 "}"))  
 .andExpect(*status*().isOk());  
 }  
//we did static import to make code cleaner  
//import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*; - for post() method  
//import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*; - for status() method  
 @Test  
 void fetchDepartmentById() throws Exception {  
 Mockito.*when*(departmentService.fetchDepartmentById(1L)).thenReturn(department);  
 //we passed 1 because we are having only 1 as id in department object  
 mockMvc.perform(*get*("/departments/1")  
 .contentType(MediaType.*APPLICATION\_JSON*))  
 .andExpect(*status*().isOk())  
 .andExpect(*jsonPath*("$.departmentName").value(department.getDepartmentName()));  
 }  
}

## Adding Config in Properties

To read any value from properties file we use @Value(“${keyName}”) annotation.

welcome.message=Welcome to Daily Code Buffer!!!!!!!!

Hello Controller

@Value("${welcome.message}")  
private String welcomeMsg;  
@GetMapping("/welcome")  
public String welcome(){  
 return welcomeMsg;  
}

**Since facing some issues while entering key in yaml file, commented this code.**

## Adding Application.yml

Yml we can remove the redundant part and it is in good format. Most of the technologies uses yml file.

We create it under resources folder.

To convert it we can simply open any converter - <http://mageddo.com/tools/yaml-converter>

server:  
 port: '8082'  
springdoc:  
 api-docs:  
 path: /api-docs  
spring:  
 datasource:  
 username: root  
 url: jdbc:mysql://localhost:3306/dcbapp  
 driverClassName: com.mysql.jdbc.Driver  
 password: Chetan  
 jpa:  
 hibernate:  
 ddl-auto: update  
 show-sql: 'true'  
welcome:  
 message: Welcome to Daily Code Buffer!!!!!!!!

## Springboot Profiles

We might have different environments to run our application. That is why we create different profiles to run application on dev, qa, prod environments

We can create new document in yaml file using 3 hyphens ---

*#spring:  
# h2:  
# console:  
# enabled: true  
# path: /h2  
# datasource:  
# url: jdbc:h2:mem:dcbapp  
# username: sa  
# password: password  
# driverClassName: org.h2.Driver  
# jpa:  
# spring.jpa.database-platform: org.hibernate.dialect.H2Dialect*server:  
 port: '8082'  
  
springdoc:  
 api-docs:  
 path: /api-docs  
  
---  
spring:  
 config.activate.on-profile: dev  
 datasource:  
 username: root  
 url: jdbc:mysql://localhost:3306/dcbapp  
 driverClassName: com.mysql.jdbc.Driver  
 password: Chetan  
 jpa:  
 hibernate:  
 ddl-auto: update  
 show-sql: 'true'  
welcome:  
 message: Welcome to Daily Code Buffer!!!!!!!!  
  
---  
spring:  
 config.activate.on-profile: qa  
 datasource:  
 username: root  
 url: jdbc:mysql://localhost:3306/dcbapp-qa  
 driverClassName: com.mysql.jdbc.Driver  
 password: Chetan  
 jpa:  
 hibernate:  
 ddl-auto: update  
 show-sql: 'true'  
welcome:  
 message: Welcome to Daily Code Buffer!!!!!!!!  
  
---  
spring:  
 config.activate.on-profile: prod  
 datasource:  
 username: root  
 url: jdbc:mysql://localhost:3306/dcbapp-prod  
 driverClassName: com.mysql.jdbc.Driver  
 password: Chetan  
 jpa:  
 hibernate:  
 ddl-auto: update  
 show-sql: 'true'  
welcome:  
 message: Welcome to Daily Code Buffer!!!!!!!!

To deploy our jar file in production first we have to pass application version in our pom.xml for our project artefact.

<groupId>com.dailycodebuffer</groupId>  
<artifactId>spring-boot-tutorial</artifactId>  
<version>1.0.0</version>

Then do mvn clean install. This will generate the jar file in target directory

**PS D:\Latest Learning\Springboot\spring-boot-tutorial> mvn clean install**

Now navigate to target folder and run java –jar spring-boot-tutorial-1.0.0.jar –spring.profiles.active.prod

**PS D:\Latest Learning\Springboot\spring-boot-tutorial> cd .\target\**

**PS D:\Latest Learning\Springboot\spring-boot-tutorial\target> java -jar .\spring-boot-tutorial-1.0.0.jar --spring.profiles.active=prod**

It will start the application with prod profile and in our prod schema table will get created.

Or we can also use **maven profiles** in pom.xml plugins

<plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 <configuration>  
 <excludes>  
 <exclude>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </exclude>  
 </excludes>  
 <profiles>  
 <profile>qa</profile>  
 </profiles>  
 </configuration>  
 </plugin>

Then run it using mvn spring-boot:run

## Springboot Actuator

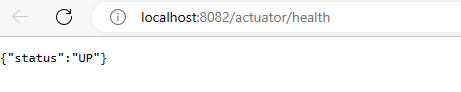
When we deploy our application in environment we have to monitor our application for how our application is performing what all beans are created, is application healthy, different object created, memory utilized. Springboot actuator provides functionality to monitor such things.

We need to add dependency for spring-boot-starter-actuator in our pom.xml

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

By adding this we are done with default configuration.

By default if we hit endpoint /actuator then we can see two endpoints to check health status by default. However we can create lot more if needed by doing configuration



**To expose all options add this in yml file**

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