PROJECT (steps) :

Spring boot + hibernate + maven:

1. first we create the structure of spring boot project using sprint booven maven initialzr (web tool) specifying maven project, java language, spring boot version, the groupId and artifactId & following dependencies:
    a. **WEB:** build web, including restful applications using spring mvc. Uses apache tomcat as the default embedded container.
    b. **MySQL:** jdbc.
    c. **JPA:** persists data in sql stores with java persistence api using spring data and hibernate.
    d. **DevTools:** provide fast application restarts, liveReload and configurations for enhanced development experience.
2. in src/main/java you will already have package that you had given as group (in initializr)
    a. create java class (our main class) in this package and inside main method:

```
SpringApplication.run(main_class_name.class, args);
```

    b. annotate this class with **@SpringBootApplication** and import corresponding package.
    //refer INTERNAL WORKING (project): for more.
    Note: class for springbootapplication annotation is present in org.springframework.boot package.
    c. create packages: controller,service,dao,model inside this package.
        i. controller package: EmployeeController class.
        ii. service: EmployeeService interface, EmployeeServiceImpl class.
        iii. dao: EmployeeDAO interface, EmployeeDAOImpl class
        iv. model: EmployeeBO class.
3. then download the mysql community server from official website, configure server like port, username and password. Then you will get MySql shell or Workbench(prefered) to work:
    a. create table employee with whom we will be dealing with.
4. in src/main/resources/**application.properties** you need to configure the datasource url, username and password.
    a. normally to connect java application with mysql database we need 3 steps
        i. **Driver class:** com.mysql.jdbc.Driver
        ii. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/db** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and db is the database name

            // localhost:3306 : A TCP/IP connection is always made to an IP address (you can think of an IP-address as the address of a certain computer, even if that is not always the case) *and* a specific (logical, not physical) port on that address.
            there are several processes running in our computer and they are assigned different port, like if you have local mysql server, it will run bydefault on 3306. To uniquely identify the process and connect to it we use this ip:port.
        iii. **Username and password**.
    b. but in our case we do not need to specify driver class, spring boot is automatically detect the driver class name based on connection url.
5. Configuring the classes with annotations:

a. EmployeeBO:
Note: classes following annotations are present inside javax.persistence package i.e JPA.

i. **@Entity** (on class): specifies that the class is an **entity** and is mapped to a database table.

ii. **@Table(name="*table_name*")** (on class): specifies the name of the database table to be used for mapping for this class.

iii. **@Column** (on property) **:** to specify the mapping of a property (i.e variable) to specific column of the table, note that if column name and variable name is same then we don't need to specify the name property inside the column annotation otherwise you need to specify column name like this @Column(name="col_name")**.**

iv. **@Id** (on property)**:** to specify that the property is the primary key.
@GeneratedValue(strategy=GenerationType.Identity) (on property): In business scenario's we generally use the actual values, however incase we want spring to generate the unique values for us we use GeneratedValue annotation on primary key. Strategy field allows you to use strategy provided by the databases to generate the primary key. eg., Identity for MySQL, Sequence for Oracle. Incase you don't know the generation type you just need to select Auto and hibernate generate the unique key by itself.
//https://www.youtube.com/watch?v=gUWwUG1UFME

b. EmployeeDAO:
no annotation used.

c. EmployeeDAOImpl:

i. **@Repository** (on class): indicate that the class provides the mechanism for storage, retrieval, search, update and delete operation on objects.
class for the same is present inside org.springframework.stereotype package.

ii. **@Override** (on overriden methods): indicates that the method is overriding.

iii. **@Autowired** (instance object reference): it is used for any dependency injection. In our case ,we are uisng it to get a bean(nothing but a pojo class) of Entity Manager.

iv. Incase of spring mvc and spring boot, we inject session factory. In jpa implementation we will **use Entity Manager** to get the current session. And we will autowire this.
//present inside javax.persistance

```
@Autowired
private EntityManager entityManager;
```

v. Inside method:

1. create session using unwrap method of entityManager object. Pass Session.class as parameter where Session is from org.hibernate.Session.
2. create query using currentSession.

```
@Override
public List<Employee> get() {
   Session currentSession = entityManager.unwrap(Session.class);
   //1. to get employees present in table.
   Query<Employee> query = currentSession.createQuery("from Employee
   List<Employee> list = query.getResultList();
   return list;
}

//2. to add row in table.
currentSession.save(employee);

//3. to get employee with given id.
```

```
Employee employee = currentSession.get(Employee.class, id);

//4. to delete an employee with given id.
first get that employee object using above method, then call:
currentSession.delete(employee);

//5. to update an employee with give id & insert if not present.
currentSession.saveOrUpdate(employee);
```

d. EmployeeService:
no annotation used.

e. EmployeeServiceImpl:
   i. **@Service** (on class): indicates that a particular class serves the role of a service layer.
   ii. **@Override** (on overriden methods).
   iii. **@Transactional** (on method which are going to interact with database): we start transaction to ensure the ACID(atomicity, consistency, isolation, durability) properties.
   it's mandatory otherwise you will get exception at runtime.
   iv. In methods, call the dao..which will return required result.

f. EmployeeController:
   i. **@RestController** (on class): it is used to create restful web services.
   ii. **@RequestMapping** (on class): class level mapping of url.
      1. it can be applied on methods as well, but additionally we will need to specify the method type like get/post that's why we use:
         a. **@GetMapping or @PostMapping**
         b. **@DeleteMapping or @PutMapping**
   iii. **@Autowired** (instance object ref): used for dependency injection.
   iv. **@RequestBody** (on params): binds the http request body to java object.
   v. **@PathVariable** (on params): binds the value passed through the (incoming request) request url to the pathvariable.

```
@GetMapping("/getEmployeeWithId/{id}")
public Employee get(@PathVariable int id){
   Employee employee = employeeService.get(id);
   return employee;
}
//suppose, we are hitting url localhost:8080/api/getEmployeeWithId/2,
because of {id} we can hit the url and using the pathVariable this value
binds to id.
```

6. Now to run the application, you need to specify the fully classified name of main class in <start-class> in properties tag of pom.xml file.
7. update the maven project→ run as maven build→ run as java application(main file).
   a. hit the url localhost:8080/api/employee from postman and you will get the response.
