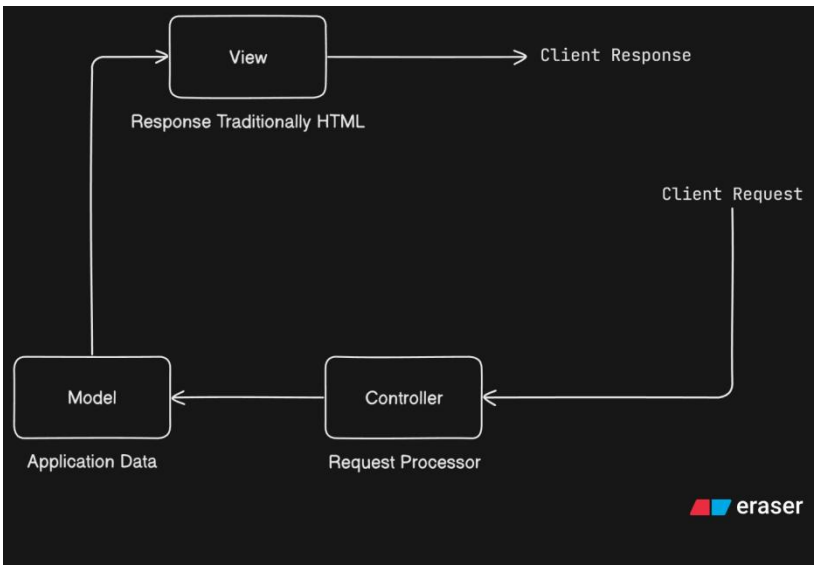
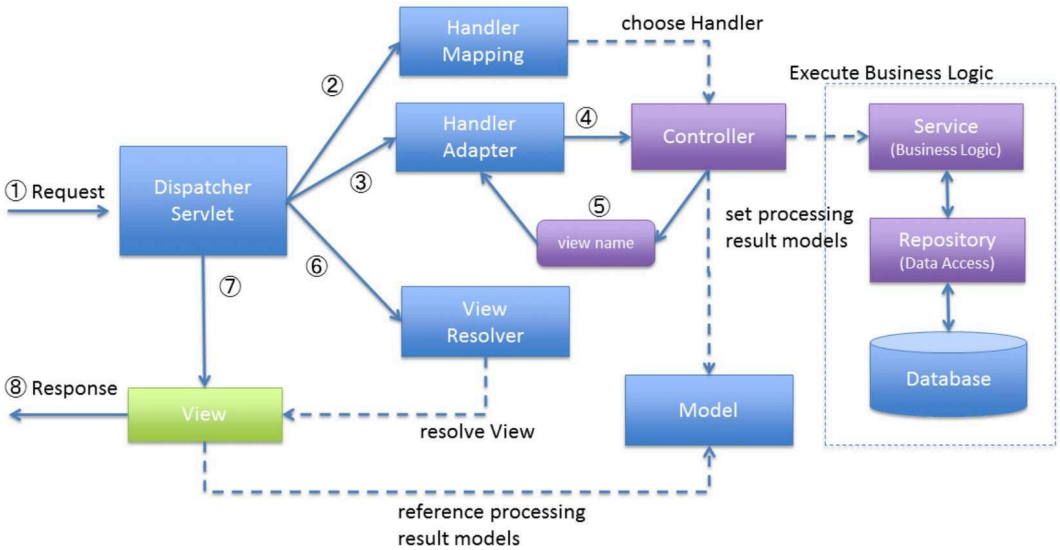
MVC Architecture



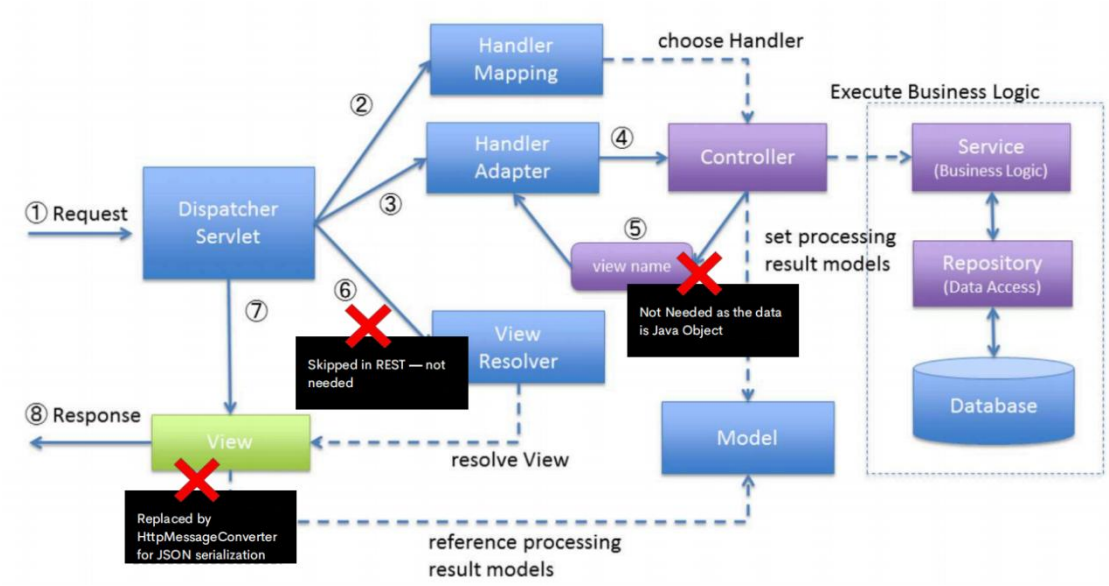
- Controller
  - Different APIs (Get, Post etc) will be having different controller
- Model
  - Handling the data
  - Talking to Database
  - In case of Java, Model is present in the Object
- View
  - HTML or JSON
  - Send the data to the client

- **Spring MVC** (Legacy)



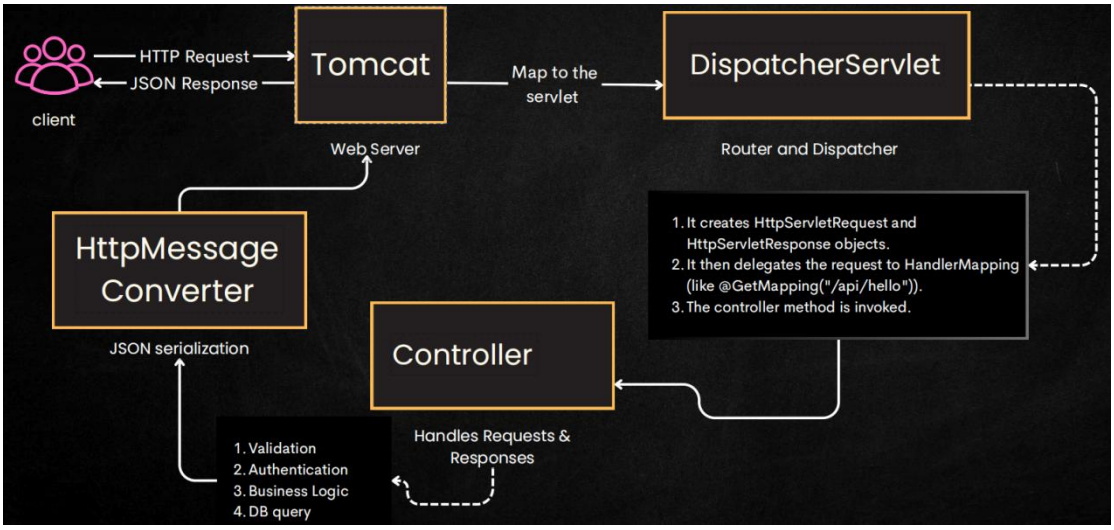
- View Resolver will take a look at the Dispatcher Servlet that which kind of user has requested and then it converts the Model to that format i.e. XML, HTML, JSON etc.

➢ **Spring MVC** (Modern)

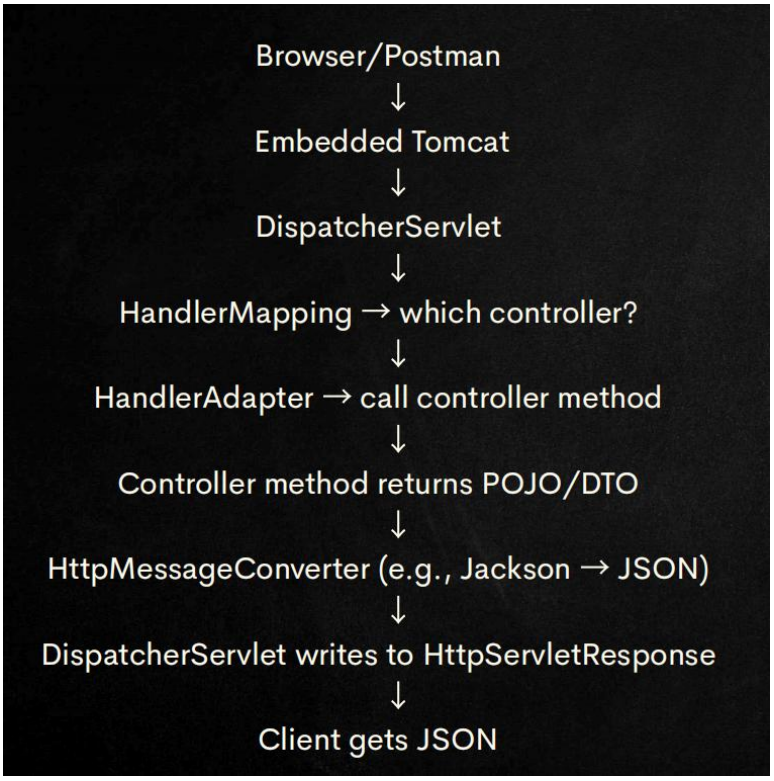


- In recent times, JSP is not used.
- React or Angular etc are used to create Frontend and use Spring to create JSON.

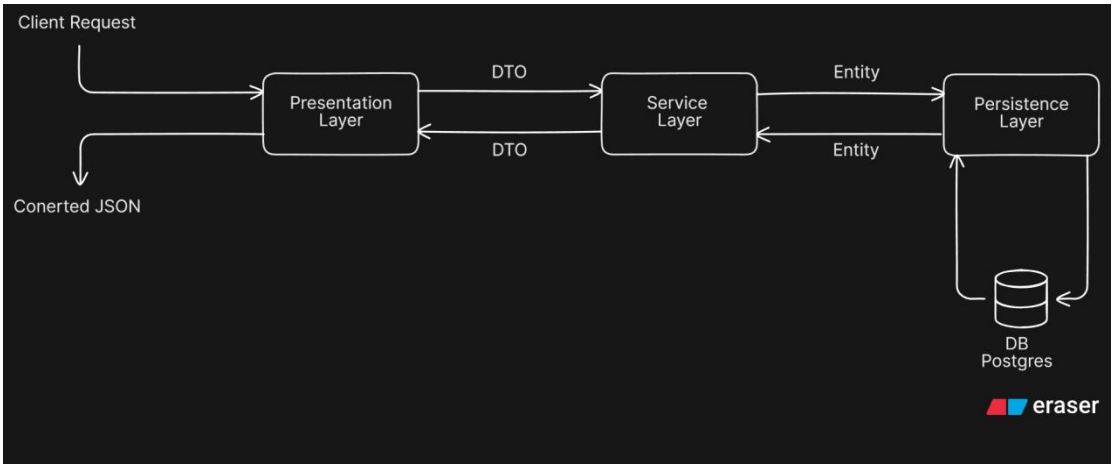
➢ **How does a Web Server works in Spring Boot?**



- The client request cannot go to the **Dispatcher Servlet** directly as it's a Java/Spring entity and not able to handle http request directly.
- So the request go to a **web server** (tomcat, jetty etc.. Spring Boot uses Tomcat by default) and then it transfer those requests to **Dispatcher Servlet**.

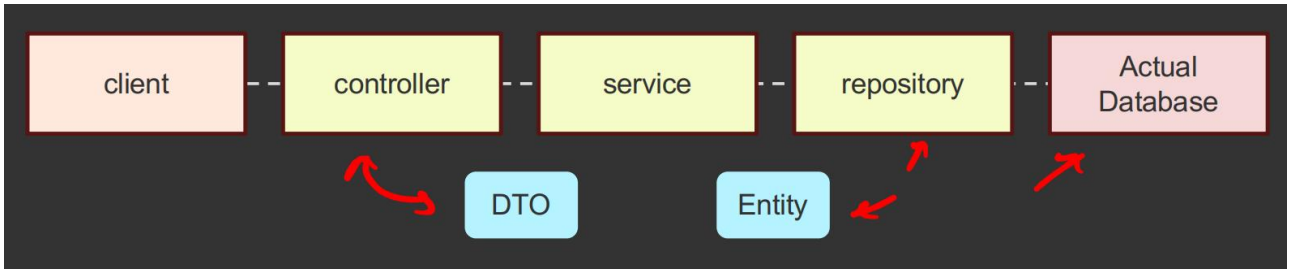
- We can have multiple requests like *profile request* or *admin request* or *reels request* etc etc. So these routing part is done in here i.e. **Dispatcher Servlet**.
  - 2 Objects i.e. **HttpServletRequest** and **HttpServletResponse** are created in this part only.
  - In that **HttpServletRequest** contains everything like headers, query, ids params.
  - After doing the things, we can update the **HttpServletResponse** and return it as the response.
- 

- 3 layered architecture



- **Layers**
  - **Presentation Layer** *(Handle Request and Response)*
    - Request from client will be handled by this
    - Communication between *Presentation* and *Service* layer happens with transfer of **DTO** *(Data Transfer Object)*
    - Lets say user is trying to login, so payload of the request may be containing username, password, email etc.. those things will be converted to an Object (DTO) and sent to the *Service* Layer.
  - **Service Layer** *(Business Logic)*
    - It'll check the things like is the user is valid with the help of Persistence Layer.
    - Service and Persistence layer have conversation by transferring *Entity*.
  - **Persistence Layer** *(Data Access Layer)*
    - It has only the access to database.
    - So, *Service* layer verifies the things with the help of this *Persistence* layer only.
- In simple terms: A DTO is the structure received from (or sent to) the user in API requests/responses, and an Entity is the structure used to store data in the Database."

Presentation Layer

- 

- **@RestController**
  - It marks a class as a REST controller and automatically applies *@ResponseBody* to every method.
  - ```
    @RestController  no usages
    public class EmployeeController {
    ```
  - You can see in the **@RestController**, it has one annotation called **@ResponseBody**
    - ```
      @Controller
      @ResponseBody
      public @interface RestController {
          @AliasFor(
                  annotation = Controller.class
      ```
    - It ensures the returned Java object is converted to JSON/XML and written to the HTTP response body using message converters.
  - To convert incoming JSON/XML to Java objects, Spring uses **@RequestBody**, not **@ResponseBody**.
  - If you write **@Controller** instead of **@RestController** then it'll think "user is the name of a view template (i.e. User.jsp, User.html) and it'll try to load the UI page and fail".
    - To make it return JSON then you need to write **@ResponseBody** in each method.
- **NOTES**
  - Spring MVC provides an annotation-based programming model where **@Controller** and **@RestController** components use annotations to express request mappings, request input, exception handling, and more.
  - The **@RestController** annotation is a shorthand for **@Controller** and **@ResponseBody**, meaning all methods in the controller will return JSON / XML directly to the response body.

- With **@RestController**, we don't need to explicitly write **@ResponseBody** for each methods to make it return *JSON*. (But with **@Controller** we need to write explicitly).

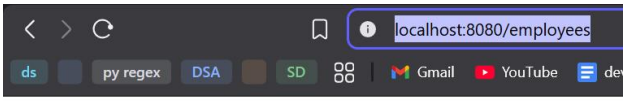
- **Request Mappings**
  - You can use the **@RequestMapping** annotation to map requests to controllers methods. It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.
  - **@GetMapping**, **@PostMapping** ..etc are there for **GET, POST, ..**etc type of requests.
- See the below code

```
@RestController  no usages
public class EmployeeController {

    @GetMapping(path = "/getSecretMessage")  no usages
    public String getMySuperSecretMessage() {
        return "Secret Message: abcd87957hw4895";
    }
}
```

  - Now this path **"/getSecretMessge"** is exposed.
  - But the question is how was it become live just by writing the code here? We didn't even used this method anywhere to make it live.
    - So, as we discussed earlier, Component Scan happens for all the files present on the same or child directories where the **main** method file is placed.
    - **@RestController** contains **@Controller** inside it; and **@Controller** contains **@Component** inside it.
    - It'll check for all the *annotations* and makes Bean out of those directly and use.
- To create the DTO, there is no specific annotations.

```
public class EmployeeDTO {  no usages
    private Long id;  no usages
    private String name;  no usages
    private String email;  no usages
    private Integer age;  no usages
    private LocalDate dateOfJoining;  no usages
    private Boolean isActive;  no usages
}
```

- See the below use cases to pass **params** in URL
  - ```
    @GetMapping("/employees/{employeeId}/{someMore}")  no usages
    public EmployeeDTO getEmployeeById(@PathVariable String employeeId,
                                       @PathVariable String someMore) {
    ```
    - If you see here I have set the **argument** names in the method same as the **params** name in the URL.
    - If you give different names then it'll not work.
    - The fix for this i.e. if you want to give different argument names then use **@PathVariable("real-param-name") DataType newArgument**
    - ```
      @GetMapping("/employees/{employeeId}/{someMore}")  no usages
      public EmployeeDTO getEmployeeById(@PathVariable("employeeId") String id,
                                         @PathVariable("someMore") String more) {
      ```
  - You can see the output will be in JSON format even if we didn't explicitly convert this to JSON.
    - ```
      {
          "id": 89734394873849,
          "name": "Alok",
          "email": "something@gmail.com",
          "age": 24,
          "dateOfJoining": "2020-01-01",
          "active": true
      }
      ```
    - There is a library **jackson** which does this things.
- To get the **query** values, use the annotation **@RequestParam**
  - ```
    @GetMapping(path = "/employees")  no usages
    public String getAllEmployees(@RequestParam Integer age) {
    ```
  - The URL should be like: **/employees?age=40** (something like this)
  - But if you open only **/employee** it'll not load the page
    - 

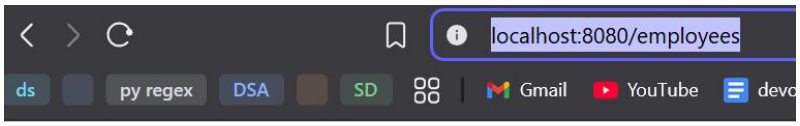
      Whitelabel Error Page

      This application has no explicit mapping for /error, so you are seeing th

      Wed Dec 10 02:18:42 IST 2025
      There was an unexpected error (type=Bad Request, status=400).
  - It is because the query param is mandatory here.
  - We need to make it optional. **@RequestParam(required = false)**

```
@GetMapping(path = "/employees")  no usages
public String getAllEmployees(@RequestParam(required = false) Integer age) {
```

- Now you can see the page is loading



Hi age = null

- Instead of writing **/employees** everywhere, we can set the path for **@RequestMapping** at the top level.

```
@RestController  no usages
@RequestMapping(path = "/employees")
public class EmployeeController {
```

- Now you don't need to write **/employee** in every methods.

```
@RestController  no usages
@RequestMapping(path = "/employees")
public class EmployeeController {

    @GetMapping   // here no need to define anything  no usages
    public String getAllEmployees(@RequestParam(required = false) Integer age) {
        return "Hi age = " + age;
    }

    @GetMapping("/{employeeId}")  no usages
    public EmployeeDTO getEmployeeById(@PathVariable Long employeeId) {
        return new EmployeeDTO(employeeId);
    }
}
```

- Other mappings are also there: **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**, **@PatchMapping**
- There are so many **@Request** annotations to get the **query, body, header** etc etc.

```
@ RequestParam or
@ RequestMapping
@ RequestBody org
@ RequestAttribut
@ RequestHeader o
@ RequestPart org
@ RequestScope or
```

---------- **SOME POINTS** ----------

- There is a dependency called **lombok** which provides **getters, setters, constructors** by default. No need of defining this by our own.

```
@Getter  7 usages
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class EmployeeDTO {
    private Long id;
    private String name;
```

- **spring.datasource**
  - It is a spring boot configuration property used to set up *database connection*.
  - It tells spring boot:
    - Which database to connect to
    - How to connect
    - Which driver to use
    - Database username & password.

```
spring.datasource.url=jdbc:mysql://localhost:3306/mydb
spring.datasource.username=root
spring.datasource.password=1234
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

- **Presentation Layer**, **Service Layer**, **Persistence Layer**
  - These 3 are just architectural patterns. It has nothing to do with **spring/spring boot** framework.
  - Spring Boot just makes it easy to implement these architectural patterns.
  - Spring Components for the Layer
    - **Presentation** -------- **Controller** (**@RestController**)
    - **Service** ---------- **Service** (**@Service**)
    - **Persistence** ---------- **Repository** (**@Repository**)

```
Controller → Service → Repository → JPA → Hibernate → JDBC → Driver → Database
```

  - This is how the work happens between each layers.

Level	What It Does	VERY Short Explanation
Repository (Spring Data)	Converts repository methods (`save`, `findAll`) into JPA API calls	"Method → JPA function"
JPA (Specification)	Provides API (`persist`, `find`, `remove`) but does NOT implement them	"Rules + interfaces only"
Hibernate (JPA Provider)	Implements JPA functions → generates SQL → talks to JDBC	"Real ORM + SQL generator"
JDBC	Sends SQL to the database and gets results	"Communication pipe"
Database Driver	Converts JDBC calls to DB-specific protocol	"Translator"
Database	Stores and returns actual data	"Storage engine"

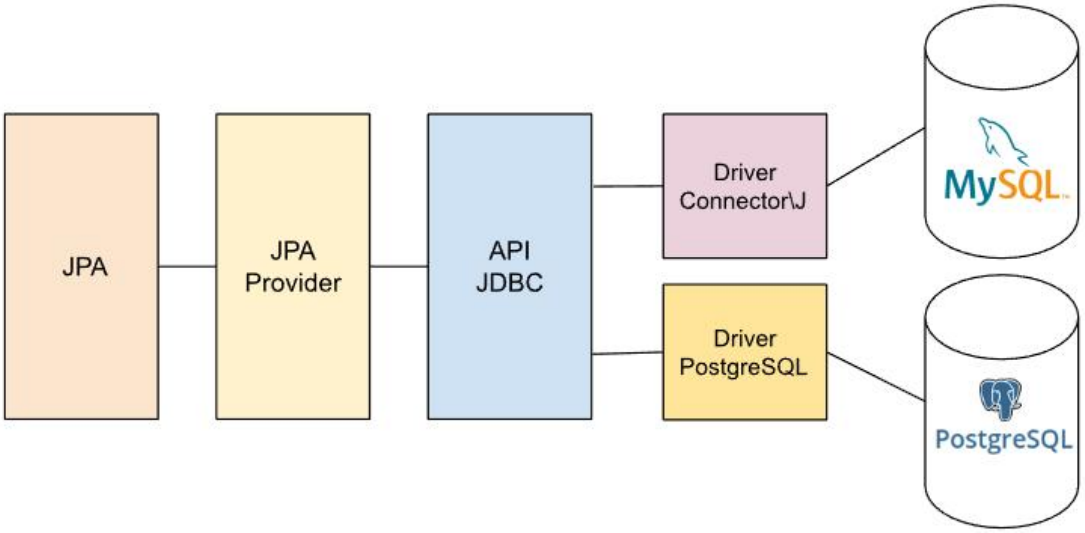
- You call high-level functions *(save(), findAll())*
- Spring Data + JPA + Hibernate automatically generate optimized SQL for you

---

➢ WORK-FLOW FROM REPOSITORY TO DATABASE

- There are **two** levels in Spring Data JPA:
  - **Repository** level
  - **EntityManager** (JPA) level.
- JPA itself consists of only **interfaces** (no implementation) and the **EntityManager** API.
- When we call a repository method like *findById()*, Spring Data JPA creates a **proxy implementation of the repository interface** that contains implementations of high-level methods (findAll, findById, save, etc.).
  - **Proxy implementation** means Spring creates a class at runtime that acts as the real implementation of your repository interface — even though YOU never wrote that class.
  - It takes the implementations from a class called **SimpleJpaRepository** which has implementation of all the abstract methods of **JpaRepository** interface.
- That proxy internally delegates to the **EntityManager**, which triggers low-level JPA operations such as persist(), merge(), find(), and remove().
  - These methods are low level
  - It is there in between Repository and Hibernate level
- **Up to this point, everything was at the JPA level (only rules + API, no actual SQL).**
- Next, the call goes to the **JPA provider (Hibernate)**.
  - Hibernate implements JPA and **converts entity operations into actual SQL** queries.
- After Hibernate generates SQL, it sends the SQL to **JDBC**.
- JDBC is responsible for:
  - **creating/maintaining a secure connection** to the database, and acting as the **transport layer to send Hibernate's SQL to the database driver**.
- Finally, the database driver sends the SQL to the database server for execution.

Persistence Layer



-
- **JPA**: Java Persistence API, nowadays its called "***Jakarta Persistence API***"
- Databases are also some servers but they hold some unique logic so that they can hold some data and query through it.
- Databases provide **JDBC** *(Java Database Connection)* drivers. Java uses JDBC to communicate with the database using those drivers.
- JPA is a specification. Hibernate is the most commonly used JPA provider that actually performs **ORM (object–relational mapping)**.
  - **Object-relational mapping** means *JPA* works with Java *objects*, the *database* works with *tables*, and the *JPA provider* (like Hibernate) maps the Java *objects* to the database *tables* and vice versa.
  - We can imagine JPA as a set of rules, and a JPA provider as the implementation of those rules.
  - JPA defines interfaces and abstract classes that describe how entities should be mapped, persisted, and managed.
  - But JPA itself does not contain actual persistence logic.
  - A JPA provider (like Hibernate) implements those rules and performs the real work such as generating SQL, mapping objects to database tables, managing relationships, and handling transactions.
- **JPQL**
  - It is an object-oriented query language used in JPA to query entities, NOT database tables.
  - You write queries on entity classes, not table names
  - You use field names, not column names

- It works on Java objects, not database records
- *The JPA provider (Hibernate) converts **JPQL → SQL** internally*

➢ In simple terms:

- **JPA** = What to do (not how to do it)
- **JPS Provider (Hibernate)** = How to do it
- **JDBC** = The pipe through which SQL is sent to the DB
- **Driver** = Translator for a specific database
- **Database** = The storage engine

➢ **@Entity** is used to define a class as an entity.

- ```
  @Entity  no usages
  public class EmployeeEntity {
  ```
- It'll just tell the Spring Data JPA (Hibernate) that "**this is the Java class. You need to convert it to table and store in the database**".
- ```
  @Entity  no usages
  @Table(name = "employee")
  public class EmployeeEntity {
  ```
  - If you don't give that **@Table** then it'll create the table with name **EmployeeEntity**

➢ There are annotations to define the SQL things. For example:

- ```
  @Id  no usages
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
  ```
- Here **@Id** means ***Primary Key*** in SQL.
- **@GeneratedValue** with **strategy** means the strategy of incrementing the **Id**.
- Different databases like mysql, postgres are having different type of strategy. So I selected ***AUTO*** so that it'll work everywhere.

➢ After creating the **Entity**, you need to create the **Repository**.

- The annotation will be **@Repository**.
- ```
  @Repository  no usages
  public interface EmployeeRepository extends JpaRepository<EmployeeEntity, Long> {
  }
  ```
- You need to extend **JpaRepository** (which is an interface) which is having all the *definitions of methods* like findAll, findAllById etc etc.
- Basically **JpaRepository** provides set of **CRUD** operations and query methods.
  - Generic Interface

- Predefined Methods
- Custom Queries

- Here the type which has to be passed to **JpaRepository** generic will be:
  - 1st one is type of **entity**
  - 2nd one is type of **primary key (@Id)**
- The **@Repository** is containing **@Component** so **bean** will automatically be created by Spring.

```
@GetMapping("/{employeeId}")  no usages
public EmployeeEntity getEmployeeById(@PathVariable(name = "id") Long id) {
    return employeeRepository.findById(id).orElse( other: null);
}

Change signature
@GetMapping   // here no need to define anything  no usages
public List<EmployeeEntity> getAllEmployees() {
    return this.employeeRepository.findAll();
}

@PostMapping  no usages
public EmployeeEntity createNewEmployee(@RequestBody EmployeeEntity inputEmployee) {
    return this.employeeRepository.save(inputEmployee);
}
```

- Created some controller methods inside the **EmployeeController** class using **EmployeeEntity**.
- But this shouldn't be done; Repository part and Controller part should not be mixed up.
- I have used the dependency **h2** which is used to simulate database things without using a real database.
- **If Entity class has private variables and you have not defined Getters and Setters then you will not get the result because it'll not be able to get those values.**

- You can see in the terminal
  - `Added connection conn0: url=jdbc:h2:mem:0703e0a3-b9ec-498e-902e-8abbb19c1115 user=SA`
  - In between the URL link, there is something **"mem"** ; it means it'll store the entries in the memory till the application is running.
- **spring.jpa.hibernate.ddl-auto** tells Hibernate how to handle your database tables when the application starts.

Service

- It will be having access to both **DTO** and **Entity**.
- As it'll fetch the things from **Repository**, so it'll be in the form of **Entity**. But whatever it'll return, that will be going to **Controller** layer, so that return type should be of type **DTO** (not Entity).
  - To convert this, we can get the values of Entity one by one, then create one DTO type of object and return.
  - Or we can take help of the dependency: "**modelmapper**"
- As we need to convert **Entity** to **DTO** or vice-versa, so **ModelMapper** object will be used multiple times.
  - Its better to create one **Bean** of that and use that everywhere.
  - So, create one class **MapperConfig** and write the **@Configuration** and **@Bean** there.

```
@Configuration  4 usages
public class MapperConfig {

    @Bean  no usages
    public ModelMapper getModelMapper() {
        return new ModelMapper();
    }
}
```

  - Now make sure, **Controller** part will be using **DTO**; **Repository** part will be using **Entity**; and **@Service** will be used to map those 2 properly depending upon the usage.

```
@Service  3 usages
public class EmployeeService {
    private final EmployeeRepository employeeRepository;  4 usages
    public final ModelMapper mapper;  5 usages

    public EmployeeService(EmployeeRepository employeeRepository, ModelMapper mapper) {  no us
        this.employeeRepository = employeeRepository;
        this.mapper = mapper;
    }

    public EmployeeDTO getEmployeeById(Long id) {  1 usage
        EmployeeEntity employeeEntity = employeeRepository.findById(id).orElse( other: null);
        return mapper.map(employeeEntity, EmployeeDTO.class);
```

➢ Understand the flow

- We have 3 things: **Controller, Service, Repository**
- Usage
  - *Controller* uses *DTO*
  - *Service* uses both *DTO* and *Entity*
  - *Repository* uses *Entity*
- So, the requests (Get, Post, Put, Patch, Delete) comes to *controller*. It'll get the data as *DTO* object (if payload is there) using **@RequestBody** annotation.
  - Then it'll pass that *DTO* object (if present) to *Service*.
  - If DTO is not there, it'll simple call the *Service method*.
  - So, Controller will be having objects of **DTO** and **Service**.
- Now It'll come to *Service* layer
  - *Service* layer has access to *Repository*.
  - Here it'll be having some business logic and it'll then send the request to *Repository*.
  - So, **Service** will be having objects of **DTO** (to convert Entity to DTO), **Entity** (return type of Repository methods), **Repository**
- Now it'll come to *Repository*
  - We **don't have to write** any methods inside the Repository (till not its an interface only) as everything is already there.
  - So, there is no need of doing anything in the Repository layer.

- **@PutMapping**
  - Controller
    -
```
@PutMapping(path = "/{employeeId}")  no usages
public EmployeeDTO updateEmployeeByID(@PathVariable(name = "employeeId") Long id,
                                      @RequestBody EmployeeDTO employeeDTO) {
    return employeeService.updateEmployeeById(id, employeeDTO);
}
```
  - Service
    -
```
public EmployeeDTO updateEmployeeById(Long id, EmployeeDTO employeeDTO) {  1 usage
    // converted the DTO to Entity
    EmployeeEntity employeeEntity = mapper.map(employeeDTO, EmployeeEntity.class);

    // set the id (if employee is there then id will be same only;
    // else new employee will be having this id)
    employeeEntity.setId(id);

    // save the employee
    EmployeeEntity savedEmployee = employeeRepository.save(employeeEntity);

    return mapper.map(savedEmployee, EmployeeDTO.class);

}
```
  - In **PUT** the whole object body should be updated; not partial
  - Simple concept is:
    - Getting the data payload (i.e. **DTO**)
    - Setting the id of this
      - In this case, if object is already there in the database then it'll update the id of that entry
      - If the object is not there, it'll create one new entry in the database
  - Now just save the new updated employee.
  - In this case, if you give partial values then other values in the database will become NULL.

Reflection in Java

- Lets say we have created one object of a class. When we call the method **getClass()** it'll return the **class object** of the class out of which the object is created.
  - For example: lets say the class name is **Car** and we created one object **carObj**.
  - **carObj.getClass()** will give the **object of Car class** which is **Car.class**
- **The class object is used for reflection.**
- Reflection means **inspecting/modifying** classes at **runtime**.
  - Reading fields
  - Modifying fields
  - Invoking methods
  - Listing constructors
  - Creating objects dynamically
  - Reading annotations
- Sometimes you'll get some object at runtime, but you don't know which class does that belong to.
  - In that case you can use **?** type generic
  - ```
    Class<?> clazz = obj.getClass();
    ```
  - ```
    SomeClass obj = new SomeClass();

    Class<?> clazz = obj.getClass();
    Class<?> clazz2 = SomeClass.class;

    System.out.println(clazz == clazz2);  // true
    ```
- There are 2 fields: **getField()** and **getDeclaredField()**
  - **getField()**
    - It can only access **public fields**.
    - It *searches for the field in super classes* as well.
  - **getDeclaredField()**
    - It can access **all type of fields**.
    - It **doesn't** search in super classes.
  - ```
    Field nameField = clazz.getField(name: "name");
    Field nameField2 = clazz.getDeclaredField(name: "name");
    ```
  - In the class, **name** is a private field of type *String*.
  - **getField()** will not be able to get this. But **getDeclaredField()** can access this.
- ```
  Field nameField = clazz.getDeclaredField(name: "name");
  nameField.setAccessible(flag: true);
  ```

- Now, I extracted the field **name** from the class using **getDeclaredField()**
- Then I set it accessible using **setAccessible(**true**)** method.
- **setAccessible(true) doesn't make the field public. It just allows you to read/write after getting it.**

```
SomeClass obj = new SomeClass();
Class<?> clazz = obj.getClass();

Field nameField = clazz.getDeclaredField(name: "name");
System.out.println(nameField.get(obj));
```

  - This will give error because the **name** field is private.

```
Field nameField = clazz.getDeclaredField(name: "name");
nameField.setAccessible(flag: true);
System.out.println(nameField.get(obj));
```

  - Now it can be accessible because we made it accessible using the method **setAccessible**.

➢ I have written 2 private methods in the class named as **display** and **randomMethod**

```
private void display() {
    System.out.println(x: "hello");
}

private String randomMethod(String name, int age) {
    return name + " " + age;
}
```

  - Both are private

```
SomeClass obj = new SomeClass();
Class<?> clazz = obj.getClass();

Method displayMethod = clazz.getDeclaredMethod(name: "display");
displayMethod.setAccessible(flag: true);
displayMethod.invoke(obj);
```

  - Here I am getting the **display** method using the function **getDeclaredMethod()** and making it accessible
  - Then **invoke** means it'll run the function. You need to pass the **instance of the class** if it's a non static method.

```
Method otherMethod = clazz.getDeclaredMethod(name: "randomMethod",
                                             ...parameterTypes: String.class, int.class);

otherMethod.setAccessible(flag: true);

Object res = otherMethod.invoke(obj, ...args: "Alok", 34);
System.out.println(res);
```

- As the other method is returning something and also it has some arguments, so we need to pass the **class object** of all the *arguments*.
  - Here *String.class, int.class*
- And the **invoke** method returns a Object type value so the *res* is of type Object.
- We can also **downcast** here.
  -
```
Method otherMethod = clazz.getDeclaredMethod(name: "randomMethod",
        ...parameterTypes: String.class, int.class);

otherMethod.setAccessible(flag: true);

String res = (String) otherMethod.invoke(obj, ...args: "Alok", 34);
System.out.println(res);
```

➢ Reflection in Spring

- Spring uses reflection to inspect classes, discover annotations, create objects, inject dependencies, and call methods — all at runtime.
- Dependency Injection
  -
```
@Autowired
private EmployeeService service;
```
(we write like this)
  -
```
Field f = clazz.getDeclaredField("service");
f.setAccessible(true);
f.set(object, dependencyBean);
```
(internal of spring)
- Reading Annotations
  -
```
@Controller
@Service
@Entity
@Autowired
@RequestMapping
```
(we write these)
  -
```
clazz.getAnnotations();
field.getAnnotations();
method.getAnnotations();
```
(spring does this)
- **ReflectionUtils** is a Spring utility class that makes using Java Reflection easier, safer, and cleaner.
  - Java Reflection = raw, low-level, verbose
  - Spring ReflectionUtils = wrapper that simplifies reflection
- Comparision

- ```
  Field field = ReflectionUtils.findField(Employee.class, "name");
  ```
  - Spring ReflectionUtils
- ```
  Field field = Employee.class.getDeclaredField("name");
  ```
  - Java Reflection

➢ Some advantages of Spring **ReflectionUtils** over Java Reflections

- It handles exceptions by itself
- It marks **setAccessible(true)** by default
- Handles super class traversal as well.
- Provides iteration helpers
  - ```
    ReflectionUtils.doWithFields(clazz, field -> {
        System.out.println(field.getName());
    });
    ```

➢
    ```

- **@PatchMapping**
  - **Approach-1 (my approach)**
  - In **PATCH** some part of the object should be updated; not full

```java
@Configuration  2 usages
public class MapperConfig {

    @Bean  no usages
    public ModelMapper getModelMapper() {
        ModelMapper modelMapper = new ModelMapper();
        modelMapper.getConfiguration().setSkipNullEnabled(true);
        return modelMapper;
    }
}
```

  - I added one line here. It means, if I am transferring the new values from **new** object to **old** object, then the **null** values of **new** object **will not be updated** in the **old** one.
    - Lets say in my database, the user's entry is there.
    - But I want to update the specific values that are being sent in the **patch** request.
    - If I directly update then it'll update the null values as well.

```java
public EmployeeDTO updateEmployeeById(Long id, EmployeeDTO employeeDTO) {  1 usage
    EmployeeEntity dbEmployee = employeeRepository.findById(id).orElse( other: null);
    EmployeeEntity employeeEntity = mapper.map(employeeDTO, EmployeeEntity.class);

    EmployeeEntity toSave;
    if(dbEmployee == null) { // if no employee, then create one
        employeeEntity.setId(id);
        toSave = employeeEntity;
    } else { // if employee present, then update
        mapper.map(employeeEntity, dbEmployee); // it'll update the "dbEmployee" object
        toSave = dbEmployee;
    }
    EmployeeEntity savedEmployeeEntity = employeeRepository.save(toSave);
    return mapper.map(savedEmployeeEntity, EmployeeDTO.class);
}
```

    - This is the function to update the **employee** .
    - It is written inside the **Service**.

- **Approach-2**

```
public EmployeeDTO updatePartialEmployeeById(Long id, Map<String, Object> updates) {
    boolean exists = isExistsByEmployeeId(id);
    if(!exists) return null;
    EmployeeEntity employeeEntity = employeeRepository.findById(id).get();

    updates.forEach(( String field,  Object value) -> {
        Field fieldToBeUpdated = ReflectionUtils.findField(EmployeeEntity.class, field);
        fieldToBeUpdated.setAccessible(true);
        ReflectionUtils.setField(fieldToBeUpdated, employeeEntity, value);
    });

    return mapper.map(employeeRepository.save(employeeEntity), EmployeeDTO.class);
}
```

  - **Service** method

```
@DeleteMapping(path = "/{employeeId}")  no usages
public EmployeeDTO deleteEmployeeById(@PathVariable(name = "employeeId") Long id) {
    return employeeService.deleteEmployeeById(id);
}
```

  - **Controller** method

- Here I have taken **Map<String, Object>** instead of **EmployeeDTO** for the payload.

- ---------- **@JsonProperty** ----------

  - It **should be used in DTO** (should **not be used in Entity**)
  - Whatever field you pass inside **@JsonProperty**, the incoming payload (request body) should contain that field otherwise it'll not work.

```
@JsonProperty("isActiveNew")
private Boolean isActive;
```

  - Consider this case, the object name is **isActive** but the **@JsonProperty** contains **isActiveNew**.
  - In this case, in the request payload, **isActiveNew** should be passed instead of **isActive** like the below:

```
"name": "Alok",
"email": "alok@gmail.com",
"age": "23",
"dateOfJoining": "2024-01-12",
"isActiveNew": true
```

  - **Jackson** is the one who handles these things.
    - **Deserialization** : JSON to Java Object
    - **Serialization** : Java Object to JSON

- **Request body (JSON) to DTO Object (Deserialization)** and **DTO object to Response body (JSON) (Serialization)** is done by **Jackson**.

- Using **@JsonProperty** in **Entity** has **no** effect of **DB** mapping. Even if you write something in **@JsonProperty** inside **@Entity**, the DP mapping will only happen on the basis of **field name not JsonProperty field name**.
- If you want to specify some column name, then you can use **@Column** annotation in the Entity. It'll make the **DB** column with that particular name.

- To send the Status Codes, use **ResponseEntity**

```
@GetMapping("/{employeeId}")  no usages
public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "id") Long id) {
    EmployeeDTO employeeDTO = employeeService.getEmployeeById(id);
    if(employeeDTO == null) return  ResponseEntity.notFound().build();
    return ResponseEntity.ok(employeeDTO);
}
```

- This is my implementation. (Controller class)

- The below is proper implementation

```
public Optional<EmployeeDTO> getEmployeeById(Long id) {  1 usage
    return employeeRepository
            .findById(id)
            .map( EmployeeEntity employeeEntity -> mapper.map(employeeEntity, EmployeeDTO.class));
}
```

- **Service**
- **Optional<T>.map()** is just used to convert **Optional<T>** to **Optional<U>** type.
-
```
public <U> Optional<U> map( @NotNull Function<? super  @NotNull T, ? extends U> mapper) {
```

  - Here **? super T, ? extends U**
  - Here both the "**?**" are different not same.
- In our case
  - **T** : **EmployeeEntity**
  - **U** : **EmployeeDTO**
  - **? Super T** : **?** is **EmployeeEntity**
  - **? Extends U** : **?** is **EmployeeDTO**

```
@GetMapping("/{employeeId}")  no usages
public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);
    return employeeDTO
            .map( EmployeeDTO employeeDTO1 -> ResponseEntity.ok(employeeDTO1))
            .orElse(ResponseEntity.notFound().build());
```

- **Controller**
- Instead of returning **EmployeeDTO**, it should return **ResponseEntity<EmployeeDTO>** to send the status code along with response.
- As the **Service** method is returning **Optional** type, so in the **Controller** method we need to use Optional type variable to handle that.
- In that, we can also use **Constructor reference** like below

```
return employeeDTO
        .map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
```

```
@PostMapping  no usages
public ResponseEntity<EmployeeDTO> createNewEmployee(@RequestBody EmployeeDTO inputEmployee) {
    EmployeeDTO savedEmployee = employeeService.createNewEmployee(inputEmployee);
    return new ResponseEntity<>(savedEmployee, HttpStatus.CREATED);
}
```

- In this case we created the object of **ResponseEntity** and returned here.
- If we want some custom Http Status Code then we can create one **ResponseEntity** object and pass our object and the custom status code inside it.

- **DeleteMapping**

```
public EmployeeDTO deleteEmployeeById(Long id) {  1 usage  1 related problem
    EmployeeEntity deletedEmployee = employeeRepository.findById(id).orElse( other: null);
    employeeRepository.deleteById(id);
    return mapper.map(deletedEmployee, EmployeeDTO.class);
}
```

- Service method to delete the employee.

Input Validation

- spring-boot-starter-data-jpa it is the spring helper package for validating fields.
- Some Annotations for validations are listed below
  - **@NotNull**
  - **@NotEmpty**
    - not null and size should be greater than 0;
    - Used in Collections, arrays, string
  - **@NotBlank**
    - string is not null and ***trimmed*** length is greater than 0)
  - **@Size**
    - element's size should fall within specified range)
  - **@Max**
    - annotated element should be a number & less than or equal to provided value
  - **@Min**
  - **@Email**
    - Annotated string should be a valid email
  - **@Pattern**
    - Annotated string should match the specified *regular expression*.
  - **@Positive**
    - Annotated element should be a *positive number*.
  - **@PositiveOrZero**
  - **@Negative**
  - **@NegativeOrZero**
  - **@Past**
    - Annotated date or calendar value should be in the past
  - **@PastOrPresent**
  - **@Future**
  - **@FutureOrPresent**
  - **@Digits**
    - Annotated number has up to specified number of integer and fraction digits
    -
```
@Digits(integer = 6, fraction = 2, message = "Salary can be in the form: XXXXXX.YY")
private Double salary;
```
  - **@DecimalMin**

- Annotated element should **>=** specified minimum, allowing decimal points

```
@Digits(integer = 6, fraction = 2, message = "Salary can be in the form: XXXXXX.YY")
@DecimalMin(value = "100000.99")
private Double salary;
```

- **@DecimalMax**
- **@AssertTrue**
  - Annotated field should be *boolean* and should be *true*
- **@AssertFalce**
- **@Valid**
  - Validates the associated object recursively
  - Applies bean validation to nested objects.

➢ As the first point is **DTO**, we need to specify the validation at the DTO level

```
public class EmployeeDTO {
    private Long id;

    @NotBlank(message = "Name of the employee cannot be blank")
    @Size(min = 3, max = 10, message = "characters should be in range 3 to 10")
    private String name;
```

  - Here I gave **@NotBlank** and **@Size** for the field **name**.
- But giving validation alone in DTO won't work; you need to mention in the **Controller** as well.

```
@PostMapping  no usages
public ResponseEntity<EmployeeDTO> createNewEmployee(@RequestBody @Valid EmployeeDTO inputEmployee)
```

  - When you write **@Valid** here then it'll go and check the *validations* of DTO.

Annotation

- An annotation is metadata added to Java code to give information to the compiler, tools, or runtime. It does not affect program logic directly but describes how the code should be treated.
- Types:
  - SOURCE
    - Available only in .java file
    - Removed during compilation
    - Used by compiler and IDE
    - e.g., **@Override, @SuppressWarnings**
  - CLASS
    - Stored in .class file
    - Not available at runtime
    - Default retention
  - RUNTIME
    - Stored in .class file
    - Available at runtime via reflection
    - e.g., custom annotations, **@Deprecated**
  - **RetentionPolicy.SOURCE**, **RetentionPolicy.CLASS**, **RetentionPolicy.RUNTIME**
- Targets (Where annotations can be applied) **ElementType**
  - TYPE → class, interface
  - METHOD → method
  - FIELD → variable
  - PARAMETER → method parameter
  - CONSTRUCTOR → constructor
  - LOCAL_VARIABLE → inside methods
  - ANNOTATION_TYPE → another annotation
  - PACKAGE → package-info.java
  - Example: **ElementType.METHOD**
-
```
@Target({ElementType.ANNOTATION_TYPE, ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
```
  - You can give **multiple targets** as well.
- Spring uses annotation to do everything
  - It scans the annotations using **reflection**.

- Create objects and injects according to those
- For example: **clazz.isAnnotationPresent(Component.class)** then it'll create one Bean.
- It helps it maintain the code clean.

Custom Annotation

- Created one custom annotation to validate the Role of the employee.

```java
@Retention(RetentionPolicy.RUNTIME)  1 usage
@Target({ElementType.FIELD, ElementType.PARAMETER})
public @interface EmployeeRoleValidation {
    String message() default "Role of the employee cannot be blank";

    Class<?>[] groups() default {};  no usages

    Class<? extends Payload>[] payload() default {};  no usages
}
```

  - Here **message** means the error message when the validation fails.
  - Rest methods like **groups** and **payload**. It'll be updated later. **@TODO**
  - So, we created the interface now. But we have to implement the logic to make sure when this validation will be valid and when it'll be invalid.
- We have to create one class inheriting the interface **ConstraintValidator** passing the *annotation interface* and *data type that validator will check* (in our case String).

```java
public class EmployeeRoleValidator implements ConstraintValidator<EmployeeRoleValidation, String> {

    @Override
    public boolean isValid(String inputRole, ConstraintValidatorContext constraintValidatorContext) {
        List<String> roles = List.of("USER", "ADMIN");
        return roles.contains(inputRole);
    }
}
```

  - And here write the logic of **isValid** method to validate the constraint.
- Now, we implemented the **annotation** to validate, and created one **validator** class to write the validate logic.
  - But, the **annotation** and **validator class** are not connected till now.
  - The connection will be build using **@Constraint** annotation.

```java
@Retention(RetentionPolicy.RUNTIME)  1 usage
@Target({ElementType.FIELD, ElementType.PARAMETER})

@Constraint(validatedBy = {EmployeeRoleValidator.class})
public @interface EmployeeRoleValidation {
    String message() default "Role of the employee cannot be blank";
```

  - Now whenever the *custom annotation* **@EmployeeRoleValidation** will be triggered, it'll go to **EmployeeValidator** class and run the validation method which is **isValid**.
  - **@Constraint** is what tells the Bean Validation (**Hibernate Validator**) system that your **annotation is a VALIDATION ANNOTATION**.

```
@NotBlank(message = "Role of the employee cannot be blank")
  @Pattern(regexp = "^(ADMIN|USER)$")
@EmployeeRoleValidation
private String role; // ADMIN, USER
```

- Now used the custom **@EmployeeRoleValidation** annotation in **EmployeeDTO** to validate the input.

- There are multiple use-cases of **annotations**, in our case the use case was to **validate something**.
  - Annotations can be used for Annotations can be used for validation (your case), dependency injection, REST mapping, JSON mapping, database mapping, configuration, logging, security, AOP ..etc
  - 3 types of annotations:
    - **Marker Annotations**
      - No methods; only used to mark something
      - Ex: @Deprecated, @Override ..etc
    - **Single-Value Annotations**
      - Have exactly one method (usually named as **value**)
      -
```
public @interface Check {
    String value();
}
```
    - **Multi-Value Annotations**
      - Have multiple methods; most commonly used.
      -
```
public @interface MyAnnotation {
    String message();
    int length();
    boolean enabled() default true;
}
```
      - **@EmployeeRoleValidation** was a *Multi-Value Annotation*.

- **Validation Annotation** will be **multi valued** only.
  - You need to give **message**, **groups**, **payloads** though you are not using those.
  - Its related to **Java Bean (not Spring Bean)**
    - **Java Bean** means one **simple class containing private fields and getters, setters.**
    - **Spring Bean** means **Spring Managed Objects**.

Custom Exception Handling

- **@ExceptionHandler** annotation is used to define custom exception handler.

```
@ExceptionHandler(value = NoSuchElementException.class)  no usages  new *
public String handleEmployeeNotFound(NoSuchElementException exception) {
    return "Employee was not found";
}
```

  - Here (in **EmployeeController**) I created one **method** using the annotation **@ExceptionHandler**.
  - This annotation takes the **class object** of the particular exception that it needs to catch.
  - So now, whenever any **NoSuchElementException** is thrown from *EmployeeController*, this method will catch that and handle that.

- Now, we need to throw the proper **exception** to make the method (Exception handler method) work.

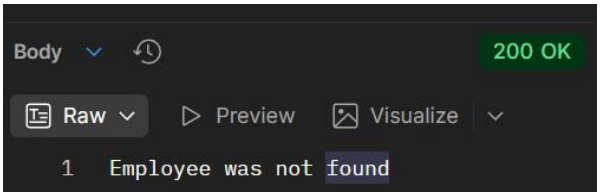
```
@GetMapping("/{employeeId}")  no usages  Alok905
public ResponseEntity<EmployeeDTO> getEmployeeById(@PathVariable(name = "employeeId") Long id) {
    Optional<EmployeeDTO> employeeDTO = employeeService.getEmployeeById(id);

    return employeeDTO
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
}
```

  - This is the *previous implementation* of our **getEmployeeById** method.
  - Here we need to throw that exception i.e. **NoSuchElementException** so that our **ExceptionHandler** method can handle the particular exception.

```
return employeeDTO
        .map(ResponseEntity::ok)
        .orElseThrow(() -> new NoSuchElementException("Employee Not Found"));
```

  - It is an optional class's method and it expects to return one **Exception object**.
  - Don't write **throw new NoSuchElementException("…bla bla…")** here.

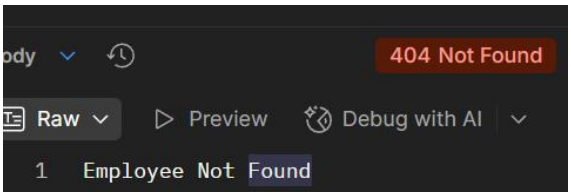
- In the previous implementation of **exception handler method**, we are not returning the **ResponseEntity** so that even if the request fails, it'll give status code **200**.



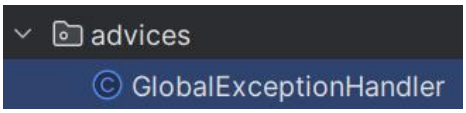
  - So we need to return **ResponseEntity** instead of **String**.

```
@ExceptionHandler(value = NoSuchElementException.class)  no usages  new *
public ResponseEntity<String> handleEmployeeNotFound(NoSuchElementException exception) {
    return new ResponseEntity<>( body: "Employee Not Found", HttpStatus.NOT_FOUND);
}
```

- Now the **status code** will be sent properly.



- We have one **catch** here
  - This exception handler is only handling the exception of **EmployeeController**; but we want one method that will handle exceptions globally.
  - **@RestControllerAdvice** is used to write logic that will apply globally for all **@RestController** classes. (not just *Exception Handling*)
    - It is basically **@RestController + @ControllerAdvice**



    - Here I created one class *GlobalExceptionHandler*

```
@RestControllerAdvice  no usages
public class GlobalExceptionHandler {

    @ExceptionHandler(value = NoSuchElementException.class)  no usages
    public ResponseEntity<String> handleResourceNotFound(NoSuchElementException exception) {
        return new ResponseEntity<>( body: "Resource Not Found", HttpStatus.NOT_FOUND);
    }
}
```

  - Created one **ApiError** class inside that **Service** directory

```
@Data  1 usage
@Builder
public class ApiError {

    private HttpStatus httpStatus;
    private String message;
}
```

    - **@Data** is lombok annotation that is combination of **@Getter @Setter** and so many.
    - **@Builder** is also lombok annotation that allows you to create an object **without using constructor**. And provide chaining like the below

```
Employee emp = Employee.builder()
        .name("Alok")
        .age(25)
        .role("ADMIN")
        .build();
```
(just example)

```
public class Employee {
    private String name;
    private int age;
    private String role;

    public static EmployeeBuilder builder() { return new EmployeeBuilder(); }

    public static class EmployeeBuilder {
        private String name;
        private int age;
        private String role;

        public EmployeeBuilder name(String name) {
            this.name = name;
            return this;
        }

        public EmployeeBuilder age(int age) {
            this.age = age;
            return this;
        }

        public EmployeeBuilder role(String role) {
            this.role = role;
            return this;
        }

        public Employee build() {
            return new Employee(name, age, role);
        }
    }
}
```

- It'll be like this under the hood.
- You need to call **builder()** in the *beginning*, **build()** at the *end* to create the object.

```java
@RestControllerAdvice  no usages
public class GlobalExceptionHandler {

    @ExceptionHandler(value = NoSuchElementException.class)  no usages
    public ResponseEntity<ApiError> handleResourceNotFound(NoSuchElementException exception) {
        ApiError apiError = ApiError.builder()
                .httpStatus(HttpStatus.NOT_FOUND)
                .message("Resource not found")
                .build();

        return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
    }
}
```

- Now the error response will be well formatted.

- Now you'll get a proper formatted response.

➢ Refactoring of the codes:

- Instead of using **NoSuchElementException**, we should create our own **Exception**.
  - (create one package **exceptions** and write your own exceptions inside that)

```java
public class ResourceNotFoundException extends RuntimeException{

    public ResourceNotFoundException(String message) {  1 usage
        super(message);
    }
}
```

- Instead of writing some fixed error message, we should use **exception.getMessage()**

```java
@RestControllerAdvice  no usages
public class GlobalExceptionHandler {

    @ExceptionHandler(value = ResourceNotFoundException.class)  no usages
    public ResponseEntity<ApiError> handleResourceNotFound(ResourceNotFoundException exception) {
        ApiError apiError = ApiError.builder()
                .httpStatus(HttpStatus.NOT_FOUND)
                .message(exception.getMessage())
                .build();

        return new ResponseEntity<>(apiError, HttpStatus.NOT_FOUND);
    }
}
```

- Created one exception handler for all those exceptions which are not being handled yet by giving the parent **Exception** class's object.

```
@ExceptionHandler(value = Exception.class)  no usages
public ResponseEntity<ApiError> handleInternalServerError(Exception exception) {
    ApiError apiError = ApiError.builder()
            .httpStatus(HttpStatus.INTERNAL_SERVER_ERROR)
            .message(exception.getMessage())
            .build();
    return new ResponseEntity<>(apiError, HttpStatus.INTERNAL_SERVER_ERROR);
}
```

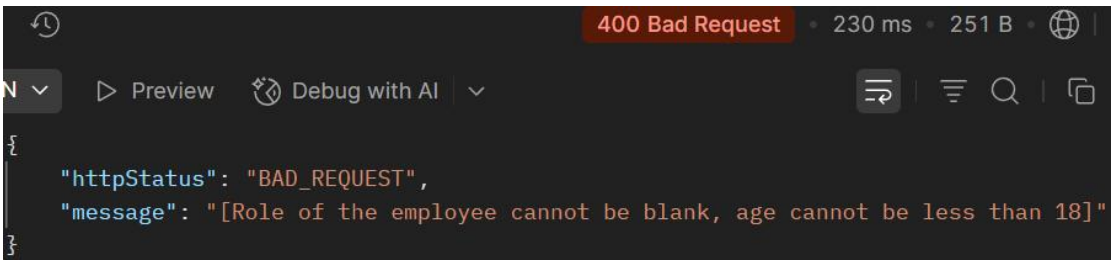
  - But it'll give the response with some messy messages where the input validation failed. (with the validation annotations)

```
"httpStatus": "INTERNAL_SERVER_ERROR",
"message": "Validation failed for argument [0] in public org.springframework.http.
    ResponseEntity<com.codingshuttle.springbootwebtutorial.springbootwebtutorial.dto.
    EmployeeDTO> com.codingshuttle.springbootwebtutorial.springbootwebtutorial.controllers.
    EmployeeController.createNewEmployee(com.codingshuttle.springbootwebtutorial.
    springbootwebtutorial.dto.EmployeeDTO) with 2 errors: [Field error in object 'employeeDTO'
    on field 'role': rejected value [ADMINx]; codes [EmployeeRoleValidation.employeeDTO.role,
    EmployeeRoleValidation.role,EmployeeRoleValidation.java.lang.String,EmployeeRoleValidation];
    arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes
    [employeeDTO.role,role]; arguments []; default message [role]]; default message [Role of the
    employee cannot be blank]] [Field error in object 'employeeDTO' on field 'age': rejected
    value [3]; codes [Min.employeeDTO.age,Min.age,Min.java.lang.Integer,Min]; arguments [org.
    springframework.context.support.DefaultMessageSourceResolvable: codes [employeeDTO.age,age];
    arguments []; default message [age],18]; default message [age cannot be less than 18]] "
```

- Created one more **exception handler** for the **MethodArgumentNotValidException**
  - This exception is thrown for **failure of validation annotations.**

```
@ExceptionHandler(MethodArgumentNotValidException.class)  no usages
public ResponseEntity<ApiError> handleInputValidationErrors(MethodArgumentNotValidException exception) {
    List<String> errors = exception
            .getBindingResult()  BindingResult
            .getAllErrors()  List<ObjectError>
            .stream()  Stream<ObjectError>
            .map(DefaultMessageSourceResolvable::getDefaultMessage)  Stream<String>
            .toList();
    ApiError apiError = ApiError.builder()
            .httpStatus(HttpStatus.BAD_REQUEST)
            .message(errors.toString())
            .build();
    return new ResponseEntity<>(apiError, HttpStatus.BAD_REQUEST);
    }
```

  - Now it'll come proper



Transforming API responses Globally

- Before this lets learn about **@RestControllerAdvice**
  - It is used to apply GLOBAL logic to ALL @RestController classes.
  - The use-cases are given below:
    - Exception Handling (most common use-case)
      - In this case, no need to inherit any class/interface.
    - Global Response Body Modification
      - In this case, implement **ResponseBodyAdvice** interface.
    - Global Request/Response Binding Configuration
- How?
  - Extend your class with **@ResponseBodyAdvice<Object>** to define the custom return type.
  - Use **@RestControllerAdvice** for global API response transformation
  - Return appropriate HTTP status codes and error messages.
  - You can also return the timestamp of the API response.
-

```
@RestControllerAdvice  no usages
public class GlobalResponseHandler implements ResponseBodyAdvice<Object> {
```

  - Here you need to implement **ResponseBodyAdvice** interface passing **Object** as the type.
- Inside **ResponseBodyAdvice**, 2 abstract methods are there
  - **supports()** and **beforeBodyWrite()**
  - **boolean supports(returnType, converterType)**
    - Depending upon the returnType and converterType (html, json, string etc) you can choose to use the method *beforeBodyWrite()*
    - If you return **false** from **support()** method, then **beforeBodyWrite()** method will not be used.
    - If you return **true** from **support()** method, then beforeBodyWrite() method will be used.

- Create one **ApiResponse** class inside **advices** directory which will be the global API response format

```
@Data  2 usages
public class ApiResponse<T>{

    private LocalDateTime timeStamp;
    // when data is there, error will be null
    // when error is there, data will be null
    private T data;
    private ApiError error;

    public ApiResponse() {  2 usages
        this.timeStamp = LocalDateTime.now();
    }

    public ApiResponse(T data) {  1 usage
        this(); // this is the reason of the default constructor being called
                // everytime to initialize the timestamp
        this.data = data;
    }

    public ApiResponse(ApiError error) {  no usages
        this(); // this is the reason of the default constructor being called
                // everytime to initialize the timestamp
        this.error = error;
    }
}
```

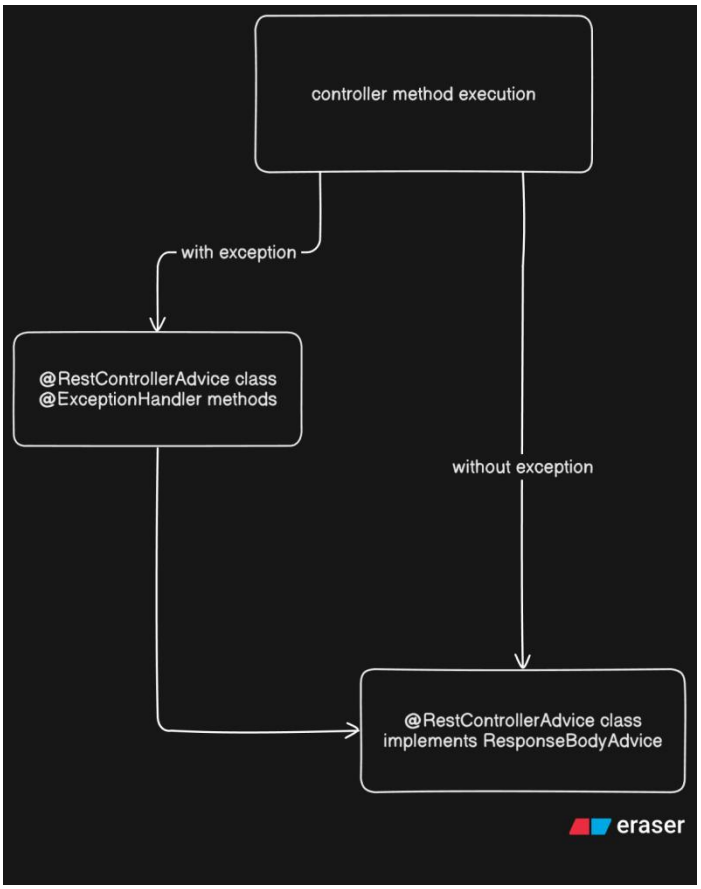
- And update the **beforeBodyWrite** method

```
@Override  no usages
public Object beforeBodyWrite(Object body, MethodPara
    if(body instanceof ApiResponse<?>)  return body;

    return new ApiResponse<>(body);
}
```

- **NOTE**
  - The response from Exception handler and Rest controller methods contains header i.e. **ResponseEntity** but when it reaches to the **response handler**'s **beforeBodyWrite** method, only the body of the response is being passed here. Not with the ResponseEntity.

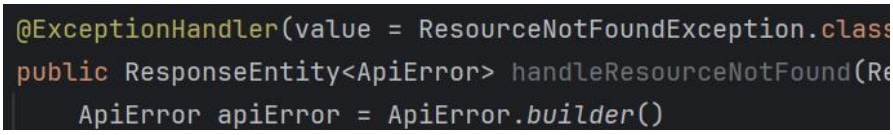
Flow of Exceptions and Response

- Whenever any error is thrown from any controller method, it'll find any class having **@RestControllerAdvice** which is having **@ExceptionHandler** methods, then it'll trigger that method.
  - Now you can say, if **@ExceptionHandler** method is only needed, then to create the bean of the class, we can simply use **@Component** instead of **@RestControllerAdvice**.
  - But it'll not work because Spring only check classes annotated with **@RestControllerAdvice** or **@ControllerAdvice**.
- After the **exception handler method's** work is finished, it's returned object will be sent to **@RestControllerAdvice** class which is implementing **ResponseBodyAdvice** interface.
  - If there was no **exception**, then it'll skip the **exception handler class**, and directly comes to **Response handling** class.
- Now, whatever you want you can do in this class and return the response. That'll go to the user as response.

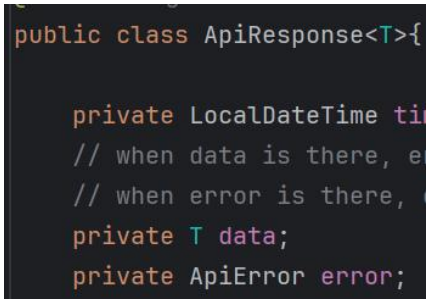


- So, we were returning **ApiError** type of object from **exception handler** and **ApiResponse** type of object from **response handler**.
  - So, in case of exception, it the ApiError object was being passed to **response handler** as the data.
  - So, the output was not proper

    - **error** should contain those error data. And **data** should be **null**.



    - Instead of sending **ApiError** object, we can send **ApiResponse** object as ApiResponse has one property already there of type ApiError.



  - And also, in **response handler**, we have written the condition that if the object is of type **ApiResponse** then return the object directly.
    - So if we return **ApiResponse** type of object having **data=null** and **error** will be **error data**, then it'll send the proper response to the user.

    - Now its coming properly

Annotations that were being used in this module

- **Controller Layer**
  - **@RestController** : make the class **Rest** (**@Controller + @ResponseBody**)
  - **@RequestMapping** : define common end point
  - **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping**
  - **@PathVariable** : get the URL param (you can change the name as well)
  - **@RequestParam** : get the URL query values
  - **@RequestBody** : Request payload will be get (type will be **DTO**)
  - **@RestControllerAdvice** : creating class for handling global exceptions.
  - **@ExceptionHandler** : used to create exception handler **methods**
- **Service Layer**
  - **@Service** : to create the service
- **Repository Layer**
  - **@Repository** : to create the repository (it should also inherit **JpaRepository**)
- **Entity / JPA**
  - **@Entity** : to create Entity (entity is nothing but the schema of SQL table)
  - **@Table** : to give *custom name* to Database **table**
  - **@Id** : high-level annotation used inside Entity to make the field Primary-Key
  - **@Column** : to give *custom name* to the column.
  - **@GeneratedValue** : used in case of **id**, to specify type of increment of ids.
- **Jackson**
  - **@JsonProperty** : used for Serialization (obj to json) and Deserialization (json to obj)
- **Lombok**
  - **@Data** : combination of **@Getter, @Setter** and so many annotations.
  - **@Builder** : used to create object of class without using constructor.
- **Custom Annotation**
  - **@Target** : specify the target**(s)** like class, methods, fields, parameters etc.
  - **@Retention** : when to use annotation (pre-compile, runtime, post-compile)
  - **@Constraint** : used in case of **validation annotation** to map the validator class.
    - Validator class should inherit **ConstraintValidator**