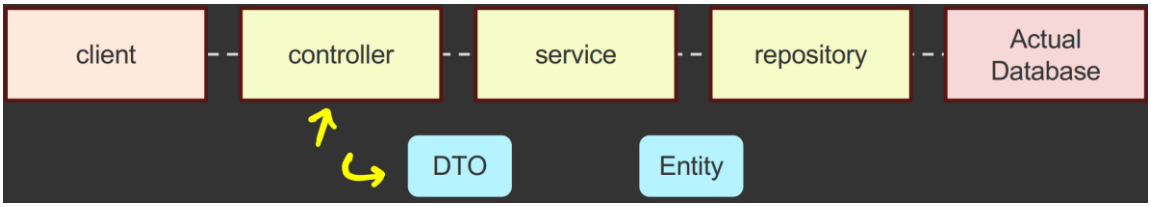
2.2 The Presentation Layer, DTO and Controllers

Presentation Layer:

The ***presentation layer*** is the ***first point*** of ***contact*** between the ***user*** and the ***application***, and it **communicates** with the **backend** using ***HTTP requests*** and ***responses***.

Spring Boot Web Project Structure:



Controller:

Spring MVC offers an **annotation-driven** programming model that allows developers to define ***request mappings***, ***handle request inputs***, ***manage exceptions,*** and more through ***annotations***. The ***@Controller*** and ***@RestController*** annotations are central to this model. Specifically, the ***@RestController*** annotation serves as a convenient shorthand for **combining** ***@Controller*** and ***@ResponseBody***, ensuring that all handler methods within the controller return data (such as ***JSON*** or ***XML***) directly in the **HTTP response body**, rather than *rendering a view*.

Creating controller in Spring Boot:

In Spring Boot, you typically use:

- ***@RestController*** for **REST APIs** (**JSON** response)
- ***@Controller*** for **web pages** (**Thymeleaf**, **JSP**, etc.)

Ex. *Method – 1* | *Method – 2*

```
@RestController
public class HomeController {
    @GetMapping("/home")
    public String homepage() {
        return "Hello Home";
    }
}
```

```
@Controller
public class HomeController {
    @GetMapping("/home")
    @ResponseBody
    public String homepage() {
        return "Hello Home";
    }
}
```

Thymeleaf is a **Java-based template engine** used for rendering dynamic HTML content in web applications. It is commonly used in **Spring Boot applications** to create web pages.

It allows you to:

- Generate HTML views on the server side.
- Dynamically display data in HTML.
- Bind form data to Java objects.
- Create layouts and reusable page fragments.

🏷️ ***@ResponseBody*** **Annotation in Spring Boot**

- Converts **Java Objects** (like POJOs, Strings, Collections) to **JSON** or **XML** (based on Content-Type).
- **Bypasses the View Resolver** — no HTML or JSP rendering.
- Commonly used in **REST APIs** to return API responses.

@RestController	*@Controller*
• **Purpose:** Designed for building **REST APIs**. • It directly returns data (JSON / Plain Text) as HTTP Response Body. • No view resolution happens. • @RestController = @Controller + @ResponseBody	• **Purpose:** Designed to return **View Names** (like **home.html** or **home.jsp**). • **Spring Boot Flow:** You return a **String** → Spring assumes it's a **View Name**. • It looks for a template file (like Home.html) in /templates folder. • Renders that HTML file as a **webpage**. • **If no file found?** → You get **404 Not Found** or **Whitelabel Error Page**.

🤔 How ***@RrestController*** or ***@ResponseBody*** converts Java objects to JSON:

In **Spring Boot**, **Jackson** plays a very important role when it comes to **converting** ***Java Objects*** to ***JSON*** and **vice versa**. This process is called ***Serialization*** and ***Deserialization***.

- ***Jackson*** is a **Java library (JSON Processor)** that handles:
  - ***Serialization*** **:** Java Object → JSON (when sending responses)
  - ***Deserialization*** **:** JSON → Java Object (when receiving requests)
- Spring Boot starter includes **Jackson by default** via: ***Dependency***

Dependency throw which Jackson added to Spring boot.

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>
```

🧐 *What is Request Mapping ?*

You can use the ***@RequestMapping*** annotation to ***map*** requests to **controller methods**. It has various attributes to match by ***URL***, ***HTTP method***, ***request parameters***, ***headers***, and ***media types***.

Ex.

```
@RestController
public class EmployeeController {
   @RequestMapping(
         value = "/employee/{id}",
         method = RequestMethod.GET
   )
   public EmployeeDTO getEmployeeById(@PathVariable("id") Long id) {
      return new EmployeeDTO(id, "Siba",
            "sibasundarj8@gmail.com",
            22, LocalDate.now());
   }
}
```

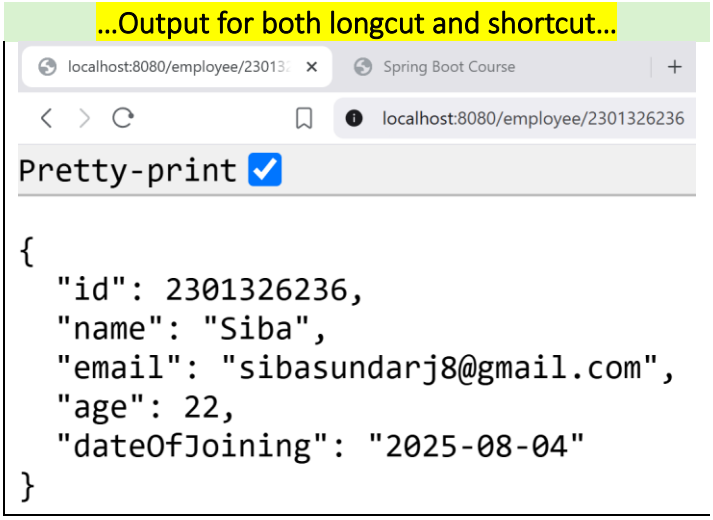
There are also ***HTTP method*** specific ***shortcut*** variants of ***@RequestMapping***:

- ***@GetMapping----------> @RequestMapping(***method = RequestMethod.***GET)***
- ***@PostMapping----------> @RequestMapping(***method = RequestMethod.***POST)***
- ***@PutMapping----------> @RequestMapping(***method = RequestMethod.***PUT)***
- ***@DeleteMapping----------> @RequestMapping(***method = RequestMethod.***DELETE)***
- ***@PatchMapping----------> @RequestMapping(***method = RequestMethod.***PATCH)***

Ex.

```
@RestController
public class EmployeeController {
   @GetMapping("/employee/{id}")
   public EmployeeDTO getEmployeeById(@PathVariable("id") Long id) {
      return new EmployeeDTO(id, "Siba",
            "sibasundarj8@gmail.com",
            22, LocalDate.now());
   }
}
```

...Output for both longcut and shortcut...



Dynamic URL Paths in Spring Boot:

Dynamic URLs are those where certain ***parts*** of the ***URL*** are ***variable*** and determined at ***runtime***.

In Spring Boot, you can handle these dynamic parts using:

- ***@PathVariable:*** Use path variables when the parameter is an ***essential*** part of the **URL** path that identifies a resource.
  Example: /user/{id} where {id} is dynamic.
- ***@RequestParam:*** Use query parameters when the parameter is ***optional*** and used for ***filtering***, ***sorting***, or other modifications to the request.
  Example: /user?id=10&name=Siba

@PathVariable	*@RequestParam*
Extracts values from the ***URL path***	Extracts values from ***query parameters***
Example URL: **/product/5**	Example URL: **/product?id=5**
Mandatory if defined in path	***Optional*** (can define default values)
Typically used for **resource identifiers**	Used for optional **filters**, **sorting**, etc.

Code example of taking input by dynamic URL path:

Previously we already seen the example of *@PathVariable* example. Now let's see the how take input using *@RequestParam* from user.

```
@RestController
@RequestMapping("/employee")
public class EmployeeController {
  // example of @RequestParam
  @GetMapping()
  public String getData(@RequestParam("name") String name,
                        @RequestParam(value = "ade", required = false) Integer age) {
    return "name: " + name + "<br>age: " + age;
  }
}
```

Default Behaviour:

- If you don't specify **required** = *true/false*, it's *treated* as *required*.
- If the **request param** is *missing*, Spring will *throw an error*.
- However, you can make it *optional* explicitly by setting *required = false*.

Note: When a **controller**'s *GET* method returns a *plain String*, the **web browser** interprets it as an *HTML response*. As a result, **newline** characters (\n) are **not** rendered as **line breaks**. To achieve line breaks in the browser, **HTML tags** such as
 should be used instead.

```
// example of @pathVariable
@GetMapping("/{id}")  no usages
public EmployeeDTO getEmployeeById(@PathVariable("id") Long id) {
    return new EmployeeDTO(id,  name: "Sibasundar Jena",  email: "sibasundarj8@gmail.com",
}
```

If you noticed these three name are always *same*, lets know why...

1. @GetMapping("/{id}")
   - *This tells Spring*: "*Expect a dynamic value* in the *URL* at *{id}* position".
   - Example URL: */employee/100*
2. @PathVariable("id")
   - This annotation *binds* the *{id}* from the *URL* to the *method parameter*.
   - "*id*" here refers to the *path variable name* defined in *{id}* in the *URL* pattern.
3. Long id
   - This is the *Java method parameter* where the actual value (e.g., 100) will be stored after extracting it from the URL.

What is @PostMapping ?

- Maps *HTTP POST* requests to a specific handler method in a Spring Controller.
- Commonly used for *submitting data* to the server, such as form submissions, creating resources (*Create* in CRUD), sending JSON payloads, etc.
- It is a specialized version of *@RequestMapping*(method = RequestMethod.*POST*).

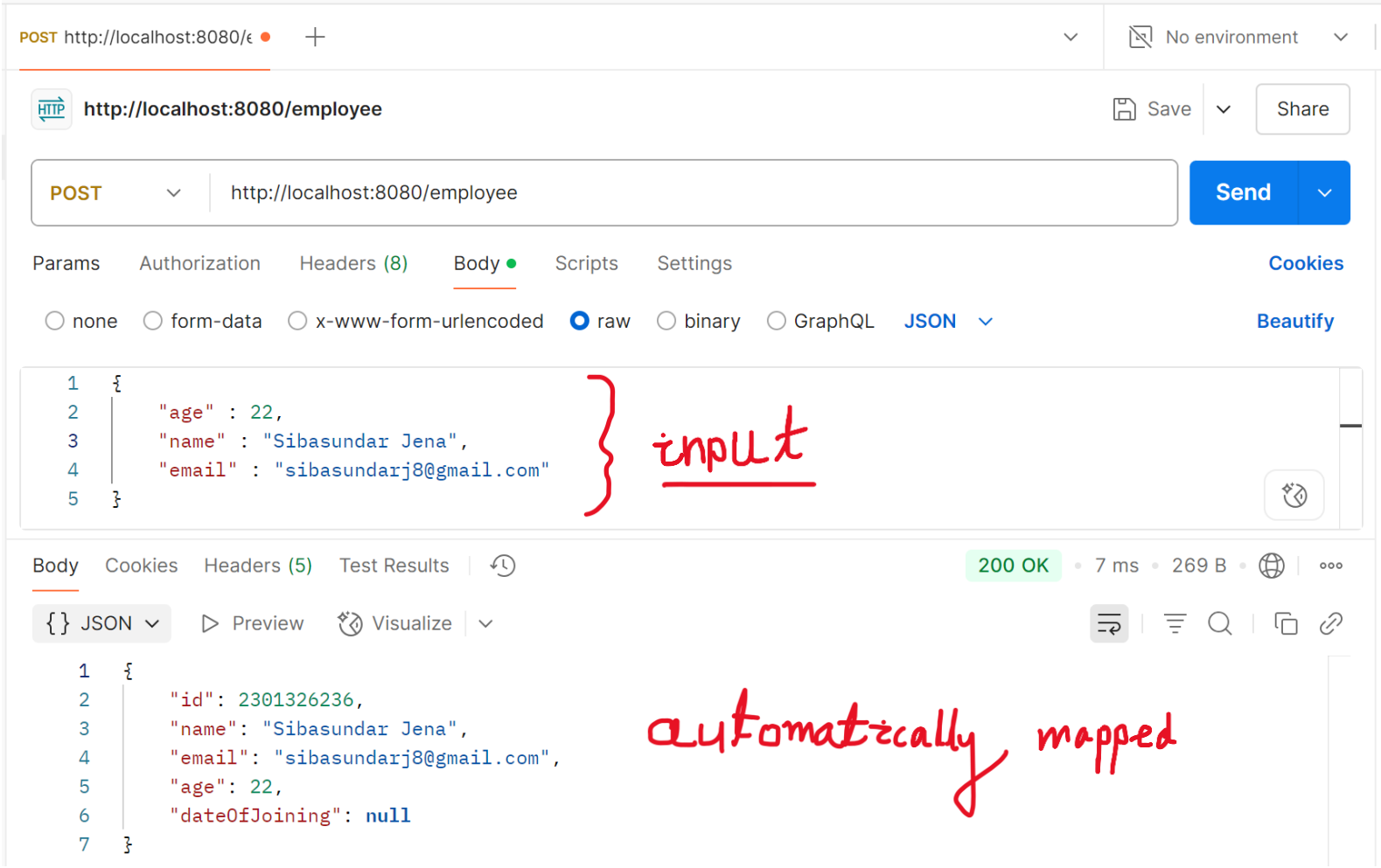
There is an annotation called *@RequestBody* which tells Spring:

"Take this incoming JSON from request body and convert it to this Java object parameter."

Example:

```
@PostMapping
public EmployeeDTO createNewEmployee(@RequestBody EmployeeDTO employee){
  employee.setId(2301326236L);
  return employee;
}
```

Here this method taking employee data as **JSON** file in parameter and converting it into EmployeeDTO **object** further. Spring reads the **JSON** body and uses ***Jackson*** (the default JSON processor) to **deserialize** it into a Java object (like DTO/Entity).


POST http://localhost:8080/e
No environment
http://localhost:8080/employee
Save
Share
POST
http://localhost:8080/employee
Send
Params
Authorization
Headers (8)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"age" : 22,
"name" : "Sibasundar Jena",
"email" : "sibasundarj8@gmail.com"
}
input
Body
Cookies
Headers (5)
Test Results
200 OK
7 ms
269 B
JSON
Preview
Visualize
{
"id": 2301326236,
"name": "Sibasundar Jena",
"email": "sibasundarj8@gmail.com",
"age": 22,
"dateOfJoining": null
}
automatically mapped