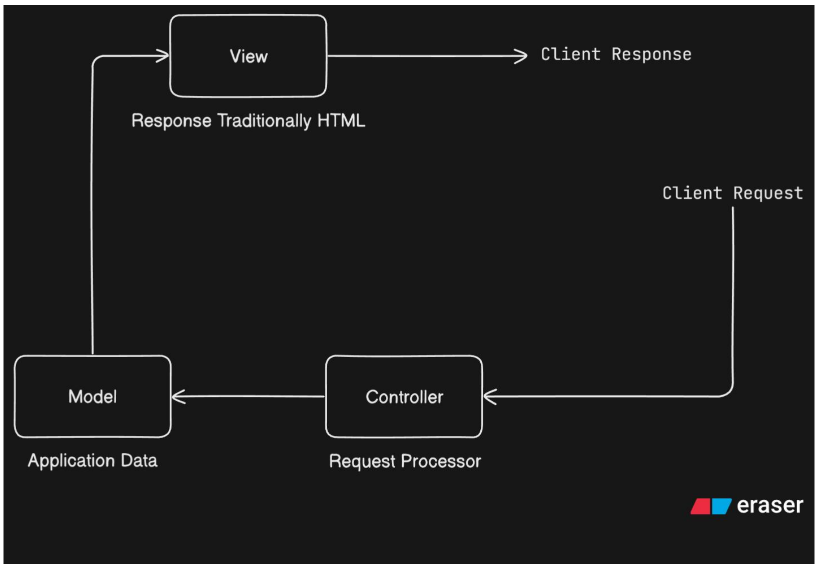
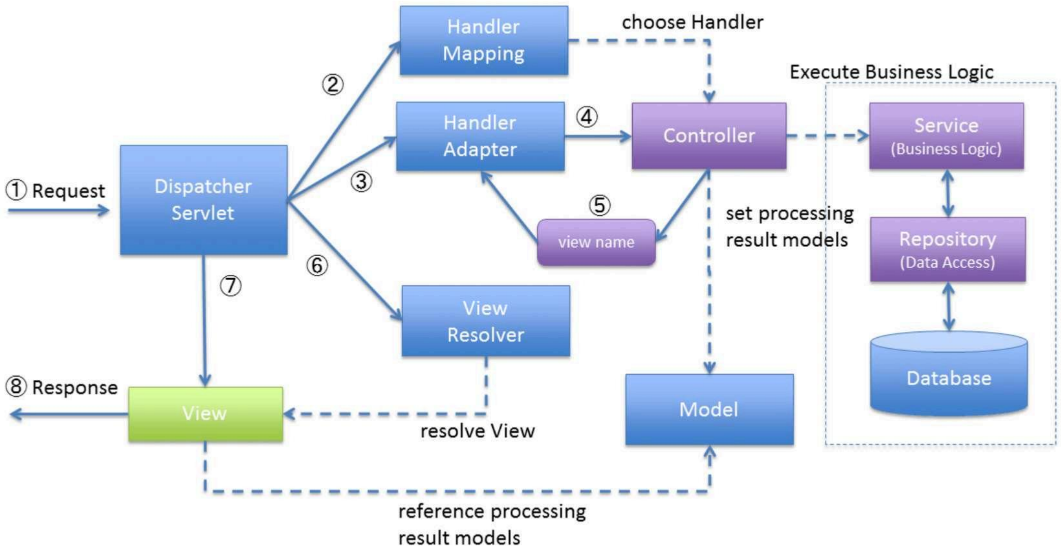
MVC Architecture

- 

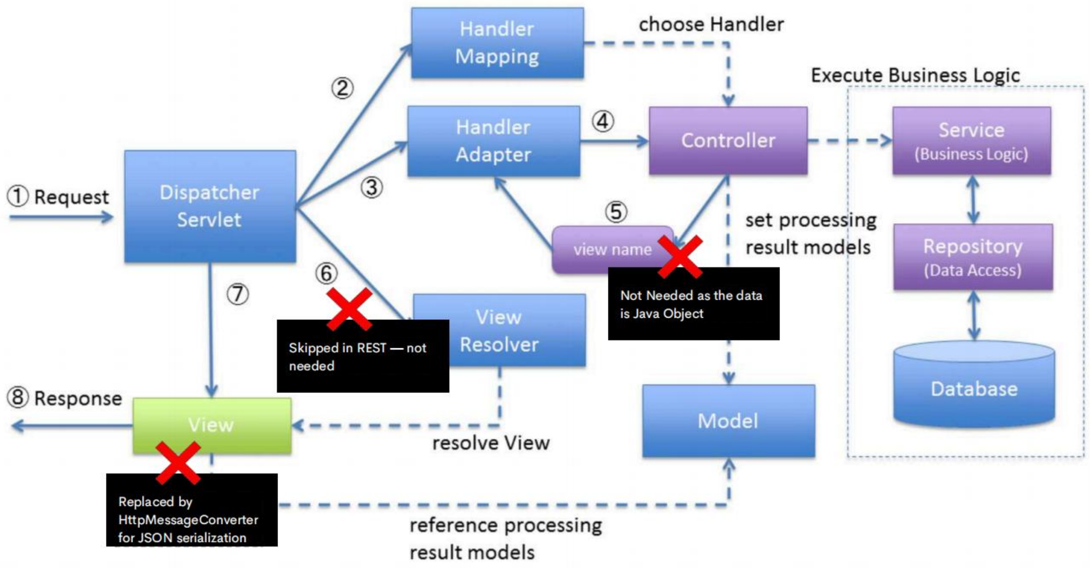
    - Controller
        - Different APIs (Get, Post etc) will be having different controller
    - Model
        - Handling the data
        - Talking to Database
        - In case of Java, Model is present in the Object
    - View
        - HTML or JSON
        - Send the data to the client

- **Spring MVC** (Legacy)



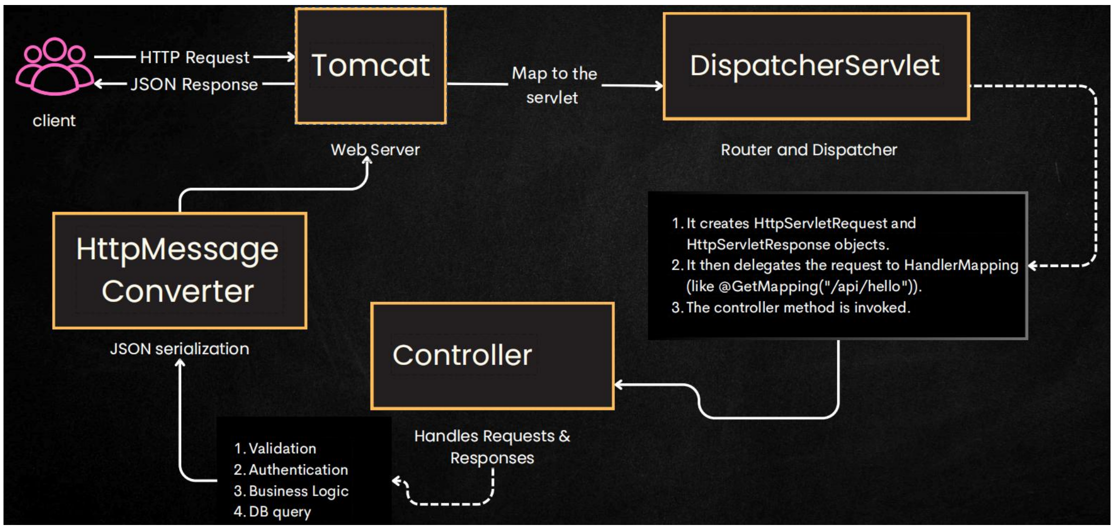
- View Resolver will take a look at the Dispatcher Servlet that which kind of user has requested and then it converts the Model to that format i.e. XML, HTML, JSON etc.

➢ **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.

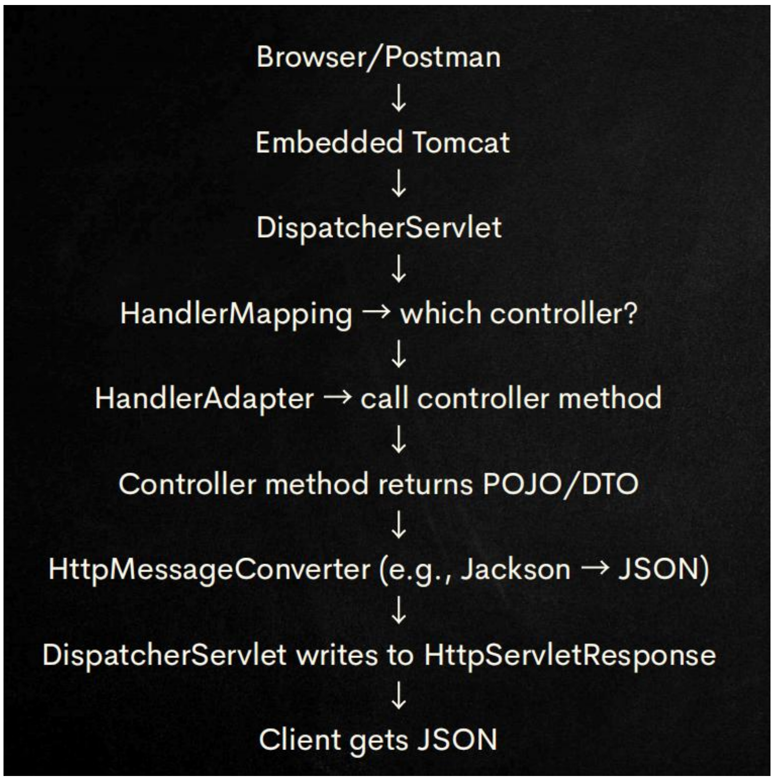
➢ **How does a Web Server works in Spring Boot?**

- 

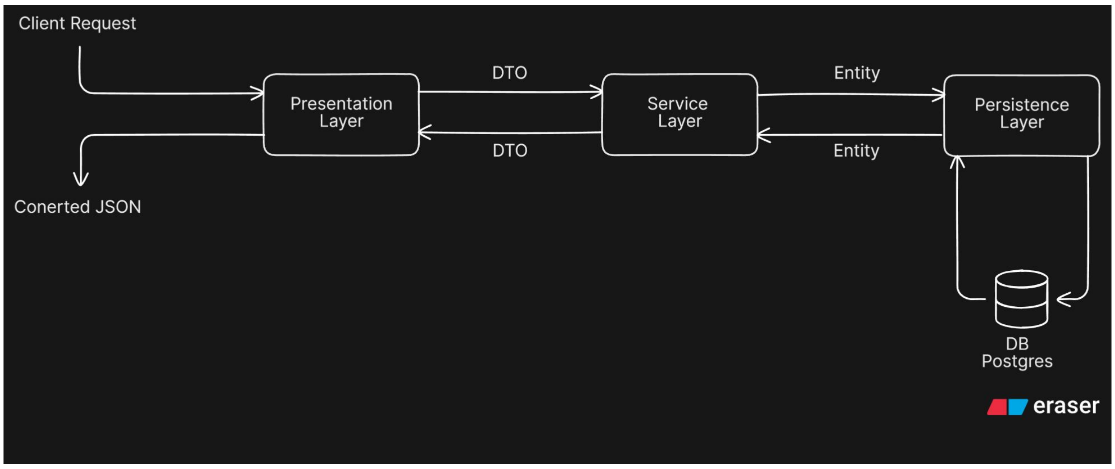
1. It creates HttpServletRequest and HttpServletResponse objects.
2. It then delegates the request to HandlerMapping (like @GetMapping("/api/hello")).
3. The controller method is invoked.

1. Validation
2. Authentication
3. Business Logic
4. DB query

- 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**.

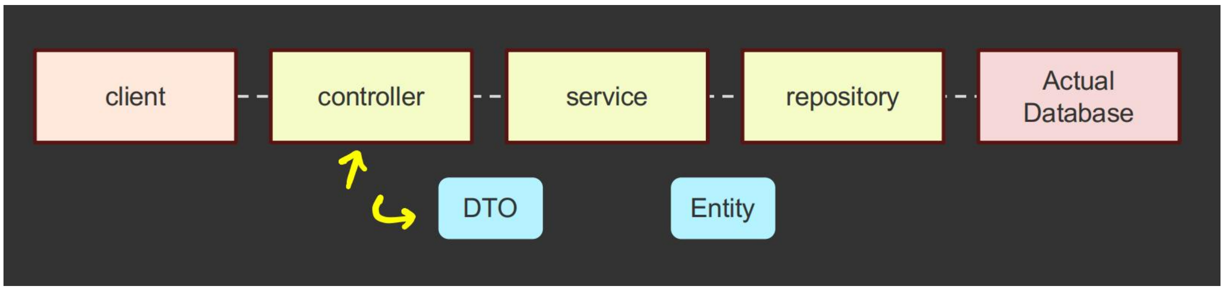
- We can have multiple requests like *profile request* or *admin request* or *reels request* etc etc. So these routing part is done in here i.e. **Dispatcher Servlet**.
  - 2 Objects i.e. **HttpServletRequest** and **HttpServletResponse** are created in this part only.
  - In that **HttpServletRequest** contains everything like headers, query, ids p
  - arams.
  - After doing the things, we can update the **HttpServletResponse** and return it as the response.
- 
Browser/Postman
↓
Embedded Tomcat
↓
DispatcherServlet
↓
HandlerMapping → which controller?
↓
HandlerAdapter → call controller method
↓
Controller method returns POJO/DTO
↓
HttpMessageConverter (e.g., Jackson → JSON)
↓
DispatcherServlet writes to HttpServletResponse
↓
Client gets JSON

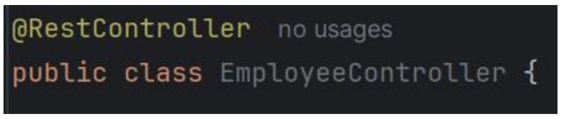
- 3 layered architecture



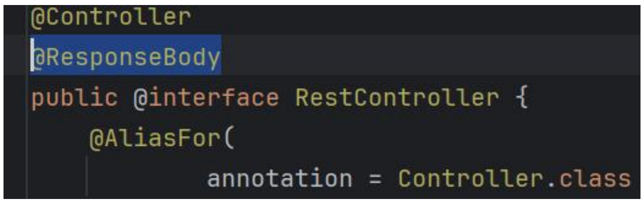
- <u>**Layers**</u>
  - **Presentation Layer** *(Handle Request and Response)*
    - Request from client will be handled by this
    - Communication between *Presentation* and *Service* layer happens with transfer of **DTO** *(Data Transfer Object)*
    - Lets say user is trying to login, so payload 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.
  - 

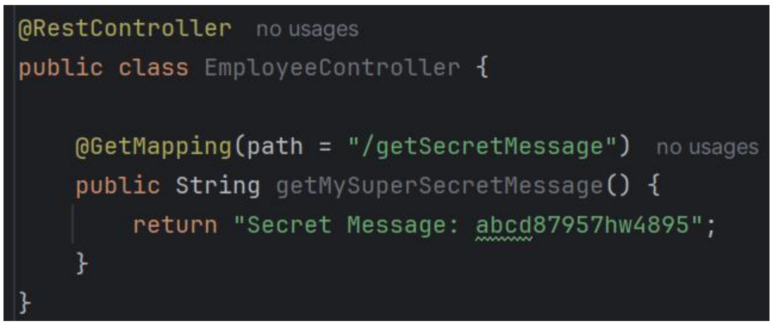
```
@RestController  no usages
public class EmployeeController {
```

  - You can see in the **@RestController**, it has one annotation called **@ResponseBody**
    - 

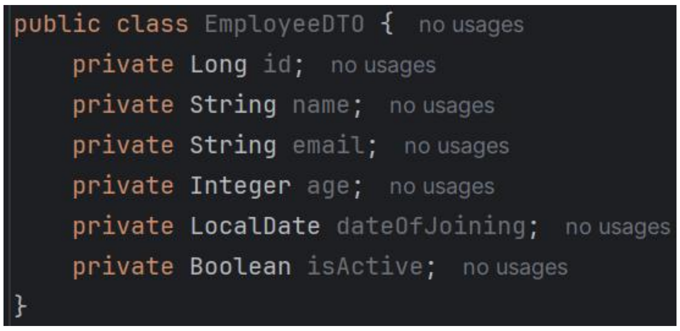
```
@Controller
@ResponseBody
public @interface RestController {
    @AliasFor(
            annotation = Controller.class
```

    - 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.

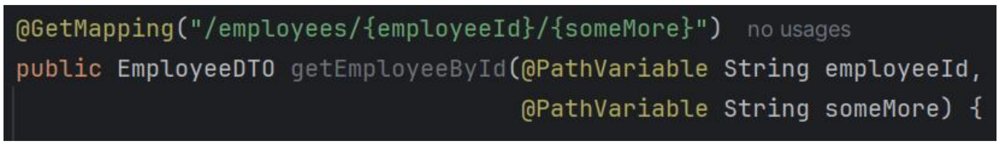
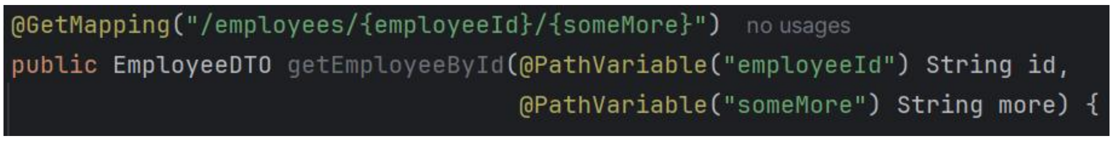
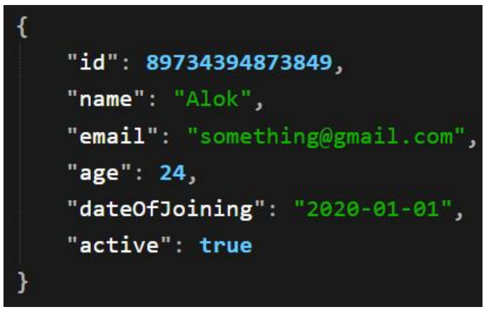
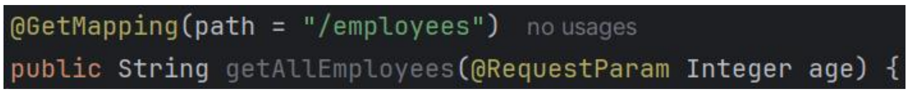
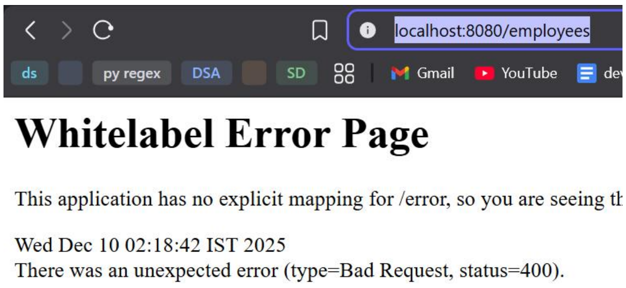
- With **@RestController**, we don't need to explicitly write **@ResponseBody** for each methods to make it return *JSON*. (But with **@Controller** we need to write explicitly).
-

- **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.
    - 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.
  - 

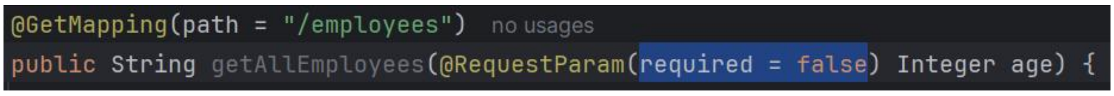
```
public class EmployeeDTO {  no usages
    private Long id;  no usages
    private String name;  no usages
    private String email;  no usages
    private Integer age;  no usages
    private LocalDate dateOfJoining;  no usages
    private Boolean isActive;  no usages
}
```

- 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

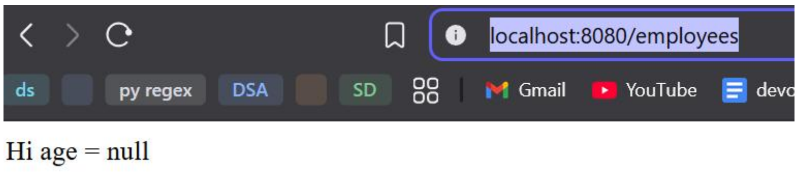
    This application has no explicit mapping for /error, so you are seeing tl

    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) {
```

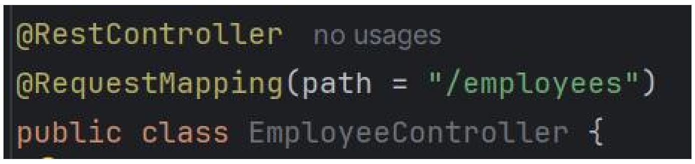
Now you can see the page is loading



```
localhost:8080/employees
```

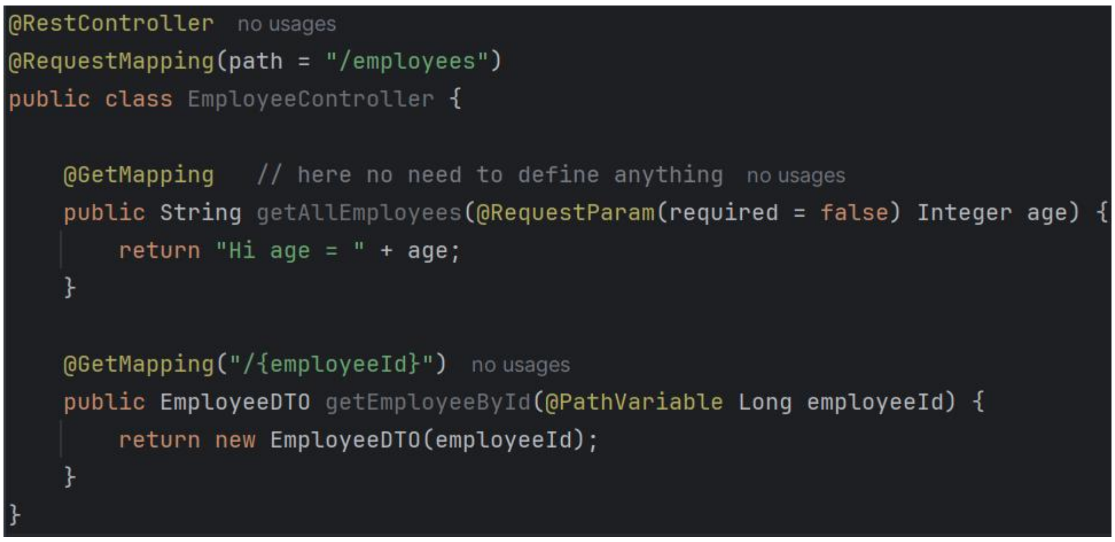
```
Hi age = null
```

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



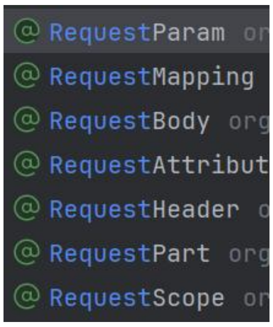
```
@RestController  no usages
@RequestMapping(path = "/employees")
public class EmployeeController {
```

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



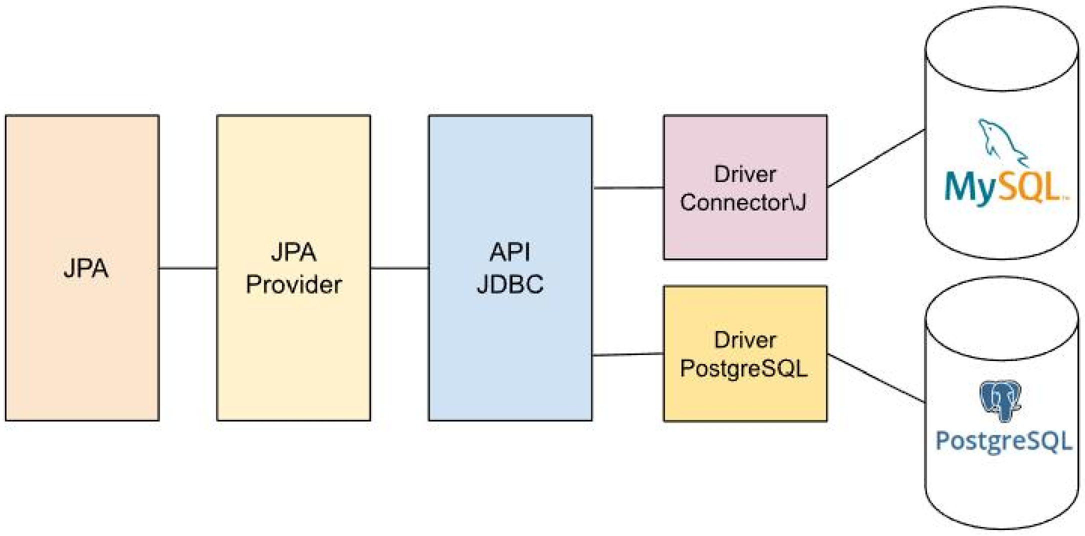
```
@RestController  no usages
@RequestMapping(path = "/employees")
public class EmployeeController {

    @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
```

Persistence Layer

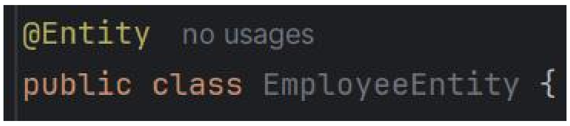
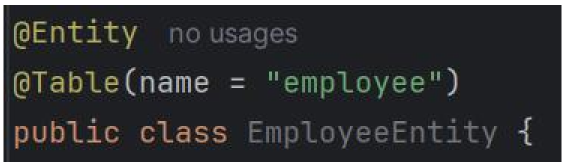


-
- **JPA**: Java Persistence API, nowadays its called **"Jakarta Persistence API"**
- Databases are also some servers but they hold some unique logic so that they can hold some data and query through it.
- Databases provide **JDBC** *(Java Database Connection)* drivers. Java uses JDBC to communicate 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**
  -
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F

- F
- F
- F
- F
- F
- F
- F
- F
- F
- Ff
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
- F
-