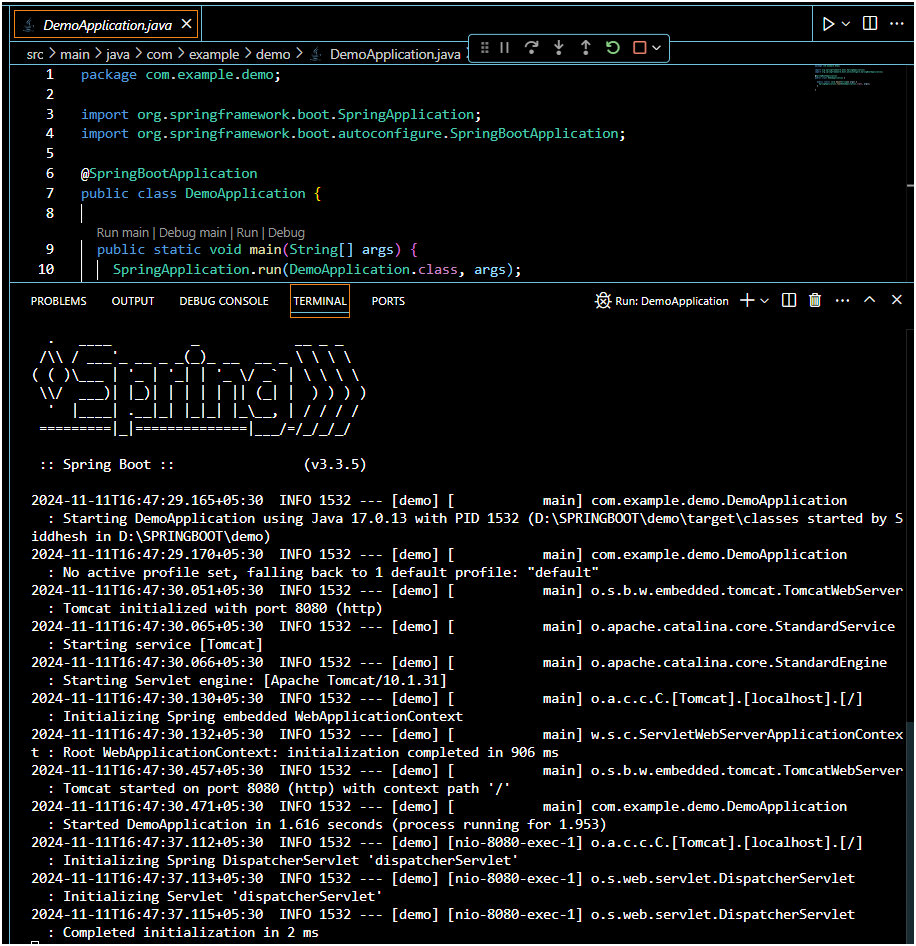
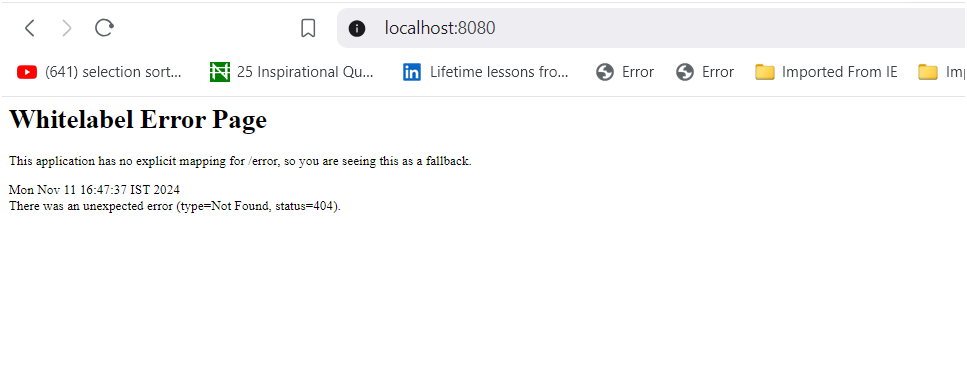


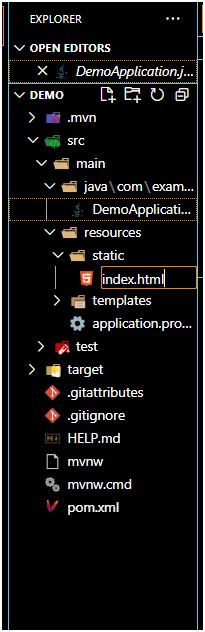
THEN RUNJAVA TO RUN UR CODE

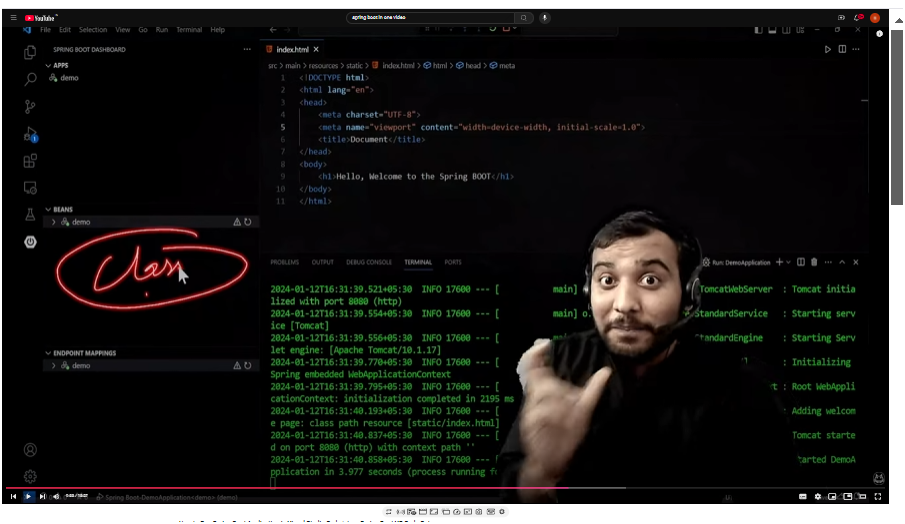


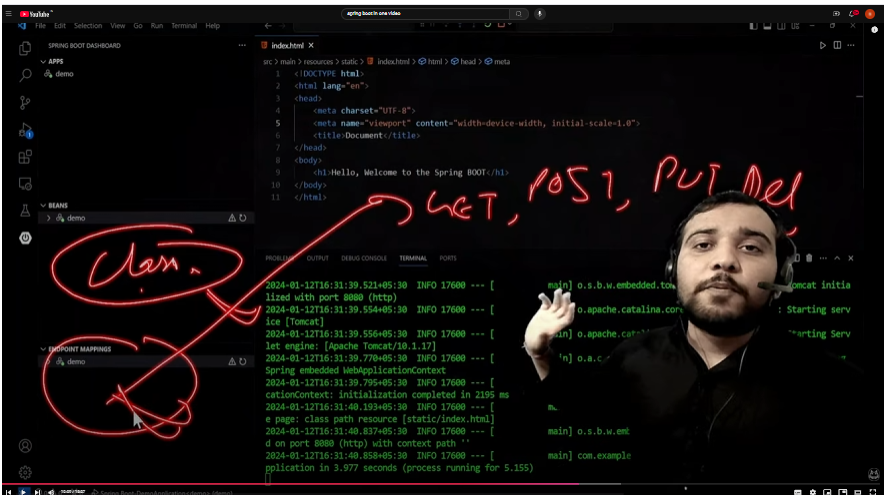
JUST RUN JAVA THEN UR TOMCAT SERVER WILL START AUTOMATICALLY



Then make index.html file in static folder







**MAVEN:-**

**Why Spring Initializr Uses Maven**

Spring Initializr uses Maven (or Gradle) by default because:

* It simplifies setting up the project.
* It automatically includes all the dependencies you select when creating the project.
* It sets up everything you need to get started quickly with Spring Boot.

**1. Dependency Management**

Every Spring Boot project usually needs several libraries (dependencies) like Spring Web, Spring Data, Spring Security, etc. Maven makes it easy to add these dependencies to your project.

With Maven:

* You just list the libraries you need in a file called pom.xml (Project Object Model file).
* Maven automatically downloads these libraries for you, so you don’t have to do it manually.

**2. Project Structure**

Maven creates a standard folder structure for your project. This structure makes it easier for other developers to understand and work with your project.

For example:

* src/main/java is where you put your main Java code.
* src/main/resources is where you put your configuration files.
* src/test/java is for your test files.

**3. Building the Project**

When you’re ready to run or deploy your Spring Boot application, Maven can package everything together. For example:

* Maven compiles your code.
* It runs any tests you’ve written.
* It packages your application into a file (like a .jar file) that can be run or deployed.

**RESTAPI:-**

Example:

If you're using a weather app:

* The app sends a request to the REST API, saying "Get the current weather for my location."
* The API asks the server for the weather data.
* The server replies with the weather info, and the API brings it back to the app, so you see the weather forecast.

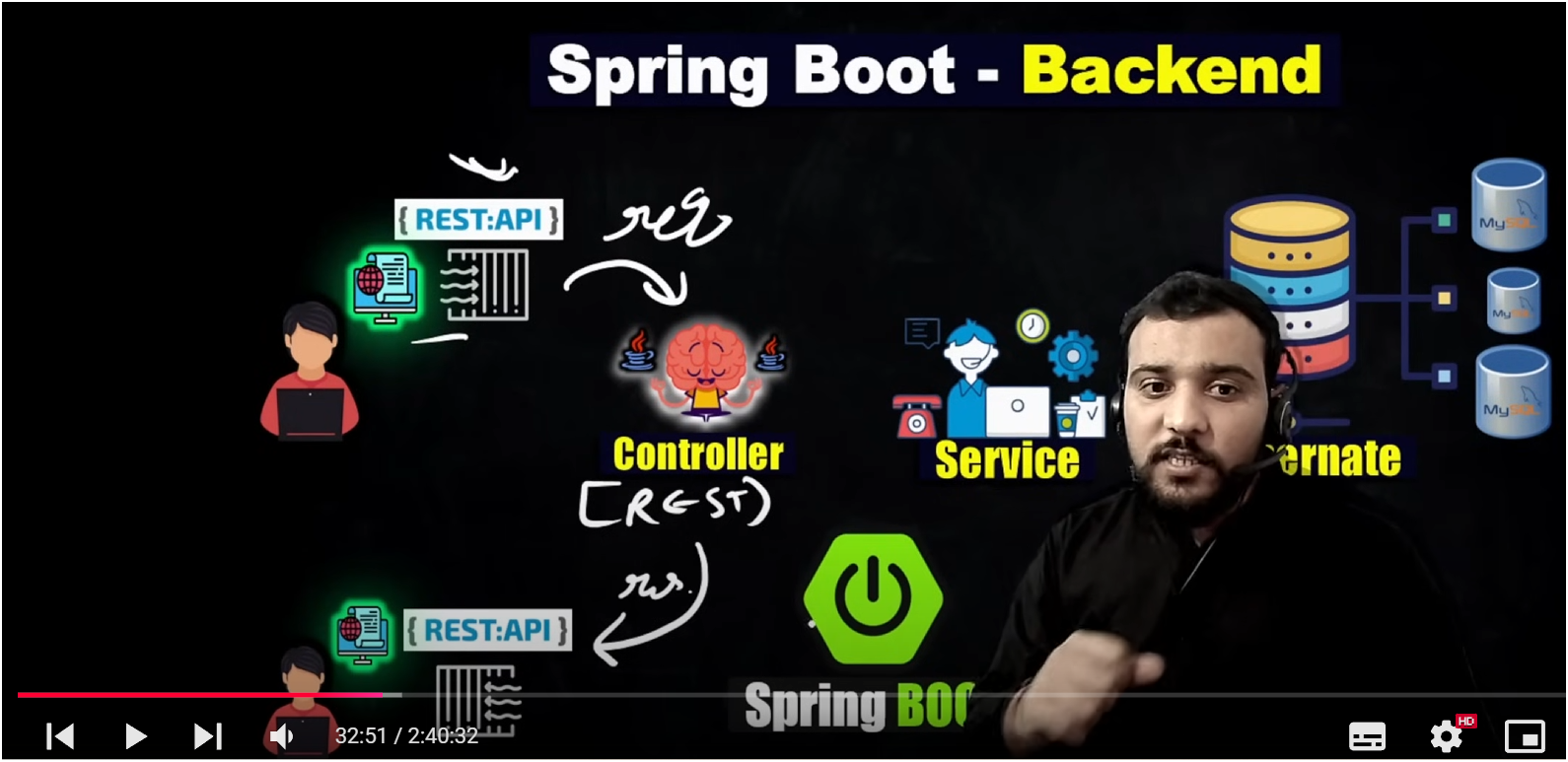
In short, a REST API is like a messenger that allows two different software systems (like a mobile app and a server) to talk to each other and exchange data easily over the web.

**CONTROLLER:-**

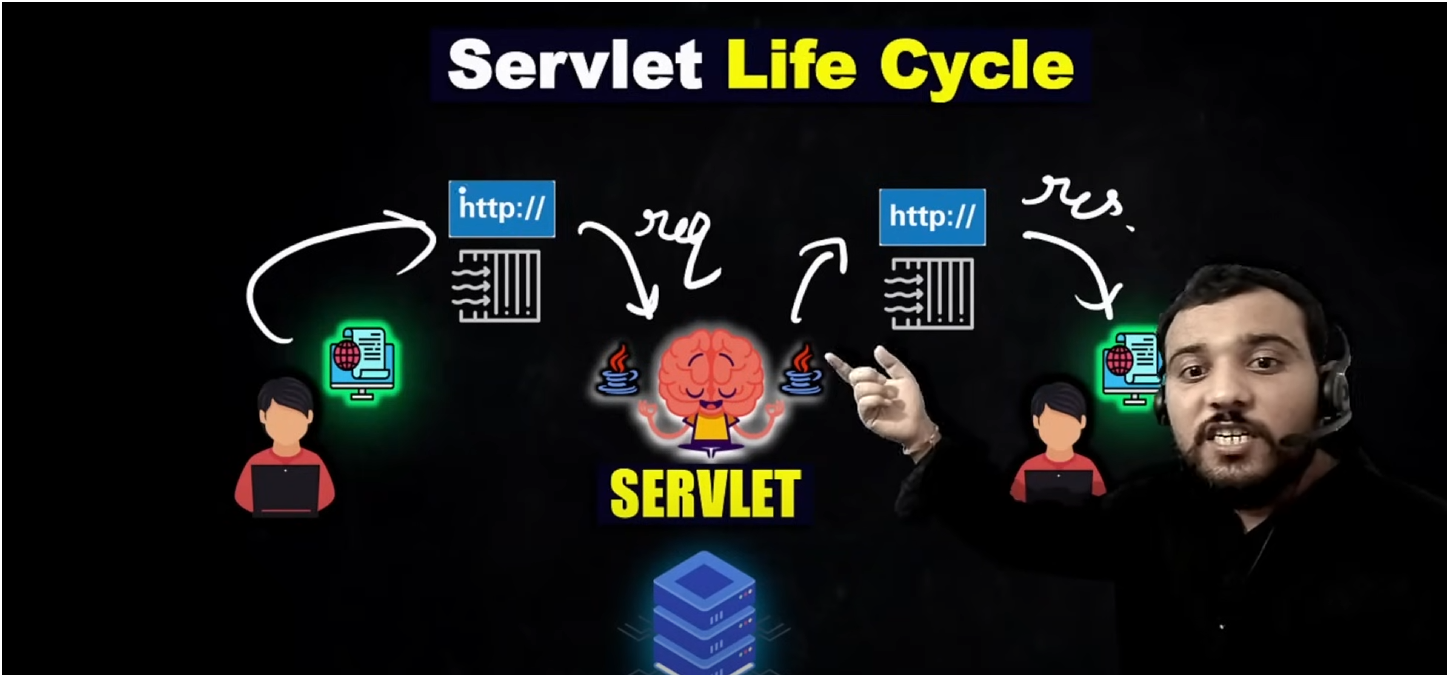
In Spring Boot, a **controller** is a class that handles incoming requests and sends responses back to the user. Think of it as the part of your application that "controls" the flow of data between the user and the backend logic of your application.Only returns text,xml,list not integer

**What a Controller Does**

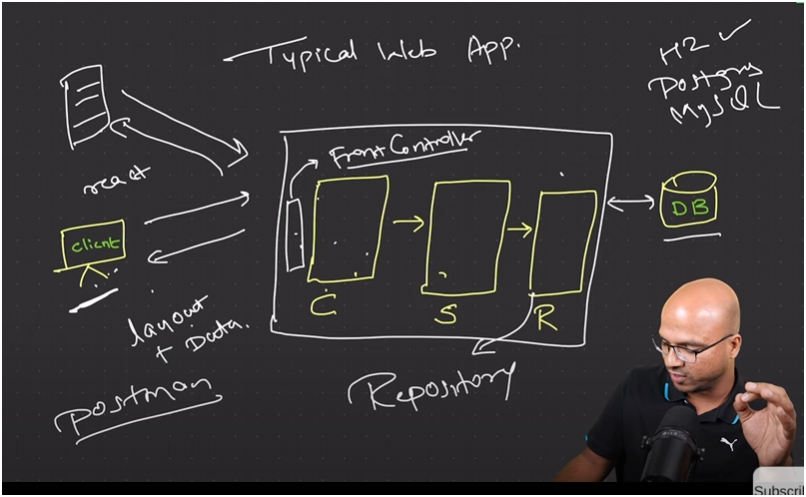
1. **Listens for Requests**: The controller waits for specific types of requests (like requests for data or actions like "create" or "delete").
2. **Processes the Request**: When it receives a request, it may call a service or perform a specific action to handle the request.
3. **Returns a Response**: After processing, it sends back a response (like some data in JSON format or a status message).



FIRST SERVLET WAS RESPONSIBLE TO GET THE REQUEST AND THEN SEND THE RESPONSE BUT IT REQUIRES A LOT MANNUAL CONFIGURATION TO AVOID THESE WE MAKE USE OF REST CONTROLLER

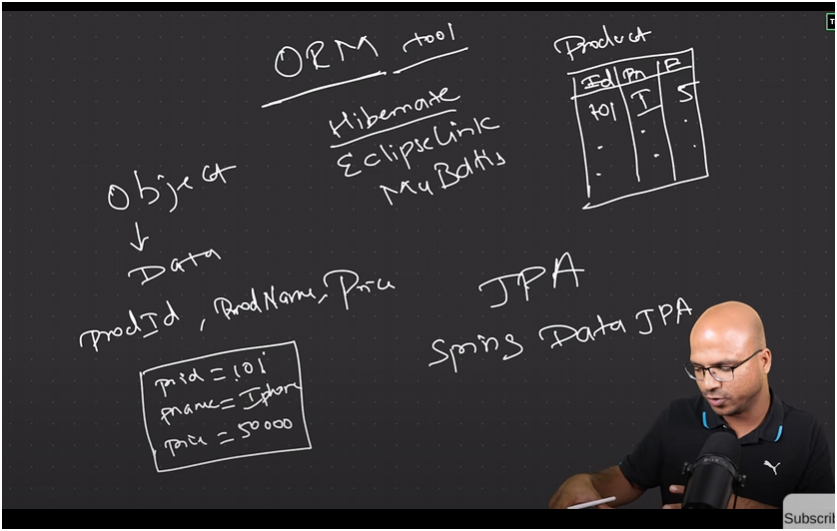


**SPRING DATA JPA(DATABASE CONNECTIVITY) FOR WE NEED TO WOR WITH REPOSITORY LAYER**



ORM (OBJECT RELATIONAL MAPPING)

WHEN U WANT TO MAP OBJECTS WITH DATABASE



**Key Points:**

1. **ORM Tool**:
   * ORM tools help map **objects in code** (like Java objects) to **database tables**.
   * Examples of ORM tools are **Hibernate, EclipseLink,** and **MyBatis**.
   * The idea is to take an object with properties (e.g., product ID, product name, price) and store it in a database table without writing complex SQL queries.
2. **Objects and Database Tables**:

* An object with properties like prodId, prodName, and price can be represented as a record in a database table called Product.NO OF ROWS IS EQUIVALENT TO NO OF OBJECTS

1. **JPA (Java Persistence API)**:

* JPA is a standard API used in Java for ORM.
* **Spring Data JPA** is a popular implementation of JPA that simplifies database interactions in Spring applications.

**@RequestMapping in a REST Controller:-**

@RequestMapping in a **REST Controller** is an annotation in Spring Boot that helps map web requests to specific methods in a controller.

**Simple Explanation:**

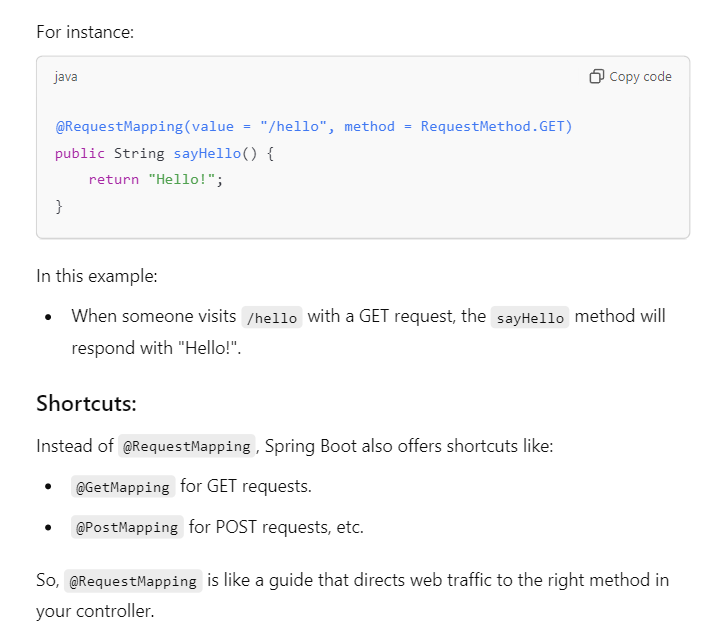
Think of @RequestMapping as an address label that tells your REST API where to go when someone makes a request.

For example:

* If you want a certain method to respond when someone visits a URL like /hello, you put @RequestMapping("/hello") above that method.
* Then, whenever someone sends a request to /hello, this specific method will be called to handle it.

**Key Parts:**

* @RequestMapping can be used to specify:
  + **Path** (URL) where the request should go.
  + **HTTP Method** (GET, POST, etc.) so the server knows what type of action is requested.



**IoC Container (Inversion of Control Container):**

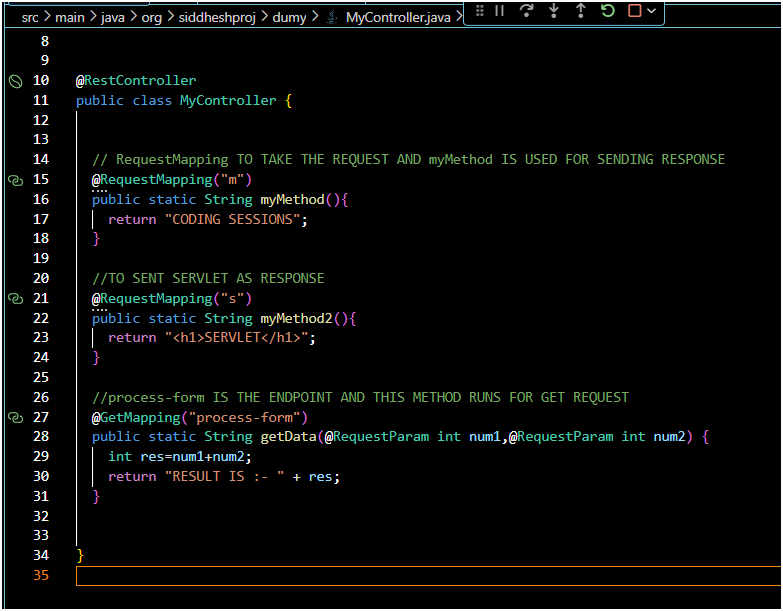
Think of the IoC container as a smart warehouse in Spring that **creates and manages objects for you**. When you need something, you don’t have to build it yourself — you just ask the container for it, and it gives you a ready-made object.

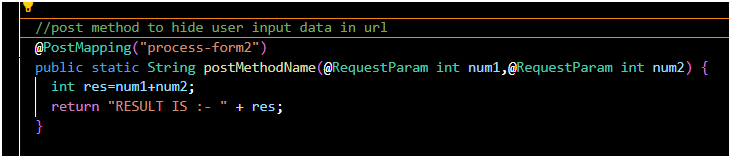
**Dependency Injection (DI):**

Dependency Injection is like having the IoC container **supply the parts an object needs automatically**. Imagine you have a car, and the engine and tires are "dependencies" that the car needs to work. Instead of the car building its own engine and tires, the IoC container provides these parts and "injects" them into the car, so everything is ready to run.

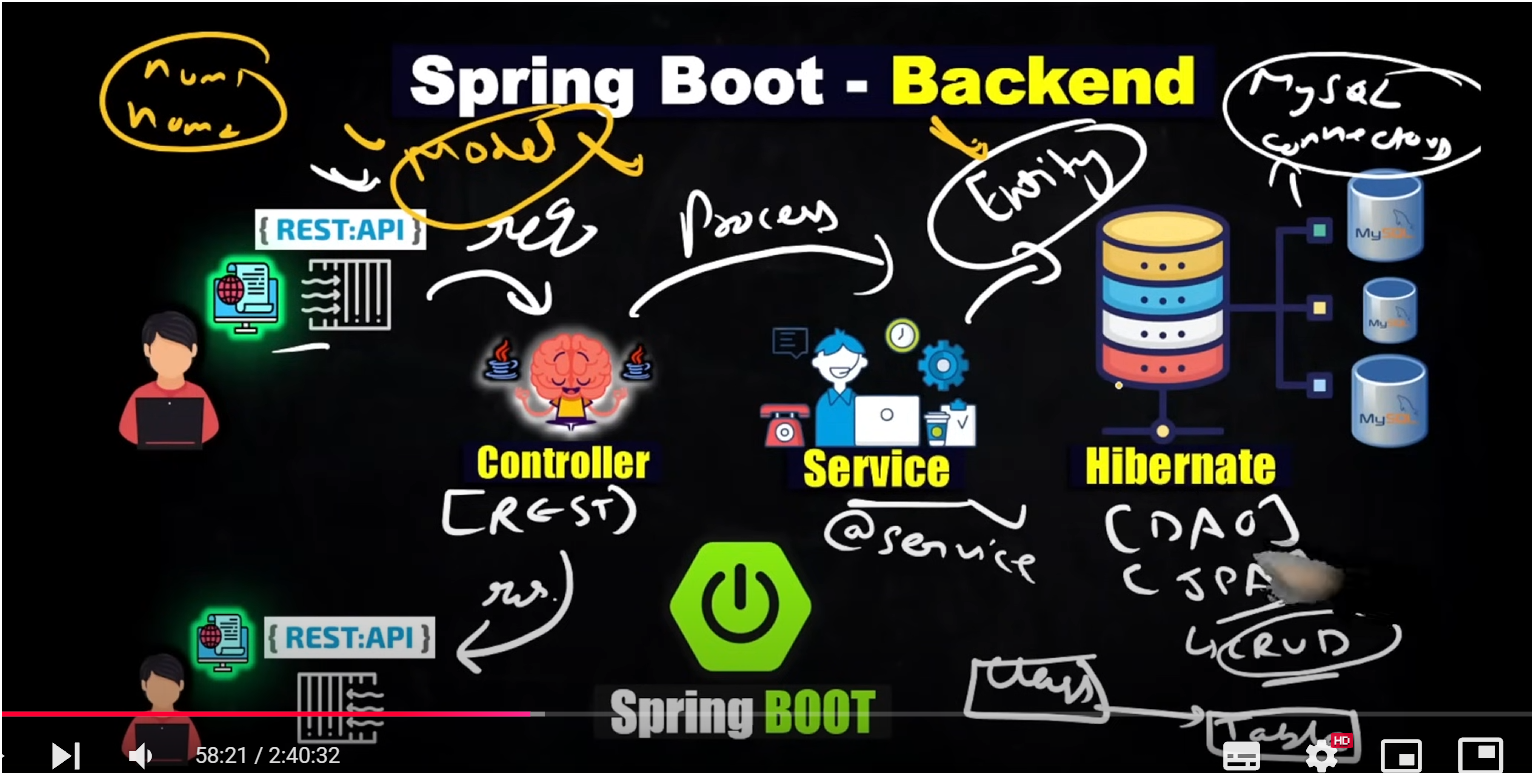
WORKING HOW TO SEND REQUEST FROM USER TO CONTROLLER AND THEN RESPONSE FROM CONTROLLER TO USER

* **@GetMapping("process-form")**: This maps the HTTP GET request to the method. When you visit /process-form?num1=5&num2=10, this method will execute.
* **@RequestParam int num1, @RequestParam int num2**: These annotations bind the query parameters num1 and num2 from the request to the method parameters.





STRUCTURE



1. **Controller**: The controller is the entry point for the application, where it handles incoming requests (often through REST APIs). It receives data from the client (like a web or mobile app), processes it, and directs it to the right part of the backend.
2. **Service**: This layer contains the main business logic of the application. The controller passes data to the service layer, where operations and processes are performed. It's typically marked with @Service annotations in Spring Boot, making it easy to identify and manage.
3. **Entity**: Entities represent the data structure of the application, often corresponding to tables in the database. For instance, if you're managing user data, a User entity would contain fields like name, email, etc.
4. **Hibernate and DAO/JPA**: Hibernate is a framework for managing the database, and it allows you to map Java objects to database tables. DAO (Data Access Object) and JPA (Java Persistence API) are used to perform CRUD (Create, Read, Update, Delete) operations on the database.
5. **MySQL Database**: This is where data is stored. Spring Boot connects to the MySQL database (or any other supported database) to store, retrieve, update, and delete data.

In short:

* The **Controller** receives requests.
* The **Service** processes them.
* The **Entity** defines data structure.
* **Hibernate** handles database operations.
* The **MySQL Database** is where data is stored.

**CRUD OPERATIONS:-**

* **GET**: You want to view the profile of a user (GET /user/1).
* **POST**: You want to create a new user (POST /user).
* **PUT**: You want to update an existing user's profile (PUT /user/1).
* **DELETE**: You want to delete a user (DELETE /user/1).

**IN SPRING BOOT PROJECT WE BASICALLY WORKS IN MAIN AND TEMPLATES FOLDER**

**Summary:**

1. **pom.xml**: A configuration file for managing project dependencies and build process.
2. **Main folder (src/main/java)**: Contains the application's Java code, including the entry point (@SpringBootApplication class), controllers, services, and repositories.
3. **Templates folder (src/main/resources/templates)**: Stores HTML view templates (like Thymeleaf or FreeMarker templates) that are rendered and sent to the browser.

These components work together to structure a Spring Boot application. The pom.xml manages dependencies, the main folder holds your business logic, and the templates folder contains the frontend views that are served to the user.

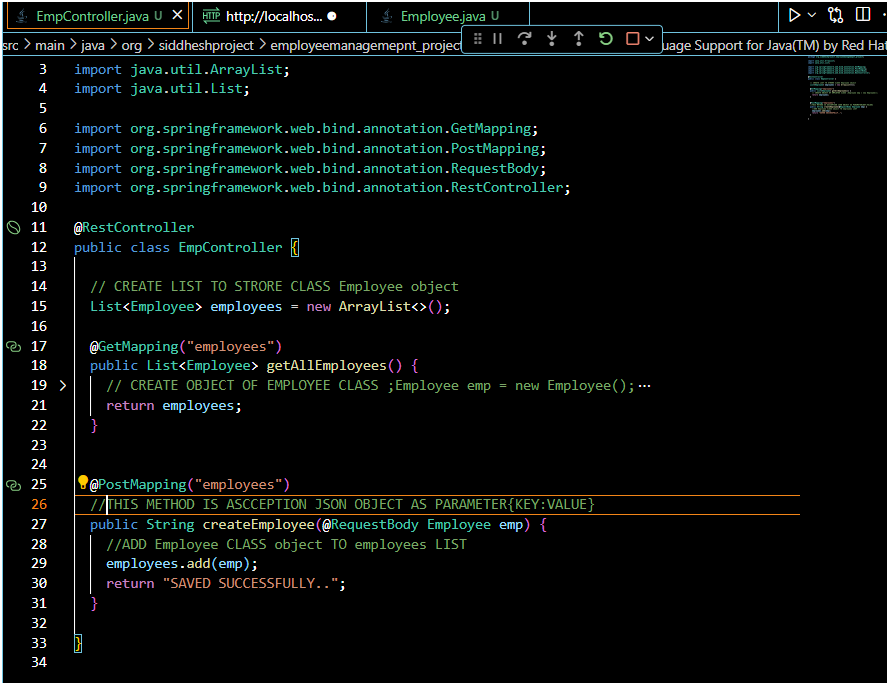
**Standalone project:-**

A **standalone project** is a software project that **works independently**, meaning it **does not require any external system or platform** to run. It can be executed on its own without needing to connect to other services, applications, or servers.

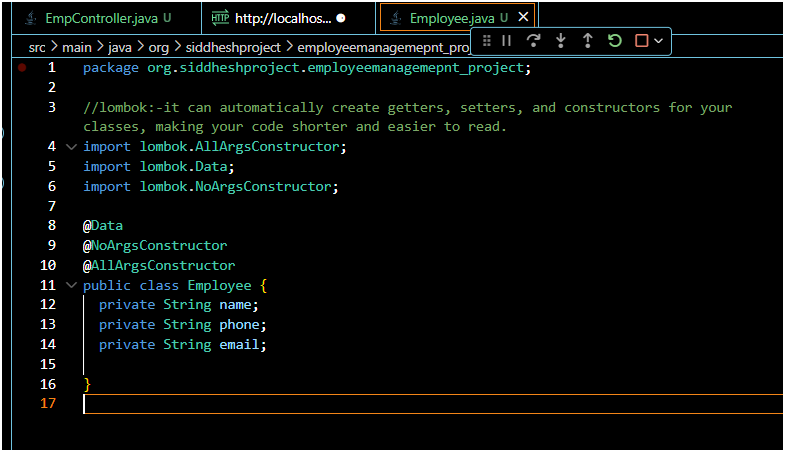
**Examples of Standalone Projects:**

1. **Desktop Applications**:
   * A calculator app on your computer that doesn't need to connect to the internet or any other system to function
   * **Simplicity**: Easier to develop, test, and deploy since they don’t rely on external systems.
   * **Portable**: You can move or share the project without worrying about complex configurations or dependencies.

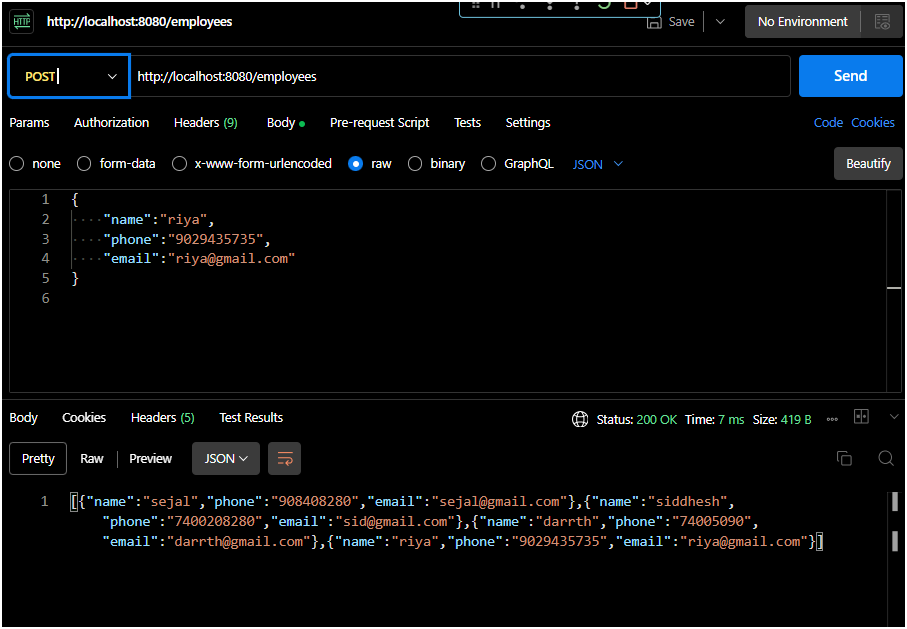
TO ADD EMPLOYEE DETAILS IN A LIST



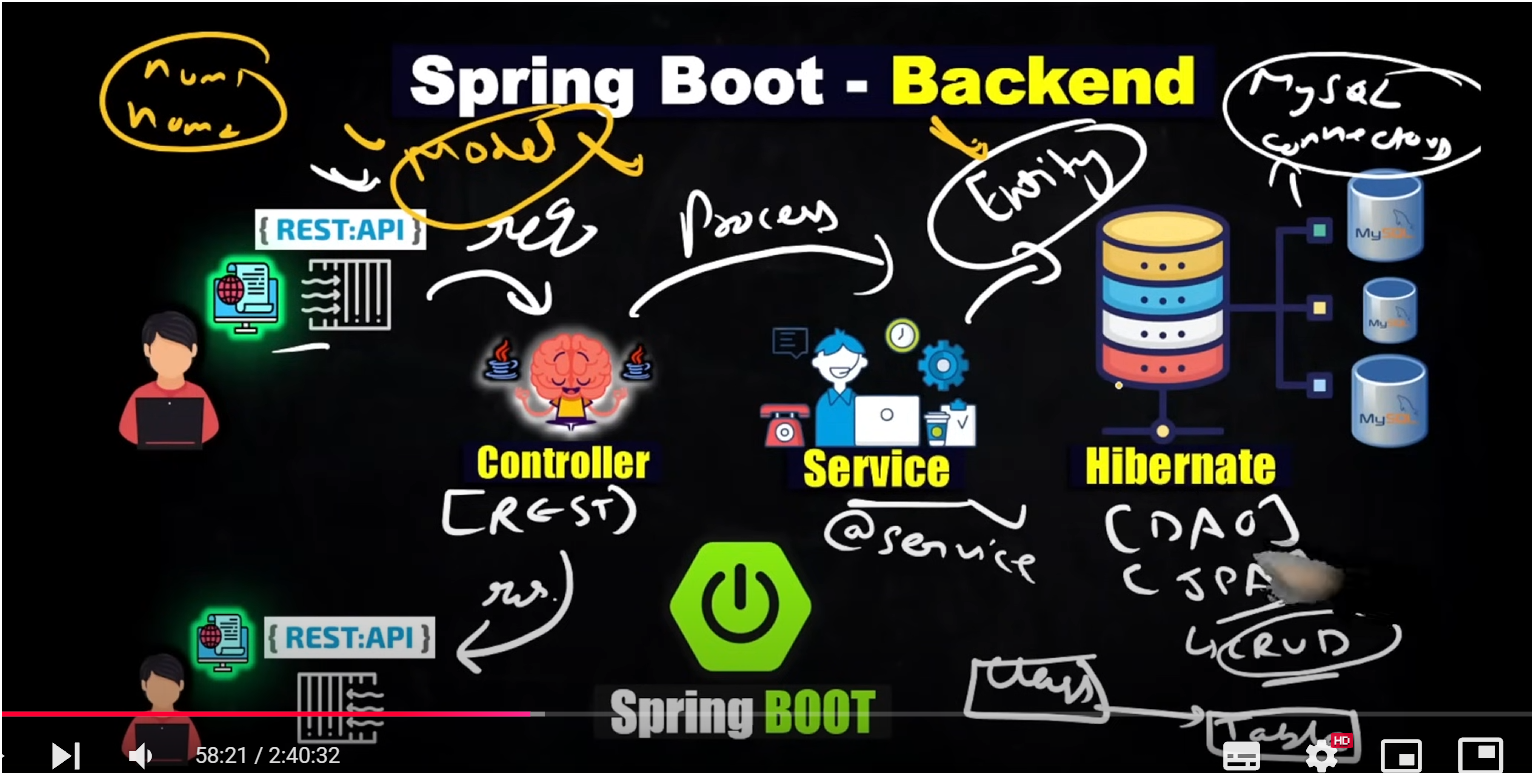
CREATE EMPLOYEE OBJECT THAT CONTAIN ALL PARAMETERS. VARIBLES



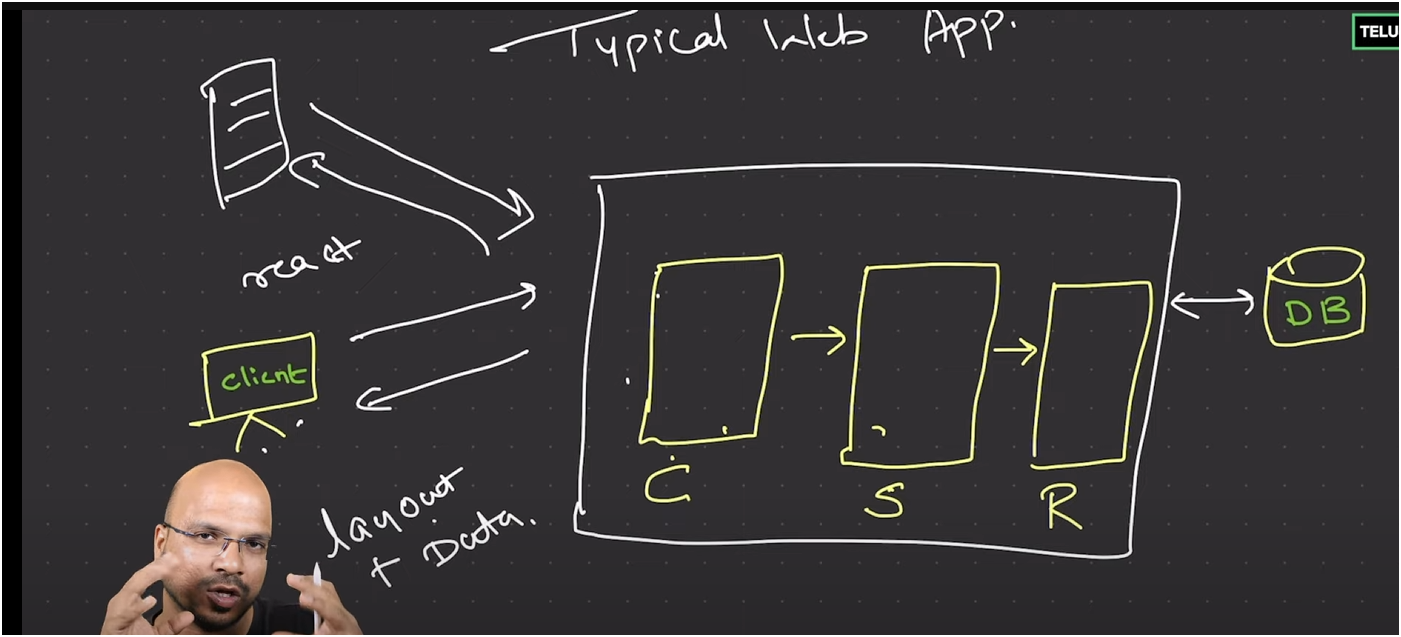
**POSTMAN**



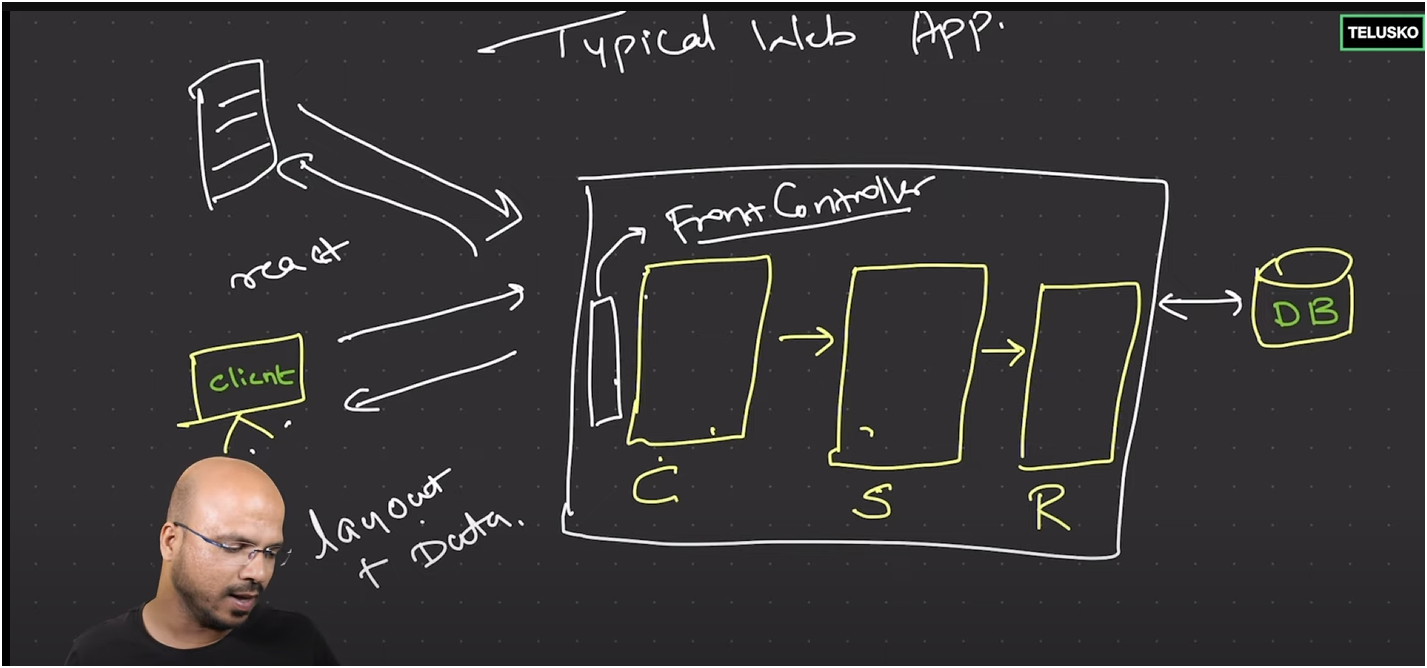
**REST CONTROLLER TALKS WITH SERVICE LAYER AND SERVICE LAYER TALKS WITH DATA BASE**



CREATE CLASS FOR DIFFERENT LAYERS



IN SPRINGB BOOT WE CAN CREATE MULTIPLE CONTROLLER FRONT CONTROLLER IS THE ONE WHICH STATES WHICH REQUEST IS FOR WHICHH CONTROLLER AND WHICH CONTROLLER WE NEED OT SELECT



**SERVICE LAYER**

CONTROLLER JUST USED FOR ACCEPTING AND SENDING THE RESPONSE BUT SERVICE LAYER IS THE ONE WHICH IS RESPNSIBLE FOR BUSINESS LOGIC

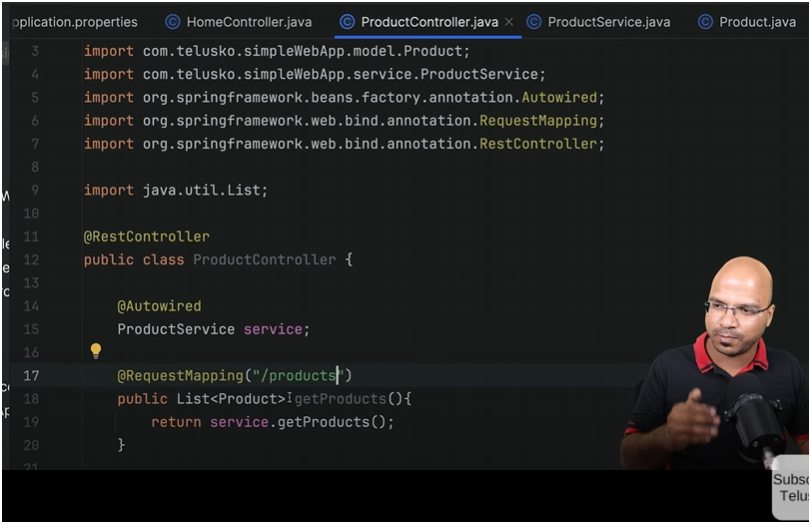
ALWAYS REMEMBER JUST WRITE @Service NOTATION FOR SERVICE LAYER OTHER WISE OBJECT WONT BE CREATED FOR SERVICE LAYER CLASS

MVC STANDS FOR

MODEL(DATA) VIEW(SERVICE LAYER) CONTROLLER

**Autowired**

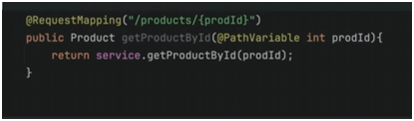
USED FOR CREATING OBJECT OF SERVIDE LAYER AS WE WRITE THIS IT CREATES OBJECT AUTOMATICALLY FOR U



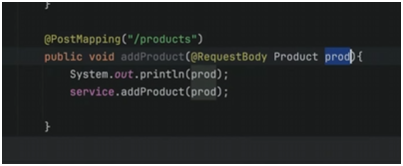
**TO GET A PARTICULAR PRODUCT DETAILS OF PARTICULAR PRODUCT ID**

{prodId}=>DYNAMIC CHANGING VALUE

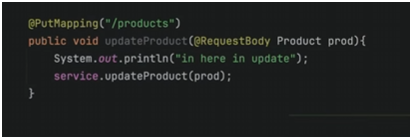
@PathVariable=>TO GET PRODUCTID DETAILS FROM URL PATH



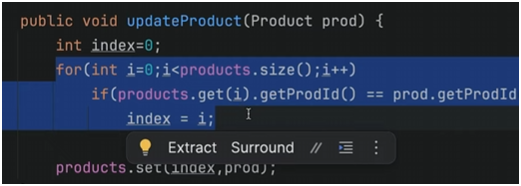
@RequestBody WHEN U WANT TO SENT Product CLASS OBJECT



**TO UPDATE A PRODUCT DETAILS**

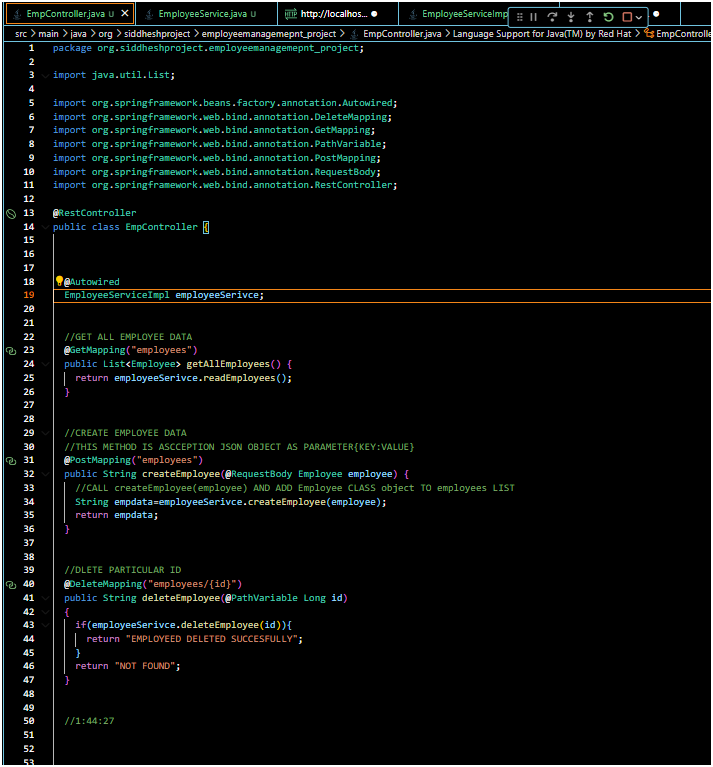


SERVICE LAYER LOGIC FOR UPDATE PRODUCT DETAILS



CODNG WALLAH SIR

**CONTROLLER LAYER(TO MANAGE THE REQUESTS)**



**SERVICE LAYER (MANAGE ALL THE LOGIC OF CRUD OPERATIONS)**

