SpringBoot CodingShuttle

Week 1:

1.3: Beans:

1.4 Dependency Injection:

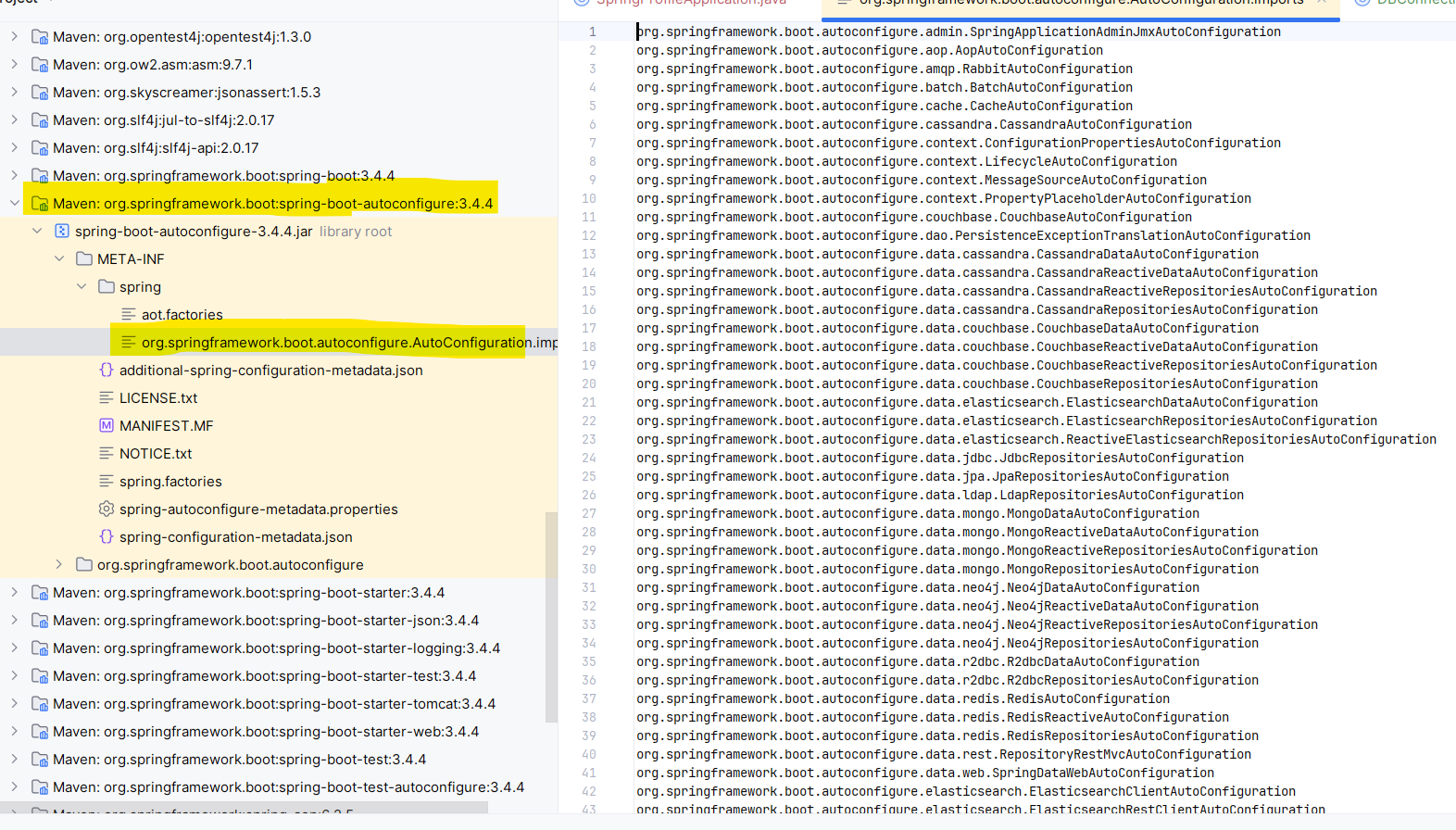
1.5 Spring VS SpringBoot

1.6 AutoConfiguration and Spring Boot Internal Flow:

In pom.xml all dependencies are specified.

Maven is popular build tool used in many java projects. In a Spring Boot Project, Dependencies are specified inside pom.xml file. Maven then resolves dependencies ( it load from mvn repository)and includes them in the class path.

This is where all AutoConfiguration are present: Around 152 autoconfiguration.



When you run the main method of Spring Boot Application Spring Boot will automatically register 17 of the property sources for us. Ex: application.properties , env-variables and many more .

So basically, SpringBoot is just a bunch of AutoConfiguration classes (== normal spring @Configurations), that creates @Beans for you if certain conditions are met!!

Spring Boot Internal Flow :

1. Initialization: when you start your spring boot application, the main entry point is typically a class annotated with @SpringBootApplication (or its meta-annotations). This annotation combines several other annotations such as @Configuration , @EnableAutoConfiguration and @ComponentScan.
2. Spring Application Context Creation: Spring Boot creates an application context, which serves as the container for managing beans and their dependencies. It scans the classpath for components, configurations, and auto-configurations and initializes the application context based on the detected classes and dependencies.
3. Auto-Configuration: Spring boot auto-configures beans and components based on the classpath and detected dependencies. It uses conditional annotations (@ConnditionalOnClass, @CondtionalOnBean etc.) to conditionally configure beans only if certain conditions are met.
4. Externalized Configuration: Spring Boot loads configuration properties from various sources such as property files, YAML files, env vars and command line arg. It provides sensible default values for configuration properties and allows them to be easily overridden or customized.
5. Embedded web server initialization
6. Application Startup: Spring Boot invokes lifecycle callbacks such as @PostConstruct methods and initialization callbacks on beans as the application context is being initialized. Beans are instantiated, dependencies are injected, and any necessary initialization logic is executed.
7. Application Ready: Once initialization process is complete, the application context is fully initialized and ready to handle requests. The embedded server is up and running and the application is ready to serve incoming HTTP requests.

1.7Maven Build Tool:

Week 2: Spring Boot MVC and RESTful APIs:

2.1: Introduction To Spring Boot Web MVC, MVC Architecture :

API’s: REST, SOAP, Web Sockets.

REST (Representational State Transfer) APIs (Application Programming Interfaces) are set of rules and conventions for building and interacting with web services.

REST API’s is stateless.

We can different request in REST API’s:

* GET /users : Retreve a list of all users.
* GET /users/{id}: Retrieve a specific user by ID
* POST /users: create a new user
* PUT /users/{id}: Update an existing user by ID
* PATCH /users/{id}: Partially update an existing user by ID
* DELETE /users/{id}: Delete a user by ID.

For REST API’s we need user Spring-boot-starter-web dependency:

Which contain following dependencies:

* Spring boot starter
* Jackson (JSON ⬄ Java Object)
* Spring-core
* Spring-mvc
* Spring-boot-starter-tomcat

Why to use MVC architecture:

1.Seperation of concerns 2. Reusability 3. Testability 4. Scalability (as code increases here it’s easy to expand)

2.2: Presentation Layer, DTO and Controllers:

1. Presentation Layer: It will write all logic to communicate between client and application. First point of contact between our client and spring boot application with the help of HTTP connection.

How client talk with application? As soon as you run your application server will starts. It goes to all the controllers and tell servlet dispatcher.

DispatcherServlet is a core component of the Spring MVC framework, acting as the front controller that handles all incoming HTTP requests. It receives requests and delegates them to other components for processing, such as controllers, and then returns the result as a response.

The workflow of DispatcherServlet involves the following steps:

* Receives an HTTP request.
* Determines the appropriate controller to handle the request using HandlerMapping.
* Invokes the selected controller through HandlerAdapter.
* Processes the controller's response, which might be a model and view.
* Resolves the view using ViewResolver.
* Renders the view and sends the response back to the client.

Annotated Controllers:

1. @RestController: It is used on top of controller class which accepts all the http request and return response. It is combination of @Controller + @ResponseBody: It means all method in the controller will return JSON/XML directly to the response body.
2. @RequestMapping : Annotation to map requests to controllers methods. It have various attributes to match by URL, HTTP method, request parameters ,headers and media types .

Class level:

You can use @RequestMapping on a class to define a base URI for all methods within that class. For instance, @RequestMapping("/myapp") at the class level would mean all methods inside that class would handle requests starting with /myapp/.

Method-Level:

You can also use @RequestMapping on a method to further refine the mapping for that specific method. For example, if you had @RequestMapping("/myapp") at the class level and @RequestMapping("/hello") on a method, then requests to /myapp/hello would be handled by that method.

@GetMapping:

To get the data two ways:

* 1. @PathVariable : /employees/123

Use path variables when the parameter is **essential** part of the URL path that defines the resource

*@GetMapping(path = "/employees/{employeeId}")*

*public EmployeeDTO getEmployeeByID(@PathVariable(required = false) Long employeeId){*

*return new EmployeeDTO(employeeId,"Sachin","sachin@gmail.com",24, LocalDate.of(2023,9,7),true);*

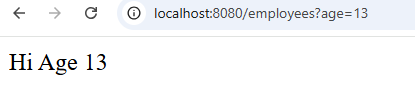
*}*

* 1. @RequestParam : /employees?id=123 (Query Parameter)

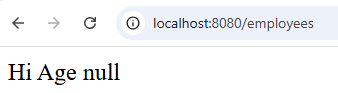
Use query parameters when the parameter is **optional** and used for filtering ,sorting or other modifications to the request.

http://localhost:8080/employees?age=13

@GetMapping(path = "/employees")  
public String getAllEmployees(@RequestParam(required = false) Integer age){  
 return "Hi Age "+ age;  
}



But is its optional if you don’t give then also it works:



Note: When you hit the browser, all requests are type of get only. For post put patch you need to have some UI / frontend / client or Postman (web client) to test web API’s .

* 1. @RequestBody : It is used to bind the HTTP request body to Java Object . When a client sends data in the body of a request ( e.g. JSON or XML),@RequestBody maps this data to a java object.

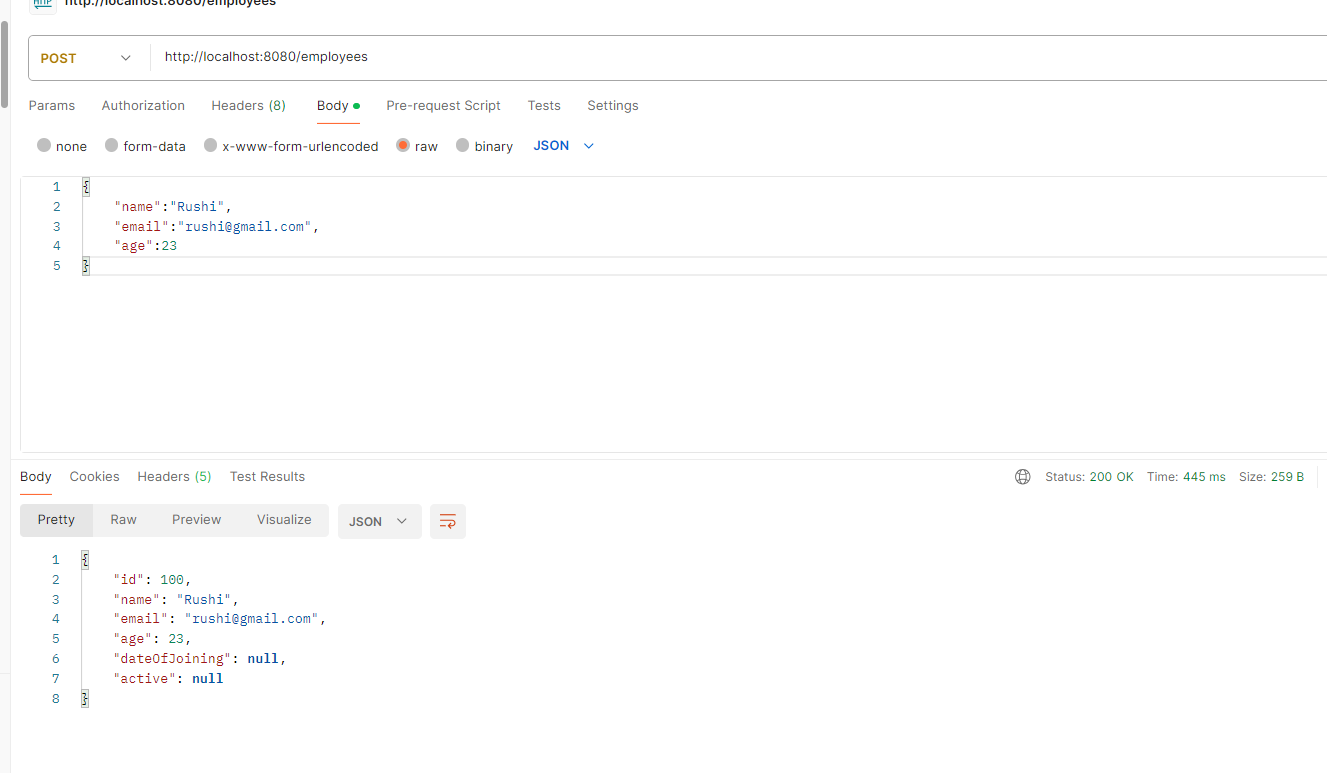
@PostMapping

public EmployeeDTO createNewEmployee(@RequestBody EmployeeDTO employeeDTO){

employeeDTO.setId(100L) ;//we are saying don't give id I will give

return employeeDTO;

}



-----------------------------------------------------------------------------------------------------------------

* 1. Persistence Layer & JPA:

See the architecture of JPA in Dig:

H2 is in memory database: Mainly, H2 database can be configured to run as inmemory database, which means that data will not persist on the disk. Because of embedded database it is not used for production development, but mostly used for development and testing.

We will need : starter-data-jpa dependency .

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

For Database we need use H2-database dependency :   
<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>  
</dependency>

Two interfaces: 1) CrudRepository<T,ID> 2)JpaRepository

**✅ 1. CrudRepository**

**Basic Interface** for CRUD (Create, Read, Update, Delete) operations.

**🔸 Key Features:**

* Basic operations:
  + save()
  + findById()
  + findAll()
  + deleteById()
* Minimal functionality — enough for simple use cases.

**🔹 Example:**

public interface EmployeeRepository extends CrudRepository<Employee, Long> {

}

**✅ 2. JpaRepository**

**Extends CrudRepository** and **PagingAndSortingRepository** with **extra JPA-specific features**.

**🔸 Key Features:**

* Inherits all CrudRepository methods.
* Adds:
  + flush() – sync changes immediately to DB
  + saveAllAndFlush()
  + findAll(Sort sort) – for sorting
  + findAll(Pageable pageable) – for pagination
  + deleteInBatch() and deleteAllInBatch()
* Better suited for complex applications and production systems.

**🔹 Example:**

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

}

**🆚 Comparison Summary:**

| **Feature** | **CrudRepository** | **JpaRepository** |
| --- | --- | --- |
| Basic CRUD operations | ✅ Yes | ✅ Yes |
| Sorting | ❌ No | ✅ Yes |
| Pagination | ❌ No | ✅ Yes |
| Batch operations | ❌ No | ✅ Yes |
| JPA-specific methods | ❌ No | ✅ Yes |
| Recommended for real projects | ❌ Limited use | ✅ Preferred choice |

**✅ Conclusion:**

✅ Use **JpaRepository** if you want **full functionality**, especially for **pagination, sorting, and batch operations**.  
⚠️ Use **CrudRepository** only for **very simple** use cases with limited operations.

In Pom.xml Spring-boot-starter-parent will handle dependency and versions.

By default H-2 database is in memory database : It means once you close the application it will clear all the data present in that .

If you want to store that data in disk locally you can change the url :In application.properties : spring.datasource.url=jdbc:h2:file:/Users/sacshetk/SpringBoot\_0\_to\_1/week2\_2\_3/db

If you want to see h2 console:

spring.h2.console.enabled=true

If you want to add your custom username and password you can add :

spring.datasource.username=sa

spring.datasource.password=sa

If you want for first when it is running application you should do this :

spring.jpa.hibernate.ddl-auto=create

If you want from second time it should update existing table instead of creating new use following :

spring.jpa.hibernate.ddl-auto=update

Define this repository:

@Repository

public interface EmployeeRepository extends JpaRepository<EmployeeEntity,Long> {

}

To connect with repository :

1. findById(id) : return optional type entity : for get request
2. findAll() : for GET request
3. save(entity) : For post request

private final EmployeeRepository employeeRepository;

public EmployeeController(EmployeeRepository employeeRepository) {

this.employeeRepository = employeeRepository;

}

@GetMapping(path = "/{employeeId}")

public EmployeeEntity getEmployeeByID(@PathVariable(name = "employeeId") Long employeeId) {

return employeeRepository.findById(employeeId).orElse(null);

}

@GetMapping

public List<EmployeeEntity> getAllEmployees(@RequestParam(required = false) Integer age){

return employeeRepository.findAll();

}

@PostMapping

public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity employeeEntity){

return employeeRepository.save(employeeEntity);

}

For post request :

{

"name":"Subramanyam",

"email":"subramanyam@gmail.com",

"age":24,

"dateOfJoining":"2024-11-01",

"isActive":true

}

In entity class we need to do follow some things :

1. @Entity : To convert entity class into table
2. @Table : To change the name of table else it will give name to table as classname
3. @Id : To denote primary key
4. @GeneratedValue(strategy= GenrationType.AUTO)

@GeneratedValue is an annotation used in JPA to specify that the value of the primary key should **be automatically generated** (i.e., not set manually).

It works together with **@Id** (which marks a field as the primary key).

**⚙️ Strategy Options:**

|  |  |
| --- | --- |
|  |  |
| AUTO (default) | JPA picks the best strategy depending on the database |
| IDENTITY | Uses DB identity columns (e.g., AUTO\_INCREMENT in MySQL) |
| SEQUENCE | Uses database sequence object (best for Oracle/PostgreSQL) |
| TABLE | Uses a table to simulate sequence (less common, slower) |
| **💡 What Is Lombok?**  **Lombok** is a **Java library** that helps reduce **boilerplate code** in Java classes.  It does this by using **annotations** to automatically generate commonly used methods like:   * getters * setters * constructors * toString() * equals() and hashCode() * and more...   **✅ Why Use Lombok?**  Java code can be **verbose**. For example, if you have a class with 3 fields, you typically write:   * Getters * Setters * Constructor * toString()   That’s a lot of **repetitive code**.  With Lombok, you can avoid writing all that manually.  **🔧 Setup**   1. **Add Lombok to pom.xml** (if you're using Maven):   xml  CopyEdit  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <version>1.18.30</version> <!-- or latest -->  <scope>provided</scope>  </dependency>   1. **Enable annotation processing** in your IDE (important):    * In IntelliJ: Preferences → Build, Execution, Deployment → Compiler → Annotation Processors → ✅ Enable.   **🧪 Example Without Lombok:**  public class User {  private String name;  private int age;  public User() {}  public String getName() { return name; }  public void setName(String name) { this.name = name; }  public int getAge() { return age; }  public void setAge(int age) { this.age = age; }  @Override  public String toString() {  return "User{name='" + name + "', age=" + age + "}";  }  }  **✅ Same Example With Lombok:**  import lombok.Data;  @Data  public class User {  private String name.  private int age;  }  ➡️ This one annotation @Data automatically generates:   * Getters and setters * toString() * equals() and hashCode() * Required constructor   **📌 Common Lombok Annotations**   | **Annotation** | **Description** | | --- | --- | | @Getter | Generates getter methods | | @Setter | Generates setter methods | | @Data | Combines @Getter, @Setter, @ToString, etc. | | @NoArgsConstructor | Generates no-arg constructor | | @AllArgsConstructor | Generates constructor with all fields | | @Builder | Adds builder pattern support | | @ToString | Generates toString() | | @EqualsAndHashCode | Generates equals() and hashCode() |   **✅ Conclusion:**  Lombok helps write **clean, short, and readable Java code** by removing boilerplate. It's widely used in **Spring Boot projects** to keep entities, DTOs, and models simple. |  |

------------------------------------------------------------------------------------------------------------------

* 1. Service Layer, Writing our business logic:

Key role of service layer:

The service layer acts as bridge between the persistence layer (responsible for data access) and the presentation layer (handing user interaction).

It encapsulates the business logic of the application, shows interactions between different components, and provides a clean interface for external clients to interact with the system.

By abstracting away the complexities of data access and business operations, the service layer promotes modularity, maintainability, and scalability.

Important things: In service layer we don’t return entity to controller we return DTO for that we need to map Entity with DTO.

For that we need add dependency of Mapper in pom.xml and need to create Bean of that by using config :

Dependency :

<!-- https://mvnrepository.com/artifact/org.modelmapper/modelmapper -->

<dependency>

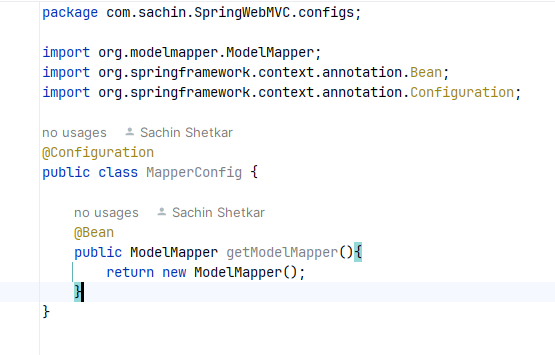
<groupId>org.modelmapper</groupId>

<artifactId>modelmapper</artifactId>

<version>3.0.0</version>

</dependency>

Config class:



We need use ModelMapper in now service:

ModelMapper has map method(source , target.class)

private final EmployeeRepository employeeRepository;  
  
**private final ModelMapper modelMapper;**  
public EmployeeService(EmployeeRepository employeeRepository, ModelMapper modelMapper) {  
 this.employeeRepository = employeeRepository;  
 this.modelMapper = modelMapper;  
}  
  
public EmployeeDTO getEmployeeById(Long employeeId) {  
 EmployeeEntity employeeEntity = employeeRepository.findById(employeeId).orElse(null);  
 return modelMapper.map(employeeEntity,EmployeeDTO.class);  
}  
  
public List<EmployeeDTO> getAllEmployees() {  
 List<EmployeeEntity> employeeEntities = employeeRepository.findAll();  
 return employeeEntities  
 .stream()  
 .map(employeeEntity -> modelMapper.map(employeeEntity,EmployeeDTO.class))  
 .collect(Collectors.*toList*());  
}  
  
public EmployeeDTO createNewEmployee(EmployeeDTO employeeDTO) {  
 EmployeeEntity toSaveEntity = modelMapper.map(employeeDTO,EmployeeEntity.class);  
 EmployeeEntity savedEmployeeEntity = employeeRepository.save(toSaveEntity);  
 return modelMapper.map(savedEmployeeEntity,EmployeeDTO.class);  
}  
  
  
public EmployeeDTO updateEmployeeById(EmployeeDTO employeeDTO, Long employeeId) {  
 EmployeeEntity employeeEntity = modelMapper.map(employeeDTO,EmployeeEntity.class);  
 employeeEntity.setId(employeeId);  
 EmployeeEntity savedEmployeeEntity = employeeRepository.save(employeeEntity);  
 return modelMapper.map(savedEmployeeEntity,EmployeeDTO.class);  
}

**When to Use ModelMapper:**

* To reduce boilerplate code for object mapping.
* In layered apps (like Spring Boot), where you separate:
  + Controllers (accept request DTO)
  + Services (use entity/domain objects)
  + Responses (return response DTO)
  1. : PUT , PATCH and DELETE Mappings in Spring web MVC :

1. PUT Mapping : It is used to update the Entity :

* What we are doing we are passing DTO and Id if Id is present then Update the DTO else create new employee:

For PUT Mapping also we need to use : repo.save(entity) it returns entities

Write API : in controller

@PutMapping(path = "/{employeeId}")

public EmployeeDTO updateEmployeeById(@RequestBody EmployeeDTO employeeDTO,@PathVariable Long employeeId){

return employeeService.updateEmployeeById(employeeDTO,employeeId);

}

Write logic in Service Layer :

public EmployeeDTO updateEmployeeById(EmployeeDTO employeeDTO, Long employeeId) {  
 EmployeeEntity employeeEntity = modelMapper.map(employeeDTO,EmployeeEntity.class);  
 employeeEntity.setId(employeeId);  
 EmployeeEntity savedEmployeeEntity = employeeRepository.save(employeeEntity);  
 return modelMapper.map(savedEmployeeEntity,EmployeeDTO.class);  
}

What we did here: step 1 : First We Mapped DTO to Entity becz repository works with entities

Step 2 : We use setter to setId

Step 3 : We saved the entity in repo

Step 4 : we returned the Updated DTO by mapping

1. DELETE Mapping : It is used to delete DTO by using id

For Delete Mapping we use method : repo.deleteByID(id) return type is void

Write API in controller:

@DeleteMapping(path = "/{employeeId}")  
public void deleteEmployeeById(@PathVariable Long employeeId){  
 employeeService.deleteEmployeeById(employeeId);  
}

Write Logic in Service Layer :

public void deleteEmployeeById(Long employeeId) {

employeeRepository.deleteById(employeeId);

}

But here we are not handling if employee is not present with employeeId

To handle basic we are checking whether employee exist or not by using employeeId

Controller :

@DeleteMapping(path = "/{employeeId}")

public Boolean deleteEmployeeById(@PathVariable Long employeeId){

return employeeService.deleteEmployeeById(employeeId);

}

In Service Layer:

public boolean deleteEmployeeById(Long employeeId) {

Boolean isPresent= employeeRepository.existsById(employeeId);

if(!isPresent) return false;

employeeRepository.deleteById(employeeId);

return true;

}

3.PATCH Mapping: It is used for partial Mapping you don’t need to pass DTO from Postman only you will pass fields :

In controller:

@PatchMapping(path = "/{employeeId}")  
public EmployeeDTO updatePartialEmployeeById(@RequestBody Map<String, Object> updates,@PathVariable Long employeeId){  
 return employeeService.updatePartialEmployeeById(employeeId,updates);  
}

It accepts as Map of updates.

In Service Layer:

They are using Reflection concept: to set fields

public EmployeeDTO updatePartialEmployeeById(Long employeeId,Map<String, Object> updates) {  
 if(!isExistsByEmployeeId(employeeId)) return null;  
 EmployeeEntity employeeEntity = employeeRepository.findById(employeeId).get();  
 updates.forEach((field,value)->{  
 Field fieldToBeUpdated = ReflectionUtils.*findRequiredField*(EmployeeEntity.class,field);  
 *//we are allowing to access the fields of Entity* fieldToBeUpdated.setAccessible(true);  
 ReflectionUtils.*setField*(fieldToBeUpdated,employeeEntity,value);  
 });  
 return modelMapper.map(employeeRepository.save(employeeEntity),EmployeeDTO.class);  
}

Note : If you are using any Boolean field in DTO or in Entity like name starting with isActive or isPresent or isAvaiable then in that it if create getter and setter by using IDE not by using Lombok it will create getters and setter like this and value it will give null :

public void setActive(Boolean active){  
 isActive=active;  
}

But while deserilazation Jackson will look for :

Public void setIsActive(Boolean active){  
 isActive=active;  
}

We can do this manually or you can use:

@JsonProperty(“isActive”)  
private Boolean isActive;

It will consider name isActive only not as active . For getter and setter you can create method manually or you can use lombook :

If you doing manually then create as : setIsActive( ) and getIsActive( )

ResponseEntity in Spring Boot:

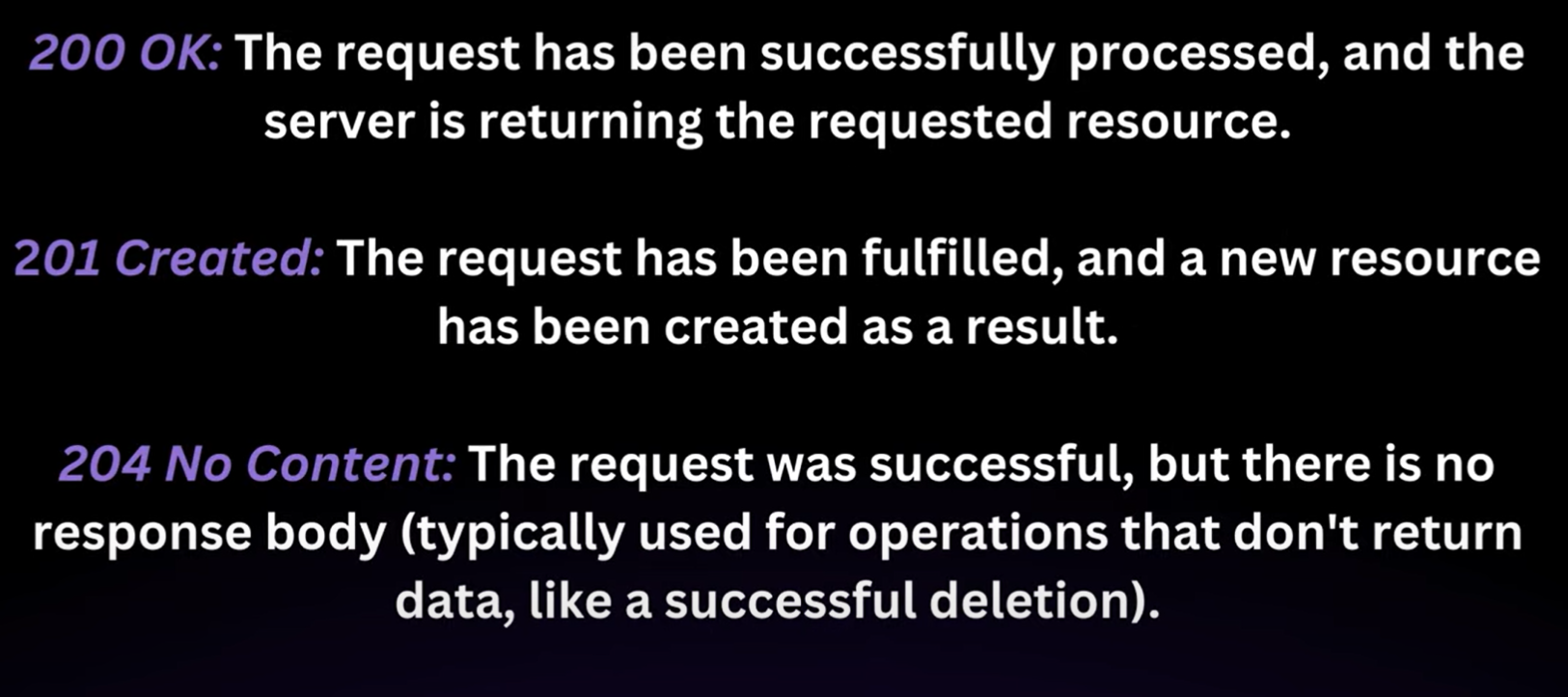
Client (who send request here postman is client) ⬄ Server (here we are having tomcat server) ( who give response to request)

If in some API it is not working properly how it will not it is updated or added new entry how we know for that we need status codes : By using status codes client can know what happened at server .

HTTP status code is three-digit code return by a web server as part of the response to an HTTP request made by client.

Status code which starts with 1 : 1XX : It is used for informational purpose

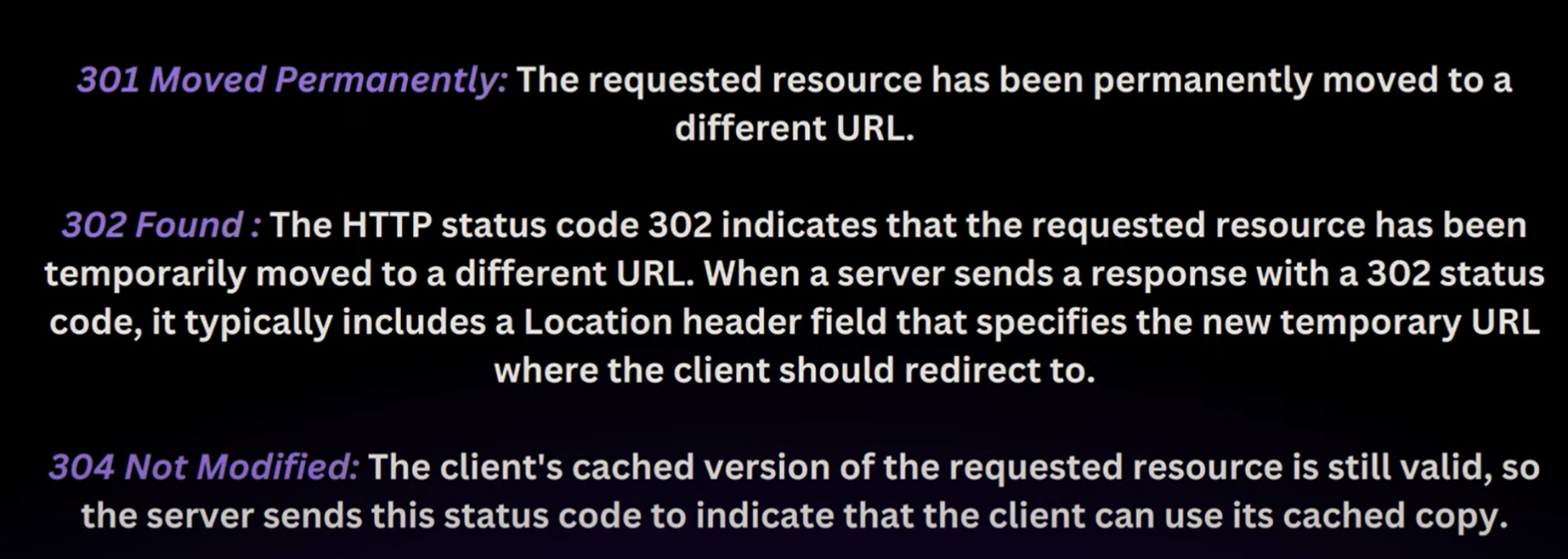
Status code which starts with 2 : 2XX : Which indicates that the request was successfully received understood and proceed by the server.



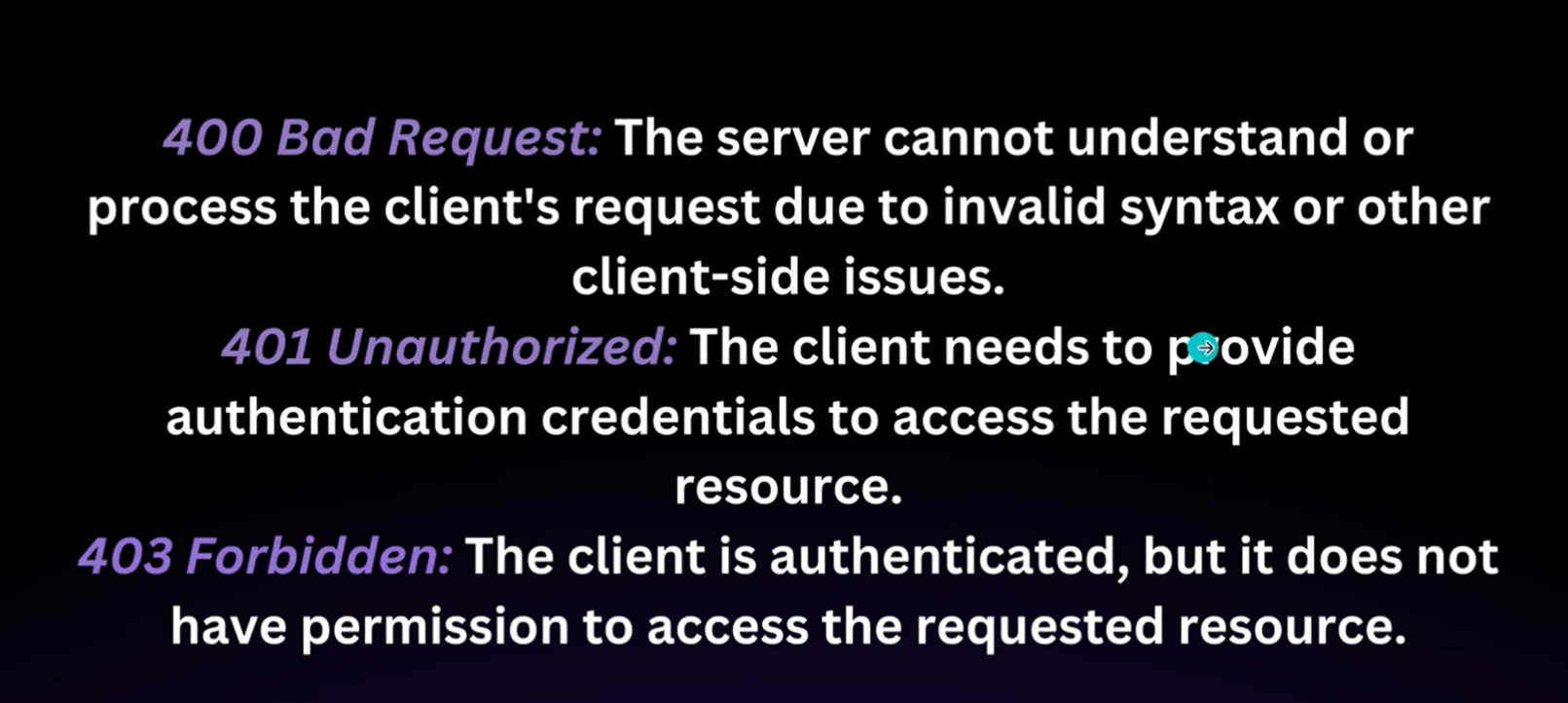
201 : Is in post request

204 : In delete request

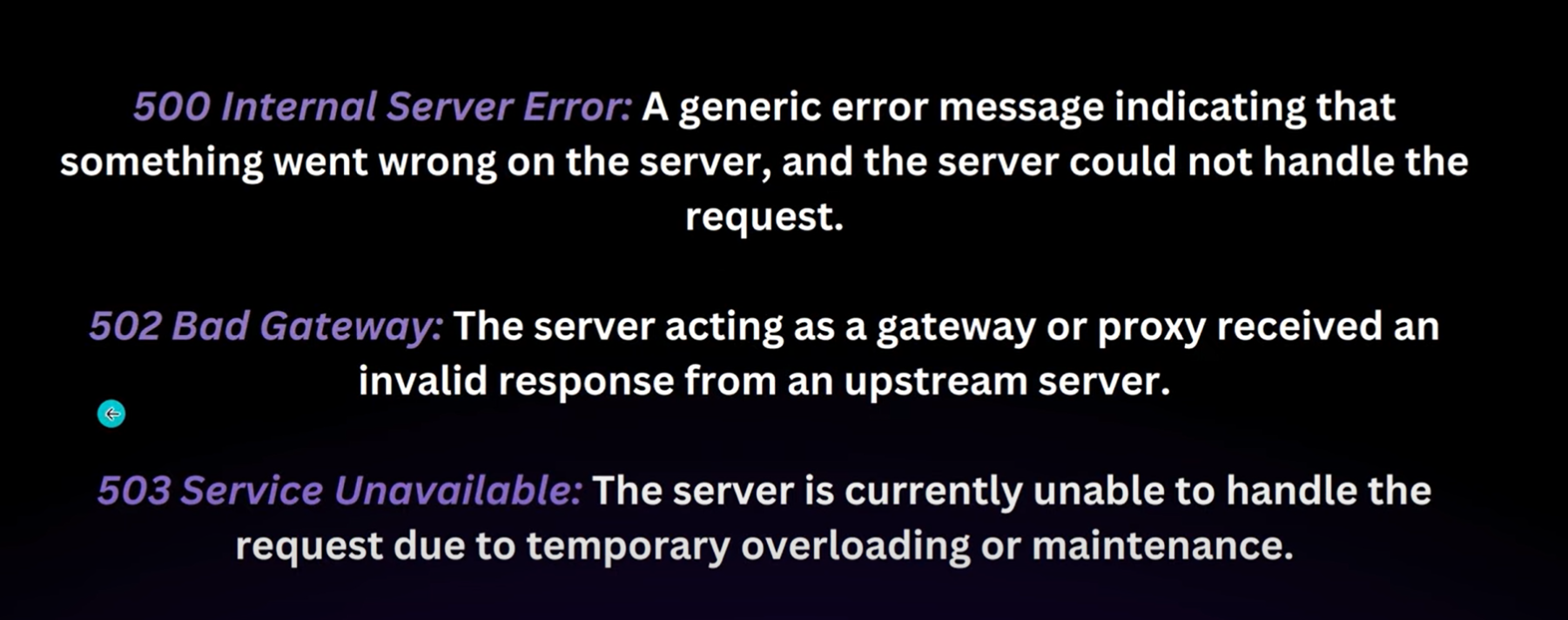
Status code starts with 3 : 3XX : These status code indicates that further action is needed to complete the request. They are used when the client needs to take additional steps to access the requested resource.



Status code starts with 4 : 4XX : It indicates that there was error on the client’s part such as malformed request or authentication issues



Status code which starts with 5 : 5XX : The status code indicates that there was an error on the server’s part while trying to fullfill the request . That is in springboot application



ResponseEntity: It is class part of spring framework and is commonly used in spring boot applications to customize the HTTP response.

It provide method for setting the response status , headers and body . We can return different data JSON , XML.

findById(id) => return optional

* if(optional.isPresent()){

return new ResponseEntity<>(optional.get(),HttpStatus.OK)l

}else{

Return new ResponseEntity<>(HttpStatus.NOT\_FOUND);//404

For PostMapping:

* public ResponseEntity<EmployeeDTO> createEmployee(@RequestBody EmployeeDTO employeeDTO){  
   try{  
   employeeService.createEmployee(employeeDTO);  
   return new ResoponseEntity<>(employeeDTO,HttpStatus.CREATED);  
   }catch(Exception e){  
   return new ResponseEntity<>(HttpStatus.BAD\_REQUEST);

For DeleteMapping :   
public ResoponseEntity<?> deleteEmployeeById(@PathVariable Long employeeId){

employeeService.deleteEmployeeById(employeeId);  
 return new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

For Patch Mapping :

@PatchMapping(path = "/{employeeId}")  
public ResponseEntity<EmployeeDTO> updatePartialEmployeeById(@RequestBody Map<String, Object> updates,@PathVariable Long employeeId){  
 EmployeeDTO employeeDTO = employeeService.updatePartialEmployeeById(employeeId,updates);  
 if(employeeDTO == null) return ResponseEntity.*notFound*().build();  
 return ResponseEntity.*ok*(employeeDTO);  
}

For PUT Mapping :

@PutMapping(path = "/{employeeId}")  
public ResponseEntity<EmployeeDTO> updateEmployeeById(@RequestBody EmployeeDTO employeeDTO,@PathVariable Long employeeId){  
 return ResponseEntity.*ok*(employeeService.updateEmployeeById(employeeDTO,employeeId));  
}

Note: status 100: Information status 200 : for successful status 300 : redirectional 400 : client error 500 :server error .

* 1. Input Validation Annotations:

We want something i.e. before going to service layer it should validate at controller level only we are giving valid fields or not.

Ex salary should be positive, gmail should contain domain name , phone number should be 10 digits , name should not be empty , salary should be in range

To validate 1) firstly we need to add dependency i.e :

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

Second we need use @Valid in Controller :

In method argument only we need use **@Valid** :

@PostMapping("/users")

public ResponseEntity<String> createUser(@Valid @RequestBody UserDTO user) {

// handle user creation

return ResponseEntity.ok("User created");

}

Now we need to add whichever validation we want to use in DTO class because user is interacting with DTO not entity:

public class EmployeeDTO{

private Long id;

@NotNull(message=”Required Field is Employee : name”)   
 private String name;

Now if you not give name is post in JSON it will give error till now it was giving null value for name if we are not giving :

{

"email":"rama@gmail.com",

"age":27,

"dateOfJoining":"2021-01-01",

"isActive":true

}

Here I have not given name so it will give error : 404 badrequest :

But issues **@NotNull** in name is that if we give empty string with zero character it will work example :

{

“name” : “”,

“age”,27,

….

}

It will work but to overcome that we use :

**@NotEmpty** : It ensures that the annotated filed is not null and it’s size/length is greater than zero

But issue is that it will count space : if I give : in json : “name” : “ “, It will not give any error but it is not correct for that we use

**@NotBlank** : Ensure that annotated String is not null and its trimmed length is greater than zero .

**@Size** : Validates that the annotated element’s size falls within the specified range : @Size(min=,max=,message=””)

On name , array … we can use :

Ex :

@NotBlank(message = "Name of the employee can't be blank")  
@Size(min=2,max = 10,message = "Number of characters in name should be in the range : [3,10]")  
private String name;

**@Email(message=””)**

Ex : @Email(message = "Email should be valid ")  
private String email;

**@Max(value=,message=””)  
 @Min(value= , message=””)**

Example :

@Max(value = 80,message = "Age of employee can't be greater than 80")  
@Min(value = 18,message = "Age can't be less than 18")  
 private Integer age;

**@Pattern(regexp= , message)** : It is used for regular expression we can use this for checking 10 digit phone number : @Pattern(regexp=”\\d{10}”,message=””)

Example :

@Pattern(regexp = "^(ADMIN|USER)$",message = "Role of Employee can be USER or ADMIN")  
private String role;//ADMIN,USERIt will check role should be ADMIN or USER only other than that it will not accept

**@Postive** : Ensures that the annotated element is a positive number(greater than zero)

@Positve(message =”Salary should be positive”)  
private Integer salary;

**@PositveOrZero , @Negative , @NegativeOrZero**

**@Past** : ensures that the annotated date or calendar value is in the past or present.

**@PastOrPresent** : date or calender value is past or present

**@Future** : date or calender value is in the future.

Example :

@PastOrPresent(message = "date of joining employee can't be in the future")  
 private LocalDate dateOfJoining;

**@Digits,@DecimalMin,@DecimalMax :**

@Digits(integer = 6,fraction = 2,message = "The salary can be in the form of XXXXX.YY")

@DecimalMin(value = "100.10")

@DecimalMax(value = "100000.20")

private Double salary;

**@AssertTrue** : Ensures that the annotated boolean filed is true

**@AssertFalse** : Ensures that the boolean field is false.

@JsonProperty("isActive")

@AssertTrue(message = "Employee should be active")

private Boolean isActive;

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------

* 1. : Exception Handling in Spring Boot MVC:

Benefits of Exception Handling:

* Prevents application crashes.
* Provide user friendly error responses.
* Facilitate debugging and maintenance.
* Ensure consistent error handling across the application.

Use @ExceptionHandler to handle specific exceptions in controllers

Use @RestControllerAdvice for global exception handling

Return appropriate HTTP status codes and error messages.

Use custom error response class to provide structured error details.

**✅ What is Exception Handling in Spring Boot?**

Exception handling in Spring Boot is a way to manage errors in a clean, centralized, and user-friendly manner so that the application doesn't crash and returns a proper response to the user when something goes wrong.

**💬 How do I handle exceptions in Spring Boot?**

You can say:

I use a **Global Exception Handler** class with @RestControllerAdvice to catch and handle all exceptions across the application.  
Inside it, I define methods with @ExceptionHandler to handle different types of exceptions like resource not found, validation errors, and generic exceptions.

**🔧 What it Looks Like (Example)**

Say this:

For example, if I have a CRUD API for employees and someone tries to fetch an employee that doesn’t exist, I throw a custom exception like ResourceNotFoundException.  
Then, in the global exception handler, I catch it and return a clean message like "Employee not found with id 10" instead of crashing the app.

* 1. : Transforming API response:

You need to complete this.

--------------------------------------------------------------------------------------------------------------

3 : Hibernate and Spring Data JPA

* 1. : Installing MySQL for Database and DBeaver for Analytics:

You can install MySQL from: <https://dev.mysql.com/downloads/installer/>

To login in MySQL through commond line use this command : mysql -u root -p

DBeaver is mostly used for good practices : There we can connect any database

3.2: Hibernate and JPA :

**✅ What is Hibernate in Java?**

**Hibernate** is an **Object-Relational Mapping (ORM)** framework for Java.  
It helps you **map Java objects (classes)** to **relational database tables**, and automatically handle common database operations like insert, update, delete, and query.

**🔸 Why is Hibernate Used?**

Without Hibernate, you'd use **JDBC**, which requires:

* Writing SQL queries manually
* Handling connections and result sets
* Mapping result rows to Java objects manually

✅ Hibernate **automates** all that.

**🔧 Example:**

Instead of writing this in JDBC:

INSERT INTO users (id, name, email) VALUES (?, ?, ?);

You just write:

session.save(user);

Hibernate converts your Java object into SQL and executes it.

**✅ Key Features of Hibernate**

| **Feature** | **What it does** |
| --- | --- |
| **ORM** | Maps Java classes to DB tables |
| **HQL (Hibernate Query Language)** | Lets you query the database using object-oriented syntax |
| **Automatic Table Creation** | Can create/update DB schema based on entity classes |
| **Caching** | Improves performance by avoiding repeated queries |
| **Lazy Loading** | Loads data only when needed |
| **Transactions Support** | Works with JDBC or JTA transactions |
| **Annotations/XML Config** | You can configure mappings using annotations or XML |

**✅ Real-World Use of Hibernate**

Used in:

* Spring Boot applications (with JPA)
* Any Java EE project that interacts with a relational DB (like MySQL, PostgreSQL, Oracle, etc.)

**🧠 Summary**

| **Concept** | **Description** | | |
| --- | --- | --- | --- |
| Hibernate | Java ORM tool to work with databases easily | | |
| Benefit | No need to write SQL manually, reduces boilerplate | | |
| Alternative to | | JDBC (lower-level manual database handling) |

**✅ What is JPA?**

**JPA (Java Persistence API)** is a **standard specification** (like a rulebook) for **ORM** in Java.

It defines:

* **How** Java objects should be mapped to database tables
* **How** to perform operations like save, update, delete, and query

But JPA **doesn't do anything by itself** — it's just a set of **interfaces** and **annotations**.

**✅ Hibernate vs JPA**

| **Feature** | **JPA** | **Hibernate** |
| --- | --- | --- |
| What it is | Specification (API) | Implementation of JPA (a tool) |
| Who provides it | Part of Java EE | Open-source framework |
| Needs implementation? | Yes | Hibernate **is** an implementation |
| Uses annotations | Yes | Yes (same ones as JPA) |
| Standard or tool? | Standard | Tool that implements the standard |

**💡 Example Analogy:**

Think of **JPA as an interface** and **Hibernate as a class that implements it**.

* 🔹 JPA: "Here are the rules you must follow to be a good ORM."
* 🔹 Hibernate: "Okay, I’ll follow those rules and provide real working code."

**✅ Common JPA Implementations**

* **Hibernate** ✅ Most popular one
* **EclipseLink**
* **OpenJPA**
* **TopLink (Oracle)**

**✅ In Spring Boot**

When you use Spring Boot with JPA:

<dependency>

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

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

Spring uses **Hibernate** as the default JPA implementation **behind the scenes**.

**✅ Summary**

| **Term** | **Meaning** |
| --- | --- |
| JPA | Standard/specification for ORM in Java |
| Hibernate | A tool that implements JPA |
| Spring Boot JPA | Uses Hibernate by default |

**🧠 Step-by-Step Flow: From Spring to Database**

**✅ 1. Spring Data JPA (Top layer)**

You write a simple **interface** like this:

public interface UserRepository extends JpaRepository<User, Long> {

User findByName(String name);

}

👉 Spring Data JPA **automatically provides** the implementation.

**✅ 2. JPA (Java Persistence API)**

* Spring Data JPA uses **JPA annotations and API** like @Entity, @Id, @Query, etc.
* These are **standard rules** that say how Java objects (entities) should map to DB tables.

👉 JPA is **just a specification** (rulebook). It can’t do the actual database work.

**✅ 3. Hibernate (JPA Provider)**

* Hibernate is the **actual tool** that implements the JPA specification.
* It converts your entity classes and JPA queries into **SQL queries**.
* Hibernate also handles things like caching, dirty checking, lazy loading, etc.

👉 So JPA talks to Hibernate under the hood.

**✅ 4. JDBC (Java Database Connectivity)**

* Hibernate internally uses **JDBC** to interact with the database.
* JDBC provides the API to send raw SQL to the database.

👉 Hibernate creates and sends SQL through JDBC.

**✅ 5. JDBC Driver (Connector)**

* JDBC calls are translated into **native protocol** by the JDBC **driver**.
* Example: MySQL JDBC driver knows how to talk to MySQL using its protocol.

👉 JDBC Driver sends the actual SQL to the database server.

**✅ 6. Database**

* The **database** (like MySQL, PostgreSQL, Oracle) receives the SQL,
* Executes it,
* Sends results back up the chain.

**🔁 Flow Diagram (Simplified)**

Spring Data JPA (you use)

⬇️

JPA (standard annotations & API)

⬇️

Hibernate (implementation of JPA)

⬇️

JDBC (low-level Java DB API)

⬇️

JDBC Driver (e.g., MySQL Connector)

⬇️

Database (e.g., MySQL)

**📦 Real-Life Code Sample**

You call:

java

CopyEdit

userRepository.findById(1L);

➡️ Spring Data JPA builds a JPA query  
➡️ JPA delegates it to Hibernate  
➡️ Hibernate creates SQL: SELECT \* FROM users WHERE id = 1  
➡️ JDBC sends SQL to MySQL through the driver  
➡️ MySQL executes and sends the result back

**✅ Spring Data JPA**

🔹 A **Spring framework module** that makes it easier to work with JPA.  
🔹 It provides **ready-made interfaces** like JpaRepository, CrudRepository that reduce boilerplate code.

**📌 What it contains:**

* Predefined interfaces (like JpaRepository, PagingAndSortingRepository)
* Auto-implemented query methods (like findByName())
* Integration with Spring Boot and Hibernate

**✅ JPA (Java Persistence API)**

🔹 A **specification** (i.e., a set of rules or interface definitions) provided by Oracle  
🔹 It defines **how** ORM (Object-Relational Mapping) should work in Java

**📌 What it contains:**

* Annotations like @Entity, @Id, @GeneratedValue
* Interfaces like EntityManager, Query
* It does **not** provide implementation

**🔄 How They Work Together**

| **Layer** | **Role** |
| --- | --- |
| **Spring Data JPA** | Helps you use JPA with less code (interfaces) |
| **JPA** | Tells how to map Java objects to DB (standard) |
| **Hibernate** | Does the actual work (implementation of JPA) |

**✅ Example Flow**

public interface UserRepository extends JpaRepository<User, Long> {

}

1. UserRepository → Comes from **Spring Data JPA**
2. Uses annotations like @Entity → Comes from **JPA**
3. Spring uses **Hibernate** (by default) to implement those JPA rules

So yes 👇:

✅ **Spring Data JPA** contains helpful **interfaces**  
✅ **JPA** is a **specification** (no real code, just rules)

Let me know if you want a diagram or code example to understand this better!

JPA Provides a standard of ORM in java applications, ensuring that developers can switch between different JPA providers without changing their code.

And Hibernate is one such JPA provider.

However,

Hibernate is a specific implementation of JPA and a powerful ORM framework on its own. It offers additional features and optimizations beyond the JPA specification, making it a popular choice for ORM in Java applications.

For Spring Boot JPA we need following dependencies:   
Spring-boot-starter-data-jpa:

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

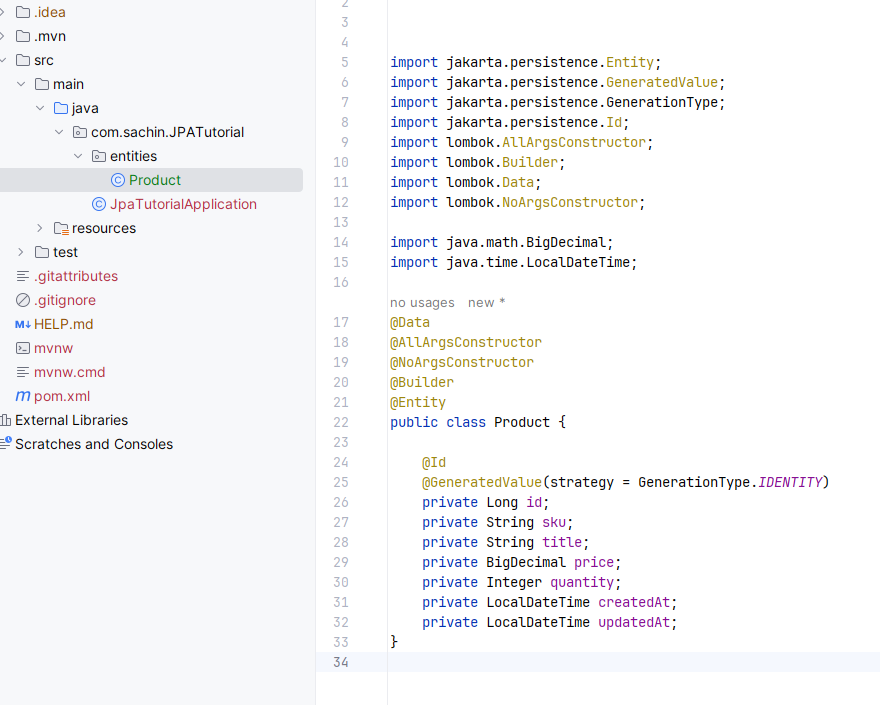
For MySQL Driver :

<dependency>  
 <groupId>com.mysql</groupId>  
 <artifactId>mysql-connector-j</artifactId>  
 <scope>runtime</scope>  
</dependency>

Spring-boot-starter-web:

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

After this we will create entity:



So here to create or to map our entity class with table we used @Entity annotation.

@Id : It is used to denote the primary key of our table .

@GeneratedValue(startergy = GenrationType.IDENTITY) : It means it will generate values of id sequentially we don’t need to give it.

@Data : will automatically add Getter,setter,toString() , hashcode and equals method, and constructor with final fields.

@AllArgsConstructor : It is used to create constructor with all fields

@NoArgsConstructor : it is used to create no args constructor.

@Builder : By using this we can easily create DTO from entity . In one sentence it is used to create object by using build() method. Useful when only some fields are needed or fields are optional.

**✅ What is @Builder in Lombok?**

**@Builder** is a Lombok annotation that lets you **create objects in a flexible and readable way**, using the **Builder design pattern**.

**🔧 What It Does**

When you use @Builder on a class or constructor, Lombok automatically:

* Creates a **static inner builder class**
* Adds **chained setter methods** for all fields
* Adds a build() method to create the object

**🧠 Why Use It?**

* Avoid long constructors with many parameters
* Improve **readability** and **flexibility** when creating objects
* Useful when only some fields are needed or fields are optional

**📦 Example Without @Builder**

public class User {

private String name;

private int age;

private String city;

public User(String name, int age, String city) {

this.name = name;

this.age = age;

this.city = city;

}

}

You would create the object like:

User user = new User("John", 25, "Pune");

If you forget the order or skip a field — it’s messy.

**✨ Example With Lombok @Builder**

import lombok.Builder;

@Builder

public class User {

private String name;

private int age;

private String city;

}

Now you can create the object like this:

User user = User.builder()

.name("John")

.age(25)

.city("Pune")

.build();

Much cleaner, flexible, and **no need to remember parameter order**.

**✅ Bonus: You can use @Builder with constructors or methods too!**

@Builder

public User(String name, int age) {

this.name = name;

this.age = age;

}

**🚫 Note**

* If you're using @Builder and @Data together, you **may want to use @AllArgsConstructor or @NoArgsConstructor** explicitly if needed.
* Fields must be set through the builder, not through setters unless you also use @Setter.

Common Hibernate Configurations:

* 1. **spring.jpa.hibernate.ddl-auto=update/create/validate/create-drop/none**

ddl: data definition layer: It is used to manipulate schema i.e. table

update: It is used to update the table every time we run it will use previous data.   
create: Every time you run the application it will create new table.

Validate: It will validate whether table we have and entity we are getting if both different validate will give error, so we use in production only.

create-drop: It will create table as soon as server i.e. application run and drop the table as soon as we stop server i.e. application.

none : used if we don’t want to change anything when entity is update .

**🔍 Available Options (with Explanation & Use Case)**

| **Option** | **Meaning** | **When to Use** |
| --- | --- | --- |
| **none** | Do **nothing** with schema | When your schema is **managed manually**, like in production |
| **create** | **Drops** and **recreates** tables **every time** app starts | For **testing or prototyping** — ⚠️ all data will be lost on restart |
| **create-drop** | Same as create, but **drops tables when app stops** | Ideal for **JUnit tests** or temporary testing with in-memory DBs |
| **update** | **Modifies schema to match Entity** classes without dropping data | ✅ Safe for **development** (but not production) — auto-applies small changes |
| **validate** | Just checks if schema matches Entities — throws error if not | ✅ Good for **production** to **ensure consistency** (no auto changes) |
| **none** | Completely disables Hibernate DDL management | For **full manual control** |

**✅ Which One Should You Use?**

| **Environment** | | **Recommended ddl-auto** | | **Why?** | |
| --- | --- | --- | --- | --- | --- |
| 🧪 Local dev | | update | | Keeps data and applies changes automatically | |
| 🧪 Testing | | create-drop | | Clean DB each time; great for test isolation | |
| 🚀 Production | | validate or none | | Prevents accidental schema changes | |
| 🛠 Staging | validate or manual scripts | | Closest to production, for final testing | |

2.spring.jpa.show-sql=true: it will tell hibernate every query you are creating show us.

3. spring.jpa.properties.hibernate.format\_sql=true: Normally query will show in one line but if we want to see in beautiful way query, we can use this.

4. spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect (Optional)  
It is used to define the rule that the hibernate will use to convert JPQL to the corresponding query language. here SQL.

We need to add this configuration in application.properties

Entity Annotations:

@Entity: It will tell hibernate to create table of this class which is annotated with @Entity.  
@Table: By using we can change default table name any many more

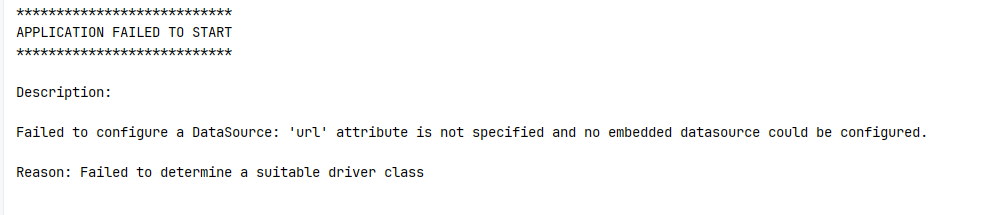
Ex :

@Table(  
 name = "product\_table",  
 uniqueConstraints = {  
 @UniqueConstraint(name = "sku\_unique",columnNames = {"sku"}),  
 @UniqueConstraint(name = "title\_price\_unique",columnNames = {"title\_x","price"})  
 },  
 indexes = {  
 @Index(name = "sku\_index",columnList = "sku")  
 }  
)

name = to change the default name of table  
@UniqueConstraint : it will tell which coloumn should have unique value  
If you give more than one it means combination of this should have unique value : Example in second uniqueconstraint : you given two cols i.e title\_x and price it means for example : parleG is title and price is 50 , it can new record of parleG title and price any but not 50.  
and if price is 50 then it can any title but not parleG.  
@Index : It is used for fast processing : It will create one separate index table for fast access . but default index table is create for @Id column.

@Id  
@GeneratedValue(startergy=GenerationType.IDENTITY)  
@Column : We can change the default name of column in table we can apply restrictions ,length . Ex : @Column(name=”name”,nullable=false,length=50)  
@CreationTimestamp and @UpdateTimestamp : It will store creation time stamp of record and updated time stamp of record by default

If we run application it this stage we will get error :



Because still we haven’t given the data source URL : for mysql because we are not using H2.

spring.datasource.url=jdbc:mysql://localhost:3306/sachin?useSSL=false  
spring.datasource.username=root  
[spring.datasource.password=MySQL@45](mailto:spring.datasource.password=MySQL@45)

Note : We can give data to table from file also : We use this in dev for testing our application

We need create file with extension : .sql : Ex : data.sql in resource folder.

We need add configuration in application.properties :

*#we are telling database is initialized by us*spring.jpa.defer-datasource-initialization=true

*#we want always initialize with given data*spring.sql.init.mode=always  
*#path of data*spring.sql.init.data-locations=classpath:data.sql

data.sql :

INSERT INTO sachin.product\_table(price,quantity,sku,title\_x)

VALUES

(12.4, 4, 'parle734', 'Parle Buscuit'),

(14.4, 1, 'pepsi123', 'Pepsi'),

(16.4, 3, 'pepsi156', 'Mazza');

**✅ 1. spring.jpa.defer-datasource-initialization=true**

📌 **Meaning:**  
Tells Spring Boot to **wait** for the **database schema to be ready** before running data.sql.

🔧 **Why needed?**

* By default, Spring Boot initializes data **before** JPA creates tables — which can cause errors if your data.sql tries to insert into tables that don’t exist yet.
* This line delays the data.sql execution **until after** the schema is created by JPA.

**✅ 2. spring.sql.init.mode=always**

📌 **Meaning:**  
This forces Spring Boot to **always run SQL init scripts**, even when a real database (not in-memory) is used.

🔧 **Without this**, Spring only runs data.sql with in-memory DBs like H2 by default.

**✅ 3. spring.sql.init.data-locations=classpath:data.sql**

📌 **Meaning:**  
This tells Spring Boot **where to find your SQL file**.

✔️ It will look for data.sql inside the resources folder (because classpath: means the src/main/resources directory).

**📦 Folder Structure Example**

Make sure you place your file like this:

css

CopyEdit

src/

└── main/

└── resources/

└── data.sql

**✅ Summary**

| **Property** | **Purpose** |
| --- | --- |
| spring.jpa.defer-datasource-initialization=true | Wait for JPA schema before inserting data |
| spring.sql.init.mode=always | Always run data.sql even with real DBs |
| spring.sql.init.data-locations=classpath:data.sql | Points to your SQL file for loading data |

Key Features of JPA:

* Entity Management: Defines how entities (Java Objects) persisted to the DB
* Query Language: Provides JPQL (Java Persistence Query Language) for querying entities.
* Transactions: Manage transactions, making it easier to handle db operations within a transactional context
* Entity Relationships: Supports defining relationships between entities (e.g., One-to-one, One to many, many to one, many to many)
  1. **: Spring Data JPA Interfaces and Dynamic Query Methods:**

Spring Data JPA: It is part of the larger spring data family

* It builds on top of JPA, providing a higher-level and more convenient abstraction for data access.
* Spring Data JPA makes it easier to implement JPA-based repositories by providing boilerplate code, custom query methods, and various utilities to reduce the amount of code you need to write.

Which are the interfaces present for Spring Data JPA:

JpaRepository<T, ID> : interface ( which has most capability)  
PagingAndSortingRepository<T,ID> : interface ( for pagination and sorting)  
CrudRepository<T,ID> : interface (basic crud operations)  
Repository<T,ID> : interface ( parent of CrudRepository)  
QueryByExampleExecutor<T> : interface

-JpaRepository extends PagingAndSortingRepository, QueryByExampleExceutor

-PagingAndSortingRepository extends CrudRepository

-CrudRepository extends Repository

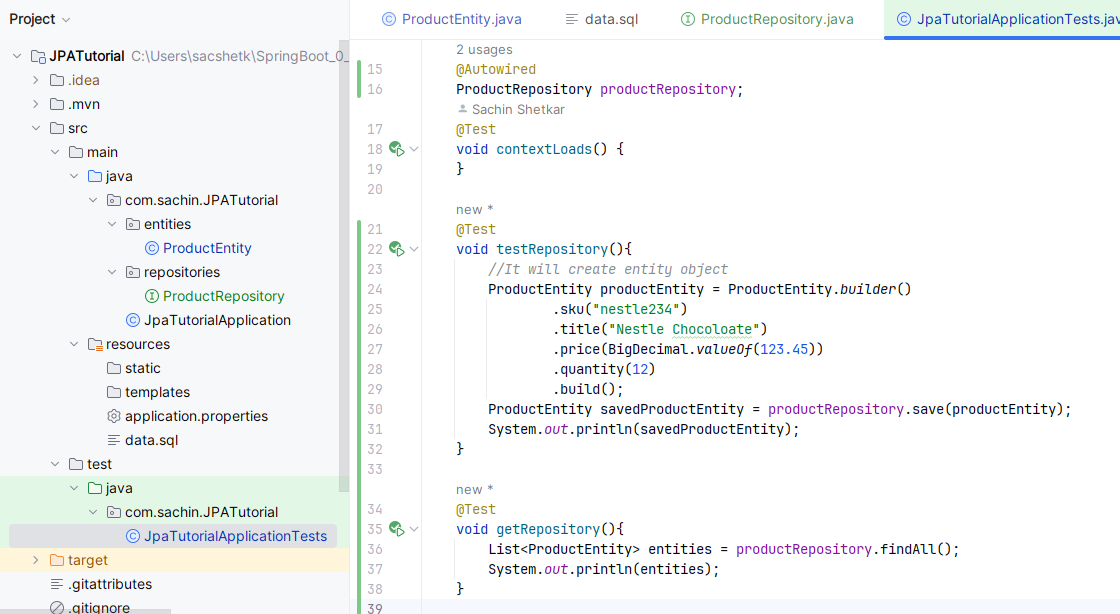
Who is implementing all these methods of this interfaces? SimpleJpaRepository class.

Simple We need to create one interface which extends JpaRepository in repositories package :



We can use now all the queries i.e. methods of all the interfaces.

For testing we can write test cases :



Only you need to inject your repository by using @Autowired annotation.

---------------------------------------------------------------------------------------------------------Key Features of Spring Data JPA:

* Repository Abstraction: Provides a Repository interface with methods for common data access operations.
* Custom Query methods: Allows defining custom query methods by simply declaring method names.
* Pagination and Sorting: Offers built-in support for pagination and sorting.
* Query Derivation: Automatically generates queries from method names.

Rules for Creating Query Methods:

List<Product> findByDateCreatedBetween(LocalDateTime startDate,LocalDateTime endDate);

Return type should be mostly: 1. List<T> 2.Optional<T> 3.T  
method name i.e Query subject + Query Predicate(DateCreated + Between(Operator))  
Input Params : should be type of Query Predicate

Rules for method names:

* The name of our query method must start with one of the following prefixes : find… By , read…By, query….By and get…By

Ex : findByName,readByName,queryByName,getByName : All are for retrieving data from db .

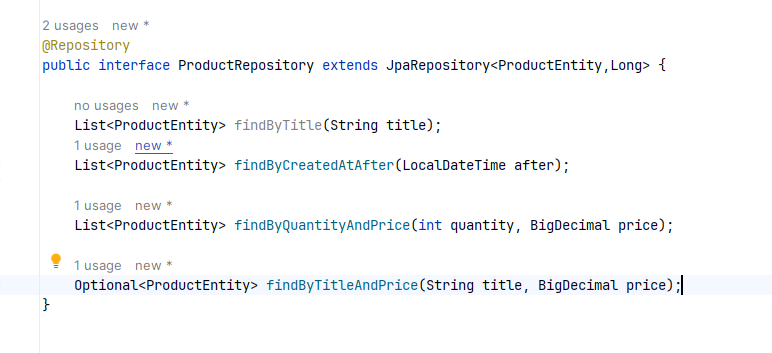
* If we want to limit the number of returned query results, we can add the first or the Top keyword before the first By word

Ex : findFirstByName, readFirst2ByName , findTop10ByName

* It we want to select unique results , we have to add the distinct keyword before the first by word.

Ex : findDistinctByName or findNameDistinctBy

* Combine property expression with AND or OR.  
  ex : findByNameOrDescription, findByNameAndDescription





By writing this method you will not be able to write custom queries that may be complex like aggregate function second max like many other for that you need write your custom query by using annotation @Query : in JPQL or in SQL

@Query("select e from ProductEntity e where e.title=?1 and e.price=?2")  
Optional<ProductEntity> findByTitleAndPrice(String title,BigDecimal price);

e means in JPQL : everything : like \* in SQL

Notes :

**✅ 1. Using Method Name Conventions (Derived Queries)**

Spring Data JPA generates the query based on the method name.

**✅ Rules:**

* Must follow specific **naming keywords** like findBy, countBy, existsBy, deleteBy.
* Property names must exactly match the **field names** in the Entity.
* Can chain conditions with And, Or, Between, In, Like, etc.

**✅ Examples:**

java

CopyEdit

// Find by single field

List<ProductEntity> findBySku(String sku);

// Find by multiple fields

List<ProductEntity> findByTitleAndPrice(String title, BigDecimal price);

// Find by range

List<ProductEntity> findByPriceBetween(BigDecimal min, BigDecimal max);

📌 Fields in method name must match **Java property names**, not DB column names.

**✅ 2. Using @Query Annotation (JPQL or SQL)**

**✅ Syntax:**

java

CopyEdit

@Query("SELECT p FROM ProductEntity p WHERE p.price > :price")

List<ProductEntity> findProductsCostlierThan(@Param("price") BigDecimal price);

**✅ Rules:**

* Use **entity class names** and **field names**, not table or column names.
* JPQL supports JOINs, WHERE, ORDER BY, etc.

**✅ Native SQL:**

java

CopyEdit

@Query(value = "SELECT \* FROM product\_table WHERE sku = :sku", nativeQuery = true)

ProductEntity findBySkuNative(@Param("sku") String sku);

✅ Use nativeQuery = true only when needed — for complex joins, DB functions, etc.

**✅ 3. Using @Modifying for Update/Delete**

Use this for update or delete queries.

**✅ Example:**

java

CopyEdit

@Modifying

@Query("UPDATE ProductEntity p SET p.quantity = :qty WHERE p.sku = :sku")

int updateQuantity(@Param("sku") String sku, @Param("qty") int qty);

**✅ Rules:**

* Must add @Transactional at method or class level.
* Only works with @Modifying.

**✅ 4. Optional Return Types**

* You can return:
  + Optional<T>
  + List<T>
  + T
  + Page<T> (for pagination)
  + Slice<T>

**🚫 Common Mistakes to Avoid**

| **Mistake** | **Fix** |
| --- | --- |
| Using DB column names in method name | Use **Java field names** |
| Missing @Param("...") in @Query | Always bind parameters |
| Not adding @Modifying to update/delete | Add @Modifying |
| Using nativeQuery without column mapping | Ensure column names match DB table |
| Wrong return type | Match the query result type |

**✅ Summary**

| **Approach** | **Use When** |
| --- | --- |
| Method name query | Simple queries, quick to write |
| @Query with JPQL | Custom logic, joins, flexible |
| @Query with native SQL | Complex DB-specific logic |
| @Modifying + @Query | Update/delete queries |

Let me know if you want examples for:

* Pagination with Pageable
* Custom DTO projection
* Complex joins in JPQL

**✅ Common Chain Conditions You Can Use in Method Names**

| **Keyword** | **Description** | **Example** |
| --- | --- | --- |
| And | Both conditions must be true | findByTitleAndPrice |
| Or | Either condition can be true | findByTitleOrSku |
| Between | Value lies between two bounds | findByPriceBetween(BigDecimal start, BigDecimal end) |
| LessThan / LessThanEqual | Field is less than (or equal) to value | findByPriceLessThan(100) |
| GreaterThan / GreaterThanEqual | Field is greater than (or equal) to value | findByQuantityGreaterThan(5) |
| IsNull / IsNotNull | Check for null or not null | findByUpdatedAtIsNull() |
| In | Field value is in the list | findBySkuIn(List<String> skus) |
| NotIn | Field value is not in the list | findBySkuNotIn(List<String> skus) |
| Like | SQL LIKE with % | findByTitleLike("%pepsi%") |
| NotLike | SQL NOT LIKE | findByTitleNotLike("%coke%") |
| StartingWith | Starts with (LIKE 'xxx%') | findBySkuStartingWith("pepsi") |
| EndingWith | Ends with (LIKE '%xxx') | findBySkuEndingWith("123") |
| Containing / Contains | Contains substring (LIKE '%xxx%') | findByTitleContaining("milk") |
| Not | Negation of condition | findBySkuNot("abc123") |
| True / False | For boolean fields | findByActiveTrue() |
| IgnoreCase | Case-insensitive comparison | findByTitleIgnoreCase(String title) |

**✅ Examples of Chained Conditions:**

java

CopyEdit

// AND chaining

List<ProductEntity> findBySkuAndQuantity(String sku, Integer quantity);

// OR chaining

List<ProductEntity> findBySkuOrTitle(String sku, String title);

// Range chaining

List<ProductEntity> findByPriceBetweenAndQuantityGreaterThan(BigDecimal low, BigDecimal high, int qty);

// Complex chaining

List<ProductEntity> findByTitleContainingAndPriceLessThanEqualAndQuantityGreaterThanEqual(

String title, BigDecimal maxPrice, int minQty);

**🧠 Tips**

* Always use **Java field names**, not column names.
* Chained methods can get long — use @Query if it becomes too complex.
* For better performance, consider indexing fields used in chained conditions.
  1. : Sorting and Pagination in Spring Data JPA:

How to sort the data?

Ex : If we go on amazon website we can sort by prize(low to high & high to low) . How we can do that.

How to get the view in paginated view?

Ex : We will not get the data on first page only in amazon it will divided in multiple page .   
i.e Pagination. It will faster our pc.

* Sorting :

First way : by using : “OrderBy” function.

@Repository

Public interface EmployeeRepository extends JpaRepository<Employee,Long> {

List<Employee> findByOrderByNameAsc();

List<Employee> findByOrderByNameDesc();

}

If we are using OrderBy we will need many methods for different field example OrderBy ex : findByOrderByPrice() , findByOrderByQuantity(), findByOrderByCreatedAt(), many more.

To overcome this tight coupling to specific field in method we can use different way :

i.e Sorting with the Sort class

@Repository  
public interface EmployeeRepository extends JpaRepository<Employee,Long> {

List<Employee> findByDepartment(String department,Sort sort);

}